DOCKER CONTAINER PLACEMENT IN DOCKER SWARM CLUSTER BY USING WEIGHTED RESOURCE OPTIMIZATION APPROACH

Jalpa M Ramavat[1], Dr Kajal S Patel[2]

•

Research Scholar, Gujarat Technological University[1]
jalpa.ramavat.2012@gmail.com
Associate Professor, VGEC, Chandkheda[2]
kajalpatel@vgecg.ac.in

Abstract

The use of Docker containers and their orchestration tools is rapidly improving as Web application deployment shifts from a server- or VM-based approach to a container-based approach. Docker Swarm is a flexible and simple container orchestration tool. it is widely used by application developers for the deployment of their applications in a containerized environment. Docker Swarm uses the default spread strategy for placing new containers on cluster nodes. This strategy distributes containers evenly on all nodes of the cluster, but it will not consider the current resource utilization of nodes or heterogeneous resource availability on cluster nodes. Again, all task containers are treated similarly, irrespective of their specific resource-oriented nature. This paper proposes the weighted resource optimization algorithm for calculating the weighted score of each node. Score depends on CPU and memory weight for a given task and the availability of that resource on the node. The task container is placed on the node with the highest score. This approach improves CPU and memory load balancing in a Docker cluster and also improves the completion time of the task container as compared to the spread strategy.

Keywords: Cloud, VM, Containers, Docker Swarm, Orchestration

I. Introduction

Virtual machines (VMs) are widely utilized in organizations, even for running minor applications, leading to system inefficiencies. There's a growing demand for lightweight application deployment to enhance efficiency. Docker containers have emerged as a solution, offering lightweight virtualization [2].

As the demand for containers grows, efficient scheduling and orchestration mechanisms are paramount for distributing container executions across numerous nodes in cloud clusters. Consequently, various container scheduling tools have emerged, including Docker Swarm from Docker, Mesos from Apache, and Kubernetes from Google [10].

Docker Swarm is a container orchestration tool developed by Docker [11]. The Docker Swarm cluster has two types of nodes: masters and workers. Any node can leave the cluster at any time. Again, a new node can be added to the cluster by using tokens generated [12]. A Docker Swarm cluster has components like tasks, services, Raft Consensus Groups, internal distributed state stores, managers, and worker nodes [13]. A task is a combination of a container and a command to run it, and a service is represented by a single task or replicas of a task [13]. Manager nodes handle API requests, orchestration, allocation of IP addresses, scheduling, and control over worker nodes [11].

Worker nodes receive task containers and execute them [11]. Raft Consensus Groups are for handling fault tolerance by selecting a new leader if the current one becomes unavailable [14]. Docker Swarm integrates seamlessly with the Docker environment, simplifying container deployment and management. It allows decentralized design, declarative model, scaling, error correction, networking, load balancing, security, and version updates with restore points [15]. External load balancing and graphical interface for management were added after May 2019 [1].

Docker Swarm is commonly implemented without prior information about the workload or the specific resource requirements of containers. Consequently, it exclusively utilizes a single scheduling strategy referred to as "Spread" [10].

This policy aims to ensure that the workload is evenly distributed. It tries maintain equal number of running containers on each node within the cluster. If sufficient resources are available on a node having least number of running containers than new container will be placed on that node irrespective of resource utilization of that node by currently running containers. So, it tries to spread container equally among all Docker cluster nodes.

The spread strategy will not consider current resource utilization of node before placing containers. There might be load imbalance when containers are specific resource oriented i.e. CPU bound, Memory bound etc.

Over the past years, container scheduling has garnered considerable research attention. The author [7] introduces an Availability-Based Prioritization (ABP) scheduler to enhance service availability and scheduling efficiency within Docker Swarm. It begins by detailing a Service Availability Model. This model calculates service availability based on task distribution across nodes and considers mean time between failures (MTBF) and mean time to repair (MTTR). This model defines availability as the valid probability of nodes hosting service replicas, with replica requirements determined by node validity. The Modules Design section outlines the functionality of various components within Docker Swarm, with the Spread strategy serving as the default scheduling mechanism. The heart of the paper lies in the Scheduling Strategy of the ABP Scheduler. The periodic decisions are made to scale services based on task requirements and current distributions. These decisions prioritize tasks and nodes based on replica gaps, image layer coincidence, and dominant shares, ensuring efficient resource utilization. The Scale-Out and Scale-In decisions sections elaborate on how the ABP scheduler handles service scaling. Overall, the ABP scheduler presents a promising approach to dynamically adjusting service replicas to meet availability targets while optimizing resource allocation within Docker Swarm.

Mao et al. [4] analyzed two cloud platforms, Docker and Kubernetes for finding how well resource management is done on both of these domains. Author created a container monitoring system using Prometheus and Grafana which continuously monitor how much resource usage is done by each job on the worker nodes. Results show that by altering the default configurations on Docker and Kubernetes, the completion times were lowered by up to 79.4% and 69.4%.

The paper introduces ECSched [8], a container scheduling solution for heterogeneous clusters. This approach overcomes limitations of traditional queue-based schedulers by leveraging a graph-based approach and modeling scheduling as a Minimum Cost Flow Problem (MCFP). While ECSched demonstrates superior performance in terms of container completion time and resource utilization compared to baseline schedulers. But it introduces increased complexity in algorithm runtime, particularly noticeable when processing large numbers of concurrent container requests. This heightened computational demand could potentially pose challenges in highly dynamic or resource-constrained environments which is limiting the scalability of ECSched in certain scenarios. For container deployment of application tasks Wu and Xia [5] proposed a model for deploying containers at lowest possible deployment cost. They also proposed an improved PSO, known as the CD-PSO algorithm, to offer the best solution for application task loading

The adCFS [9] (Adaptive CPU Fair Sharing) policy aims to dynamically adjust CPU resource

allocation in containerized workflow systems based on workload characteristics, such as task runtime, CPU usage, and number of tasks. It utilizes a CPU State Predictor (CSP) to forecast CPU usage states and a Container's CPU Weight Scaler to redistribute CPU resources among containers accordingly. Two variations of the policy, soft (L1) and force (L2), are implemented based on CPU contention levels (cautious and severe states). L2 is enforcing strict CPU allocation based on estimated weights. This adaptive approach aims to improve fairness and efficiency in CPU sharing for scientific workflow processing. Experimental findings demonstrate a 12% improvement in container response time when compared to the default CFS policy.

