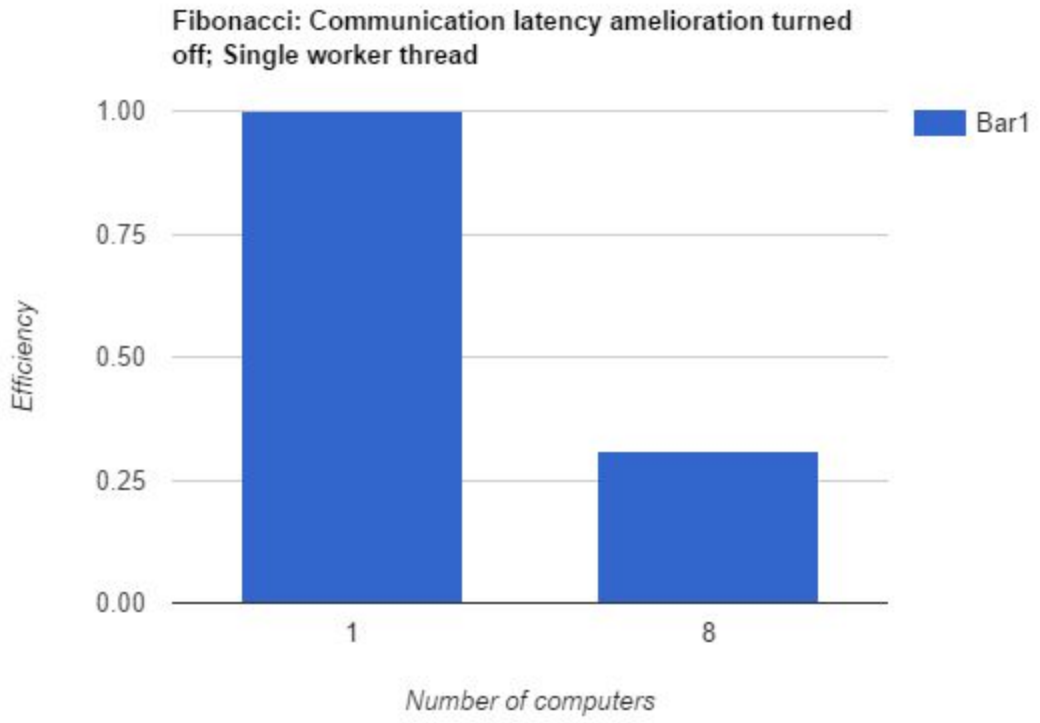
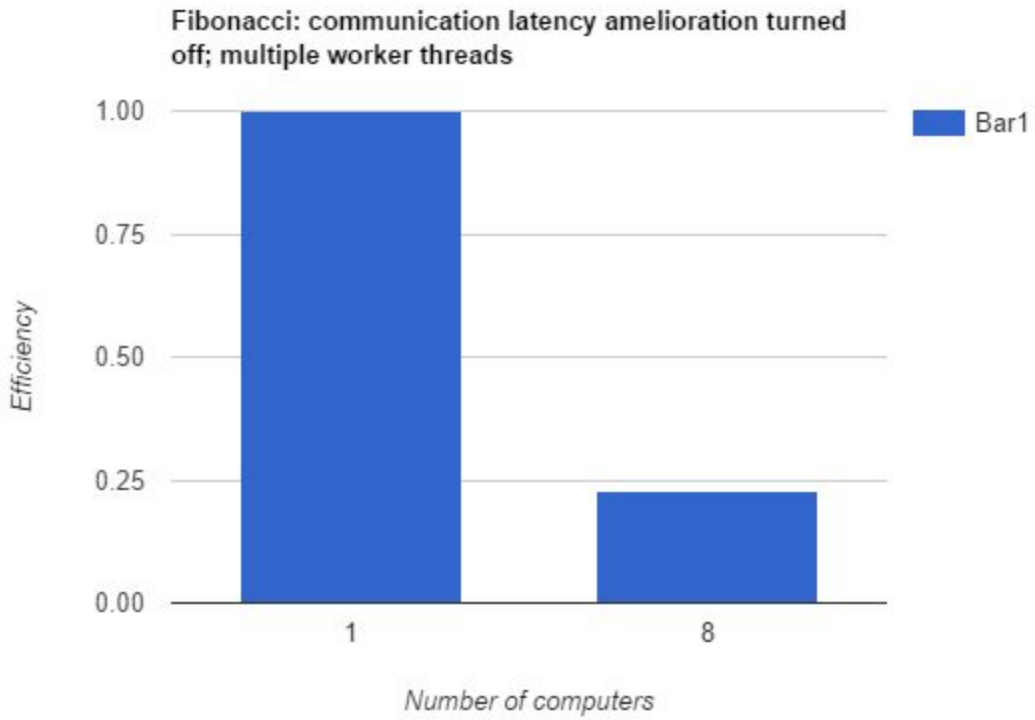
