Volume 4, Issue 7, July 2014 ISSN: 2277 128X

International Journal of Advanced Research in Computer Science and Software Engineering

Research Paper

Available online at: www.ijarcsse.com

Mathematical modeling of load distribution problem in distributed computing environment: A State of Art

Ruchi Gupta[*]
OIMT, Rishikesh,
India

Dr. Pradeep Kumar Yadav
Building Research Institute, IIT, Roorkee,
India

Abstract—Load balancing is the process of improving the performance of a parallel and distributed system through a redistribution of load among the processors. In this paper we present the performance analysis of various load balancing algorithm based on different parameters, considering two typical load balancing approaches Mathematical technique and soft computing Technique. The analysis indicates that both types of algorithm can have advancements as well as weaknesses over each other. Load balancing is the process of distributing the load among various nodes of a distributed system to improve both resource utilization and job response time while also avoiding a situation where some of the nodes are heavily loaded while other nodes are idle or doing very little work. One of the biggest issues in such systems is the development of effective techniques/algorithms for the distribution of the processes/load of a parallel program on multiple hosts to achieve goal(s) such as minimizing execution time, minimizing communication delays, maximizing resource utilization and maximizing throughput, Reallocation Cost etc.

Keywords— Load Balancing, Static load balancing, dynamic load balancing, soft computing

I. INTRODUCTION

A distributed computing system (DCS) consists of a set of multiple processors interconnected by communication links. A very common interesting problem in DCS is the task allocation. This problem deals with finding an optimal allocation of tasks to the processors so that the system cost (i.e. the sum of execution cost and communication cost) is minimized without violating any of the system constraints [1]. In DCS, an allocation policy may be either static or dynamic, depending upon the time at which the allocation decisions are made. In a static task allocation, the information regarding the tasks and processor attributes is assumed to be known in advance, before the execution of the tasks [2]. Load balancing is an efficient strategy to improve throughput or speed up execution of the set of jobs while maintaining high processor utilization. The purpose of task allocation problem is to assign tasks to processors such that some objective function can be maximized or minimized subject to some constraints. . In general, load balancing provides parallel and distributed systems with an ability to avoid the situation where some resources of the systems are overloaded while others remain idle or under loaded. It is well understood that excessively overloading a portion of resources can substantially reduce the overall performance of the systems. In the last years, distributed computing systems have become a key platform for the execution of hydrogenous applications. The major problem encountered when programming such a system is the problem of tasks allocation. A best allocation of tasks leads to a best balancing of the system, [3] Load-balancing algorithm must deal with different unbalancing factors, according to the application and to the environment in which it is executed.[4] A distributed computing system (DCS) is defined as a computing system consisting of at least two autonomous processors connected by a network. Within a DCS, many processors may share resources; which includes files, printers and CPU's. Since many processes may try to access a particular resource or may demand multiple resources at the same time, access to the resources are scheduled to avoid conflict and to optimize the resource utilization for optimum performance.

Load balancing differs with properties of the tasks:

- **Tasks costs**
  - ✓ Do all tasks have equal costs?
  - ✓ If not, when are the costs known?
  - ✓ Before starting, when task created, or only when task ends

- **Task dependencies**
  - ✓ Can all tasks be run in any order (including parallel)?
  - ✓ If not, when are the dependencies known?
  - ✓ Before starting, when task created, or only when task ends

- **Locality**
  - ✓ Is it important for some tasks to be scheduled on the same processor (or nearby) to reduce communication cost?
  - ✓ When is the information about communication known?

Load balancing algorithms vary in their complexity where complexity is measured by the amount of communication used to approximate the least loaded node. Two broad categories of load balancing algorithms are commonly recognized. **Static algorithms** collect no information and make probabilistic balancing decisions, while **Dynamic algorithms** collect varying amounts of state information to make their decisions. The most significant parameter of the system was found to be the cost of transferring a job from one node to another. It is this cost that limits the dynamic algorithms, but at the high end of complexity are the dynamic algorithms which do collect varying amounts of information.

The four policies that govern the action of a load-balancing algorithm when a load imbalance is detected deal with information, transfer, location, and selection.

The **information Policy** is responsible for keeping up-to-date load information about each node in the system. A global information policy provides access to the load index of every node, at the cost of additional communication for maintaining accurate information. [5]

The **transfer policy** deals with the dynamic aspects of a system. It uses the nodes' load information to decide when a node becomes eligible to act as a sender (transfer a job to another node) or as a receiver (retrieve a job from another node). Transfer policies are typically threshold based. Thus, if the load at a node increases beyond a threshold Ts, the node becomes an eligible sender. Likewise, if the load at a node drops below a threshold, the node becomes an eligible receiver.

The **location policy** selects a partner node for a job transfer transaction. If the node is an eligible sender, the location policy seeks out a receiver node to receive the job selected by the selection policy. If the node is an eligible receiver, the location policy looks for an eligible sender node [6]. Once a node becomes an eligible sender, a **selection policy** is used to pick which of the queued jobs is to be transferred to the receiver node. The selection policy uses several criteria to evaluate the queued jobs. Its goal is to select a job that reduces the local load, incurs as little cost as possible in the transfer, and has good affinity to the node to which it is transferred. A common selection policy is latest-job arrived which selects the job which is currently in last place in the work queue [6].

Many heuristic strategies have been proposed for scheduling tasks for execution on the processors of a homogeneous multiprocessor [7] using task models that address a wide range of task granularities. In some models, a task refers to a single iteration of a parallel loop, while in

other models, a task may refer to a module in a single program or to separate programs run by one or more users. The simplest scheduling heuristic statically assigns tasks to processors in a round robin fashion such that processor 0 executes tasks 1, $p + 1$, $2p + 1$. . . Processor 1 executes tasks 2, $p + 2$, $2p + 2$. . . and so on, where p is the number of processors in the system. Static scheduling generates essentially no run-time overhead, but, since the tasks may have non-uniform execution times, static scheduling can severely unbalances the computational load on the processors leading to excessively long execution times.

The problem of task assignment in heterogeneous systems deals with finding proper assignment of tasks to processors in order to optimize some performance metric such as the system utilization and the turnaround time. We consider the task assignment problem with the following characteristics. The tasks are modeled using a task interaction graph (TIG). In the TIG model, the vertices of the graph correspond to the tasks and the edges correspond to the inter task communications. There is no precedence relation among tasks. In other words, processor p being faster than processor q on task i, e.g., *xip_xiq* , does not imply anything about their speeds for another task. In general, cost models are composed from constituent cost components that reflect the application activities. In these compositional models, cost components such as local disk I/O costs are modeled separately.[8] heterogeneous load balancing approaches attempt to boost the performance of heterogeneous clusters, which comprise a variety of nodes with different performance characteristics in computing power, memory capacity, and disk speed.

