# A Pragmatic Analysis On Assorted Load Balancing Multiprocessing Scheduling Algorithms

**Shreya Chauhan**

## ABSTRACT

Parallel processing is one of the important processing types of applications that are used to execute multiple tasks on different number of processors at the same time. To utilize the processors in optimized manner, various scheduling algorithms are used. In multiprocessor scheduling algorithms, scheduling may be required for both related and unrelated tasks. The main objective of any multiprocessor scheduling algorithm is to schedule the tasks or jobs in optimized way so that performance of the processors may be optimized. In this paper, we have discussed about multiprocessor scheduling algorithms.

*Keywords: Global EDF scheduling, LLREF scheduling, LRE-TL scheduling, PF scheduling, Multiprocessor scheduling, Time sharing scheduling.*

## 1. INTRODUCTION

Parallel processing is one of the promising impressions that are used to execute multiple tasks on different number of processors at the same time [1]. Multiprocessor task scheduling problem is essentially classical machine scheduling problem in which a number of tasks are to be processed on more than one processor at a time and the objective is to minimize the overall task execution time. Scheduling of multiple tasks is highly critical to the performance of a multiprocessor computing system. It requires the allocation of individual tasks to suitable processor in some designated order the objective being to minimize the overall completion time. A Number of multiprocessor list scheduling heuristics have been proposed in the last few years, some of these consider communication costs also. To be more realistic a scheduling algorithm should exploit the parallelism by identifying the task graph structure and take into consideration task granularity, arbitrary computation, and communication costs. The priorities should also be determined statically before the scheduling process begins. There are two types of scheduling techniques: list scheduling and timely scheduling. The main problem with list scheduling algorithms is that static priority assignment may not always order the nodes for scheduling according to their relative importance. However, timely scheduling of the nodes can lead to a better schedule eventually. However, the drawback of this approach is that an inefficient schedule will only be generated if a relatively less important node is chosen for scheduling before the more important ones (priority assignment may not capture the variation of the relative importance of nodes during the scheduling process). For example the algorithms such as highest levels first with estimated times (HLFET), the modified critical path (MCP) algorithm, the dominant sequence clustering (DSC) algorithm, and the mobility directed (MD) algorithm does not precisely identify the most important node at each scheduling step because these order nodes while assigning each of them a static attribute that does not depend on the communication among nodes [2].

## 1.1 MULTIPROCESSOR SCHEDULING

The problem statement for multiprocessing scheduling can be defined as: "Given a set *S* of jobs where job $S_j$ has length $t_j$ and a number of processors *n*, what is the minimum possible time required to schedule all jobs in *S* on *n* processors such that none of them overlaps[2]? Mathematically, the problem of multiprocessor scheduling can be defined as:

Given n identical processors $N_1, \ldots, N_n$ and j jobs $S_1, S_2, \ldots, S_j$. Job $S_J$ has a processing time tj $\geq 0$ and the goal is to assign jobs to the processors so as to minimize the maximum load where the load of a processor is defined as the sum of the processing times of jobs that are assigned to that processor.

## 1.2 UNIPROCESSOR VERSUS MULTIPROCESSOR SCHEDULING

The term Multiprocessor scheduling [3] is the combination of terms multiprocessor and scheduling. Multiprocessor system means the computing system where more than one processor is working in parallel to execute a single process or multiple processes. The term scheduling means to decide which process or job is to be processed next. On a Uniprocessor, scheduling is one dimensional means only to decide "Which process should be run next?" On a multiprocessor, scheduling is two dimensional. The scheduler has to decide which process to run and which CPU to run it on. Due to the extra dimension scheduling on multiprocessors becomes complicated. The main factor that needs consideration is that in some systems, all the processes are unrelated like timesharing systems in which independent users start up independent processes. The processes are not related and each one can be scheduled not including the other processes.

## 1.3 TIMESHARING

Let us first address the case of scheduling independent processes; later we will consider how to schedule related processes. The simplest scheduling algorithm for dealing with unrelated processes (or threads) is to have a single system wide data structure for ready processes, possibly just a list, but more likely a set of lists for processes at different priorities as depicted in Fig. 1(a).

Here the 16 CPUs are all currently busy, and a prioritized set of 14 processes are waiting to run. The first CPU to finish its current work (or have its process block) is CPU 4, which then locks the scheduling queues and selects the highest priority process, A, as shown in Fig. 1(b). Next, CPU 12 goes idle and chooses process B, as illustrated in Fig. 1(c). As long as the processes are completely unrelated, doing scheduling this way is a reasonable choice.

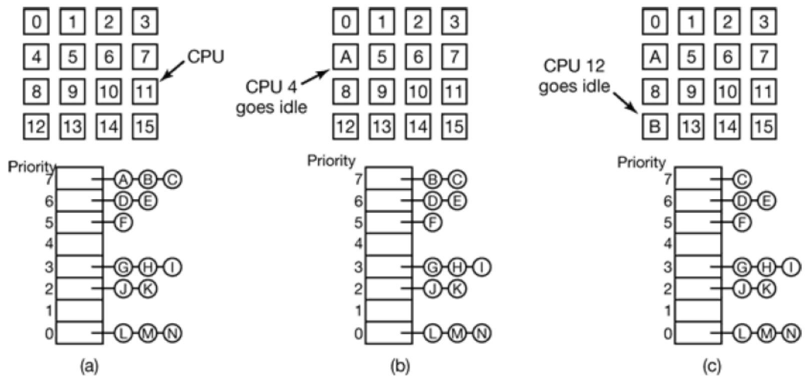


*Fig. 1: Using a single data structure for scheduling a multiprocessor [23]*

Having a single scheduling data structure used by all CPUs timeshares the CPUs, much as they would be in a uniprocessor system. It also provides automatic load balancing because it can never happen that one CPU is idle while others are overloaded. Two disadvantages of this approach are the potential contention for the scheduling data structure as the numbers of CPUs grows and the usual overhead in doing a context switch when a process blocks for I/O.

