Homework 8 - Load Balancer

Team (Group 10):

Nitesh Gupta : 43487933 (niteshg)

Pallavi Garg : 87608678 (pallavg1)

Code Structure

Tunable Variables: Following are the tunable parameters in our load balancer:

1. `L_MIN` and `L_MAX` - To set the minimum and maximum value of load units.
2. `D_MIN` and `D_MAX` - To set the minimum and maximum value of next time cycle for running load balancing activity
3. `MAX_CYCLES` - To set the maximum number of time cycles to run the load balancer in the system.
4. `BALANCED_LOAD_THRESHOLD` - To set the percentage difference in load units among neighboring processors. For example, if two processors can have 2% of load units difference, then this value should be set at 0.02.
5. `STEADY_STATE_THRESHOLD` - To set the maximum percentage of processors which can be unbalanced in a steady system. For example, if the system can tolerate 5% of unbalanced processors then this value should be set at 0.05.

Helper methods and Data Structures: We have created a data structure Processor to represent physical processors in the distributed system. Each processor is instantiated in the method `createNewProcessor(int position)`. Each processor knows its position in the ring architecture, number of load units assigned to it, and a pointer to its left and right neighbor in the system. The system is initialized in method `initializeRingSystem()`. Method `disposeRingSystem(int k)` is created to dispose of the ring architecture. `printAllProcs()` function prints the state of all the processors in the system. Helper methods such as `getUniformlyRandomLoadUnits()` and `getUniformlyRandomNextActivityTime()` are used to calculate the random value of load unit and next load balancing activity time respectively.

Algorithm and methods: The main logic of the load balancer lies in following methods:

1. `void performLoadBalancing(int k)` - This method keeps track of the time cycles and loops over all the processors using `getNextProcessor()` in the order of their `nextLoadBalanceTime` field. It stops the load balancing process if a steady state is reached and is checked using `bool isSteadyStateAchieved()`.
2. `struct Processor* getNextProcessor(struct Processor *proc)` - Returns the next processor starting from the given processor's right neighbor, who has the lowest `nextLoadBalanceTime` value.
3. `bool isSteadyStateAchieved()` - Uses the global knowledge and returns true if steady state is achieved in the system.
4. `void balanceLoad(struct Processor *proc)` - Balances load of given the processor by using local knowledge of neighboring processors. This method internally uses the `shiftLoad()` to shift the load units to its neighbors if they have less work than the average load units.

Note: These methods will make more sense in the next section where we discuss the details.

Unbalanced Load Initialization: We set the load units using `getUniformlyRandomLoadUnits()` for every 3rd processor in the system. For example, processors at positions 1, 2, 4, 5, 7… will have 0 load units in the beginning. The processors at position 3,6,9,... will have random load units.

Termination: We terminate the load balancing processor if any of the following two conditions occur:

1. The global time cycles in the system reaches `MAX_CYCLES`.
2. The system reaches a steady (possibly balanced) state which is defined in the next section.

Building and Running the Code

- To build use the command: “gcc balance.c -o lb” as shown below in Fig 1.
- To run the application, use one of the two commands: “./lb <k>”. In order to print a more detailed output use the verbose option as “./lb <k> v” as shown in Fig 2.

```
pallavi@Orion:~/Workspace/pdc/LoadBalancer$ ./lb 10
Total load units: 1570
Time cycles: 15205
Load balancing activities performed: 271
```

Fig 1: Running load balancer

```
pallavi@Orion:~/Workspace/pdc/LoadBalancer$ gcc balance.c -o lb
pallavi@Orion:~/Workspace/pdc/LoadBalancer$ ./lb 5 v
***System Configuration before load balancing***
Position = 0, Load = 212
Position = 1, Load = 0
Position = 2, Load = 0
Position = 3, Load = 764
Position = 4, Load = 0

Max load difference between two neighbors: 9
Max unbalanced processors allowed in the system: 0

***System Configuration after load balancing***
Position = 0, Load = 194
Position = 1, Load = 187
Position = 2, Load = 194
Position = 3, Load = 203
Position = 4, Load = 198
Unbalanced processors after load balancing finished: 0
Unbalanced load units after load balancing finished: 0

Total load units: 976
Time cycles: 2918
Load balancing activities performed: 25
```