II. TASK ASSIGNMENT PROBLEM AND DEFINITION

The processors are heterogeneous, i.e., the execution cost of a task depends on the processor on which it is executed. Let P be the set of n processors in the heterogeneous computing system, T be the set of m tasks to be assigned to the processors, where it is assumed that M>N, assign (allocate) each of the M modules to one of the N processors in such a manner that the IPC time is minimized and the processing load is balanced. $ETC = \{xip\}$ m×n be the expected time to compute matrix where x_{ip} denotes the execution cost of task i on processor p, and $G = (T ,E)$ be the TIG, where E is the set of edges representing the communication between tasks. Each edge $(i, j) \in E$ is associated with a communication cost c_{ij}, which incurs only when tasks i and j are assigned to different processors. The processors are heterogeneous in the sense that there is no special structure in the ETC matrix.

Parameter of different load balancing algorithm

Execution cost:-

The execution cost $\mathbf{ec_{ij}}$ is the amount of the work to be performed by the executing task $\mathbf{t_i}$ on the processor $\mathbf{p_i}$ during the kth Phase. Where $1 \le i \le m$, $1 \le j \le n$ of each task $\mathbf{t_i}$ depends on the processor $\mathbf{p_j}$ to which it is assigned and the work to be performed by each of tasks of that processor $\mathbf{p_j}$. The processing execution cost of the tasks on all the processors is given in the form of Execution Cost Matrix (ECM) of order m x n. The Execution Cost of a given assignment on each processor is calculated by the following equation:

$$PEC(j) = \sum_{j=1}^{n} ec_{i,j} x_{i,j}\, , i = 1,2,3,4,5,6, \ldots \ldots \ldots . m.$$

Where x_{ij} is the

$$x_{ij} = \begin{cases} 1, \text{ if task } \mathbf{t_i} \text{ is assigned to processor } \mathbf{p_j} \\ 0, \text{ otherwise} \end{cases}$$

Inter process communication cost:-

The Inter Processor Communication cost $\mathbf{CC_{ik}}$ of the interacting tasks $\mathbf{t_i}$ and $\mathbf{t_k}$ is the minimum cost required for the exchange of data units between the processors during the process of execution. The edges that connect node indicate data exchange between modules. Each edge is labeled with a value to represent the communication cost for exchanging the data when the modules reside on different processors.

$$IPC(j) = \sum min(CC_{i,j}), \quad Where\ i = 1,2,3 \dots\dots m.$$
$$j = 1,2,3,4 \dots\dots\dots n.$$

Response time

The Response time (RT) of a task or thread is defined as the time elapsed between the dispatch (time when task is ready to execute) to the time when it finishes its job. Response time of the system is a function of amount computation to be performed by each processor and the computation time. This function is defined by considering the processor with the heaviest aggregate computation and communication load. Response time of the system for a given assignment is defined by

$$RT(A_{alloc}) = \max_{1 \le j \le n} \{PEC(A_{alloc})_j + IPC(A_{alloc})_j\}$$

Residence Cost

Residence cost $r_{ki\,j}$ (where $1 \le i \le M \& 1 \le j \le N$) is the amount of time spent by the residing task (other than executing task) t_i on the processor p_i in k_{th} phase. These costs may come from the use of storage.
Assign the residing tasks t_g ($g = 1, 2\dots M, g =/ s$) to processors p_h at which time is
Minimum say TRC (g) k
Calculate:

$$TRC_k = \sum_{\substack{i=1 \\ i \ne s}}^{M} TRC(i)_k$$

Reallocation cost

Reallocation cost rel_{ik} is the cost of reassigning the i_{th} task from one processor to another processor at the end of k_{th} phase. When an allocated task is shifted from one processor to another processor during the next phase then reallocation cost is mentioned at the end of each phase. an amount of relocation cost for reassigning each task from one processor to the others at the end of the phases.

Data Transfer Rate

Data Transfer Rate d_{ik} is per unit cost i.e. data exchanged between tasks t_i and t_k during the program execution. The speed with which data can be transmitted from one device to another.

$$\mathbf{DTR(P_i) = min(ec_{i,j}x_{i,j}) * d_{i,j}}$$

$$\mathbf{TDT(P_i) = DTR(P_i)\ where\ i = 1,2,3,4 \dots\dots m, and}$$
$$\mathbf{j = 1,2,3,4 \dots\dots\dots n.}$$

III. CATEGORIZATION OF DIFFERENT LOAD BALANCING ALGORITHMS

1. MATHEMATICAL BASE ALGORITHM

- **ADAPTIVE LOAD SHARING ALGORITHM**

In this algorithm load sharing attempt to improve the performance of a distributed system by using the processing power of the entire system to “smooth out” period of high congestion at individual nodes. The potential attractiveness of load sharing is enhance by factors such as the increasing size of locally distributed system, the use of shared file servers, the presence of pools of computation servers, and the development of streamlined communication protocols. [1]
Two important component of load sharing policy are:

- Transfer policy
  - ➢ Locally
  - ➢ Remotely
- Location policy
  - ➢ Threshold

- Random
- Shortest

Policies that use only information about the average behavior of the system, ignoring the current state, are termed static policies. Policies main concern on the following things

- Effect of overhead
- Effect of the occasional poor decisions that inevitably will be made.
- Potential for instability.

Three policy introduced in this paper explain the new task arrive at each node at average rate λ. the average task services time (processing cost) is S. we define load factor ρ of each node to be the ratio of offered load to services.

$$\rho = \lambda S$$

Because of the task transfer, the average utilization of the nodes may be significantly greater than ρ.

Cost of task =Avg. utilization > ρ.

The cost of transferring a task from one node to another is represented by a processing cost S at the sending node whose avg. value is denoted by C(Communication network cost) .In the processing of task, any services discipline that selects task in a way that is independent of their actual service time is allowed.

In each state the arrive los the task is the sum of the rate of arrival os task transferred to this node by the remainder of the system ($\lambda_t(n)$).

Total arrival rate at the node =($\lambda + \lambda_t(T^+)$)

Where $\lambda_t(T^+)$=arrival rate of the task transferred to the node.

The random policy substantial further performance over no loads sharing. The threshold yield substantial performance improvement for the system load greater than 0.5.the shortest neliglever the neligible policy performance improvement over the threshold policy. During preceding phase the node is either idle or is processing task. During transferring phase the node is busy, either transferring task or processing task that could be transferred because of a restriction imposed by the location policy.

