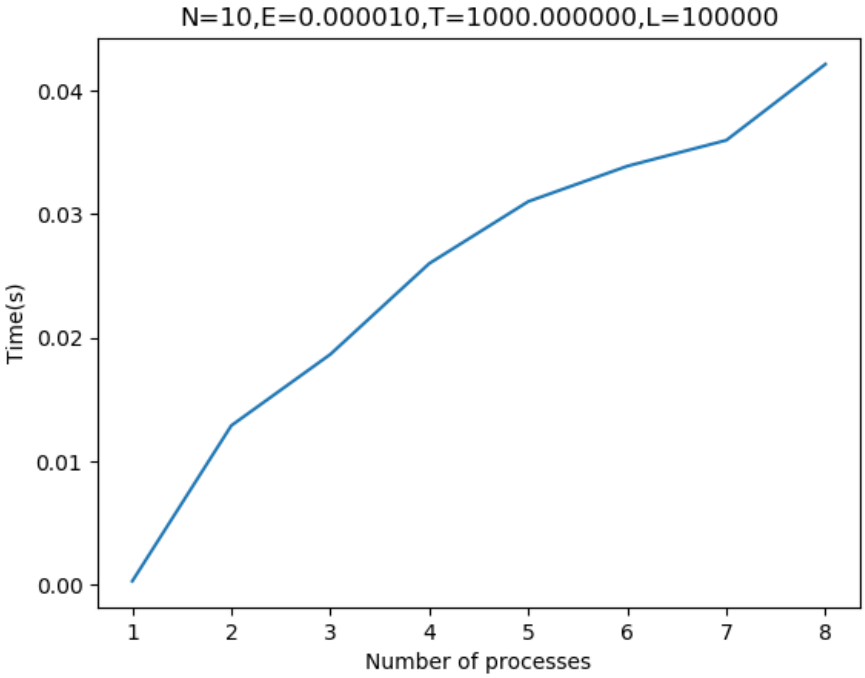
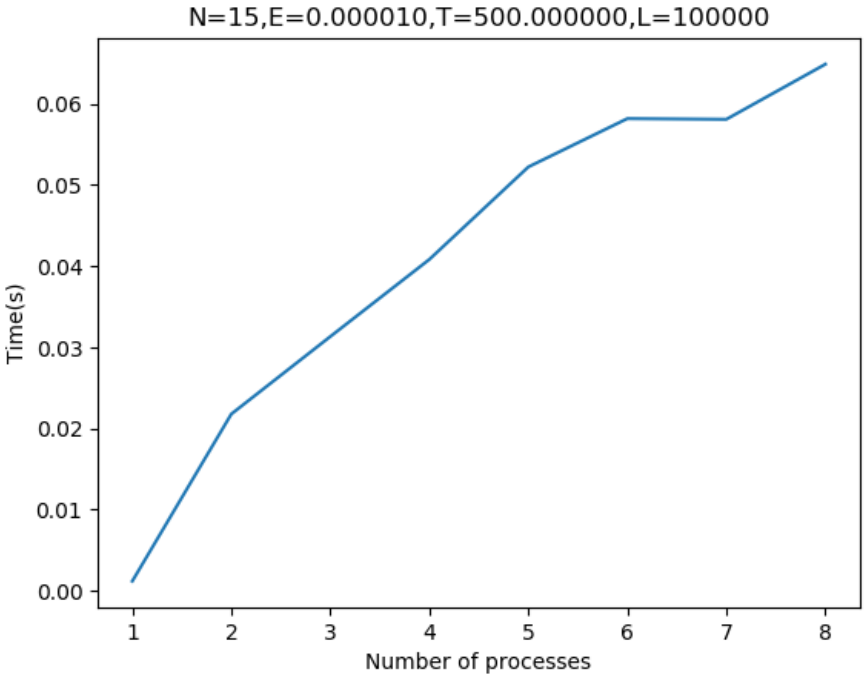
COL331: Operating Systems Assignment 2

