Exam in INF236 spring 2015
Problem set with solutions

Problem 1a

Benchmarking of a sequential program reveals that 95% of the execution time is spent inside functions that are amenable to parallelization. What is the maximum speedup we could expect from executing a parallel version of this program on 10 processes?

Solution

S = Ts/Tp
Tp <= 0.95Ts + 0.05Ts

S <= Ts / (0.95Ts/10 + 0.05Ts) = 6.9

Problem 1b

For a problem size of interest, 6% of the time of a parallel program is spent inside I/O functions that are executed on a single processor. What is the minimum number of processes needed in order for the parallel program to exhibit a speedup of 10?

Solution

We want S = Ts/Tp >= 10.
The lowest parallel running time using p processes is Tp = 0.94Ts/p + 0.06Ts.
Solving Ts/(0.94Ts/p + 0.06Ts) >= 10 gives p >= 23.5.
p must be an integer, so p=24 is the minimum number of processes needed to obtain S>=10.

Problem 1c

Mark those statements that are true

In practice it is possible to get higher speedup than p when using p processes.

True. This can happen because we now get access to more memory and the sequential parts can execute faster.

In C we should always stride through two dimensional arrays by columns.

False. One should access two dimensional arrays by rows in C.

If a loop gives the same result when executed forward as it does when executed backwards then it is always safe to parallelize using OpenMP.

False. One should be able to execute the iterations in any order, not just forward and backwards.

In MPI it is impossible to tell which process is printing a line to the screen.

False. One can print out the process ID together with the text.

Moore´s law tells us that the speed of a computer will double roughly every 18 month.

False. Moore´s law says that the number of transistors in a given area will double every 18th month. This does not necessarily result in faster computers.

Latency is the time it takes before a message start to arrive at its destination.

True.

Problem 2a

Assume that we have an array defined as follows:

double A[n][n];

where n is a postive integer value.

Now for every element A[i][j] where 0<i,j<n-1 we want to replace the value with the average of itself and its four closest neighbors. That is, we want to compute

A[i][j] = (A[i][j]+A[i+1][j]+A[i-1][j]+A[i][j+1]+ A[i][j-1])/5.0;

In the following problems we will consider how to solve this on a parallel computer with distributed memory and programming using MPI. You can assume that each process can both send and receive messages simultaneously.

First problem description starts here
Consider an MPI program using p processes for doing this where A is partitioned so that process i holds rows i*(n/p) through (i+1)*(n/p)-1 of A. Derive an expression for the time taken by the communication, where ts is the startup cost (latency) and tb is the time to send one double value.

Solution
Each process (except the topmost and bottom most ones) need to send their top and bottom rows to the adjacent process. Thus each process must send (and receive) two rows consisting of n values. Since we assume that sending and receiving happens simultaneously the communication time is given by

Time = 2(ts + n tb).

This is assuming that p>2. If there are only two processes the time is given by ts + n tb.

Problem 2b

Next, repeat the calculations from problem a) but now assuming a two dimensional partitioning of A into √p x √p square blocks each of size n/√p x n/√p.

Solution
Each process must now send and receive four messages, each of length n/√p. Thus for p>4 we get Time = 4(ts +n/√p tb).

For p=4 we get Time = 2(ts +n/√p tb).

Problem 2c

Discuss how the speedup of the methods from a) and b) will be affected if we increase either n or p (but not both).

Solution

The main observation is that when n increases the speedup will get closer and closer to p, but when p increases the speedup will converge towards a fixed limit. Intuitivly the speedup is dependent on the computation to communication ratio. Computation grows as n^2/p while communication in a) grows as n and in b) as $n/\sqrt{p}$. Thus when n increases, the amount of computation will increase faster than the amount of communication, thus giving a speedup closer to p. When p increases the computation will decrease linearly with p, while the communication in a) will remain fixed and in b) decrease by a factor of $\sqrt{p}$. Thus in this case computation is decreasing faster than communication giving a speedup further and further from p.

More formally, using the method from 2b we get

$S= n^2/((n^2 / p) + 4(ts + tb\ n/\sqrt{p}))$

If p increases then both the term (n^2 / p) and $tb\ n/\sqrt{p}$ will decrease, but ts will remain unchanged. Thus the speedup will converge to $n^2 / (1 + 4(ts+tb))$.

When only n increases one can rewrite S as follows (by multiplying though with p/n^2):
$S= p/(1 + 4\ p/n^2\ (ts + tb\ n/\sqrt{p}))$.

Then the term $(4\ p/n^2\ (ts + tb\ n/\sqrt{p}))$ will tend to zero as n increases and the speedup will get closer and closer to p.

The case when using the layout from 2a is similar but the speedup when n increases will be less than in 2b. Also note that we cannot increase p above n thus limiting any speedup to at most n.

Problem 2d

For what values of ts relative to tb, p and n is the method from b) faster than the method from a)?