- **HETEROGENEOUS MAXIMALLY-LINKED MODULES ALGORITHM**

This paper presents a new heuristic model, the *HMLM/SA*, which performs static allocation of such program modules in a heterogeneous distributed computing system in a manner that is designed to minimize the application program's parallel execution time. The new methodology augments the Maximally Linked Module concept by using stochastic techniques and by adding constructs which take into account the limited and uneven distribution of hardware resources often associated with heterogeneous systems. The execution time of the resulting *HMLM/SA* algorithm and the quality of the allocations produced are shown to be superior to that of the base *HMLM* algorithm, pure simulated annealing and the randomized algorithm when they were applied to randomly-generated systems and synthetic structures which were derived from real-world problems. The method comprises execution cost of the subtask & IPC cost which arise due to the interacting modules residing on different processor. [11]

First they calculate the Avg load to be assigned to each processor and assign initial modules to each processor. Then compute the processing loss & IPC cost for each processor. For a processor Pi, consider an unassigned module Mj inter module communications with the module cluster. If the total cost decrease assign inter module communication to Processor. This algo tries to minimize the processing loss by balancing the load and, at the same time, it also optimize inter processor communications cost.

The specific problem being addressed is as follows: Given application software that consists of *M* communicating modules (tasks), *m1, m2, ..., mM,* and a heterogeneous distributed computing system with *N* processors, *p1, p2, ..., pN*, where it is assumed that *M>>N*, assign (allocate) each of the *M* modules to one of the *N* processors in such a manner that the *IPC* time is minimized and the processing load is balanced. This is accomplished in this research by maximizing the energy function which represents the projected speedup. The following notation will be used throughout this text.

The task assignment vector *A* is defined to be $A{:}M \rightarrow P$, where $A(i) = j$ if module m_i is assigned to processor pj, $1 \le i \le M$, $1 \le j \le N$. For a certain assignment, the task set *(TSj)* of a processor *j* can now be defined as the set of tasks allocated to that processor [9, 10]:

$$TSj = \{ i \mid A(i)=j \} \quad j=1, \ldots, N \quad (1)$$

The communication set *(CS)* of a processor is the set of edges on the program graph that go between the given processor and another processor, for a given assignment *A*.

$$CSj = \{ (x,y) \mid (A(x) = j \wedge A(y)^1 \neq j \} \; j=1, \ldots, N \quad (2)$$

The communication load of a processor *j* is the sum of the values of the edges in the communication set, that is, Each element, $e_{i,j}$, depends upon the amount of computation to be performed by the module m_i as well as on the specific attributes of the processor p_j. The overall Actual Execution Time *(AET)* of a given assignment *A* is then

$$AET(A) = \sum_{1 \le i \le M} e_i \, A(i)$$

And the per processor Actual Execution Time for processor p_j is defined to be

$$AET(A) = \sum_{1 \le j \le M} e_i A(i)$$

The M xM matrix *C* is used to represent the communication time (cost) between modules in a program, where $C_{i,j}$=g *>0* if module m_i communicates with module m_j for some cost *g* when $A(i) \neq A(j)$. That is, any two modules that communicate during their execution incur a penalty if they are executed on different processors, otherwise $C_{i,j} = 0$. The overall Inter processor Communication time *(IPC)* of a given assignment *A* can be expressed by

$$IPC(A) = \sum_{\substack{1 \le i \le M \\ I+1 \le j \le M \\ A(i) \neq A(j)}} C_{A(i),A(j)}$$

And the per processor Inter processor Communication time is given by

$$IPC(A) = \sum_{\substack{1 \le k \le M \\ I+1 \le k \le M \\ A(i)-j \neq A(j)}} C_{A(i),A(j)}$$

We presume that the data about the *IPC* times are somehow available and that all the times (execution and *IPC*) are expressible in some common unit of measurement. When applied to a large number of randomly-generated systems and to three real world task structures, the methodology creates allocations which appear very competitive to those produced by other allocation methodologies.

- **NEAREST-NEIGHBOR LOAD BALANCING**

Nearest-neighbour load balancing methods rely on successive approximations to a global optimal workload distribution, and hence at each operation, need only to concern with the direction of workload migration.[12] With nearest neighbor algorithm each processor considers only its immediate neighbour processors to perform load balancing operations. A processor takes the balancing decision depending on the load it has and the load information to its immediate neighbours. By exchanging the load successively to the neighbouring nodes the system attains a global balanced load state. The nearest neighbor algorithm is mainly divided into two categories which are diffusion method and dimension exchange method. With this method a heavily or lightly loaded processor balances its load simultaneously with all its nearest neighbours at a time while in dimension exchange method a processor balances its load successively with its neighbor one at a time.

With nearest neighbor algorithm each processor considers only its immediate neighbor processors to perform load balancing operations. A processor takes the balancing decision depending on the load it has and the load information to its immediate neighbors. By exchanging the load successively to the neighboring nodes the system attains a global balanced load state. The nearest neighbor algorithm is mainly divided into two categories which are diffusion method and dimension exchange method. With this method a heavily or lightly loaded processor balances its load simultaneously with all its nearest neighbors at a time while in dimension exchange method a processor balances its load successively with its neighbor one at a time

- **CLUSTERING WITH INTER-PROCESSOR DISTANCES IN HETEROGENEOUS DISTRIBUTED COMPUTING SYSTEMS**

In this paper a mathematical model for finding optimal cost and optimal reliability to the problem is presented considering DCS with heterogeneous processors in such a way that the allocated load on each processor is balanced. The results obtained by the present model are compared with the recent models and comparison results show that the model is very effective. One approach to this problem is to use distributed computing systems (DCS) that concurrently process an application program by employing multiple processors. In DCS homogeneous or heterogeneous processors are connected together through a communication network. Distributed computing provides the capability for the utilization of remote computing resources and allow for increased levels of flexibility, reliability and modularity. The module allocation in a distributed processing system finds extensive application in the faculties where large amount of data is to be processed in a short period of time or where real time computations are required. The main incentives for choosing DCS are higher throughput improved availability and better access to a widely communicated web of information. In general the objective of modules allocation is to find an optimal allocation AO, which minimize the completion time of the program and optimize the system reliability by properly mapping the modules to the processors. In order to make the best use of the resources in a distributed computing system we would like to distribute the load on each processor in such a way that allocated load on the processors are balanced.