Fig 2: Running load balancer with verbose

Balanced Among Neighbors

A processor P_n is said to be balanced among its neighbors if both the neighbors have load units between (*Load units of P_n - balanced load threshold) and (Load units of P_n + balanced load threshold*). Balanced load threshold value is calculated once the ring architecture is initialized and load units are assigned to all the processors. It is defined as the percentage of average load per processor and is represented by `BALANCED_LOAD_THRESHOLD` value in code. For example, if a system has 1000 load units and 10 processors, then each processor in an ideally balanced system should have 100 load units. If the value of `BALANCED_LOAD_THRESHOLD` is 0.02 (2%), then two neighboring processors can have load units like (98, 100) or (102, 100) or (100, 100) or (101, 100) and (99, 100) and vice versa. If calculations with 2% come out to be less than 1, then we default it to 1.

Steady state

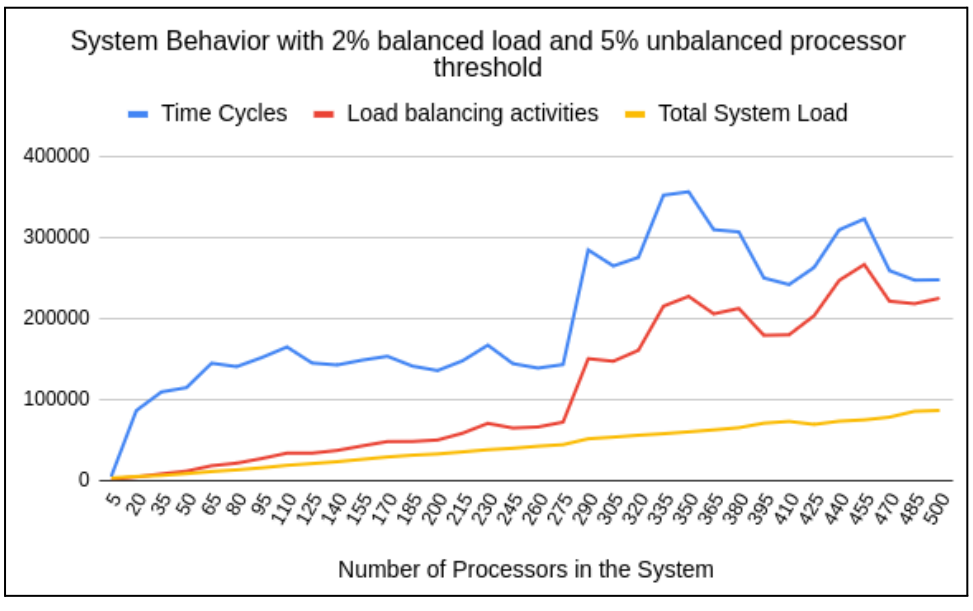
The system of k processors is said to have reached a steady state if (`k-maxUnSteadyProcs`) are balanced among their neighbors. The value of `maxUnSteadyProcs` is calculated once the ring system has been initialized in the `initializeRingSystem()` method. Its value is defined as `STEADY_STATE_THRESHOLD * k`. For example, if a system has 100 processors and `STEADY_STATE_THRESHOLD` is 0.05 (5%), then up to 5 processors are allowed to not be balanced among their neighbors. In this system, if k < 20, then `maxUnSteadyProcs`= 0 which means all k processors must be balanced. If we need to have a more strict system then the value of `STEADY_STATE_THRESHOLD` can be set to 0.00 (0%) and then all processors will need to be balanced among their neighbors to be in a steady state system.

