Elevator-Customer Simulation with MPI

Authors: Heejoon Chae, Chen Zhang

April 26, 2006

Abstract

The simulation of the elevator-customer-scheduler system is essentially a problem that can be parallelized by using multiple processes. This problem can be handled by threads, but it is more practical for threads to use and share variables for communication. In reality the data is sent between each process and cannot be shared by other means. In this paper, we shall implement the simulation of our system using MPI. From our design, each elevator, customer, scheduler process will interact and operate through MPI send and receive communications. Simulation experiments are performed on machines that support MPI to test the practicality of our implementations.

Contents

1 Introduction **3**

2 Design **3**

2.1 Communication 3

2.2 Elevator Task-Handling 4

2.3 Finding Optimal Elevator 5

3 MPI Implementation **6**

4 Optimizations **8**

5 Conclusion **9**

List of Figures

1 A simple diagram representing all communications between entity types. . 4

2 A simple table of array of tasks stored in each elevator 5

3 Simple pseudo-code for Scheduler process 7

4 Simple pseudo-code for Elevator processes 8

5 Simple pseudo-code for Customer processes. 8

1. Introduction

To simulate of the elevator-customer-scheduler system, it essentially requires us to allow multiple elevator processes to handle multiple customer processes. We must synchronize the way in which the processes can communicate to allow for good optimization and communication. In this paper, we will discuss our general design of the system, how communications are carried out, what information are shared, the implementation of each process in MPI, and finally list the optimizations we achieve by using MPI.

2. Design

In this section, we discuss our overall design of the elevator-customer-scheduler system. Since this simulation is based on a real-world situation, our design should follow the real-world situation closely. We create three types of entities for the simulation: the scheduler, responsible for taking user requests and assigning them to specific elevators to handle, the elevators, responsible for handling requests from the scheduler and customers, and the customer, who wish to ride the elevator to a new destination. Furthermore, we must find a way for the entities to communicate correctly and efficiently with one another. We also discuss in this section how the scheduler decides which elevator is selected to handle a job and how elevators handles their tasks.

2.1 Communication

The communication between the three entity types is simple and is illustrated in figure (1). All customers must initially send a request to the scheduler. Each request primarily contains the floor that the customer is on and the direction that he/she wants to go. The scheduler will take customer requests and assign jobs to the most optimal elevator to handle each request. The scheduler finds the optimal elevator based on the distance that each elevator has to travel to the floor specified by the request. The elevator should periodically send its status to both the scheduler and the customers. The elevator status consists primarily of the floor that the elevator is currently on, its current direction, and whether or not the elevator doors are open. The customers require this information to decide if he/she will be able to get on to ride the elevator to his/her destination floor. The scheduler requires this information to decide an optimal elevator to handle a user request. Once the customer is inside an elevator, he/she can send requests directly to that elevator. Finally, when the customer is inside an elevator, he/she must wait for

the elevator's status report. If the elevator is open on the floor of his/her destination, the customer will be able to exit and an interaction is completed.

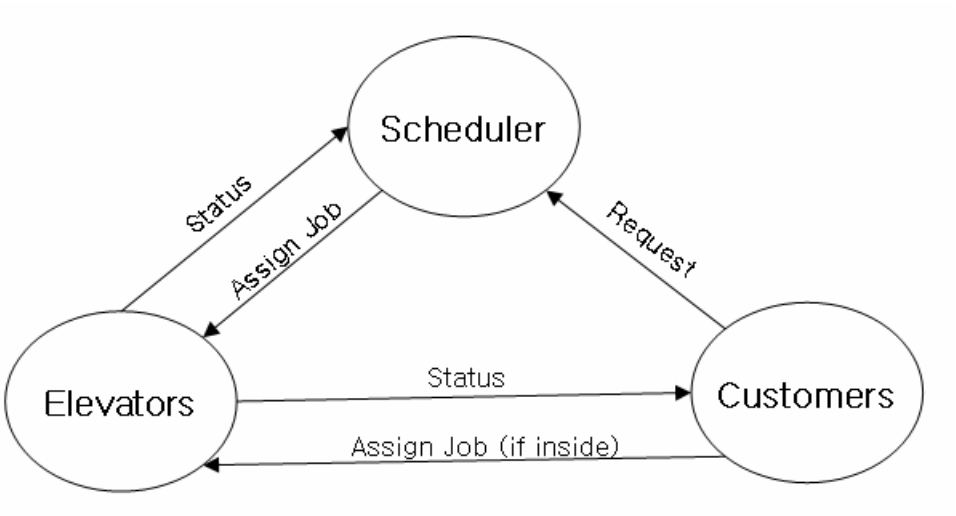

Figure 1: A simple diagram representing all communications between the three entity types.

2.2 Elevator Task-Handling

Each elevator is itself a single process that contains a set of local variables, which determines the behavior of the elevator. The elevator must store its current floor, current location, and its current set of tasks that it must complete. The scheduler and the customers inside the elevator can directly send tasks to the elevator. The elevator's tasks are stored into a two-dimensional array shown in figure (2). Since there can be at most one upward request and downward request on each floor, an elevator can have at most two tasks per floor.

	Elevator Buttons	
Floor 6	**On/Off**	**On/Off**
	On/Off	**On/Off**
.	**On/Off**	**On/Off**
.	**On/Off**	**On/Off**
.	**On/Off**	**On/Off**
	On/Off	**On/Off**
Floor 0	**On/Off**	**On/Off**
	Up	**Down**

Figure 2: A simple table representing the array of tasks stored in each elevator. Tasks include all possible upward and downward requests throughout the building. If an upward button is turned on for a floor, it could mean a customer may want to go up from that floor, or someone wants to get off on that floor and the current direction is up (the opposite is true for all downward requests).

