Operating Systems
COL331 - Lab Assignment 2
Supervisor - Prof. S.R. Sarangi

Shreshth Tuli
2016CS10680
Date: 22/03/2019

Introduction

In this assignment we implemented Parallel version of Jacobi heat distribution algorithm and Maekawa's distributed mutual exclusion algorithm in xv6. The different IPC primitives used for synchronization and data sharing among the processes were: unicast, multicast and barriers.

Performance comparison in Jacobi

We compare the time and speedup obtained for different N (Size of the Jacobi matrix) and P (Number of processes). We run the experiments on a 4 logical core machine. Time is obtained in milliseconds and Speedup is calculated as $T_{sequential}/T_{parallel}$, and Efficiency as $Speedup/Number\ of\ processes$.

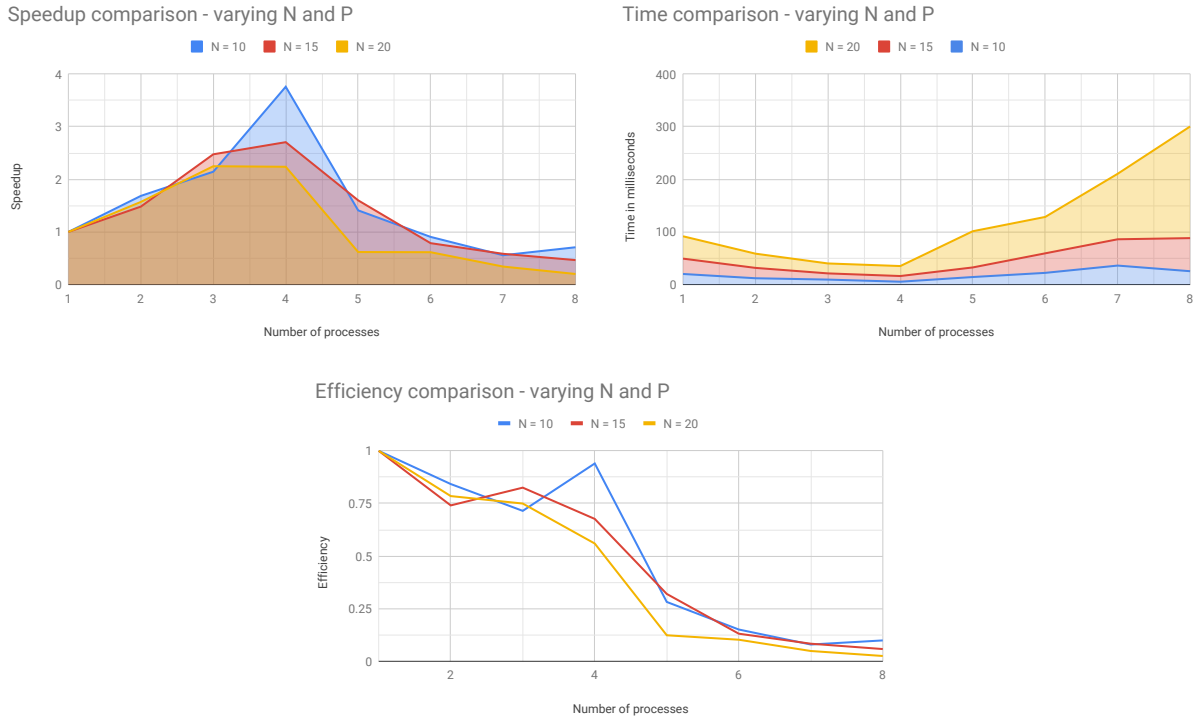

Figure 1: Performance comparison - varying N and P

Observations:

1. The first observation is that the time taken for computation of the whole Jacobi matrix increases as N increases. This is reasonable because as N increases, the size of the matrix increases and hence time also increases

2. We also observe that as the number of processes increases initially (till P = 4), the time reduces and hence speedup increases. s the number of processes increases to more than 4, the time increases and speedup decreases significantly. This is because, both computation power and communication delays decrease and increase linearly respectively till 4 processes, and for more than 4 processes computation power doesn't increase but communication delays increase linearly and hence time increases.

