

*A (Practical Training Report) on*

## **Algorithmic Load Balancing**

*undergone at*

*under* **National Institute of Technology Karnataka**

*the guidance of*

**Dr. Annappa B**

*Submitted by*

**Aparna R. Joshi (14CO204)**

**Sheetal Shalini (14CO142)**

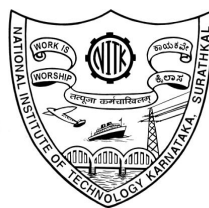
**VII Sem B.Tech (CSE)**

*in partial fulfillment for the award of the degree of*

**BACHELOR OF TECHNOLOGY**

*in*

**COMPUTER ENGINEERING**



**Department of Computer Science & Engineering  
Technology**

**National Institute of Technology Karnataka, Surathkal**

***April 2017***

## Abstract

In this project, we have implemented the paper: Algorithmic Mechanism Design for Load Balancing in Distributed Systems, (Daniel Grosu, Student Member, IEEE, and Anthony T. Chronopoulos, Senior Member, IEEE). [1]

(<http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=1262484>)

The paper solves the problem of static load balancing in heterogeneous distributed systems involving selfish agents by designing a truthful mechanism. The output function admits a truthful payment scheme satisfying voluntary participation using the optimal allocation algorithm. The algorithm proposed falls in the category of global approach that is, there is only one decision maker that optimizes the response time of the entire system over all jobs and the operating point is called *social optimum*. The optimal algorithm finds the fraction of load that is allocated to each computer such that the expected execution time is minimized. It is also shown in the paper that if the computers do not follow a truthful mechanism, the expected overall execution time will increase. This mechanism design problem for load balancing in distributed systems has been implemented.

## **TABLE OF CONTENTS**

<b>Introduction</b>	<b>3</b>
<b>Technologies Used</b>	<b>4</b>
<b>Work Done</b>	<b>4</b>
<b>Code</b>	<b>5</b>
<b>Results</b>	<b>10</b>
<b>Conclusion</b>	<b>12</b>
<b>References</b>	<b>12</b>

## INTRODUCTION

---

Each participating agent in a distributed system has a privately known function called valuation which quantifies the agent benefit or loss. The valuation depends on the outcome and is reported to a centralized mechanism. The mechanism chooses an outcome that maximizes a given objective function and makes payments to the agents. The valuations and payments are expressed in some common unit of currency. The payments are designed and used to motivate the agents to report their true valuations. Reporting the true valuations leads to an optimal value for the objective function.

The goal of each agent is to maximize the sum of her valuation and payment. In this implementation, we consider the mechanism design problem for load balancing in distributed systems. A general formulation of the load balancing problem is as follows: Given a large number of jobs, find the allocation of jobs to computers optimizing a given objective function (e.g., total execution time).

## TECHNOLOGIES USED

---

For the implementation of this project, we have used Open MPI (A High Performance Message Passing Library). Distributed loosely coupled systems lack a shared address space and thus processes in distributed systems mainly make use of message passing primitives unless it is distributed virtual shared memory.

We have used Open MPI- 1.4.5 along with boost library support.

Boost.MPI is a library for message passing in high-performance parallel applications. A Boost.MPI program is one or more processes that can communicate either via sending and receiving individual messages (point-to-point communication) or by coordinating as a group (collective communication). Unlike communication in threaded environments or using a shared-memory library, Boost.MPI processes can be spread across many different machines, possibly with different operating systems and underlying architectures. [2]

## WORK DONE

---

The load balancing problem can be formulated as:

Given a large number of jobs, find the allocation of jobs to computers optimizing a given objective function (e.g., total execution time).

Problem Statement of the paper [1] is to design protocols for resource allocation involving

selfish agents (geographically distant grids manipulated by different agents).

We have proved that using the optimal algorithm, the output function admits a truthful payment scheme. Each agent has a private valuation function that depends on the outcome (aim is to maximise the function). The payments are designed to make the agents report their true valuations. Reporting the true valuations leads to an optimal value for the objective function.

Goal of each agent : maximize the sum of her valuation and payment.

