Parallel Computing: Problem Set 3 Document

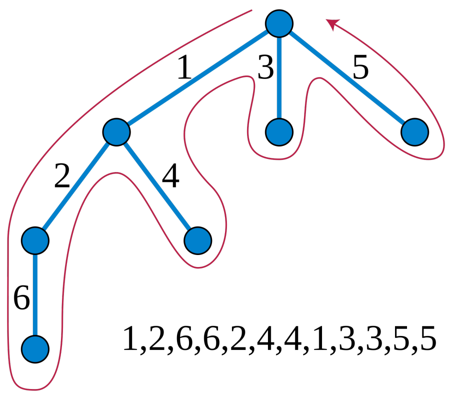
Due on Dec 5th at 23:59

Mailbox: yuzhh1@shanghaitech.edu.cn
Student ID: 2020533156
Student Name: Zhenghong Yu

Contents

1 Problem 1

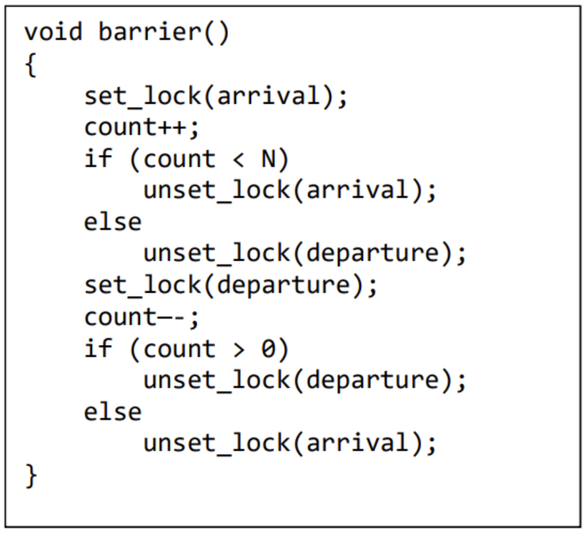
Consider the same situation as in problem 1 of problem set 2. Give another algorithm for all-to-all broadcast which takes time $(t_s + t_w m)(p - 1)$.
Hint: Try to embed a p process ring in the tree.
Solution:
I use Euler-tour representation to traverse the tree to be a ring. Here is a sample on a tree



With applying the ring-broadcast method, it will make the workload of every single directional channel to be m. Therefore, the time cost should be $(t_s + t_w m)(p - 1)$.

2 Problem 2

We discussed barrier synchronization as a way to ensure all threads reach a point in the code before any of them progress beyond the point. The figure below shows a barrier for N threads implemented using locks. Explain clearly how this code works. What should the initial values of count and the arrival and deparoure locks be? Why is it necessary to use two lock variables?



```
void barrier()
{
    set_lock(arrival);
    count++;
    if (count < N)
        unset_lock(arrival);
    else
        unset_lock(departure);
    set_lock(departure);
    count--;
    if (count > 0)
        unset_lock(departure);
    else
        unset_lock(arrival);
}
```

Solution:
The initial value of **count** should be 0, **arrival** should be unset, **departure** should be set.

The **arrival** lock make sure that the variable **count** to be increased by only one thread at the same time. Once a thread get **arrival** lock, increase count, if not all threads are arrival, the **arrival** lock will be released, and then thread will try to access **departure** lock. If the thread is the last one to arrival, the **departure** lock will be released. Thus, all not last thread will wait at setting **departure** lock until the last thread release it. All thread synchronization.
The **departure** lock protect variable **count** to be decreased by only one thread at the same time. Similarly, after all thread leave, the **arrival** lock will be released and the barrier will be back to the previous state.
The two lock is used to protect increment and decreasement atomatically to prevent more than one thread do the things which should only done by the last thread.