The execution cost eij (1£ i £ m , 1£ j£ n) of each module Mi depends on the capability of the processor Pj to which it is assigned and the work to be performed by each module.

$$PEC(j) = \sum_{\substack{1 \le i \le M \\ i \in MSj}} e_i A(i)$$

Where MSj= {i: A(i) =j, j=1, 2…n} Overall execution cost of a given allocation A, is calculated as:

$$EC(j) = \sum_{1 \le i \le M} e_i \, A(i)$$

In this paper, the matrix DT= [$\text{data}_{i,j}$] of order m x m is used to represent the data communication between the modules during the execution, where datai,j is the amount of data required to be transmitted from Module Mi to module Mj. [18] The data transfer rates, i.e. bandwidth size, (in bytes/seconds) between processors are stored in a matrix DTR= [$r_{i,j}$] of size n x n. if two interacting modules Mi and Mj are assigned to two different processors Pk and Ps respectively, then the two modules cause the inter-processor communication cost of $C_{i,j}$*d_k.

$$C_{i,j} = \begin{cases} \bar{L} + \dfrac{Data_{i,j} \quad \text{if } i \neq j}{\overline{DTR}} \\ 0 \qquad if \; i = j \end{cases}$$

The overall inter-module communication cost and inter-module communication cost for each processor for a given allocation A, are obtained by equations and respectively as:

$$\mathbf{TCC(A)} = \sum_{\substack{1 \le i \le M \\ I+1 \le j \le M \\ A(i) \neq A(J)}} \mathbf{C_{i,j} * d_{A(i),A(j)}}$$

$$\mathbf{IPCC(j)} = \sum_{\substack{1 \le i \le M \\ A(i) = j \neq A(k)}} \mathbf{C_{i,k} * d_{A(i),A(k)}}$$

Response (turnout) time of the system is a function of amount computation to be performed by each processor and the computation time. [18]

$$RT(A) = \max_{1 \le j \le n} \{PEC(j) + IPCC(j)\}$$

Thus the objective function for the response time of the system is expressed as:

$$Min\ Z = \min \ \{RT(A)\}$$
$$= min\{\max_{1 \le j \le n}\{\textstyle\sum_{i=1}^{m} e_{i,j} X_{i,j} + \sum_{i=1}^{m} \sum_{k=1}^{m} C_{i,k} * \ d_{j,1} X_{i,j} X_{k,1}\}\}$$

The run time complexity of the algorithm presented in this paper is O (m2+mn) which is less than the complexities of the models. [18]

- **MUTUALLY OVERLAPPING SUBSET USING MAEKAWA'S ALGORITHM**

In this algorithm nodes are scheduled independently and asynchronously with distinct execution initiation times corresponding to their earliest instant of being overloaded.[6] The technique handles the task of resource management by dividing the nodes of the system into mutually overlapping subsets and thereby a node gets the system state information by querying only a few nodes.[13] The approach is primarily targeted at systems that are composed of general purpose workstation computers having identical processors. Process scheduling decisions are driven by the desire to minimize turnaround time while maintaining fairness among competing applications and minimizing communication overhead. The performance analysis of the technique shows that it significantly reduces the total number of messages required for a node to take scheduling decision.
In this technique novel workload migration technique proposed by considering the homogeneous and dynamic features of distributed computing environments and dynamic and stable technique that satisfies the quick decision making capability and provides a balanced system performance with respect to scheduling overhead and the experimental results show that the proposed migration scheme outperforms common-used schemes with respect to reducing the communication cost and the application execution time.
All nodes in the system are assigned unique identification numbers from 1 to N. All the nodes in the system are fully connected. The method used as the Load Estimation policy would be the measure of CPU utilization of the nodes [12]. CPU utilization is defined as the number of CPU cycles actually executed per unit time. Process transfer policy that determines whether to execute a process locally or remotely is implemented by the double threshold policy by Alonso and Cova [12]. The node sends its state information to other nodes only when its state switches from normal load region to either overload or under load region.

- **TASK ASSIGNMENT USING THE A* ALGORITHM**

A distributed system comprising networked heterogeneous processors requires efficient task-to-processor assignment to achieve fast turnaround time. The authors propose two algorithms based on the A* technique, which are considerably faster, are more memory-efficient, and give optimal solutions. The first is a sequential algorithm that reduces the search space. The second proposes to lower time complexity, by running the assignment algorithm in parallel, and achieves significant speedup. This problem, *task assignment*, is well-known to be NP-hard in most cases. A task-assignment algorithm seeks an assignment that optimizes a certain cost function—for example, maximum throughput or minimum turnaround time. However, most reported algorithms yield suboptimal solutions. In general, optimal solutions can be found through an exhaustive search, but because there are *nm* ways in which m tasks can be assigned to *n* processors, an exhaustive search is often not possible. The A* algorithm, an informed-search algorithm, guarantees an optimal solution, but doesn't work for large problems because of its high time and space complexity. [14]

We represent the interconnection network of n processors, {p1, p2, ..., pn}, by an n´n link matrix L, where an entry Lij is 1, if processors i and j are directly connected, and 0 otherwise. We do not consider the case where i and j are not directly connected.

We can execute a task ti from the set VT on any one of the system's n processors. Each task has an associated execution cost on a given processor. A matrix X gives task execution costs, where Xip is the execution cost of task i on processor p. Two tasks, ti and tj, executing on two different processors, incur a communications cost when they need to exchange data. Task mapping will assign two communicating tasks to the same processor or to two different, directly connected processors. A matrix C represents communication among tasks, where Cij is the communication cost between tasks i and j, if they reside on two different processors. A processor's load comprises all the execution and communication costs associated with its assigned tasks. The time needed by the heaviest-loaded processor will determine the entire program's completion time. The task-assignment problem must find a mapping of the set of *m* tasks to *n* processors that will minimize program completion time. Task mapping, or assignment to processors, is given by a matrix **A**, where **A***ip* is 1, if task *i* is assigned to processor *p*, and 0 otherwise. The following equation then gives the load on *p*:

$$\sum_{i=1}^{m} X_{ip}A_{ip} + \sum_{\substack{q=1\\(q\neq p)}}^{m} \sum_{i=1}^{m}\sum_{j=1}^{m}\left(C_{i,j}A_{i,p}A_{j,p}L_{p,q}\right)$$

For the task-assignment problem under consideration,

- The search space is a tree;
- The initial node (the root) is a null assignment node—that is, no task is assigned as yet;
- Intermediate nodes are partial-assignment nodes—that is, only some tasks are assigned; and
- a solution (goal) node is a complete assignment node—that is, all the tasks are assigned.

To compute the cost function, g(n) is the cost of partial assignment (A) at node n— the load on the heaviest-loaded (p); this can be done using the equation from the previous section. For the computation of h(n), two sets Tp (the set of tasks that are assigned to the heaviest-loaded p) and U (the set of tasks that are unassigned at this stage of the search and have one or more communication link with any task in set Tp) are defined. Each task ti in U will be assigned either to p or any other processor q that has a direct communication link with p. So, you can associate two kinds of costs with each ti's assignment: either Xip (the execution cost of ti on p) or the sum of the communication costs of all the tasks in set Tp that have a link with ti. This implies that to consider ti's assignment, we must decide whether ti should go to p or not (by taking these two cases' minimum cost). Let cost(ti) be the minimum of these two costs, then we compute h(n) as:

$$h(n) = \sum_{t_i \in U} Cost(t_i)$$

In the first algorithm in the paper, Optimal Assignment with Sequential Search (OASS) algorithm uses the A* search technique, but with two distinct features. First, it generates a random solution and prunes all the nodes with costs higher than this solution during the optimal-solution search. This is because the optimal solution cost will never be higher than this random-solution cost. Pruning unnecessary nodes not only saves memory, but also saves the time required to insert the nodes into OPEN. Second, the algorithm sets the value of *f*(*n*) equal to *g*(*n*) for all leaf nodes, because for a leaf node *n*, *h*(*n*) is equal to 0. This avoids the unnecessary computation of *h*(*n*) at the leaf nodes. The parallel algorithm aims to speed up the search as much as possible using parallel processing. This is done by dividing the search tree among the *processing elements* (PEs) as evenly as possible and by avoiding the expansions of nonessential nodes—that is, nodes that the sequential algorithm does not expand.