Each computer has its true processing rate  $v_i$ . True Value  $t_i$  is  $1/u_i$  (inverse of true processing rate). Only computer  $i$  knows  $t_i$ . It reports  $b_i$  to the mechanism as the inverse of its  $v_i$ . It can be correct or wrong. The mechanism then computes

1. output function  $\lambda(b) = (\lambda_1(b), \lambda_2(b), \dots, \lambda_n(b))$
2. payment function  $p(b) = (p_1(b), p_2(b), \dots, p_n(b))$

such that overall expected execution time is minimum. Each of these is handed to the respective agents. Cost for each agent is  $t_i * \lambda_i(b)$  (the total time taken to execute all the jobs assigned to it). Profit for each agent is  $p_i(b) - cost$ . Each agent wants to maximize its profit. So the aim of the paper is to design an allocation algorithm & payment scheme such that overall expected response time is minimum - and that happens when systems give their true values.

We have implemented this algorithm and shown 2 different cases :

1. When all systems give their true values as their bid values
2. When systems give wrong values as their bid values

The overall execution time of both cases has been compared and plotted. It has been noticed that with faulty values, the expected overall execution time has decreased.

## Proposed Algorithm

### OPTIM algorithm:

**Input:** Bids  $b_1, b_2, \dots, b_n$  submitted by computers

Total arrival rate:  $\Phi$

**Output:** Load allocation:  $\lambda_1, \lambda_2, \dots, \lambda_n$ ;

1. Sort the computers in increasing order of their bids

$$(b_1 \leq b_2 \leq \dots \leq b_n);$$

2.  $c \leftarrow (\sum_{i=1}^n 1/b_i - \Phi) / (\sum_{i=1}^n \sqrt{1/b_i});$

3. **while** (  $c > \sqrt{1/b_n}$  ) **do**

$$\lambda_n \leftarrow 0;$$

$$n \leftarrow n - 1;$$

$$c \leftarrow (\sum_{i=1}^n 1/b_i - \Phi) / (\sum_{i=1}^n \sqrt{1/b_i});$$

4. **for**  $i = 1, \dots, n$  **do**

$$\lambda_i \leftarrow 1/b_i - c\sqrt{1/b_i};$$

## CODE

```
#include<iostream>
#include<algorithm>
#include<math.h>
#include<cstdlib>
#include<boost/mpi/environment.hpp>
#include <boost/mpi/communicator.hpp>
#include <boost/mpi/status.hpp>
#define ELECTED_TAG 2

using namespace boost::mpi;
using namespace std;

environment env;
communicator world;
float rates[4] = {1698.937, 1700.265, 1700.000, 1700.066}; // according to the
configuration of the system
float
bids[4], total_arr_rate, load[4], cost[4], total_cost, profit[4], pay[4], total_prof
it=0.0;
int slave_bids;
/*rates[0] = 1599.992;
rates[1] = 1600.125;
rates[2] = 1599.992;
rates[3] = 1600.125;
*/
int nprocs;
void printCost()
```

```

{
    int i;
    for(i=0;i<slave_bids;i++)
    {
        printf("Cost of processor %d = %f\n",i,cost[i]);
    }
    printf("\nTotal cost = %f",total_cost);
}

void printProfit()
{
    int i;
    for(i=0;i<slave_bids;i++)
    {
        total_profit += profit[i];
        printf("Profit of processor %d = %f\n",i,profit[i]);
    }
    printf("\nCorrect: Total profit = %f",total_profit);
}

void calculate_payment()
{
    int i;
    for(i=0;i<slave_bids;i++)
    {
        pay[i] = (bids[i]+1)*load[i];
        profit[i] = pay[i] - cost[i];
        printf("load = %f, rate = %f\n",load[i],rates[i+1]);
    }
    printProfit();
}

void calculate_cost()
{
    int i;
    for(i=0;i<slave_bids;i++)
    {
        cost[i] = load[i]/rates[i+1];
        total_cost+=cost[i];
        printf("load = %f, rate = %f\n",load[i],rates[i+1]);
    }
    calculate_payment();
}

void master()
{
    int i;
    sort(bids,bids+3);
    for(i=0;i<slave_bids;i++)
        printf("%f\n",bids[i]);
    float c,bids_total = 0.0,sqrt_bids=0.0;
    for(i=0;i<slave_bids;i++)
    {
        bids_total += 1.0/bids[i];
        sqrt_bids += sqrt(1.0/bids[i]);
    }
    c = (bids_total - total_arr_rate)/(sqrt_bids);
}