Balanced state

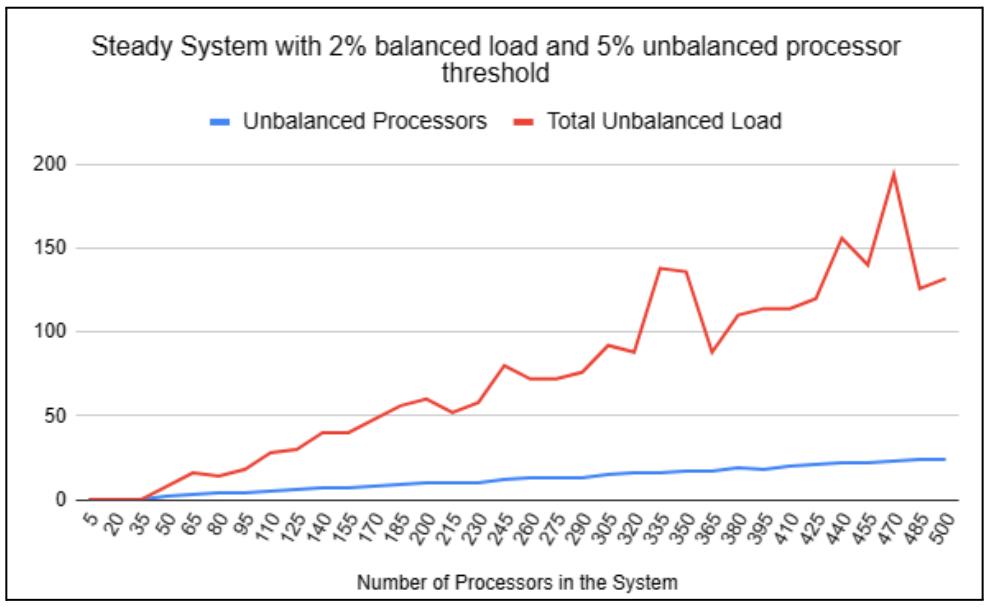
A balanced system is the one in which all the processors are balanced among their neighbors. In case a steady state is achieved in the system or the `MAX_CYCLES` are reached, then values of `unbalancedProcsDuringConvergence` and `unbalancedLoadDuringConvergence` gives the status of system as they contain the count of processors are not balanced and the number of load units which are unbalanced. These are visible in the verbose output of Fig 2 as “Unbalanced processors after load balancing finished: 0” and "Unbalanced load units after load balancing finished: 0”. If these values are non-zero, it suggests that as per our algorithm, a steady state has been reached.

Results and Observations

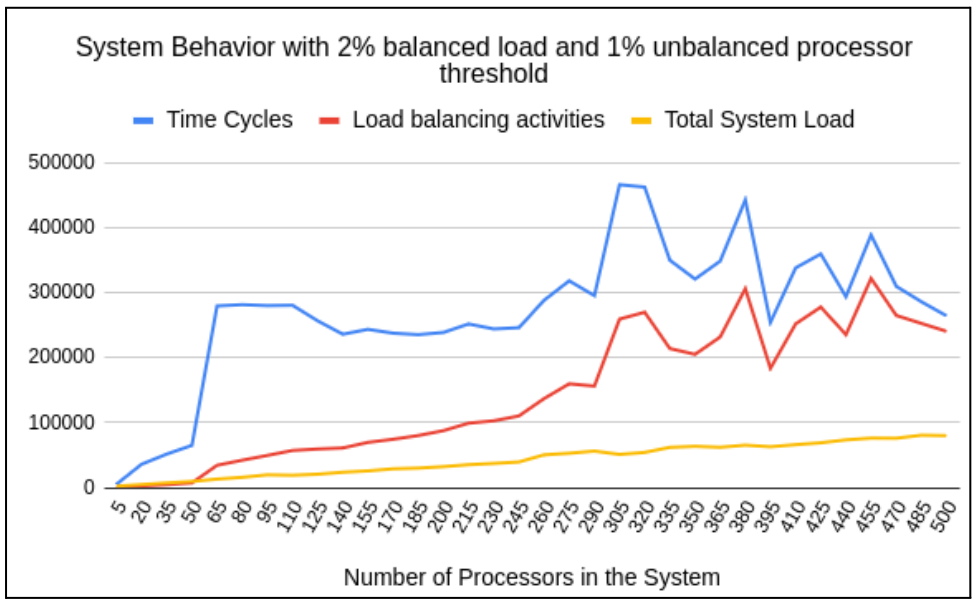
For our observations we are using `BALANCED_LOAD_THRESHOLD` is 0.02 (2%) and `STEADY_STATE_THRESHOLD` is 0.05 (5%) for our measurements.