Guanqaun Wu [6] proposed an improved ant colony algorithm to optimize Docker swarm cluster resource allocation. To reduce the start time of search they initialized pheromone by using scheduling algorithm minimum task first completion. Then they used balance factor to guide local and global pheromone updates for next iteration. For improving global search ability of algorithm, they use volatilization coefficient adjustment mechanism. The results show improvement of overall performance of cluster.

II. Methods

I. Docker Swarm Default strategy.

Docker swarm is a popular orchestration tool for containerized cloud environment. It has default scheduling strategy known as spread strategy. As its name suggest this strategy spread the container across all nodes in cluster equally. It tries to balance number of running containers on each node. However, it will not consider resource utilization of a node. This may lead to resource cluster node load unbalance and increasing execution time of service task container.

II. Proposed strategy.

To consider the resource availability and current resource utilization, this paper proposes a strategy which can be merge with existing docker swarm spread strategy to improve overall performance of the Service.

The proposed strategy will not only consider the resource of node but also focus on task type. Different Services require different resources. Services can be categorized depending on the intensity of resource they used. User has to specify the weight of resources i.e. CPU, Memory. If Service is CPU intensive than weight of CPU is higher and if Service is memory oriented then the weight of memory is higher. Weight can be given between 0 to 1.

Let N be the number of nodes in Docker cluster.

Let WCPU and WMEM be the weight assigned to CPU and Memory resources.

Let UCi is CPU usage of ith node and UMi is the memory usage of ith node in percentage.

The score Si for a node i is calculated as follows:

$Si = WCPU \times (100 - UCi) + WMEM \times (100 - UMi)$

The node selection process involves computing the score S for each node and selecting the node with the highest score:

So, Selected node is having score:

$Smax = max(S1, S2, \ldots, SN)$ Where N is total number of Nodes.

III. Proposed Algorithm (Weighted Resource Optimization):

Step 1. Start the Docker Swarm Cluster
Step 2: Input WCPU and WMEM
Step 3: For each service in the list do step 3 to 7
Step 4: For each node i in cluster
Step 5: Find score using

$$Si = WCPU \times (100 - UCi) + WMEM \times (100 - UMi)$$

Step 6: Selected node is having score Smax=max (S1, S2, …, SN)
Step 7: Create service for image and place its container on selected node

III. Results and Discussion

I. Experiment Setup

Three virtual machines were created using VirtualBox version 6.1.36. Manager virtual machine was allocated 2 CPU cores, 4 GB of RAM, and 40 GB of storage. Worker1 virtual machine was allocated 2 CPU cores, 3 GB of RAM, and 24 GB of storage. Worker2 virtual machine was allocated 2 CPU cores, 3 GB of RAM, and 26 GB of storage. Ubuntu 20.04 was installed on each virtual machine following the standard installation process.

Docker Engine version 24.0.2 was installed on each Ubuntu 20.04 virtual machine using the official Docker installation script provided by Docker. The Docker Python API (version 6.0.1) was utilized to programmatically interact with the Docker Swarm cluster for dynamic container placement. Python scripts were developed to leverage the Docker API for tasks such as creating Docker services, removing services, calculating Completion time etc.

Swarmprom[3] is a comprehensive toolkit designed for monitoring Docker Swarm environments. It includes essential monitoring components such as Prometheus, Grafana, cAdvisor, Node Exporter, Alert Manager, and Unsee, providing a complete solution for monitoring and managing Docker Swarm clusters.

Despite deploying all Swarmprom services, the focus of the experiment was primarily on utilizing Prometheus for metric collection and Grafana for visualization.

While Prometheus and Grafana were utilized for the experiment, it's noted that the Node Exporter, cAdvisor, and Docker Exporter containers were deployed on the worker nodes. These components contribute to comprehensive monitoring by collecting system and container metrics from the worker nodes.

II. Implementation and Results

Initially the status of three nodes is as shown in Figure 1 with containers for swarmprom are running in all the three nodes. It is shown that Manager node is having total 8 running containers. Worker1 and Worker2 nodes are having 3 running containers.