```

```

int n = slave_bids;
while(c > sqrt(1.0/bids[slave_bids-1]))
{
    load[n]=0.0;
    n--;
    for(i=0;i<n;i++)
    {
        bids_total += 1.0/bids[i];
        sqrt_bids += sqrt(1.0/bids[i]);
    }
    c = (bids_total - total_arr_rate)/(sqrt_bids);
}
for(i=0;i<slave_bids;i++)
{
    load[i] = 1.0/bids[i] - c * sqrt(1.0/bids[i]);
    MPI_Send(&load[i],1,MPI_FLOAT,i+1,0,world);
}
calculate_cost();
printCost();
}

int main(int argc, char const *argv[])
{
    int no_of_jobs=0,world_rank;

    MPI_Comm_size (MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    if (nprocs < 2) {
        printf ("Number of processors < 2\nExiting\n");
        exit (1);
    }
    slave_bids = nprocs-1;
    // Leader election
    bool terminated=false;
    int inmsg[2];    //incoming msg
    int msg[2];
    msg[0]=world_rank;
    msg[1]=0;
    MPI_Status Stat;
    int min=world_rank;    //initially the minimum ID is node's own ID
    int lcounter=0;    //local counter of received messages (for protocol)
    int lmc=0;    //local counter of send and received messages (for
analysis)
    int dummy=1;
    int leader;
    //send my ID with counter to the right
    MPI_Send(&msg, 2, MPI_INT, (world_rank+1)%nprocs, 1, MPI_COMM_WORLD);
    lmc++;    //first message sent

    while (!terminated){
        //receive ID with counter from left
        MPI_Recv(&inmsg, 2, MPI_INT, (world_rank-1)%nprocs, 1,
MPI_COMM_WORLD, &Stat);
        lmc++;
        lcounter++;    //increment local counter
        if (min>inmsg[0])

```



```

min=inmsg[0];    //update minimum value if needed
    inmsg[1]++;    //increment the counter before forwarding

    if (inmsg[0]==world_rank && lcounter==(inmsg[1])) {
        terminated=true;
    }

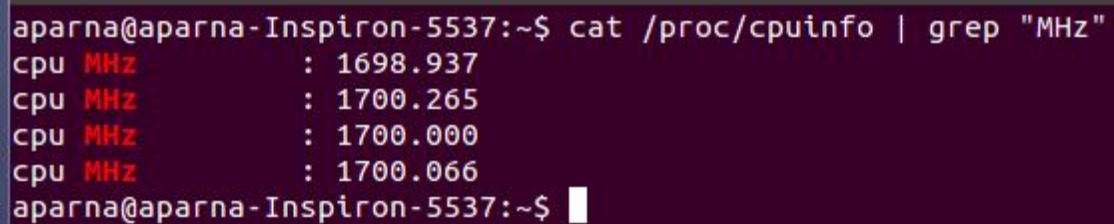
    //forward received ID
    MPI_Send(&inmsg, 2, MPI_INT, (world_rank+1)%nprocs, 1,
MPI_COMM_WORLD);
}
leader = min;
FILE *fp;
fp = fopen("cmd_lines","r");
if(fp == NULL)
{
    printf("\nCannot open file");
    return 0;
}
char c;
for(c = getc(fp); c!=EOF ; c=getc(fp))
{
    if(c=='\n')
        no_of_jobs++;
}
total_arr_rate = no_of_jobs;
fclose(fp);
if(world_rank!=leader)
{
    float proc_load;
    MPI_Send(&rates[world_rank],1,MPI_FLOAT,leader,0,world);
    MPI_Recv(&proc_load,1,MPI_FLOAT,leader,0,world,MPI_STATUS_IGNORE);
    printf("Load received by processor %d = %f\n",world_rank,proc_load);
    load[world_rank-1]=proc_load;
}
else if (world_rank == leader)
{
    float num;
    int rank;
    for (rank = 1; rank < nprocs; ++rank) {
        // Receiving the bids from slaves
        MPI_Recv(&num, 1, MPI_FLOAT, MPI_ANY_SOURCE,0, world,
MPI_STATUS_IGNORE);
        bids[rank-1]=1.0/num;
        // printf("Num %f Rank %d \n",num,rank);
    }
    master();
}
}