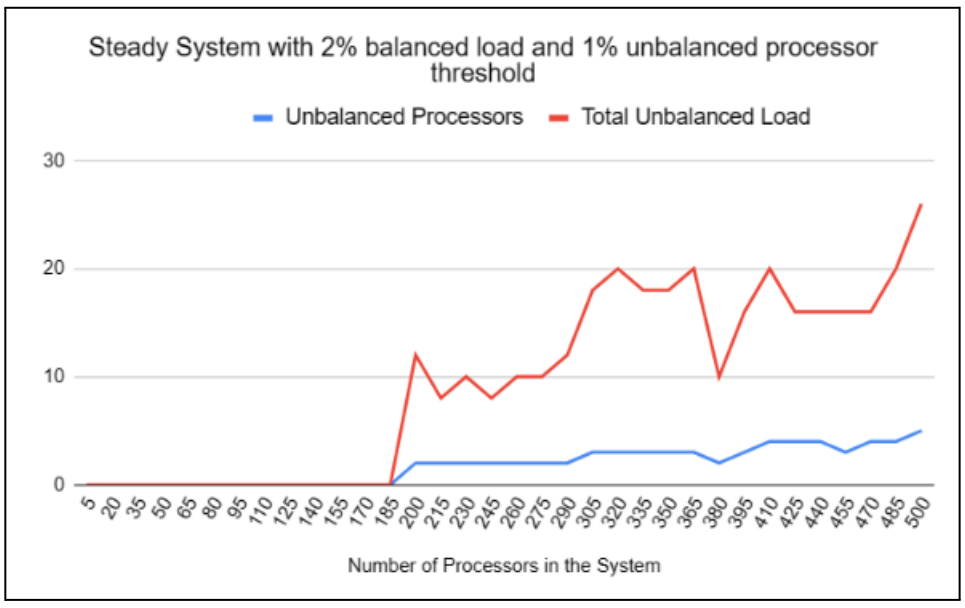
(a)



(b)



(c)



(d)

Fig 3: System Behavior with different threshold values

Please **note** that because we are using random values for load units, and for the next time to perform balancing activity, different runs of our program may lead to slightly different graphs.

Fig 3 (a) and (b) shows the system behavior i.e. load balancing results for different systems with varying processors. In these two iterations, we are using 2% balanced load difference among neighboring processors and 5% processors are allowed to be unbalanced in a steady system. As shown in Fig 3 (a), as the number of processors are increased in the system, the total loads increase and the number of load balancing activities also increase along with the total time cycles required to balance the system. We can see from Fig 3 (b) that after load balancing, systems up to 35 processors are completely balanced with all the loads balanced among all processors.

A stricter system is presented in Fig 3 (c) and (d) where we are using 2% balanced load difference among neighboring processors and 1% processors are allowed to be unbalanced in a steady system. From Fig 3 (d) we observe that now the systems with processors <= 185 are fully balanced, but we also observe from Fig 3 (c) that now the system has taken more time cycles to reach a steady state with the same number of processors.

We conclude that since load balancing is actually a system overhead we should tune these parameters in the load balancing program based on the requirements of the system. If the system can tolerate having some unbalanced processors, then we can reduce the number of load balancing activities in the entire system and keep `BALANCED_LOAD_THRESHOLD` and `STEADY_STATE_THRESHOLD` to the minimum.

We observe that as we increase the values of `BALANCED_LOAD_THRESHOLD` and `STEADY_STATE_THRESHOLD` then the program takes less time cycles to converge to achieve a steady solution but the system tends to have more unbalanced processors.