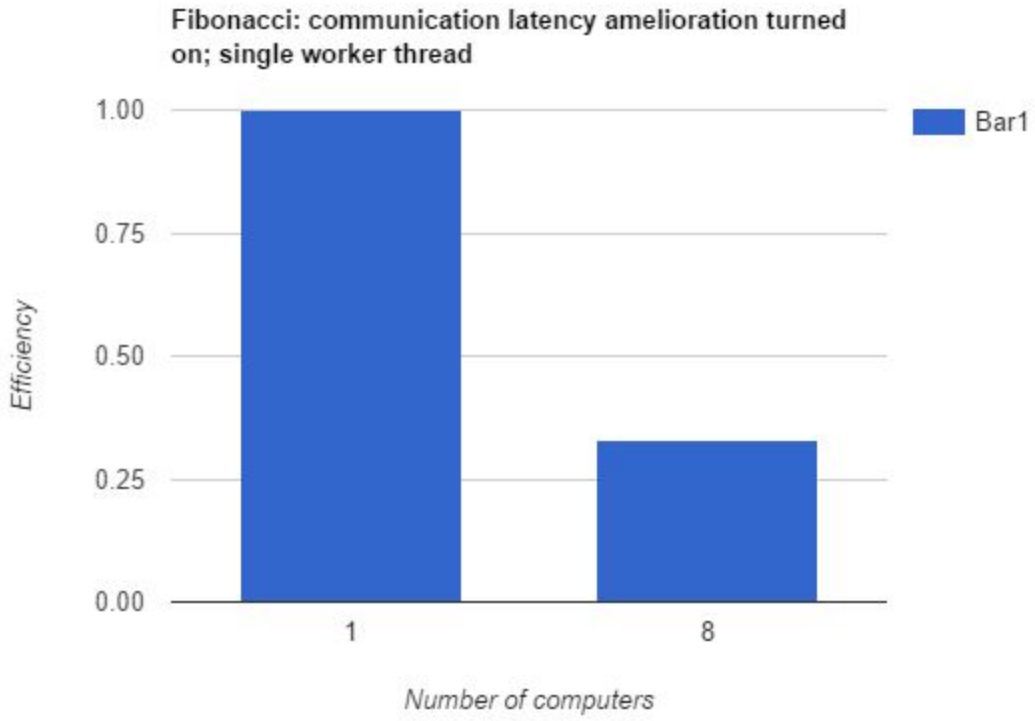
CS190B