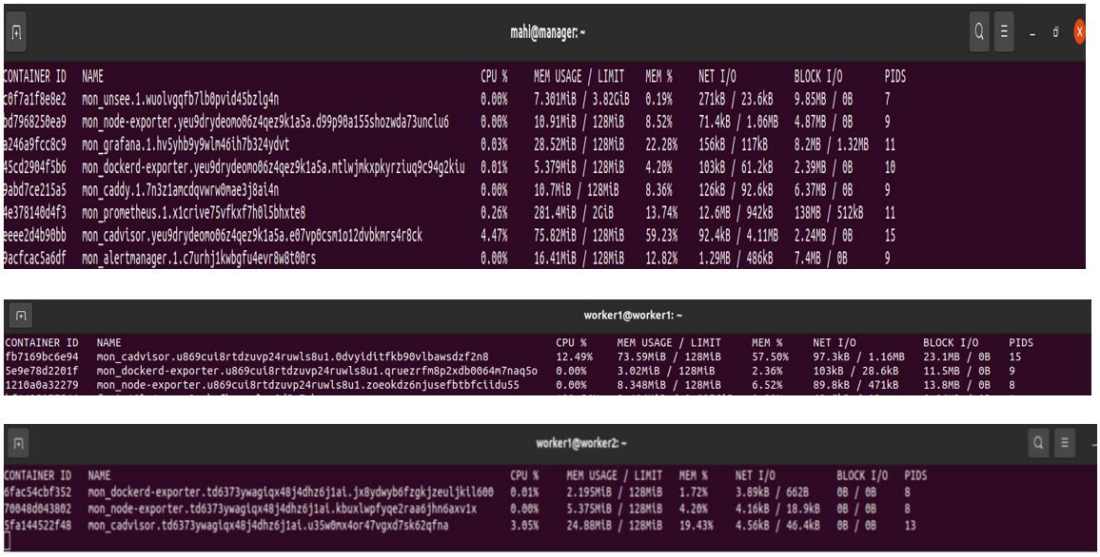

Figure 1 *Initial Docker Swarm Cluster Node -Manager, Worker1 and Worker2 with Swarmprom stack*

Initially, created images for the service of finding factorial of 500k ,300k and 100k. As factorial service requires more calculations, they are CPU intensive. Same way the images for creating and manipulating 4k by 4k and 5k by 5k arrays are also created and pushed in Docker hub. As storing 4K by 4K and 5K by 5K matrix required more memory, these are categorized under Memory intensive tasks.

After that to see the behavior of default spread strategy, five Nginx service containers are created and placed on Worker1 node by using docker placement constraint. The placement of these service containers will be as shown in Figure 2.

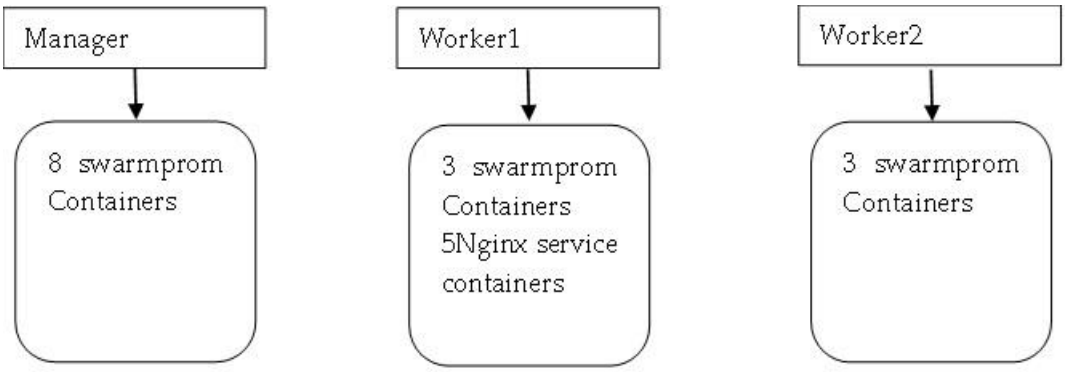

Figure 2 *Initial stat of Nodes after Swarmprom and Nginx container placement*

Now all services for images fact_500k, mem_5k, fact_300k, mem_4k and fact_100k are created and placed by default placement strategy. As spread distributes equal number of containers to all nodes the placement was seen in Figure 3. As manager is having 8 containers of swarmprom, worker1 is having 3 containers of swarmprom plus 5 containers of Nginx and worker2 is having least number of 3 containers of swarmprom stack so new created 5 service containers are placed on worker2 by Spread Strategy.

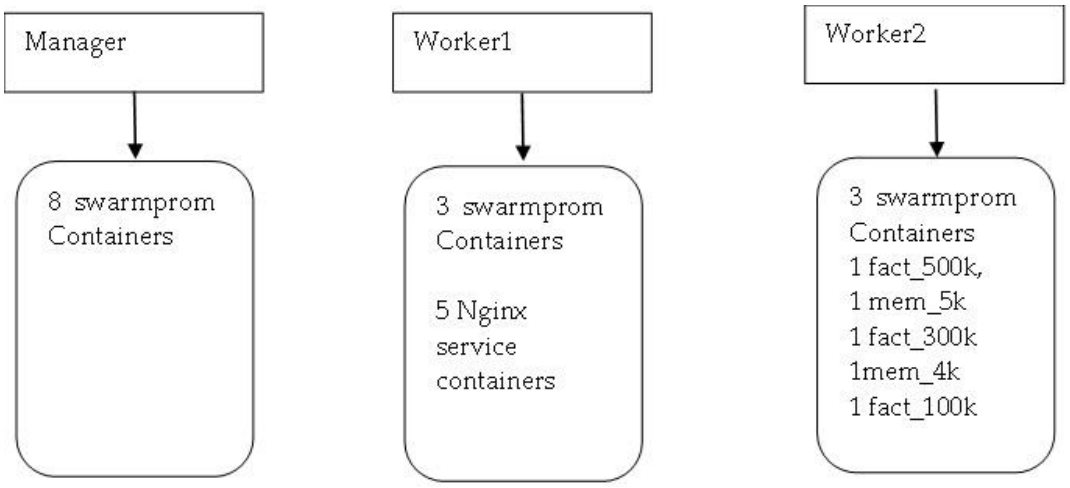

Figure 3 *Factorial and matrix manipulation task container placement (By Default Spread Strategy)*