```

### File - cmd\_lines

```
echo simulator 1
echo simulator 2
echo simulator 3
echo simulator 4
...
echo simulator 98
echo simulator 99
echo simulator 100
```

### Configuration of the System

A terminal window with a dark purple background. The prompt is 'aparna@aparna-Inspiron-5537:~\$'. The command 'cat /proc/cpuinfo | grep "MHz"' has been executed, resulting in four lines of output: 'cpu MHz : 1698.937', 'cpu MHz : 1700.265', 'cpu MHz : 1700.000', and 'cpu MHz : 1700.066'. The prompt is followed by a white cursor block.

```
aparna@aparna-Inspiron-5537:~$ cat /proc/cpuinfo | grep "MHz"
cpu MHz      : 1698.937
cpu MHz      : 1700.265
cpu MHz      : 1700.000
cpu MHz      : 1700.066
aparna@aparna-Inspiron-5537:~$
```

### Running the Program

```
mpic++ -o global_approach.cpp global_approach
mpirun -np 4 global_approach
```

## RESULTS

---

### Test case 1 :

The algorithm has been implemented with the systems sending their true values as their bids.  
The result is as follows :

```
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/DCS/Project$ mpirun -np 4 ./global
Overall expected execution time : 0.602408
Total load : 120.000000
Load received by processor 1 = 39.999874
Load received by processor 2 = 39.999874
Load received by processor 3 = 39.999874
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/DCS/Project$ █
```

The execution times have been given in terms of  $10^3$ .

The overall execution time is 0.602408

### Test case 2:

A faulty case, where systems report false bids.

```
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/DCS/Project$ mpirun -np 4 ./global_faulty1
Overall expected execution time : 0.607982
Total load : 120.000000
Load received by processor 1 = 73.235191
Load received by processor 2 = 31.413343
Load received by processor 3 = 15.351452
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/DCS/Project$ █
```

The overall execution time increased from 0.602408 to 0.607982

### Test case 3:

With faulty values again.

```
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/DCS/Project$ mpirun -np 4 ./global_faulty2
Overall expected execution time : 0.605751
Total load : 120.000000
Load received by processor 1 = 66.655190
Load received by processor 2 = 28.180563
Load received by processor 3 = 25.164251
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/DCS/Project$ █
```

The overall execution time increased from 0.602408 to 0.605751

### Test case 4:

Faulty values taken again.

```

sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/DCS/Project$ mpirun -np 4 ./global_faulty3
Overall expected execution time : 0.604008
Total load : 120.000000
Load received by processor 1 = 58.395298
Load received by processor 3 = 29.110245
Load received by processor 2 = 32.494453
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/DCS/Project$ █

```

The overall execution time increased from 0.602408 to 0.604008

### Test case 5:

The 4th faulty case.

```

sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/DCS/Project$ mpirun -np 4 ./global_faulty4
Load received by processor 1 = 97.724052
Load received by processor 2 = 12.086512
Load received by processor 3 = 10.189431
Overall expected execution time : 0.618178
Total load : 120.000000
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/DCS/Project$ █