Does the proposed strategy converge to a balanced state?

Yes, the proposed strategy converges to a balanced state within the scope of the previously defined definitions. Fig 4 shows the convergence and the state of the system after 141630 cycles for 100 processors. In this run, we used 2% balanced load threshold and 5% unsteady processor threshold. The initial load units are distributed as defined previously, that is every 3rd processor has some random load units assigned.

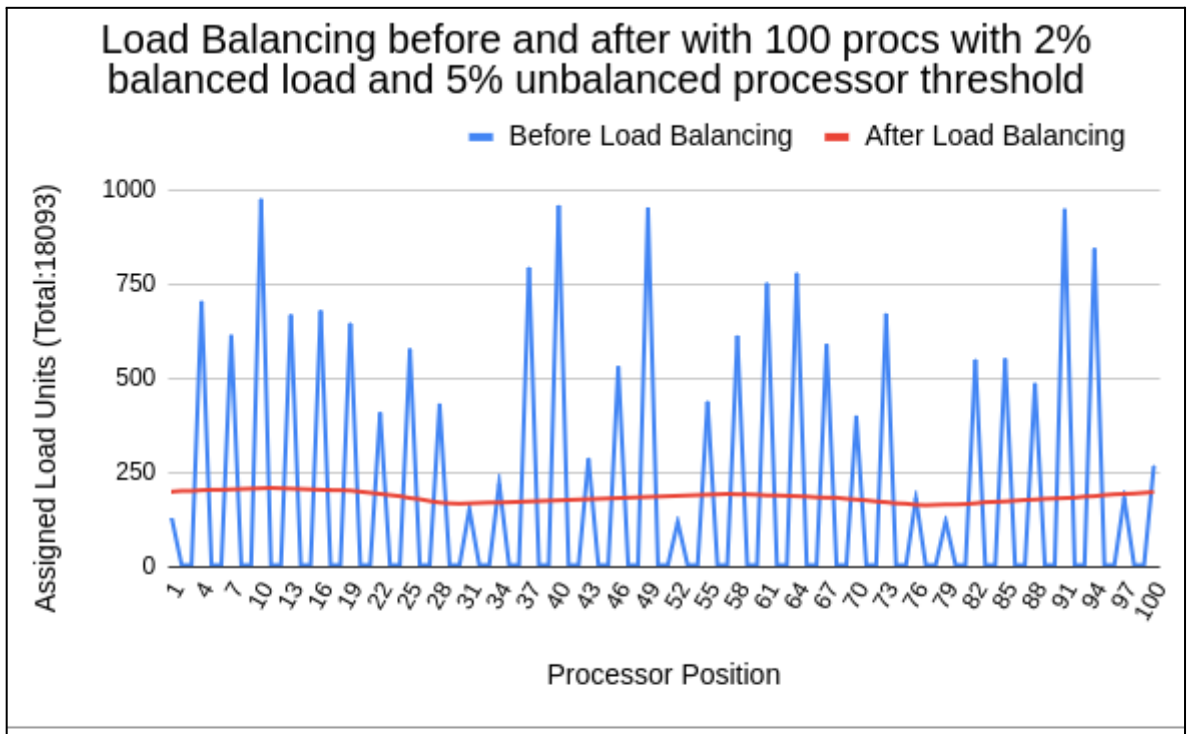


Fig 4: Convergence with 100 processors in the system

Can you prove that the strategy will converge to a balanced state for any possible initial load-units distribution?

No, we cannot prove that this strategy will converge for any initial load-units distribution. As shown in Fig 5, we can see that the load balancing process stopped because `MAX_CYCLES` have been reached and the system is not in a steady state. 29 processors are not balanced in the system. If we increase the `MAX_CYCLES` value in the system, then also there is no guarantee that this will converge. This happened because most of the cycles are wasted for those processors who did not have any loads assigned to them e.g., P10 to P19 and P31 to P60 in Fig 5. If the load was distributed more uniformly as in the previous section (Fig 4), then this case will not occur.