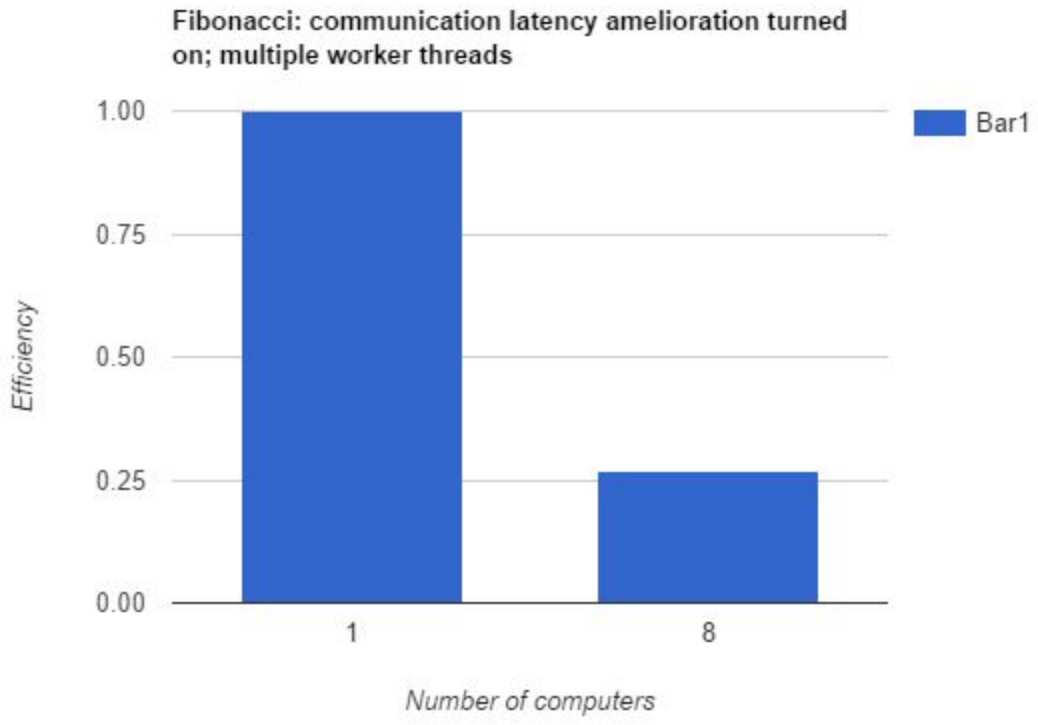
HW4 REPORT

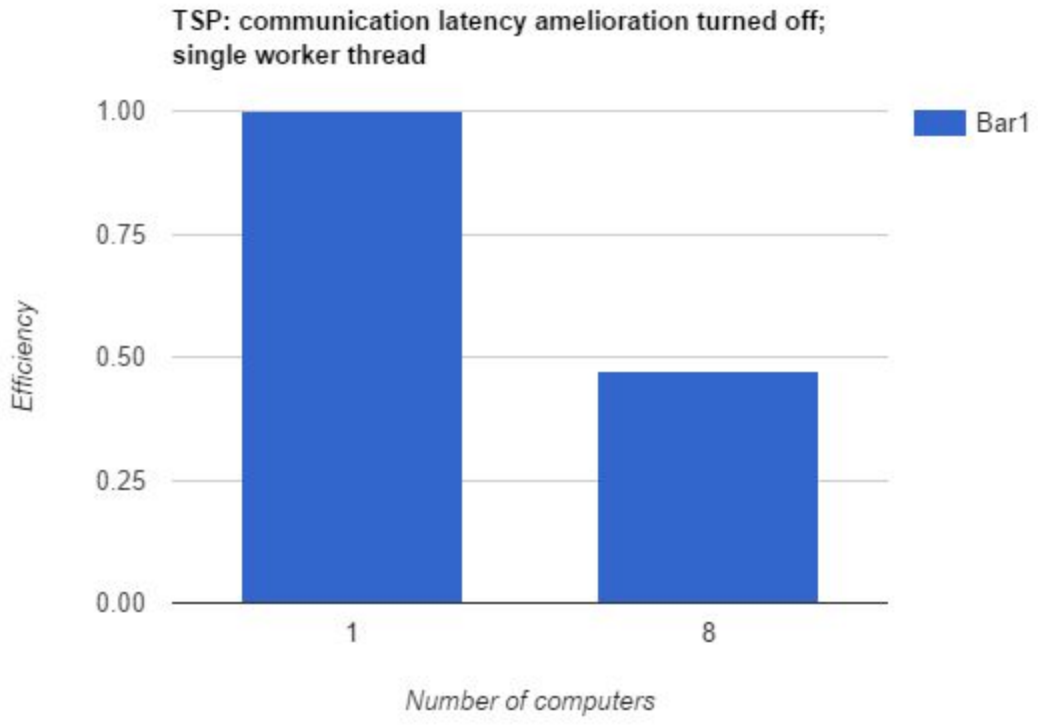
1. Performance