```

The overall execution time increased from 0.602408 to 0.618178

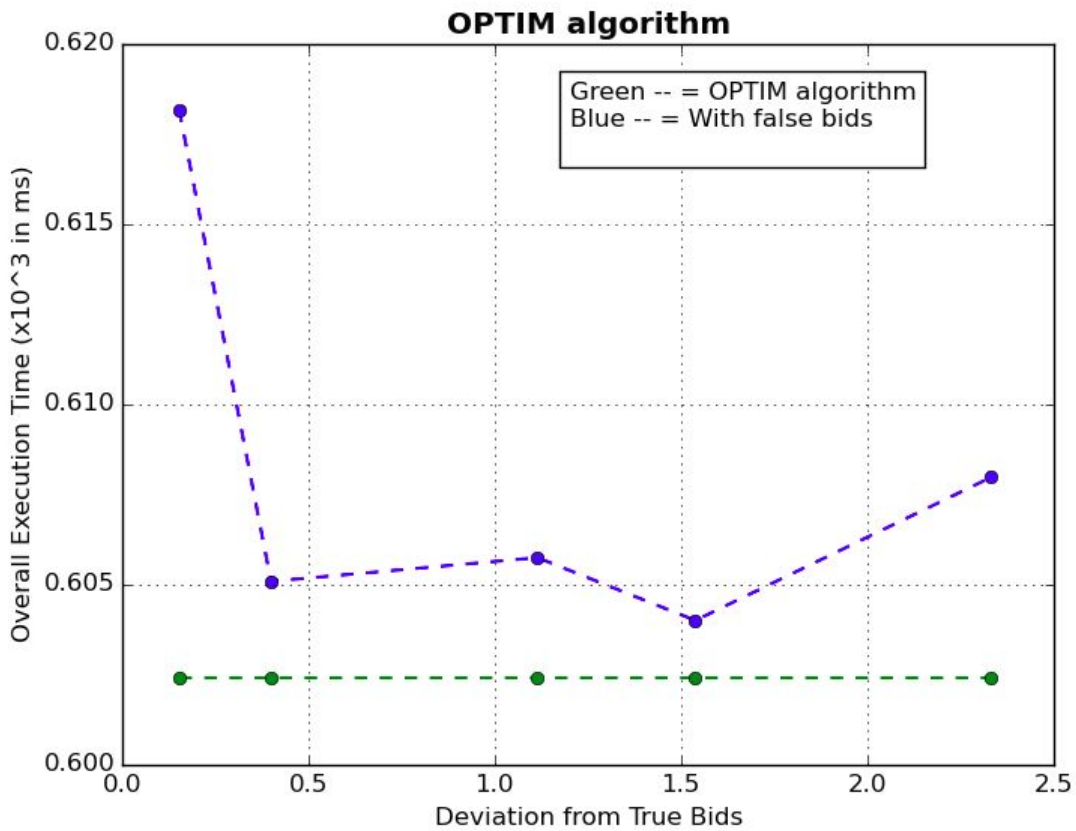
### Test case 6:

```

sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/DCS/Project$ mpirun -np 4 ./global_faulty5
Overall expected execution time : 0.605093
Load received by processor 1 = 55.758430
Load received by processor 2 = 47.919720
Load received by processor 3 = 16.321852
Total load : 120.000000
sheetal@sheetal-hp-15-notebook-pc:~/Documents/6thSemSubjects/DCS/Project$ █

```

The overall execution time increased from 0.602408 to 0.605093



This graph shows the deviation from true bids versus the overall execution time. As we can observe, the green line is drawn to indicate the overall execution time when all systems give their true bids. The blue dots will be above it.

## CONCLUSION

This paper provides an improved and less complex algorithm whose aim is to design a truthful payment scheme such that the profit of each system is maximized and their overall execution time is minimized. The algorithm has been implemented using openmpi-1.4.5, and all the different test cases have been analyzed. The overall execution time is found to increase with the increase in deviation from true bids. A graph showing the same has been plotted.

## REFERENCES

- [1] Grosu, Daniel, and Anthony T. Chronopoulos. "Algorithmic mechanism design for load balancing in distributed systems." *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 34.1 (2004): 77-84.
- [2] [http://www.boost.org/doc/libs/1\\_60\\_0/doc/html/mpi.html](http://www.boost.org/doc/libs/1_60_0/doc/html/mpi.html)