Table 1 *CPU Utilization of Docker Swarm Cluster Nodes (Spread Strategy)*

Time(in Seconds)		0	90	180	270	360	450	540	630	720	810	900
CPU Usage (in %)	Manager	12.1	8.94	8.961	13.98	10.09	12.94	9.03	11.95	25	25.8	24.89
	Worker1	14.7	15.09	15.78	19.44	16.08	14.96	14.7	14.5	16.24	15.23	14.96
	Worker2	93.43	99.9	95.23	98.76	97	94.75	96.83	99.66	97.43	99.9	98

Table 2 *Memory Utilization of Docker Swarm Cluster Nodes (Spread Strategy)*

Time(in Seconds)		0	30	60	90	120	150	180	210	240	270	300
Memory Usage (in %)	Manager	61.61	62.47	62.76	61.89	57.64	57.67	57.77	58.84	59.30	59.10	58.33
	Worker1	33.58	33.65	33.65	33.61	33.58	33.58	33.65	33.92	33.85	33.82	33.68
	Worker2	68.48	59.41	60.99	60.86	59.44	67.97	51.01	55.33	64.57	58.46	57.01

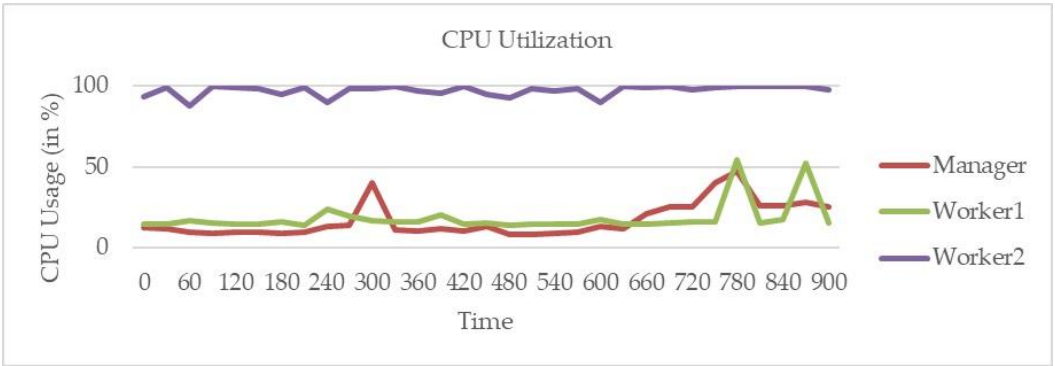

Figure 4 *Docker Swarm Cluster CPU Utilization after container placement by Default Spread Strategy*

Figure 5 *Docker Swarm Cluster Memory Utilization after container placement by Default Spread Strategy*

The CPU and Memory utilization of all cluster nodes are taken from Prometheus and Graphana as shown in Table 1 and Table 2. Now the readings are plotted as displayed in Figure 4 and 5. It is clear that the CPU load distribution is totally not even because number of containers in Worker1 and Manager node are more, all new created task containers are placed on Worker2 node by Spread Strategy without considering the resource utilization of all cluster nodes. Similarly, the memory usage is also not evenly distributed.

To avoid this unbalanced load distribution, the same containers are placed by using proposed strategy. First the results are taking using WCPU =0.7 and WMEM = 0.3. The task containers placement is shown in Figure 6.

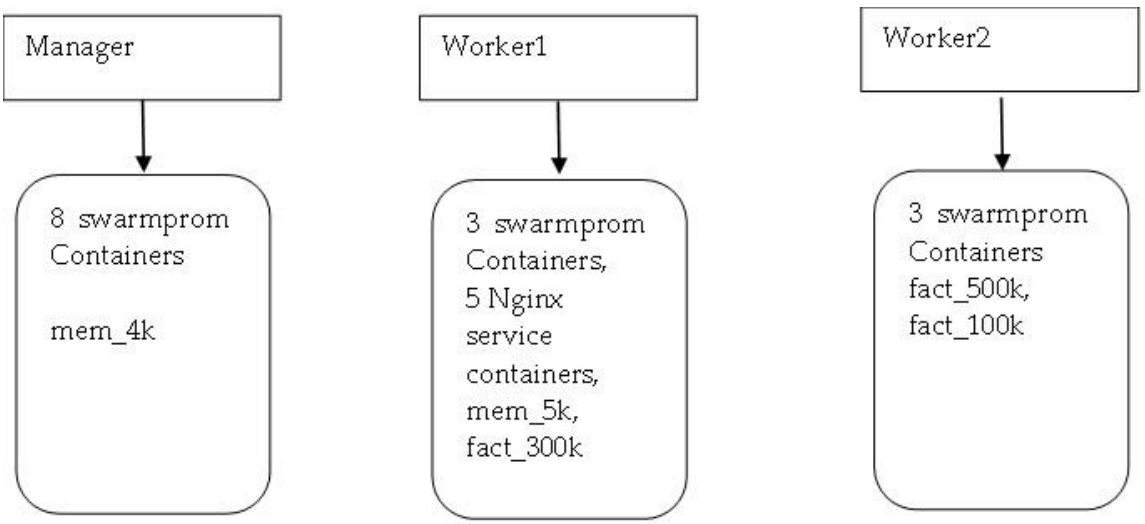

Figure 6 *Factorial and matrix manipulation task container placement (By Proposed Strategy: WCPU >WMEM)*

The CPU utilization of all three cluster nodes is as shown in Table 3 and plotted in Figure 7. The Memory utilization of all three cluster nodes is as shown in Table 4 and plotted in Figure 8.