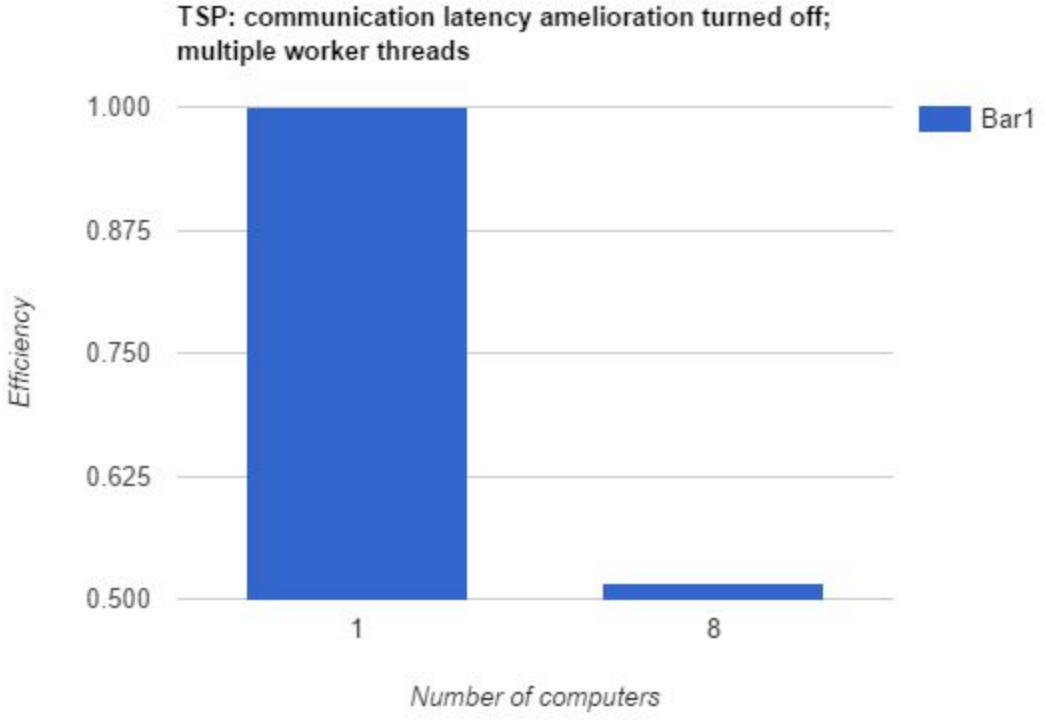


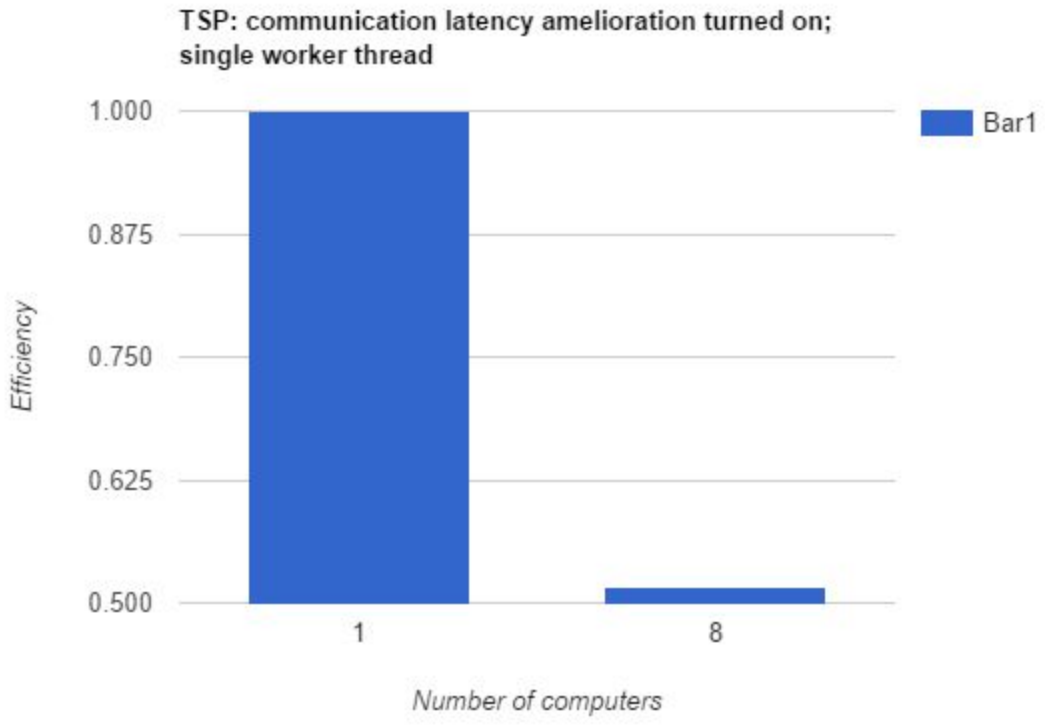

Fibonacci: communication latency amelioration turned off; multiple worker threads
1.00
0.75
0.50
0.25
0.00
Efficiency
1
8
Number of computers
Bar1