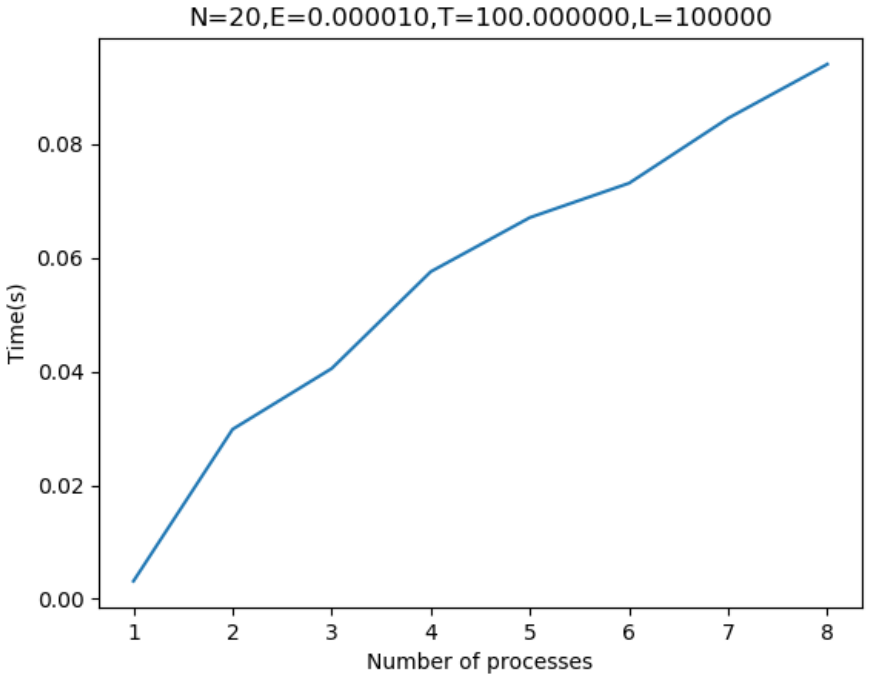
Chetan Mittal

February 2019

1 Jacobi : Performance vs Number of processes




N=15,E=0.000010,T=500.000000,L=100000
Time(s)
0.00
0.01
0.02
0.03
0.04
0.05
0.06
1
2
3
4
5
6
7
8
Number of processes


N=20,E=0.000010,T=100.000000,L=100000
Time(s)
0.00
0.02
0.04
0.06
0.08
1
2
3
4
5
6
7
8
Number of processes

1.1 Observations

It can be observed that for different N (<= 20), time taken increases almost linearly with the number of processes. This is in contrast with the expectation that the time must decrease with the increase in number of processes. However, the observation can be explained due to various reasons :

1. Simulation was done on "Ubuntu App" on Windows. It is possible that the app is not allotted sufficient number of cores.
2. Even if there are sufficient number of cores, the overhead of communication and synchronization may be greater than the actual computation time for small values of N.
3. Process synchronization is required twice in each iteration, one to communicate the minimum difference of values in successive iterations, and again to communicate values at the border with consecutive processes. Barrier synchronization has significant overhead leading to increase in execution time.

2 Maekawa : Time taken for different P, P1, P2, P3

P=4,P1=1,P2=2,P3=1
4.020236

P=4,P1=1,P2=1,P3=2
2.017071

P=9,P1=3,P2=3,P3=3
6.038079

P=9,P1=1,P2=3,P3=5
6.042270

P=9,P1=1,P2=5,P3=3
10.036886

P=16,P1=10,P2=3,P3=3
6.058625

P=16,P1=5,P2=5,P3=6
10.066820

P=25,P1=10,P2=5,P3=10
10.097984

P=25,P1=15,P2=5,P3=5

10.087617

Since there are 3 degrees of freedom {P1, P2, P3}, it is not possible to plot the data on a 2D graph to gain meaningful insights. Therefore, conclusions are made by directly observing the data.