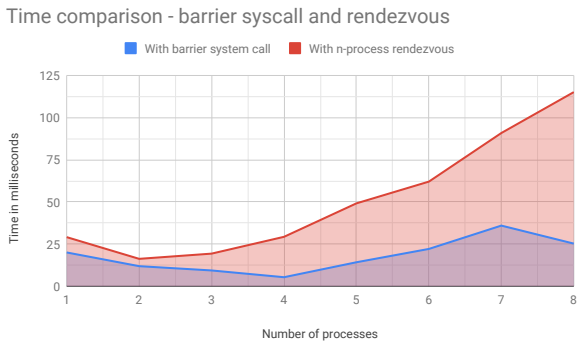

Figure 2: Performance comparison - barrier system call and rendezvous

3. We also see some other difference across N and P. For N = 10 and 20, the number of rows to be computed in the Jacobi matrix are 8 and 18 respectively. We observe that speedup in P = 2 is higher for N = 10/20 compared to that of N = 15. This is because of equal work distribution to all processes which has minimum stall time in barrier. For N = 15, number of Jacobi rows to be computed are 13 and hence the best work distribution that can be done for P = 2 is 6,7 which leads to the first process stalling for the second one in each iteration as second one has higher load.

4. Also, for N = 10, if P = 4 then division of workload is balances which this gives very high speedup (nearly 4). This is not seen in N = 15 because there are 13 Jacobi matrices which can not be divided equally for any number of processes. This observations is also not seen in N = 20 case because there are 18 rows and thus balances workload can not be assigned in 4 processes.

5. For N = 10, we also see that speedup rises slightly for P = 8 which is because of balanced workload (each process gets one row of Jacobi matrix) and hence nearly 0 synchronization overhead.

6. The speedup for N = 20 is lower than that of N = 10/15 and that of N = 15 slightly lower than N = 10, because the communication overhead increases with N but computation power doesn't.

7. Efficiency decreases as number of processes increase because of higher communication and synchronization overheads for higher number of processes. This is because high overhead results in lower CPU computation time and higher wait time which reduces CPU compute efficiency.

8. In Fig 2 we compare the synchronization and communication overhead when there is a barrier system call with respect to the case when the process send/receive messages to simulate an N-Process Rendezvous. The barrier syscall in Linux is membarrier(). We see that the barrier system call has significantly lower overhead and much better scalability with number of processes. This is because in barrier system call there is a wait queue where each process regiters itself when it reaches the barrier and when the last process registers to the queue, the kernel wakes up all processes. As this sleeps (blocks) the process for the duration, the CPU scheduler can give more timer intervals to the remaining processes and hence improves performance.

Performance comparison in Maekawa

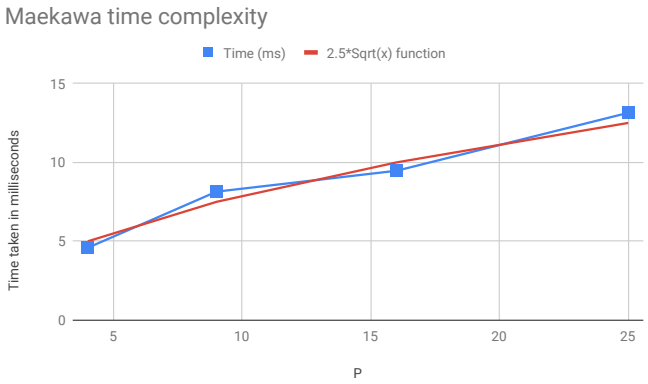

Figure 3: Time comparison for different P

Analyzing performance in different cases of Maekawa's algorithm is significantly more complex due to large number of variables on which the total time is dependent on, including P = number of processes $\in$ {4,9,16,25}, P1 = number of processes that do not acquire lock, P2, P3 = number of processes that acquire lock but for 2 and 0 seconds respectively. Fig 4 shows the comparison for different configurations of P = 4/9/25. The number of processes P1, P2, and P3 are varied and shown. The variation in graphs is along P1 and P2, P3 is calculated as $P3 = P - P1 - P2$. The total time taken is the time to run the whole program with some specific configuration i.e. P, P1, P2, P3 values. The communication time is the overhead time in execution and is calculated as $Communication\ Time = Total\ Time - 2 \times P2$. Observations:

1. As P increases the total time as well as communication time increase. We see that for higher values of P i.e. P1 + P2 + P3, the communication overhead is maximum. This is expected behaviour because with more number of process the size of quorum increases and hence the overhead time. Also for P = 4, 9, 16, 25; the communication times are approximately on an average = 4.59, 8.14. 9.47, 13.02 milliseconds. All these values when fitted with a square root function show the complexity follows nearly the function $\frac{5}{2}\sqrt{P}$. This empirically confirms that Maekawa's alorithm scales in a square root fashion with the number of processes as the number of quorums is $\sqrt{P}$. This behaviour is shown in Fig 3.

2. As shown in Fig 4 the total time increases linearly with P2 and is approximately $2 \times P2 + \delta$ where δ is the communication overhead time.

3. Another important observation from the graphs is that the communication time is high only when P2 is highest. The reason behind this behaviour is the that P2 processes contend for lock, but P1 processes do not. As P2 increases more processes contend for the lock which increases the number of messages and hence the communication time.

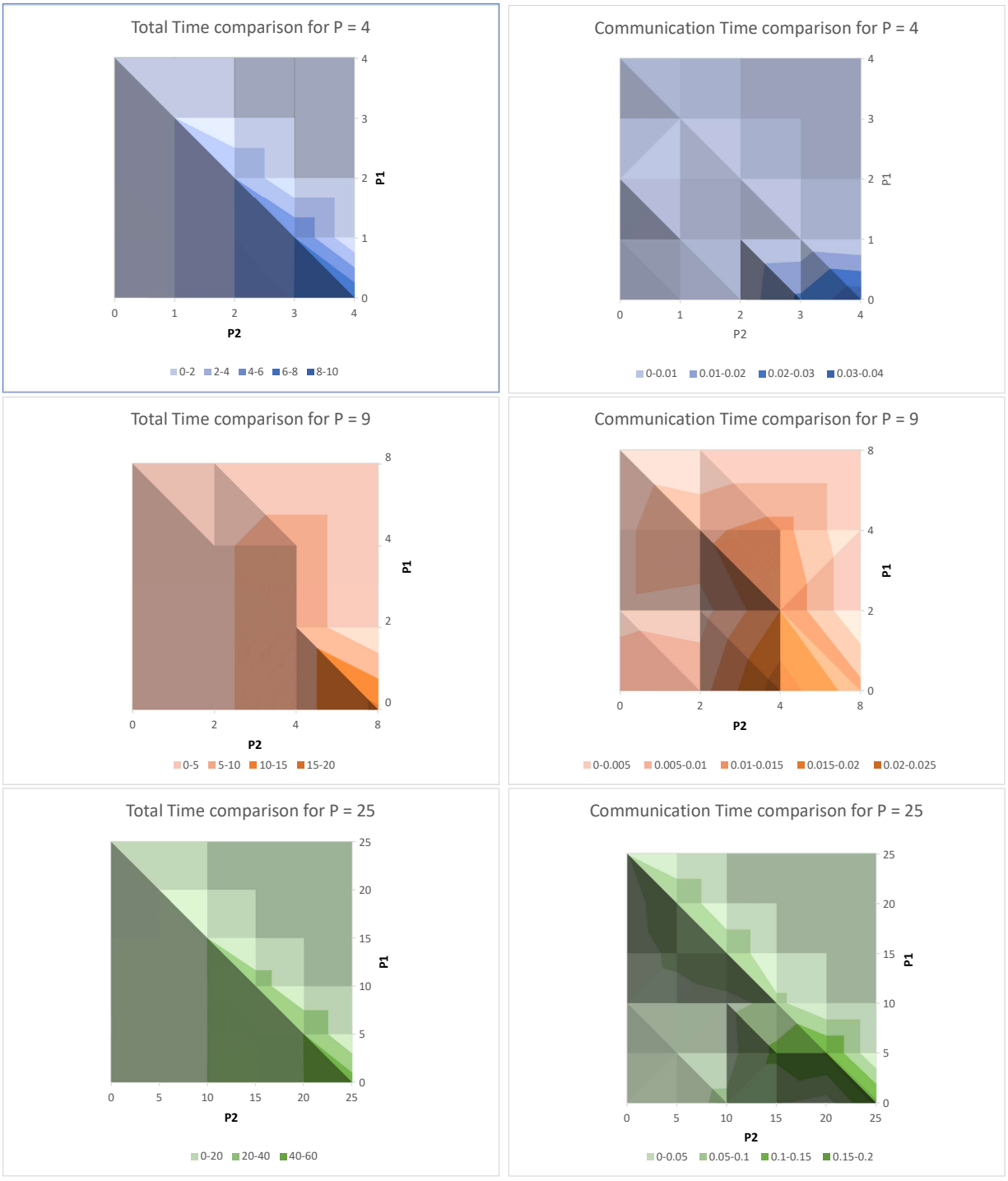

Figure 4: Performance comparison for different P

Verifying correctness of implementation

Verifying correctness of implementation of both Jacobi method and Maekawa algorithms has been done using the following steps:

Jacobi

1. The implementation of parallel version of Jaocbi method can be divided into subtasks : (1) Parametric work distribution (2) Computation of submatrices by each process (3) Communicating border values with neighboring processes (4) Getting difference of values from all processes and applying the convergence condition. Each subtask can be checked separately

2. The first idea of parametric work distribution was checked by making sure that the sub-matrices that the processes get are disjoint and their union is the complete Jacobi matrix.

3. The computation of the jacobi function on the submatrices were checked by verifying that the post computation results matched those of the sequential execution of the same program.