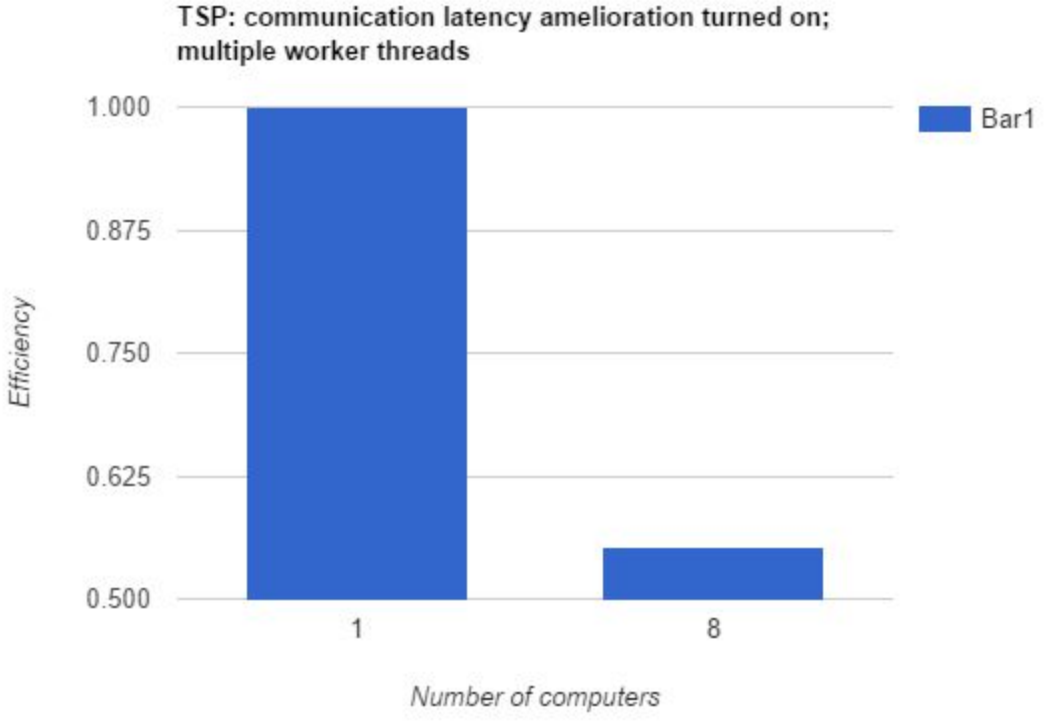

Fibonacci: communication latency amelioration turned on; single worker thread
1.00
0.75
0.50
0.25
0.00
Efficiency
1
8
Number of computers
Bar1


Fibonacci: communication latency amelioration turned on; multiple worker threads
1.00
0.75
0.50
0.25
0.00
Efficiency
1
8
Number of computers
Bar1


TSP: communication latency amelioration turned off; single worker thread
1.00
0.75
0.50
0.25
0.00
Efficiency
1
8
Number of computers
Bar1


TSP: communication latency amelioration turned off; multiple worker threads
1.000
0.875
0.750
0.625
0.500
Efficiency
1
8
Number of computers
Bar1


TSP: communication latency amelioration turned on; single worker thread
1.000
0.875
0.750
0.625
0.500
Efficiency
1
8
Number of computers
Bar1


TSP: communication latency amelioration turned on; multiple worker threads
1.000
0.875
0.750
0.625
0.500
Efficiency
1
8
Number of computers
Bar1

2. Explanation Of Our Experimental Data

Number Of Computers:

It turns out that using more computers will bring down the efficiency, which is expected since with more computers, more communication time is needed and it is more probable that computers stay idle.

Using Multithreads:

Using multithreads improves the efficiency, but not so much as we expected. We think the reason may be that in our implementation, the computer will ask proxy for new tasks only when all its threads finish their work, so that other threads will stay idle while waiting for the busiest thread. This may be a issue that makes the efficiency is not as high as we expected.

Using Amelioration (caching):

Our amelioration is caching, which is that every thread will keep one of the children tasks that its previous task generated (if it has children tasks). In this way, the computer does not need to wait for proxy sending new task, and can execute the caching task right away. Caching does improve the efficiency, but not so much as we expected either. The reason may probably be that communication time is long, so that the effect of executing a bunch of tasks is not that obvious.

3. Design Document

3.1 - Client:

The clients is the same as in homework 3.

3.2 - Space:

The space will get the number of processor that a computer has when the computer register to it. The corresponding proxy will send tasks the amount as the same number of the computer's processor if there are enough tasks in the ready queue. If not, it will take all the tasks in the ready queue and give them to the corresponding computer.

3.3 - Computer:

Our computer will create threads as the same amount of processors. When a computer has no task, it will ask proxy to give it a set of tasks, which is the same amount of multiprocessors, and each of its threads will execute a task. After executing the task, those threads will check if the task generated some ready tasks. If it did, then the threads will start to execute one of those ready tasks, and return other children tasks and the successor task to the space.

4. How We Can Do To Further Improve Our Performance

We observe that even though we use lots of threads, trying to reduce our working time to be a quarter, it seems that our computer still has lots of idle time. What I think that we can do to improve our performance, are listed as following:

1. Send preparing tasks to computer: When there are lots of tasks in the space, we can try to send tasks as double (or even more) as amount of processors to computers, so that when computers finish the first set of tasks, their threads will have immediately tasks to execute. At the same time computers may send a message to space, asking for next set of tasks. By doing this, computers do not need to wait for next set of tasks coming after executing a set of tasks. They will almost always have tasks to execute right after they finish a set of tasks.
2. Getting preparing tasks more frequently: in our design, the computer will ask for tasks only when all its threads finish their work, which is not that efficient, since all processors will need to wait for the busiest one. Instead of doing this, we can give

those resting threads some preparing tasks(mentioned in 1.) to execute, and ask proxy to give the computer tasks as the same amount of consumed tasks.