2.1 Observations

1. Majority of time is taken by processes P2 due to the sleep condition.

2. For a fixed P,P1; if we exclude the sleep time, then execution time increases as P3 increases. For eg: consider P = 9, P1 = 1 (a) P2 = 3, P3 = 5 (P2 < P3), time = 0.042 (b) P2 = 5 P3 = 3 (P2 >P3), time = 0.037

3. Execution time increases as P increases for fixed P2.

4. For fixed P, P2; execution time increases as P1 decreases. For eg: consider P = 25, P2 = 5 (a) P1 = 10, P3 = 10, time = 10.098 (b) P1 = 15 P3 = 5, time = 10.088

3 Correctness Verification

For jacobi, correctness was verfied by comparing the output with the output of the serial execution.
For maekawa, correctness was verified by observation that no two processes entered the critical section at the same time. Also, all processes that requested for lock eventually got accesss to the critical section.

4 Extra

4.1 Implementation of Barrier

I have implemented Jacobi algorithm using unicast primitive developed in assignment 1. Many students have implemented the same using other primitives (like Pipes). Using other primitives, explicit use of barrier for synchronization can be avoided, for eg: by having separate pipes for communicating "minimum difference values" and "border values between successive processes", there is no need for explicit synchronization using barriers. (I have done the same in jacob_linux). However, using the unicast primitive, it is essential to use barriers for synchronization to avoid deadlocks.

I have implemented two system calls, "set_barrier" creates a new barrier for a given number of processes, "barrier" which is the actual barrier and takes in the barrier number.

```
struct barrier_type
{
    int nproc;
    int counter; // initialize to 0
    volatile int flag; // initialize to 0
    struct spinlock lock;
};
```

```
//set barrier
int set_barrier(void)
{
  int nproc;
  if(argint(0, &nproc) < 0)
    return -1;    //cannot read arguments
  initlock(&b.lock, "barrier");
  b.nproc = nproc;
  b.counter = 0;
  b.flag = 0;
  return 0;
}
```

```
//barrier
int barrier(void)
{
  int bnum;
  if(argint(0, &bnum) < 0)
    return -1;    //cannot read arguments
  struct proc *curproc = myproc();
  acquire(&b.lock);
  curproc->local_sense = 1 - curproc->local_sense;
  b.counter++;
  int arrived = b.counter;
  if (arrived == b.nproc) // last arriver sets flag
  {
      release(&b.lock);
      b.counter = 0;
      b.flag = curproc->local_sense;
  }
  else
  {
      release(&b.lock);
      while (b.flag != curproc->local_sense); // wait for flag
  }
  return 0;
}
```

Usage :
The parent processes initializes a barrier using the "set_barrier" system calls. Then, it forks a number of child processes. Each of them call "barrier" system call at the synchronization point, and wait till each process reaches the barrier.

```
//parent sets the barrier
//P = number of processes using the barrier
int bno = set_barrier(P);
```

```
//Each process which requires synchronization calls barrier
//bno = barrier id
barrier(bno);
```

4.2 Listener threads in Maekawa

For implementing Maekawa, sending and receiving messages between processes must be non-blocking and asynchronous. For this, I have forked a child process for each process which does the actual acquire and release, while the parent continuously listens for messages. This is conceptually similar to having two threads, one which does the actual computation, and the other which handles signals.

Now, forking processes in this manner can be avoided, by continuously looping for receiving messages, even when acquiring, releasing, and doing computation (print and sleep in this case). Another way to avoid forking processes is to have cleverly placed sleep statements, which essentially make the messages sequential in some manner. Though both these methods also work, they are not capturing the true sense in which a distributed algorithm is supposed to work.

```
for(int i = 0; i < P2; i++){
  P2id[i] = fork();
  if(P2id[i] == 0){
    ...
    child_id = fork();
    if(child_id == 0){
      ... //main computation
      exit();
    }
    listen(); //listen for messages
    wait();
    exit();
  }
  pid[P1+i] = P2id[i];
}
```

```
void listen(){
    ...
    while(1){
        recv(m);
        switch(m->type){
            ...
        }
    }
}
```