Solution

This is the case if $4(ts +n/\sqrt{p}\ tb) < 2(ts + n\ tb)$.
Solving for ts we get

$ts < n\ tb(1 - 2/\sqrt{p})$

Thus it is not always the case that the 2D layout is better than the 1D.

Problem 2e

Derive the formula for how to calculate the matrix dimension n if you want to do weak scaling experiments using either the method from a) or from b). You can assume that $n = n_0$ for $p = 1$.

Solution

The sequential work for p = 1 is n_0^2. Thus we must make sure that each process gets this amount of work when p increases. In both parallel algorithms the sequential work per process is given by n^2/p. Thus we must determine n so that $n^2/p. = n_0^2$ remains true. Solving for n we get $n = n_0\sqrt{p}$.

Problem 3a
Parallelize the following code using openMP pragmas. Be sure to explicitly specify the "schedule" options that should be used, even if you want to use the default options. Rewrite the code together with the pragmas you insert. If necessary you can assume that the variable p represents the number of threads to be used. Assume that N is large (in the tens of thousands or more). You must explicitly list all variables within the range of a parallel pragma that are private using the private() directive.

```
for (i=0;i<N;i++) {
  for (j=0;j<N;j++) {
    A[i,j] = max(A[i,j],B[i,j]);
  }
}
```

Solution
```
#pragma omp parallel for private(j) schedule(static,N/p)
for (i=0;i<N;i++) {
   for (j=0;j<N;j++) {
      A[i,j] = max(A[i,j],B[i,j]);
   }
}
```

Problem 3b
Parallelize the following code using OpenMP pragmas. Be sure to explicitly specify the "schedule" options that should be used, even if you want to use the default options. Rewrite the code together with the pragmas you insert. If necessary you can assume that the variable p represents the number of threads to be used. Assume that N is large (in the tens of thousands or more). You must explicitly list all variables within the range of a parallel pragma that are private using the private() directive.

```
C[0] = 1;
for (i=1;i<N;i++) {
  C[i] = C[i-1];
  for (j=0;j<N;j++) {
    C[i] *= A[i,j] + B[i,j]
  }
}
```

Solution
```
C[0] = 1;
for (i=1;i<N;i++){
   C[i] = C[i-1];
#pragma omp parallel for schedule(static,N/p) reduction(*:product)
   for (j=0;j<N;j++){
      product *= A[i,j] + B[i,j];
   }
```

```
    C[i] = product;
}
```

Problem 3c
The transpose operation on a square matrix A[n][n] reorders the elements of A such that the element stored in position A[i][j] is moved to position A[j][i]. This is done for all values of i and j. Write a parallel code segment using OpenMP to perform this operation. You can assume that the elements of A are of type double.

You must explicitly list all variables within the range of a parallel pragma that are private using the private() directive. Be sure to explicitly specify the "schedule" options that should be used, even if you want to use the default options.

Solution

```
#pragma omp parallel for private(j,t) schedule(dynamic)
for (i=0;i<N;i++){
    for (j=i+1;j<N;j++){
       t = A[i][j];
       A[i][j] = A[j][i];
       A[j][i] = t;
    }
}
```

Note that one should use dynamic or guided load balancing as the amount of work in each iteration decreases.

Problem 3d
The following code segment is part of an algorithm for computing the minimum edit distance between two character strings stored in arrays str1[size] and str2[size]. Explain how you would rewrite the code so that it can be parallelized using OpenMP. If you have time, you can also write the code.

```
for(i=1;i<size;i++) {
  for(j=1;j<size;j++) {

    int x = cost[i-1][j] + 1;
    int y = cost[i][j-1] + 1;
    int z = cost[i-1][j-1] + (str1[i-1] != str2[j-1]);

    cost[i][j] = min(x, min(y,z));
  }
}
```

Solution
*The main observation here is that one can compute the entries of each diagonal of the cost array in parallel. Thus the code must be rewritten so that there is one loop that runs over the diagonals. There are 2*size - 3 diagonals that must be computed and for the first size-1 diagonals the length of the diagonal increases by one for each diagonal. For the remaining size-2 diagonals the length decreases by one for each diagonal. Thus it is easier to do this using two separate loops.*

```
for(i=1;i<size;i++) {
#pragma omp parallel for schedule(static,i/p)
  for(j=1;j< i+1;j++) {

// Coordinates are given by [i-j+1][j]

    int x = cost[i-j][j] + 1;
    int y = cost[i-j+1][j-1] + 1;
    int z = cost[i-j][j-1] + (str1[i-j] != str2[j-1]);

    cost[i-j+1][j] = min(x, min(y,z));
  }
}

for(i=1;i<size-1;i++) {
#pragma omp parallel for schedule(static,i/p)
  for(j=i+1;j< size;j++) {

// Coordinates are given by [size-j+1][j]

    int x = cost[size-j][j] + 1;
    int y = cost[size-j+1][j-1] + 1;
    int z = cost[size-j][j-1] + (str1[size-j] != str2[j-1]);

    cost[size-j+1][j] = min(x, min(y,z));
  }
}
```