The reason for this behavior also involves the random selection of the next time cycle to run the balancing activity. There is no logic in the proposed strategy to intelligently choose the processor at any time who should perform the load balancing activity. For example, if a processor whose time to run balancing activity is far away but it knows that its neighbor has no or little work, if allowed to diffuse its load to that neighbor at some early time, then we should be able to observe a better system. This can be implemented using some priority donation based algorithm and synchronization mechanism. Another better strategy could be to allow the processors with no or little load to ask for work from their neighbors.

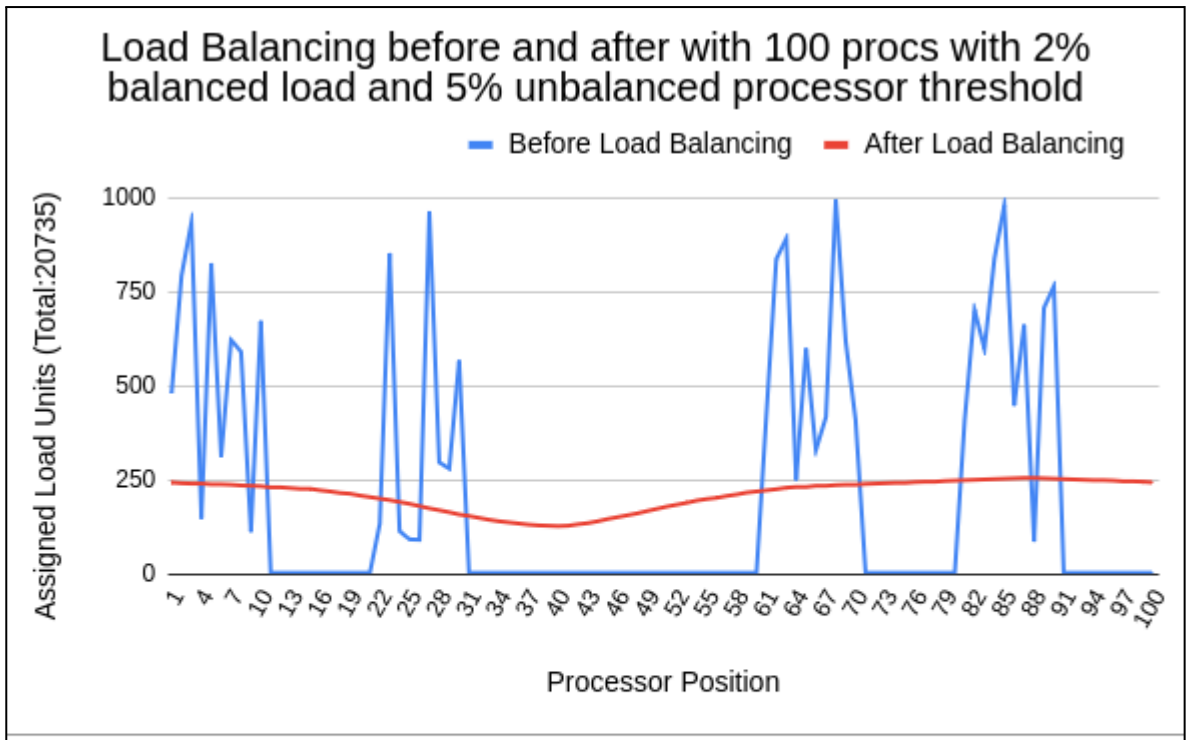


Fig 5: Non-Convergence with 100 processors in the system

What could be the “worst” possible initial load-units assignment?

The worst case initial load-units assignment would be the one where the first half of the processors have no load units assigned and the second half of the processors have all the loads assigned as shown in below Fig 6. In this distribution case, some processors did not get any load units after the load balancing ended because the `MAX_CYCLES` were reached and most cycles were wasted while working for processors with no load units assigned.

The reasons for this observation are the same as we discussed in the previous section.