The elevator's traversal of the floors inside the building is determined by its location, direction, and its set of tasks. If the elevator's direction is currently upward, the elevator will handle all upward requests for the floors that it has not yet passed. Once there are no upward requests that it hasn't passed, the elevator will handle all downward requests starting from the top-most floor with the downward request. The elevator will change its direction downward to handle all downward requests that it hasn’t passed. Then the elevator will start handling all upward requests starting from the lowest floor request, and the process repeats. By this implementation, our elevator will be able to handle all its tasks efficiently. It should be noted that the scheduler is responsible for determining the tasks to an elevator to optimize elevator traversal distance and performance.

2.3 Search of Optimal Elevator

The scheduler receives user requests to ride elevators. Once a request is received, the scheduler is responsible for finding the most optimal elevator for the task. It does this by checking each elevator's current status and finding one with the least traveling distance to reach the customer's floor. If the current elevator is idle or if the elevator is going in the same direction as that required by the customer and the elevator has not yet passed the customer's floor, then the distance is calculated simply by taking the difference of the two floor numbers. If the elevator has already passed the floor of the customer, or

if it is currently going in the opposite direction that the customer has requested, the computation of the traveling distance becomes more complicated. Essentially, the scheduler must consider the current traversal that the elevator is taking and calculate the traveling distance based on this information.

3. MPI Implementation

The design of the elevator-customer-scheduler system is straightforward, but to implement the simulation with MPI we must take into account the some of the synchronization issues. Essentially, MPI sends and receives are blocking commands, which means during each send command each process has to wait for the other end to receive, and vice versa. This could cause many problems if not carefully used. If the communication is not synchronized correctly the system can easily end in a dead-lock. Also the processes can only share data through the use of MPI send and receive. At each iteration, the elevator has to make movements upward or downward. During the time that it take an elevator to move, the elevator would not be able to receive commands. Using the Isend could cause commands to be received while the elevator is moving, but this could cause a loss of data since we do not know how long it takes between each receive.

If we can correctly synchronize the three types of entities, by using the blocking commands of MPI send and receive we can insure the reception of right data at the right time. Since the elevator is the most difficult to synchronize (since it cannot send or receive during its movement), we have the customers and the scheduler send tasks to the elevator only immediately after the elevator sends its status to them in each iteration. Below we provide a simple description of our implementation of the three kinds of processes using MPI.

In figure (3), we give an illustration of how the scheduler process should behave. The scheduler must constantly await customer requests and elevator status reports. If the scheduler receives from a customer, the scheduler must find the optimal elevator and update the collection of tasks that the elevator should handle. If the scheduler receives from the elevator, the scheduler must update the status sent by the elevator and also send that elevator the set of tasks that's must be assigned to it.

```
Scheduler{
        while (true){
                receive from customer or elevator
                if (customer){
                        record request and find optimal elevator}
                if (elevator){
                        update elevator status
                        send associated tasks to elevator}
        }
}
```

Figure 3: Simple pseudo-code describing the Scheduler process

An illustration of the behavior of the elevator is shown in figure (4). The elevator continually sends its status to the scheduler and receives any tasks from the scheduler. It updates the array of tasks and checks the direction it should take based on its current direction and makes a movement if it is not idle. It then sends its status to all customers and receives commands from customers who are inside the elevator.

```
Elevator{
        while (true){
                send status to scheduler
                receive tasks from scheduler
                update array of tasks
                check direction and move a floor/remain idle
                send status to all customers
                receive commands from all customers who are inside
        }
}
```

Figure 4: Simple pseudo-code describing the Elevator process

The pseudo-code for customers is shown in figure (5). Customers send an initial request to the scheduler. The customer must then wait for the elevator's status report while he/she is outside of the elevator. As soon as an elevator reaches his/her floor, if the elevator is going in the same direction and if the elevator's door is open, then the customer will get on and send his/her destination request to the

elevator directly. When the customer is inside the elevator, he/she again must again wait for the elevator's status report. If the elevator is open on his/her destination floor the customer can exit. This will complete the customer's interaction with the scheduler and elevators. The customer will “sleep” for a random number of iterations and use the elevator once more.

```
Customer{
        while(true){
                send request to scheduler
                while (not inside elevator){
                        receive elevator status
                        if (same direction, floor, and open){
                                get into elevator
                                send destination request direction to elevator}
                }
                while (inside elevator){
                        receive status from elevator
                        if (destination reached and open)
                                get out
                }
        }
}
```

Figure 5: Simple pseudo-code describing the Customer process

4. Optimizations

The optimization we obtain by using multiple processes in MPI is that multiple elevators can handle multiple user processes. The elevators themselves operate independently from each other. One elevator's movement does not affect the performance of any other elevator. The scheduler will find the optimal elevator to handle each customer request, which reduces the amount of time a customer must wait for service. Once a customer is inside the elevator, he/she can direction send requests to the elevator without communicating to the scheduler first.

There are additional improvements that can be made to increase optimization of our system. For instance, our current system does not consider the weight capacity of an elevator. In real life situations, an elevator can be full and not take more requests. Also the scheduler assigns an optimal elevator based

on the elevator's current traveling distance, but not on the number of its current tasks. This could cause one elevator to handle too many tasks and make too many stops, while other elevators remain idle.

5. Conclusion

From our tests and code implementations, we see that the simulation of the elevator-customer-scheduler relationship and communication is achievable with MPI. With MPI we could create separate processes to represent the scheduler, each elevator, and each customer. MPI sends and receives can be used to establish communication and the sending of data between all related processes through synchronization.

References

LAM/MPI Parallel Computing, http://www.lam-mpi.org/