Table 3 *CPU Utilization of Docker Swarm Cluster Nodes (Proposed Strategy: WCPU > WMEM)*

Time(in Seconds)		0	90	180	270	360	450	540	630	720	810	900
CPU Usage (in %)	Manager	37.53	50.40	42.46	44.45	31.57	34.56	32.44	33.23	33.00	49.81	34.26
	Worker1	76.50	71.77	77.53	76.11	73.94	75.57	84.43	77.37	83.28	77.36	79.77
	Worker2	60.43	65.83	72.80	68.23	48.31	68.67	69.23	57.33	68.33	69.17	65.30

Table 4 *Memory Utilization of Docker Swarm Cluster Nodes (Proposed Strategy: WCPU > WMEM)*

Time (in Seconds)		0	30	60	90	120	150	180	210	240	270	300
Memory Usage (in %)	Manager	53.86	53.37	53.50	59.25	53.25	56.24	53.32	53.73	53.20	68.48	62.81
	Worker1	51.18	65.75	62.88	64.30	62.91	61.77	62.41	61.46	64.67	61.83	63.65
	Worker2	32.97	33.34	33.61	33.88	33.88	34.36	34.29	34.52	34.12	34.39	34.19

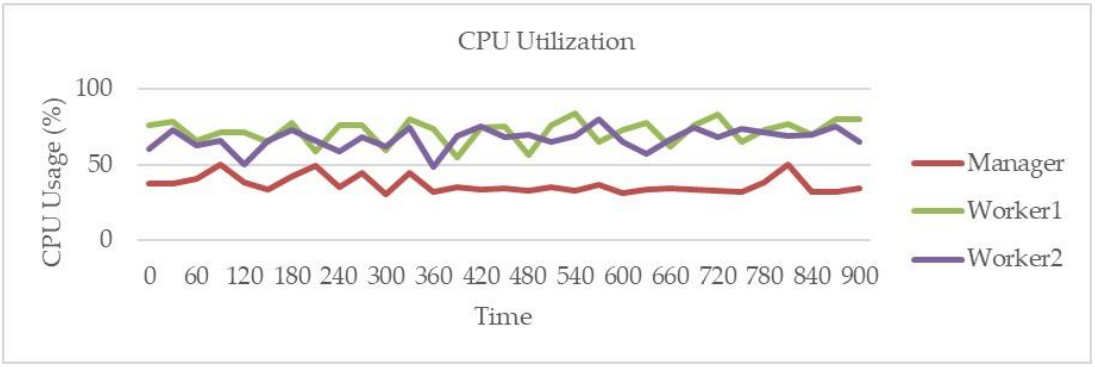

Figure 7 *Docker Swarm Cluster CPU Utilization after container placement by Proposed Strategy (WCPU > WMEM)*

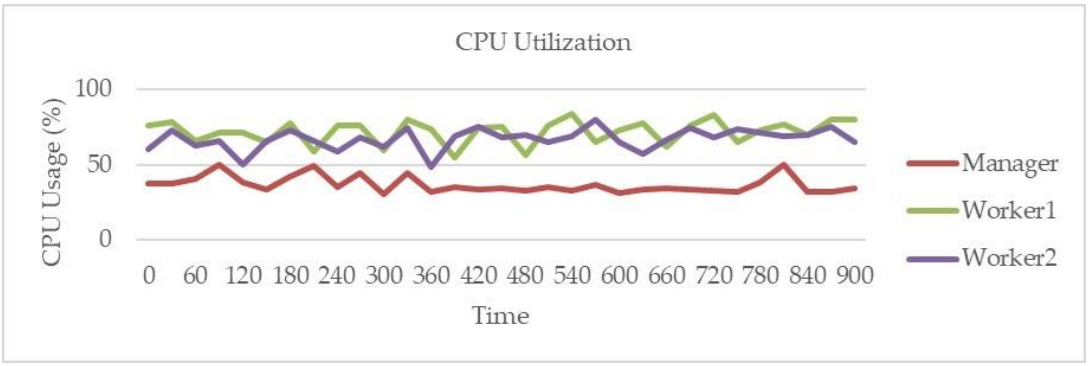

Figure 8 *Docker Swarm Cluster Memory Utilization after container placement by Proposed Strategy (WCPU > WMEM)*

As shown in above Figure 7 the CPU utilization of node is much balanced than as in spread technology shown in Figure 4. As shown in Figure 8 the memory Utilization of worker2 node is less as both are CPU oriented tasks are running on it.

Now the results are taken by using Proposed approach with W_{CPU}=0.3 and W_{MEM}-0.7. The new service containers are distributed as shown in Figure 9.

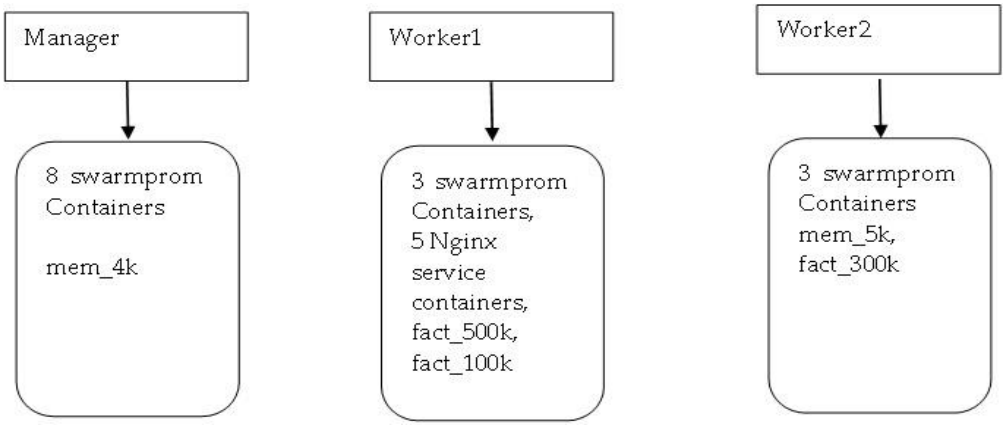

Figure 9 *Factorial and matrix manipulation task container placement (By Proposed Strategy: WCPU < WMEM)*