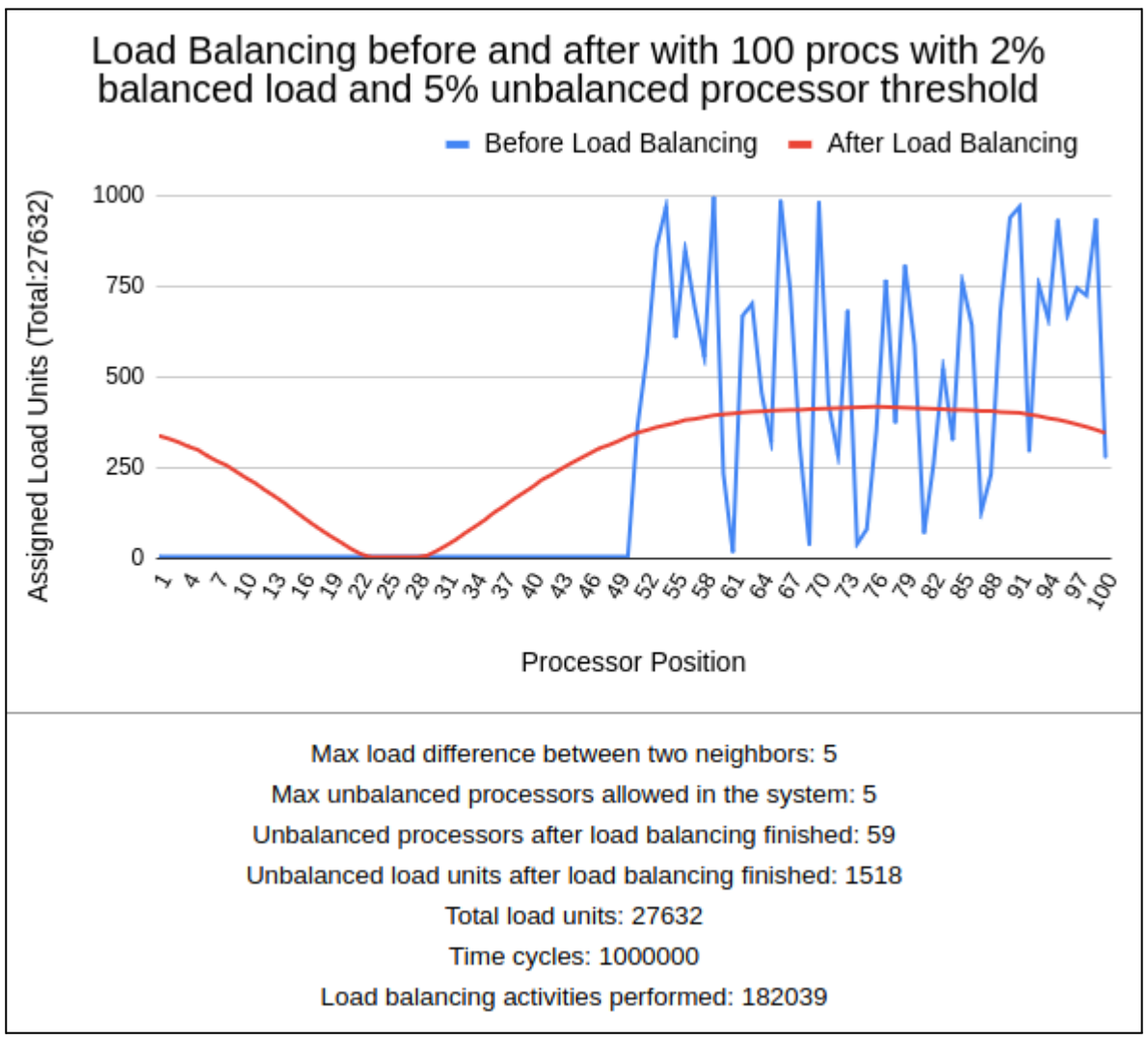


Fig 6: Worst Load Unit distribution

References

https://canvas.eee.uci.edu/courses/56554/files/23771218?module_item_id=2375249