2. SOFT COMPUTING ALGORITHM

• PARALLEL GENETIC ALGORITHM

The proposed algorithm uses multiple processors with centralized control for scheduling. Tasks are taken as batches and are scheduled to minimize the execution time and balance the loads of the processors. Genetic algorithms (GAs) are a Meta heuristic searching techniques which mimics the principles of evolution and natural genetics. These are a guided random search which scans through the entire sample space and therefore provide reasonable solutions in all situations. Genetic algorithms (GAs) are a Meta heuristic searching techniques which mimics the principles of evolution and natural genetics. These are a guided random search which scans through the entire sample space and therefore provide reasonable solutions in all situations [15]. In most parallel algorithms, the basic idea behind the algorithm is to divide the task into subtasks and use different processors to execute each subtask. This divide and-conquer approach can be applied to GAs in many different ways, and the literature contains many examples of successful parallel implementations.

A *genetic algorithm* (GA) is a heuristic used to find a vector *x* * & (a string) of free parameters with values in an admissible region for which an arbitrary quality criterion is optimized:

***f* (*x*) →*max* : *find an x * such that* ∀¥ *x* € *M* : *f* (*x*) ≤ *f* (*x**) = *f* * (1)**

A sequential GA proceeds in an iterative manner by generating new populations of strings from the old ones. Every string is the encoded (binary, real ...) version of a tentative solution. An evaluation function associates a fitness measure to every string indicating its suitability to the problem. The algorithm applies stochastic operators such as selection, crossover and mutation on an initially random population in order to compute a whole generation of new strings. There is a single panmictic population, but the evaluation of the fitness function is distributed among several processors.
Initial number of 'guesses' is made to get an optimal solution (the initial population). Each guess is evaluated and assigned a 'goodness' value (the fitness function). Those guesses with good values are selected and new guesses are made by combining the existing guesses in a particular fashion (crossover). The guesses are evolved to the next generation on a survival of the fittest basis (selecting), thereby the 'good' guesses are forwarded to the next generation and the 'bad' guesses are not. The proposed parallel genetic algorithm involves a master scheduler, which has the processor lists and the task queue. Tasks are indivisible, independent of all other tasks, arrive randomly, and can be processed by any processor in the distributed system. The master scheduler runs a sequential GA in which the fitness function evaluation alone is done by slave processors. When tasks arrive they are placed in the unscheduled task queue. They tasks are taken in batches and scheduled. Batch schedulers are shown to have higher performance than immediate schedulers in [16]. When any processor is idle, the processor asks for a task to perform and the task scheduled for that processor (if any) is given to that processor. All the task data are maintained only in the Synchronous master slave parallelization is used to evaluate the fitness function alone in a distributed fashion.
These are the steps in parallelization,

- A **master** scheduler which is the processor in charge of scheduling **chooses the slaves**. This choice is based upon the communication overhead involved and the computational potential of the slave processor.

- The **master has the population** of chromosomes for which the fitness function is to be evaluated.

- Each **slave evaluates the fitness** of a fraction (Fi) of the population in the master scheduler and returns the value. Load for each processor is considered by calculating the finishing time of a processor j. δi = (Aj/Pj), where Aj denotes the previously assigned load, measured in MFLOPs, and Pj is the current processing power in Mflop/s of processor j. The current load of each processor is calculated,

$$L_j = \left[\left(\sum_{i=1}^{n} t_i\right) / p_j\right] + \delta_j + \sum_{i=1}^{n} \Gamma_C(i,j)$$

Where ti is the processing requirements of task i in the batch (in MFLOPs) and n is the total number of tasks Assigned to the processor j. $\acute{\Gamma}_C(i,j)$ is the communication time for the ith task in the jth processor.
The mean value of Lj for all the processors is given as

$$\mu = \sum_{j=1}^{M} Lj/M$$

The relative load imbalance error of individual i is given as

$$Ei = \sqrt{\sum_{j=1}^{M} |\mu - Lj|^2}$$

This Error value denotes how unbalanced the schedule is. For instance, if all the tasks are assigned to only one processor while the others are idle, this error value will be very large. Minimizing the error value ensures that schedules which utilize more processors at the same time (increase parallelization) will be consider fitter schedules.
The fitness value of individual i is

Fi = ((max – max *span*$_i$) + 0.001*max)/E$_i$

- The **communication time** $\acute{\Gamma}C(i,j)$ is calculates as follows :

$$\acute{\Gamma}_c = \alpha * T(i-1,j) + (1-\alpha) * [\tau_c(i,j) * S(i)]$$

Where

α = factor.
τ_c = predicted communication cost.
T = time taken in second for execution of task.

- They using stochastic sampling with partial replacement selection is totally based on fitness function. In a standard weighted roulette wheel selection algorithm, the selection is totally based on the fitness function. The chromosomes are assigned slots in the roulette wheel based on their relative fitness function values. The roulette wheel is spun N times to select N chromosomes. Stochastic sampling with partial replacement selection is a simple extension of the roulette wheel selection in which the sector assigned for a particular chromosome is reduced if the chosen chromosome has a fitness value less than the average fitness value.

- **MINIMIZING COST OF DISTRIBUTED COMPUTING SYSTEMS USING GENETIC ALGORITHMS**

In this paper we present a genetic algorithm, considering DCS with heterogeneous processors in order to achieve optimal cost by allocating the tasks to the processors, in such a way that the allocated load on each processor is balanced. The algorithm is based on the execution costs of a task running on different processors and the task communication cost between two tasks to obtain the optimal solution. The execution costs of a task running on different processors are different and it is given in the form of a matrix of order m × n, named as execution cost matrix ECM. Similarly, the inter task communication cost between two tasks is given in the form of a symmetric matrix named as inter task communication cost matrix ITCCM, of order m × m. To solve the problem of task allocation in DCS via GAs, it is necessary to find a mapping of a potential candidate for a solution onto a sequence of binary digits, the so called chromosome. They consider the four components:

- ✓ An encoding method that is a genetic representation (genotype) of solutions to the program. The length of the chromosomes is given by the number of tasks that should be allocated. Every gene in the chromosome represents the processor where the task is running on.
- ✓ A way to create an initial population of chromosomes.
- ✓ The objective function is find a task allocation X such that the overall system cost is minimized.