Table 5 *CPU Utilization of Docker Swarm Cluster Nodes (Proposed Strategy: WCPU < WMEM)*

Time (in Seconds)		0	90	180	270	360	450	540	630	720	810	900
CPU Usage (in %)	Manager	47.53	28.79	32.96	26.42	33.19	31.66	42.73	37.88	36.66	36.68	37.20
	Worker1	78.53	20.50	66.80	59.57	70.05	57.35	60.23	63.43	51.16	69.87	63.73
	Worker2	69.96	55.43	65.57	72.07	65.23	47.63	60.40	56.23	72.72	68.48	60.98

Table 6 *Memory Utilization of Docker Swarm Cluster Nodes (Proposed Strategy: WCPU < WMEM)*

Time(in Seconds)		0	30	60	90	120	150	180	210	240	270	300
Memory Usage (in %)	Manager	38.60	50.46	53.67	38.94	42.95	51.67	44.27	43.70	44.03	53.66	43.39
	Worker1	66.49	56.91	36.61	44.17	68.71	53.64	65.00	62.10	71.44	41.03	40.93
	Worker2	57.26	57.52	67.26	57.26	57.31	57.95	51.39	57.03	60.57	57.57	54.16

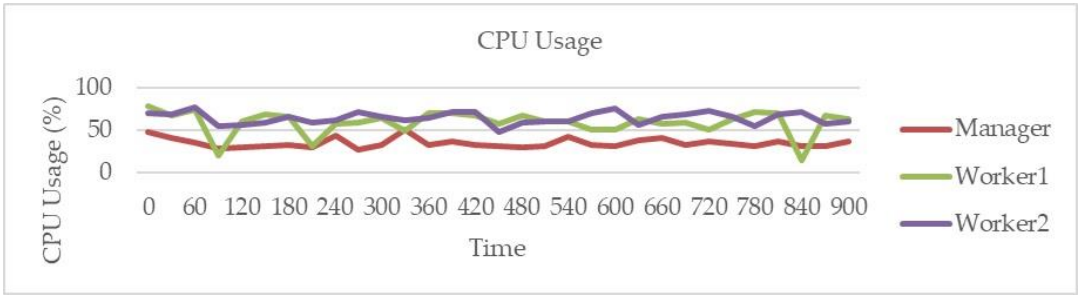

Figure 10 *Docker Swarm Cluster CPU Utilization after container placement by Proposed Strategy (WCPU < WMEM)*

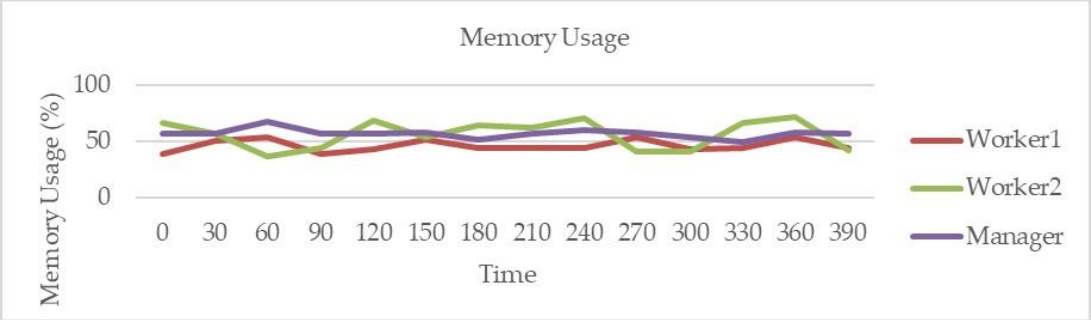

Figure 11 *Docker Swarm Cluster Memory Utilization after container placement by Proposed Strategy (WCPU < WMEM)*

Table 5 and Table 6 shows the CPU and Memory utilization of nodes by placing container using proposed algorithm by taking memory-oriented weight. As shown in Figure 11the memory load distribution is done more evenly. Also, CPU utilization is balanced as shown in Figure 10.

Now the completion time of containers of all services are calculated as shown in Table 7 and plotted in Figure 12. These readings are taken for 10 containers of each service and as factorial of 500k takes more time so 7 containers of it are taken. It can be seen from Figure 12 that the noticeable completion time reduction is achieved by using proposed approach.

Table 7 *Completion Time of task containers for Spread, Proposed (WCPU < WMEM, WCPU > WMEM)*

Services		ontainer1	ontainer2	ontainer3	ontainer4	ontainer5	ontainer6	ontainer7	ontainer8	ontainer9	ontainer10	Avg
Proposed $W_{CPU} > W_{MEM}$	fact_500k	124.65	166.55	139.70	135.67	116.23	115.80	116.90				130.79
	fact_100k	10.49	11.58	19.10	21.44	16.20	10.58	13.76	18.81	13.47	15.64	15.11
	fact_300k	49.83	51.26	45.23	46.85	43.98	44.74	47.44	44.90	44.54	46.49	46.53
	Mem_4k	7.95	8.68	8.80	7.86	8.02	7.76	9.58	7.97	8.15	8.23	8.23
	Mem_5k	14.13	16.86	14.93	17.42	18.03	11.20	10.37	10.32	12.28	11.97	13.75
Proposed $W_{CPU} < W_{MEM}$	fact_500k	121.12	126.29	125.52	123.28	119.64	123.60	123.73	-	-	-	123.28
	fact_100k	10.68	11.52	12.68	10.73	12.57	11.62	28.95	28.87	25.30	25.30	17.82
	fact_300k	43.33	43.36	40.63	39.58	42.76	41.54	41.21	43.07	43.44	38.75	41.77
	Mem_4k	8.10	7.81	8.40	8.46	8.28	8.15	7.94	8.56	8.11	8.04	8.18
	Mem_5k	12.43	10.15	9.91	9.76	11.17	12.84	10.81	10.27	11.42	9.27	10.80
Spread	fact_500k	234.44	230.93	251.08	246.21	227.47	277.72	233.15	-	-	-	243.00
	fact_100k	19.54	19.04	25.75	16.45	17.37	17.20	16.98	19.41	19.30	17.47	18.85
	fact_300k	90.72	94.23	82.33	88.72	81.10	86.17	83.51	91.76	82.94	83.31	86.48
	Mem_4k	16.31	24.48	22.00	22.57	16.31	20.22	19.02	19.29	17.52	16.57	19.43
	Mem_5k	23.88	17.84	22.80	23.95	19.56	16.61	20.12	21.68	15.89	15.77	19.81