3 Problem 3

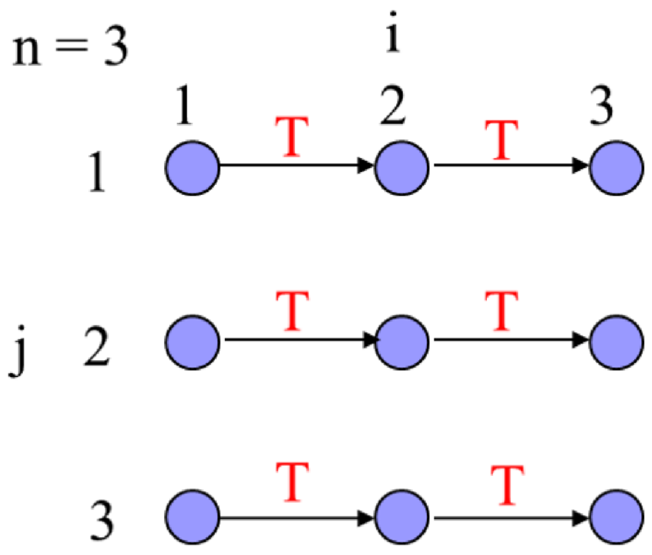
3.1 Question 1

Consider the sequential code shown in the figure below. List all the dependence relationships in the code, indicating the type of dependence in each case. Draw the loop-carried dependence graph (LDG).

```
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++) {
    S1: a[i][j] = a[i-1][j] + b[i][j];
    S2: b[i][j] = c[i][j]
  }
```

Solution:
Dependences:

- $S_1[i,j] \rightarrow T S_1[i+1,j]$
- $S_1[i,j] \rightarrow A S_2[i,j]$



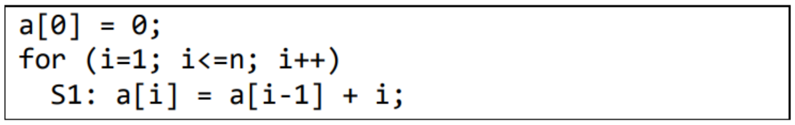
3.2 Question 2

Using the LDG derived in Q1a, describe how to obtain a parallel algorithm, explaining any modifications required to improve the efficiency. Express your parallel algorithm using OpenMP.
Solution:
Since we can see that there is no dependences between j, I choose to put j to outer iteration.

```
#pragma omp parrallel for private(i) schedule(dynamic)
for (j=1; j<=n; j++) {
    for (i=1; i<=n; i++) {
        S1: a[i][j] = a[i-1][j] + b[i][j];
        S2: b[i][j] = c[i][j];
    }
}
```

4 Problem 4

Consider the loop shown in the figure below. Indicate the loop-carried dependence relationships. By restructuring the code, is it possible to eliminate this dependence and parallelize the loop? Express your answer using OpenMP.



```
a[0] = 0;
for (i=1; i<=n; i++)
  S1: a[i] = a[i-1] + i;
```

Solution:
Dependences:

- $S_1[i] \rightarrow TS_1[i+1]$

But when we observe it, we can conclude that each $a[i] = 0 + 1 + 2 + 3 + \cdots + i = \frac{(1+i)i}{2}$,so

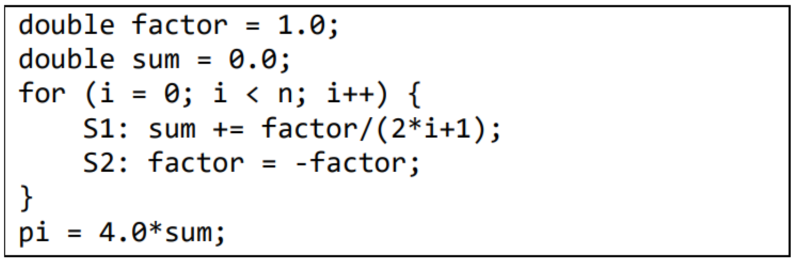
```
a[0] = 0;
#pragma omp parrallel for schedule(dynamic)
for (i=1; i<=n; i++) {
    a[i] = i * (i+1) / 2;
}
```

5 Problem 5

One way to get a numerical approximation to π is to sum the following sequence:

$$\pi = 4(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots)$$

A sequential algorithm to calculate π using this formula is given in the figure below. By analyzing the loop-carried dependences in this code, show how the algorithm may be parallelized, expressing your answer using OpenMP. Indicate clearly any modifications required to the code in order to allow parallelization and improve the efficiency.



```
double factor = 1.0;
double sum = 0.0;
for (i = 0; i < n; i++) {
    S1: sum += factor/(2*i+1);
    S2: factor = -factor;
}
pi = 4.0*sum;
```

Soltion:
Dependences:

- $S_1[i] \rightarrow TS_1[i+1]$
- $s_2[i] \rightarrow TS_2[i+1]$
- $S_2[i] \rightarrow TS_1[i+1]$

```
double factor = 0.0;
double sum = 0.0;
#pragma omp parrallel for private(factor) reduction(+:sum) schedule(dynamic)
for (i=0; i<n; i++) {
    if(i%2)
    {
        factor = 1;
    }
    else
    {
        factor = -1;
    }
    sum += factor / (2 * i + 1);
}
pi = 4.0 * sum;
```