  Min {C(X)=PEC(X)+IPEC(X)}
  Where
  PEC = *Execution Cost* of all processors.
  IPEC = inter processor communication cost for all processors

- The genetic operators (crossover and mutation) that alter the genetic composition of offspring during reproduction. The crossover operation will perform if the crossover ratio (Pc>=0.95) is verified. The cut point is selected randomly. The crossover operation is performed as Select two chromosomes randomly from the current population.[17] Randomly select the cut point. Fill the components of the chromosome. By taking the components of the first chromosome (from the first gene to the cut point) and fill up to the child. Also, tacking the components of the second chromosome (from the cut point+1 to the last gene) and fill up to the child. The mutation operation is performed on bit-by-bit basis. The mutation operation will perform if the mutation ratio (Pm) is verified. The mutation ratio, Pm in is estimated randomly. The point to be mutated is selected randomly.

- **HEURISTICS BASED GENETIC ALGORITHM**

This paper we present a genetic algorithm, the provide the systematic scheduling the problem of same execution time or completion time and same precedence in the homogeneous parallel system is resolved by using concept of Bottom-level (b-level) or Top-level (t-level). The assumption of this paper is based on the deterministic model, that is, the number of processors, the execution time of tasks, the relationship among tasks and precedence constraints are known in advance. In addition, the communication cost between two tasks is considered to be non-negligible and the multiprocessor system is not diverse and non-preemptive, that is, the processors are homogeneous, and each processor completes the current task before the new one starts its execution. The main objective is to minimize the total task completion time (execution time + waiting time or idle time).The consider a directed acyclic task graph *G* = {*V*,*E*} of *n* nodes. Each node V = {*T*1, *T2*,...., *Tn*} in the graph represents a task. Aim is to map every task to a set *P* = {*P*1,*P*2, . . . , *Pm*} of *m* processors. Each task *Ti* has a weight *Wi* associated with it, which is the amount of time the task takes to execute on any one of the m homogeneous processors. Each directed edge *eij* indicates dependence between the two tasks *Ti* and *Tj* that it connects. If there is a path from node *Ti* to node *Tj* in the graph *G*, then *Ti* is the predecessor of *Tj* and *Tj* is the successor of *Ti*. The successor task cannot be executed before all its predecessors have been executed and their results are available at the processor at which the successor is scheduled to execute. A task is "ready" to execute on a processor if all of its predecessors have completed execution and their results are available at the processor on which the task is scheduled to execute. If the next task to be executed on a processor is not yet ready, the processor remains idle until the task is ready.
Communication costs while calculating values

$$\text{tlevel}(Ti) = \max_{Tj \in \text{pred}(Ti)} \{\text{tlevel}(Tj) + Wj + cji\}$$

$$\text{blevel}(Ti) = Wi + \max_{Ti \in \text{succ}(Ti)} \{cij + \text{blevel}(Tj)\}$$

Minimum Execution Time (MET) assigns each task, in arbitrary order, to the machine with the best expected execution time for that task, regardless of that machine's availability. Min-min heuristic uses minimum completion time (MCT) as a metric, meaning that the task which can be completed the earliest is given priority. [14] Each list corresponds to computational tasks executed on a processor and order of tasks in the list indicates the order of execution. The next step in the GAs is the creation of the initial population. Number of processors, number of tasks and population size are needed to generate initial population. Each individual of the initial population is generated through a minimum execution time or min-min heuristic along with b-level or t-level precedence resolution to avoid the problem of same execution time or completion time and same precedence.

The task to be scheduled for each iteration is determined by the following rules:

i. Sort the tasks according to their execution time/completion time in ascending order according to the minimum execution time (MET)/Min-Min heuristic.
ii. Calculate the top-level of each task.
iii. Sort the tasks with the same execution time/completion time and same precedence according to their top-level in ascending order.
iv. Assign the tasks to the processors in the order of their top-level.

The elementary criterion is that of minimizing the *makespan*, that is, the time when finishes the latest job. A secondary criterion is to minimize the *flowtime* that is, minimizing the sum of finalization times of all the jobs. These two criteria are defined as follows:

$$\text{makespan}: \min_{S_i \in Sched}\{\max_{j \in Jobs} F_j\}$$
$$\text{flowtime}: \min_{S_i \in Sched}\{\sum_{j \in Jobs} F_j\}$$

This operator generate next generation by selecting best chromosomes from parents and offspring. Crossover operator randomly selects two parent chromosomes (chromosomes with higher values have more chance to be selected) and randomly chooses their crossover points, and mates them to produce two child (offspring) chromosomes.[14] Mutation ensures that the probability of finding the optimal solution is never zero. It also acts as a safety net to recover good genetic material that may be lost through selection and crossover. Implementation of two mutation operators is there in HGA. The first one selects two tasks randomly and swaps their allocation parts. [14] The second one selects a task and alters its allocation part at random.

The procedure of the Suggested Heuristics based Genetic Algorithm is:

```
Step 1: Setting the parameter
Set the parameter: Read DAG (number of tasks n, number of processors m and comm.
cost), population size pop_size, crossover probability pc, mutation probability pm, and
Maximum generation maxgen.
Let generation $gen$ = 0.
Step 2: Initialization
Generate pop_ size chromosomes using minimum execution time (MET)/Min-Min
heuristic and b-level/t-level precedence resolutions.
Step 3: Evaluate
Calculate the fitness value of each chromosome
Step 4: Crossover
Perform the crossover operation on the chromosomes selected with probability pc.
Step 5: Mutation
Perform the swap/move mutation on chromosomes selected with probability pm.
Step 6: Selection
Select pop_size chromosomes from the parents and offspring for the next generation.
Step 7: Stop testing
If $gen = maxgen$, then output best solution and stop
Else $gen = gen + 1$ and return to step 3
```

In this paper, a new genetic algorithm, named Heuristics based Genetic Algorithm for Scheduling Static Tasks in Homogeneous parallel System is presented which its population size
and the number of generations depends on the number of tasks. This algorithm tends to minimize
the completion time and increase the throughput of the system.

- **A GENETIC ALGORITHM FOR PROCESS SCHEDULING IN DISTRIBUTED OPERATING SYSTEMS CONSIDERING LOAD BALANCING**

In this paper, using the power of genetic algorithms, they evaluate the performance and efficiency of the proposed algorithm using simulation results. The proposed algorithm maps each schedule with a chromosome that shows the execution order of all existing processes on processors. The fittest chromosomes are selected to reproduce offspring; chromosomes which their corresponding schedules have less total execution time, better load-balance and processor utilization. We assume that the distributed system is non-uniform and non-preemptive, that is, the processors may be different, and a processor completes current process before executing a new one.