4. Communication of border values checking was different for xv6 and Linux implementations. In xv6, a custom barrier system call was first tested if all processes converge at the barrier call and then only continue further. This was crucial because if this was not established then the values being communicated might be stale and hence would render the whole computation useless. When this was made sure of, the communication protocol (tested before) of unicast was employed. In Linux, the message passing system calls msgsend() an msgrcv() were used.

5. The last part of collecting the diff values from each process and deciding to continue based on the convergence criteria was challenging becuase each process might have a different diff. All such values were collected using unicast to the master process and the minimum diff was used to check convergence. Based on this the master multicasts to all processes if they need to continue or not. This was tested by checking if there are any process that get stuck and hence form xv6 zombies.

Maekawa

1. The correctness verification in Maekawa was done by checking first whether the processes in a quorum are sending the request messages to their own quorum. This means that for a matrix like arrangement of processes, a process (i,j) sends requests only to (x,y) where either x = i or y = j. This condition was checked first.

2. Another operational detail that was verified was if the requests were in a global total order and the correct process i.e. the first process in that order and belonging to the set of received requests is sent the reply, and others added to the queue based on their time stamp. This total order can be maintained by using time-stamped requests or barriers to make sure that a process earlier in the order sends requests before all other that are later to it. The latter strategy was used because it was easy to check for correctness.

3. Once a process receives replies from all processes in it's quorum it was verified that it acquires the lock and waits for the delegated period, after which it releases the lock.

4. When a process in a quorum release a lock, all processes in that quorum (including the one that acquired the lock) send replies to the next process in the queue.

5. Verifying all above made sure that for any P, P1, P2, P3 permutation the mutual exclusion is maintained and all P2, P3 processes eventually get the lock.