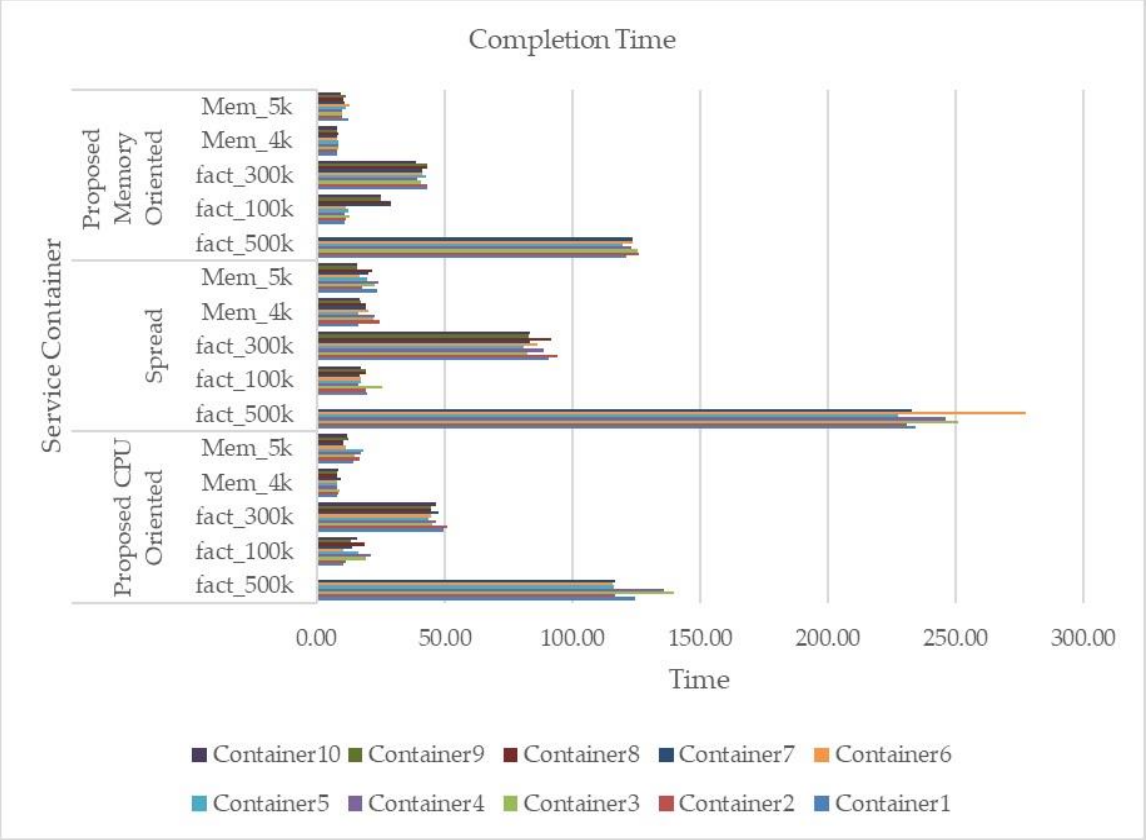

Figure 12 *Completion Time of task containers for Spread, Proposed (WCPU < WMEM, WCPU > WMEM)*

For visualizing the difference in completion time, the average completion time for the same services is calculated as displayed in Table 8 and plotted in Figure 13. It is shown that Proposed Weight resource Optimization approach is having almost same average completion time in both case (WCPU>WMEM and WCPU < WMEM), Whereas default Spread strategy is having higher average completion time for all services.

Table 8 *Average Completion Time of task containers for Spread, Proposed (WCPU < WMEM, WCPU > WMEM)*

Algo	fact_500k	fact_100k	fact_300k	Mem_4k	Mem_5k
Proposed W_{CPU} >W_{MEM}	123.6431	15.40614	46.52694	8.22835	13.75083
Proposed W_{CPU} <W_{MEM}	123.3108	17.8223	41.76793	8.184008	10.8016
Spread	243.0003	18.84889	86.47988	19.43128	19.80899