- ✓ The processor load for each processor is the sum of processes execution times allocated to that processor. However, as the processors may not always be idle when a chromosome (schedule) is evaluated, the current existing load on individual processors must also be taken into account.

- ✓ The length or *maxspan* of a schedule T is the maximal finishing time of all processes or maximum load. Also communication cost (CC) to spread recently created processes on processors must be computed.
- ✓ The Processor utilization for each processor is obtained by dividing the sum of processing times by maxspan, and the average of processors utilization is obtained by dividing the sum of all utilizations by number of processors.
- ✓ We must define thresholds for light and heavy load on processors. If the processes completion time of a processor (by adding the current system load and those contributed by the new processes) is within the light and heavy thresholds, this processor queue will be acceptable. If it is above the heavy threshold or below the light-threshold, then it is unacceptable, but what is important is average of number of acceptable processors queues.

- **MULTI-HEURISTIC DYNAMIC TASK ALLOCATION USING GENETIC ALGORITHMS IN A HETEROGENEOUS DISTRIBUTED SYSTEM**

In this paper a scheduling strategy is presented that uses a GA to schedule a set of heterogeneous tasks on to a set of heterogeneous processors in an effort to minimize the total execution time. It operates dynamically, allowing for tasks to arrive for processing continuously, and considers variable system resources, which has not been considered by other dynamic GA schedulers. To allow for efficient schedules to be produced quickly, the scheduler utilizes 8 heuristics, reducing the probability of processors becoming idle while waiting for a schedule to be generated. The scheduler has been implemented on a real-world distributed system and tested on 150 non-dedicated heterogeneous processors, with a variety of real-world problems from bioinformatics, biomedical engineering, computer science and cryptography.
The set of processors of the distributed system is heterogeneous. The available network resources between processors in the distributed system can vary over time. The availability of each processor can vary over time (processors are non-dedicated). Tasks are indivisible, independent of all other tasks, arrive randomly, and can be processed by any processor in the distributed system. When tasks arrive they are placed in a queue of unscheduled tasks. Batches of tasks from this queue are scheduled on processors during each invocation of the scheduler.
The queue of unscheduled tasks can contain a large number of tasks. If all of these task where to be scheduled at once, the scheduler could take a long time to find an efficient schedule. To reduce the execution time of the scheduler and reduce the chance of processors becoming idle, we only consider a subset of the unscheduled tasks, which we call a batch. A larger batch will usually result in a more efficient schedule, but will incur a longer running time. To do this we dynamically set the batch size according to the estimated amount of time until the first processor becomes idle.

IV. COMPRESSION PARAMETER OF DIFFERENT LOAD BALANCING ALGORITHM

PARAMETERS: The performance of various load balancing algorithms is measured by the following parameters.
Nature: - This factor is related with determining the nature or behaviour of load balancing algorithms that is whether the load balancing algorithm is of static or dynamic nature, pre-planned or no planning.

Overload Rejection: - If Load Balancing is not possible additional overload, rejection measures are needed. When the overload situation ends then first the overload rejection measures are stopped. After a short guard period Load Balancing is also closed down. Static load balancing algorithms incurs lesser overhead as once tasks are assigned to processors, no redistribution of tasks takes place, so no relocation overhead. Dynamic Load Balancing algorithms incur more overhead relatively as relocation of tasks takes place.

Reliability:- This factor is related with the reliability of algorithms in case of some machine failure occurs. Static load balancing algorithms are less reliable because no task/process will be transferred to another host in case a machine fails at run-time. Dynamic load balancing algorithms are more reliable as processes can be transferred to other machine in case of failure occurs.

Adaptability: - This factor is used to check whether the algorithm is adaptive to varying or changing situations i.e. situations which are of dynamic nature. Static load balancing algorithms are not adaptive as this method fails in varying nature problems i.e. situation in which number of processes are not fixed. Dynamic load balancing algorithms are adaptive towards every situation whether numbers of processes are fixed or varying one.

Stability: - Stability can be characterized in terms of the delays in the transfer of information between processors and the gains in the load balancing algorithm by obtaining faster performance by a specified amount of time. Static load balancing algorithm considered as stable as no information regarding present workload state is passed among processors. However in case of dynamic load balancing such kind of information is exchanged among processors.

Predictability:- This factor is related with the deterministic or nondeterministic factor that is to predict the outcome of the algorithm. Static load balancing algorithm's behaviour is predictable as most of the things like average execution time of processes and workload assignment to processors are fixed at compile-time. Dynamic load balancing algorithm's behaviour is unpredictable, as everything has been done at run time.

Forecasting Accuracy: - Forecasting is the degree of conformity of calculated results to its actual value that will be generated after execution.

Cooperative: - This parameter gives that whether processors share information between them in making the process allocation decision other are not during execution. What this parameter defines is the extent of independence that each processor has in concluding that how should it can use its own resources. In the cooperative situation all processors have the accountability to carry out its own portion of the scheduling task, but all processors work together to achieve a goal of better efficiency. In the non-cooperative individual processors act as independent entities and arrive at decisions about the use of their resources without any effect of their decision on the rest of the system.

Fault Tolerant:- It enables an algorithm to continue operating properly in the event of some failure. If the performance of algorithm decreases, the decrease is proportional to the seriousness of the failure, even a small failure can cause total failure in load balancing.

Resource Utilization:- Resource utilization include automatic load balancing A distributed system may have unexpected number of processes that demand more processing power. If the algorithm is capable to utilize resources, they can be moved to under loaded processors more efficiently.
Static load balancing algorithms have lesser resource utilization as static load balancing methods just tries to assign tasks to processors in order to achieve minimize response time ignoring the fact that may be using this task assignment can result into a situation in which some processors finish their work early and sit idle due to lack of work.
Dynamic load balancing algorithms have relatively better resource utilization as dynamic load balancing take care of the fact that load should be equally distributed to processors so that no processors should sit idle.

Process Migration: - Process migration parameter provides when does a system decide to export a process? It decides whether to create it locally or create it on a remote processing element. The algorithm is capable to decide that it should make changes of load distribution during execution of process or not.

Pre emptiveness: - This factor is related with checking the fact that whether load balancing algorithms are inherently non-preemptive as no tasks are relocated. Dynamic load balancing algorithms are both preemptive and non-preemptive.

Response Time: - How much time a distributed system using a particular load balancing algorithm is taking to respond? Static load balancing algorithms have shorter response time as one should not forget that in Static load balancing there is lesser overhead as discussed earlier so emphasis is totally on executing jobs in shorter time rather than optimally utilizing the available resources. Dynamic load balancing algorithms may have relatively higher response time as sometimes redistribution of processes takes place. Some time is being consumed during task migration

Waiting Time: - Waiting Time is the sum of the periods spent waiting in the ready queue.

Turnaround Time: - Turn interval from the time of submission of a process to the time of completion is the turnaround time.

Throughput: - Throughput is the amount of data moved successfully from one place to another in a given time period.

Processor Thrashing: - Processor thrashing occurs when most of the processors of the system are spending most of their time migrating processes without accomplishing any useful work in an attempt to properly schedule the processes for better performance. Static load balancing algorithms are free from Processor thrashing as no relocation of tasks place. Dynamic load balancing algorithms incurs substantial processor thrashing.

TABLE1: COMPARATIVE ANALYSIS OF LOAD BALANCING ALGORITHMS

Parameter	**ADAPTIVE LOAD SHARING ALGORITHM**	**MAXIMALLY-LINKED MODULES ALGORITHM**	**NEAREST-NEIGHBOR LOAD BALANCING**	**MAEKAWA'S ALGORITHM**	**A* ALGORITHM**	**PARALLEL GENETIC ALGORITHM**	**DETERMINISTIC MODEL GENETIC ALGORITHM**	**MULTI HEURISTIC**
Nature	Static	Static	Static	Static	Static	dynamic	dynamic	Dynamic
Overload Rejection:	NO	NO	NO	NO	Yes	yes	yes	Yes
Reliability	Less	Less	Less	Less	Less	More	More	More
Adaptability	Less	Less	Less	Less	Less	More	More	More

Stability	Large	Large	Large	Large	Large	Small	Small	Small
Predictability	More	More	More	More	More	Less	Less	Less
Forecasting Accuracy	More	More	More	More	More	Less	Less	Less
Cooperative	No	No	Yes	Yes	Yes	Yes	Yes	Yes
Fault Tolerant	No	No	No	Yes	Yes	Yes	Yes	Yes
Resource Utilization	Less	Less	Less	More	More	More	More	More
Process Migration	NO	NO	NO	NO	Yes	yes	yes	Yes
Pre emptiveness	Non-Preemptive	Non-Preemptive	Non-Preemptive	Non-Preemptive	Non-Preemptive	Non Preemptive & Preemptive	Non-Preemptive & Preemptive	Non Pre-emptive & Pre-emptive
Response Time	Less	Less	Less	Less	More	More	More	More
Waiting Time	More	More	More	More	Less	Less	Less	Less
Turnaround Time	Less	Less	Less	Less	More	More	More	More
Throughput	Low	Low	Low	Low	High	High	High	High
Processor Thrashing	No	No	No	No	Yes	Yes	Yes	Yes

V. CONCLUSION

In this paper we studied the load balancing strategies in detail. Load balancing in distributed systems is the most thrust area in research today as the demand of heterogeneous computing due to the wide use of internet. In this paper we have carried out the analysis of different load balancing algorithms, various parameters are used to check the results. More efficient load balancing algorithm more is the performance of the computing system. We have enumerated the facilities provided by load balancing algorithms. Finally, we studied some important dynamic load balancing algorithms and made their comparison to focus their importance in different situations. There exists no absolutely perfect balancing algorithm but one can use depending on the need. The comparative study not only provides an insight view of the load balancing algorithms, but also offers practical guidelines to researchers in designing efficient load balancing algorithms for distributed computing systems.

REFERENCES

[1] Ajith Tom P. and C. Siva Ram Murthy, An Improved Algorithm For Module Allocation in Distributed Computing Systems, Journal of Parallel and Distributed Computing Systems," 42 (1997),pp. 82-90.

[2] Sagar G. and A.K. Sarje, Task Allocation Model for Distributed System, Int. J. Systems Sci. Vol. 22, 9(1991). pp. 1671- 1678.

[3] Mostaph Zbakh, Mohamed Dafir El Kettani, A Task Allocation Algorithm For Distributed Systems, Journal of Theoretical and Applied Information Technology, Vol. 33 No. 1, 2011.

[4] A. Osman , H. Ammar ,A Scalable Dynamic Load-Balancing Algorithm for SPMD Applications on a Non-Dedicated Heterogeneous Network of Workstations (HNOW),

[5] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "Adaptive load sharing in homogeneous distributed systems," IEEE Trans. Softw. Eng., vol. 12, pp. 662-675, 1986.

[6] W. Leinberger, G. Karypis, and V. Kumar, "Load Balancing Across Near Homogeneous Multi-Resource Servers," presented at Proceedings. 9thHeterogeneous Computing Workshop (HCW 2000) Cancun, Mexico, 2000.

[7] T. L. Casavant and J. G. Kuhl, ''A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems,'' IEEE Transactions on Software Engineering, vol. 14, no. 2, pp. 141-154, February1988.

[8] F. Berman, High-performance schedulers, in: I. Foster, C. Kesselman (Eds.), The Grid: Blueprint for a New Computing Infrastructure, Morgan Kaufmann, Los Altos, CA, 1999, pp. 279–309,Chapter12.

[9] Farzad Ghannadian, Cecil O. Alford, and Ron Shonkwiler, "Application of the genetic algorithm to multiprocessor scheduling", Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA 95).

[10] Gylys, V. B., and Edwards, J. A., "Optimal partitioning of work load for distributed system", Procs. Compcon Fall 76, pp. 353-357.

[11] A. Abdelmageed Elsadek B. Earl Wells," A Heuristic model for task allocation in heterogeneous distributed computing systems".

[12] Z. Zeng and B. Veeravalli, Divisible Load Scheduling on Arbitrary Distributed Networks via Virtual Routing Approach, Proceedings of the Tenth International Conference on Parallel and DistributedSystems (ICPADS'04) 2004.

[13] Md. Abdur Razzaque and Choong Seon Hong, "Dynamic Load Balancing in Distributed System: An Efficient Approach".

[14] Kamaljit Kaur, Amit Chhabra & Gurvinder Singh ,"Heuristics Based Genetic Algorithm for Scheduling Static Tasks in Homogeneous Parallel System" International Journal of Computer Science and Security (IJCSS), Volume (4): Issue(2).

[15] R.Nedunchelian, K.Koushik, N.Meiyappan, V.Raghu "Dynamic Task Scheduling using Parallel Genetic algorithms for heterogeneous Distributed computing.

[16] AJ Page, TJ Naughton Dynamic task scheduling using genetic algorithms for heterogeneous distributed computing, - Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05).

[17] Ahmed Younes. Hamed," Task Allocation for Minimizing Cost of Distributed Computing Systems Using Genetic Algorithms" International Journal of Advanced Research in Computer Science and Software Engineering, Volume 2, Issue 9, September 2012 ISSN: 2277 128X.

[18] Manisha Sharma, Harendra Kumar, Deepak Garg "An Optimal Task Allocation Model through Clustering with Inter-Processor Distances in Heterogeneous Distributed Computing Systems", International Journal of Soft Computing and Engineering (IJSCE) ISSN: 2231-2307, Volume-2, Issue-1.