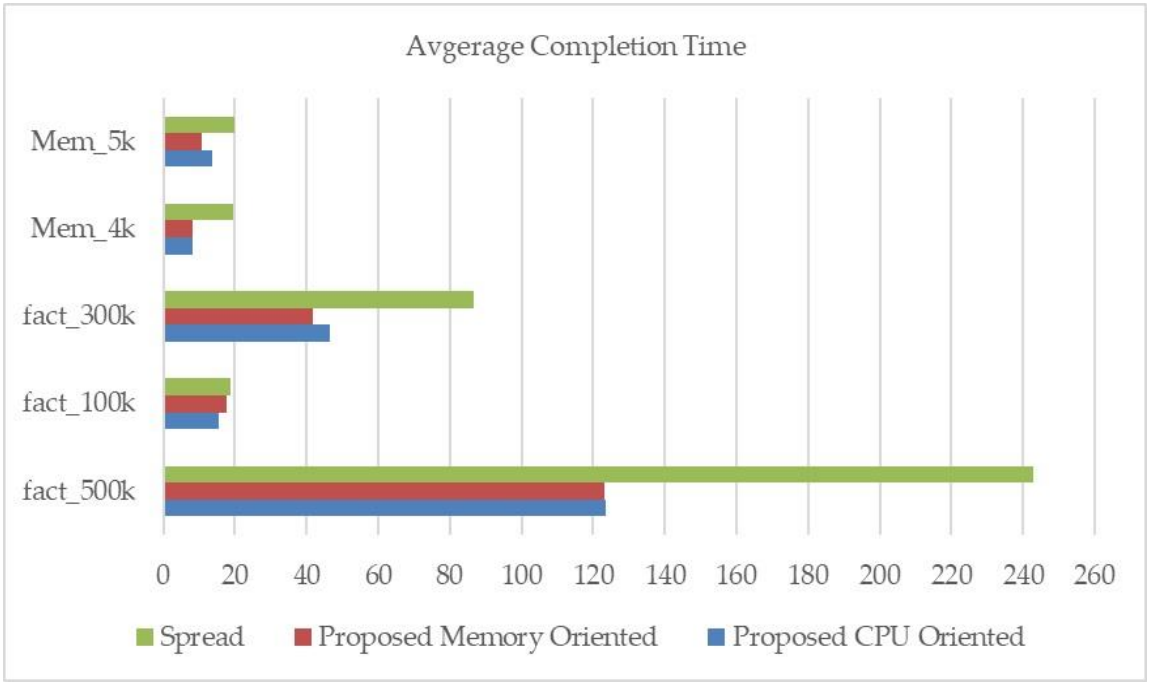

Figure 13 *Average Completion Time of task containers for Spread , Proposed (WCPU < WMEM , WCPU > WMEM)*

IV. Conclusion

This paper proposes a weighted resource optimization approach for finding a weighted score by using the available resources of nodes and the weight assigned to each resource. Resources currently considered are memory and CPU. Docker Swarm Cluster uses the default spread strategy for placing new containers on cluster nodes. This strategy tries to manage an equal number of containers on all nodes. but the spread strategy places the container without considering the resource requirements of task containers and the currently available resources of nodes. The proposed approach considers both conditions for calculating the score and places the container on the node with the highest score. The paper shows the improvement in load balancing among cluster nodes, and the completion time of task containers is also decreased. So overall performance is increased by the proposed approach.

References

[1] M. Moravcik and M. Kontsek (2020). Overview of Docker container orchestration tools. *18th* Slovenia,475-480.
[2] Potdar AM, Narayan DG, Kengond S, Mulla MM. Performance evaluation of docker container and virtual machine. Procedia Computer Science. 2020 Jan 1;171:1419-28.
[3] Swarmprom, 2021[Online]. Available https://github.com/stefanprodan/swarmprom
[4] Mao Y, Fu Y, Gu S, Vhaduri S, Cheng L, Liu Q. Resource management schemes for cloud-native platforms with computing containers of docker and kubernetes. arXiv preprint arXiv:2010.10350. 2020 Oct 20.
[5] Wu L, Xia H. Particle swarm optimization algorithm for container deployment. InJournal of Physics: Conference Series 2020 May 1 (Vol. 1544, No. 1, p. 012020). IOP Publishing.
[6] Wu G, Chen R, Zen D, Chen X. Using Ant Colony Algorithm on Scheduling Strategy Based on Docker Cloud Platform.

[7] Wu Y, Chen H. ABP scheduler: Speeding up service spread in docker swarm. In2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC) 2017 Dec 12 (pp. 691-698). IEEE.

[8] Hu Y, Zhou H, de Laat C, Zhao Z. Ecsched: Efficient container scheduling on heterogeneous clusters. InEuropean Conference on Parallel Processing 2018 Aug 1 (pp. 365-377). Cham: Springer International Publishing.

[9] Alzahrani EJ, Tari Z, Lee YC, Alsadie D, Zomaya AY. adCFS: Adaptive completely fair scheduling policy for containerised workflows systems. In2017 IEEE 16th International Symposium on Network Computing and Applications (NCA) 2017 Oct 30 (pp. 1-8). IEEE.

[10] Ghammam A, Ferreira T, Aljedaani W, Kessentini M, Husain A. Dynamic software containers workload balancing via many-objective search. IEEE Transactions on Services Computing. 2023 Feb 17;16(4):2575-91.

[11] Moravcik M, Kontsek M. Overview of Docker container orchestration tools. In2020 18th International Conference on Emerging eLearning Technologies and Applications (ICETA) 2020 Nov 12 (pp. 475-480). IEEE.

[12] Lyu T. *Evaluation of Containerized Simulation Software in Docker Swarm and Kubernetes* (Master's thesis).

[13] Pan Y, Chen I, Brasileiro F, Jayaputera G, Sinnott R. A performance comparison of cloud-based container orchestration tools. In2019 IEEE International Conference on Big Knowledge (ICBK) 2019 Nov 10 (pp. 191-198). IEEE.

[14] Raft consensus in swarm mode [Internet]. Docker Documentation. 2024 [cited 2024 Aug 21]. Available from: https://docs.docker.com/engine/swarm/raft/

[15] Swarm mode overview [Internet]. Docker Documentation. 2024 [cited 2024 Aug 21]. Available from: https://docs.docker.com/engine/swarm/