Some multiprocessors take this effect into account and use what is called affinity scheduling. The basic idea here is to make a serious effort to have a process run on the same CPU it ran on last time. One way to create this affinity is to use a two-level scheduling algorithm. When a process is created, it is assigned to a CPU, for example based on which one has the smallest load at that moment. This assignment of processes to CPUs is the top level of the algorithm. As a result, each CPU acquires its own collection of processes.

The actual scheduling of the processes is the bottom level of the algorithm. It is done by each CPU separately, using priorities or some other means. By trying to keep a process on the same CPU, cache affinity is maximized. However, if a CPU has no processes to run, it takes one from another CPU rather than go idle.

Two-level scheduling has three benefits. First, it distributes the load roughly evenly over the available CPUs. Second, advantage is taken of cache affinity where possible. Third, by giving each CPU its own ready list, contention for the ready lists is minimized because attempts to use another CPU's ready list are relatively infrequent.

## 1.4 SPACE SHARING

The other general approach to multiprocessor scheduling can be used when processes are related to one another in some way. Earlier we mentioned the example of parallel *make* as one case. It also often occurs that a single process creates multiple threads that work together. For our purposes, a job consisting of multiple related processes or a process consisting of multiple kernel threads are essentially the same thing. We will refer to the schedulable entities as threads here, but the material holds for processes as well. Scheduling multiple threads at the same time across multiple CPUs is called space sharing.

At any instant of time, the set of CPUs is statically partitioned into some number of partitions, each one running the threads of one process. In Fig. 2, we have partitions of sizes 4, 6, 8, and 12 CPUs, with 2 CPUs unassigned, for example. As time goes on, the number and size of the partitions will change as processes come and go.

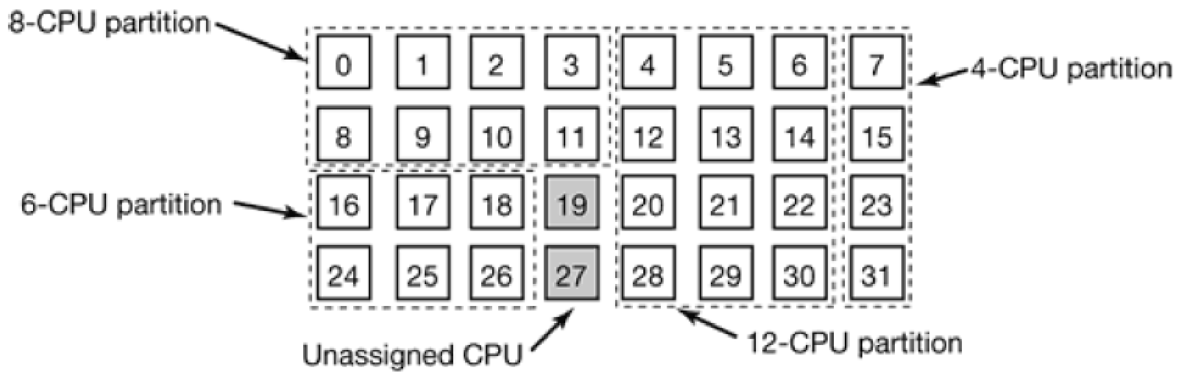


*Fig. 2: A set of 32 CPUs split into four partitions, with two CPUs available. [24]*

Periodically, scheduling decisions have to be made. In uniprocessor systems, shortest job first is a well-known algorithm for batch scheduling. The analogous algorithm for a multiprocessor is to choose the process needing the smallest number of CPU cycles that is the process having CPU-count X run-time is the smallest of the candidates. However, in practice, this information is rarely available, so the algorithm is hard to carry out. In fact, studies have shown that, in practice, beating first-come, first-served is hard to do.

In this simple partitioning model, a process just asks for some number of CPUs and either gets them all or has to wait until they are available. A different approach is for processes to actively manage the degree of parallelism. One way to do manage the parallelism is to have a central server that keeps track of which processes are running and want to run and what their minimum and maximum CPU requirements are. Periodically, each CPU polls the central server to ask how many CPUs it may use. It then adjusts the number of processes or threads up or down to match what is available. For example, a Web server can have 1, 2, 5, 10, 20, or any other number of threads running in parallel. If it currently has 10 threads and there is suddenly more demand for CPUs and it is told to drop to 5, when the next 5 threads finish their current work, they are told to exit instead of being given new work. This scheme allows the partition sizes to vary dynamically to match the current workload better than the fixed system of Fig. 2.

## 1.5 GANG SCHEDULING

A clear advantage of space sharing is the elimination of multiprogramming, which eliminates the context switching overhead. However, an equally clear disadvantage is the time wasted when a CPU blocks and has nothing at all to do until it becomes ready again. Consequently, people have looked for algorithms that attempt to schedule in both time and space together, especially for processes that create multiple threads, which usually need to communicate with one another.

To see the kind of problem that can occur when the threads of a process (or processes of a job) are independently scheduled, consider a system with threads $A_0$ and $A_1$ belonging to process *A* and threads $B_0$ and $B_1$ belonging to process *B*. threads $A_0$ and $B_0$ are timeshared on CPU 0; threads $A_1$ and $B_1$ are timeshared on CPU 1. threads $A_0$ and $A_1$ need to communicate often. The communication pattern is that $A_0$ sends $A_1$ a message, with $A_1$ then sending back a reply to $A_0$, followed by another such sequence. Suppose that luck has it that $A_0$ and $B_1$ start first, as shown in Fig. 3.

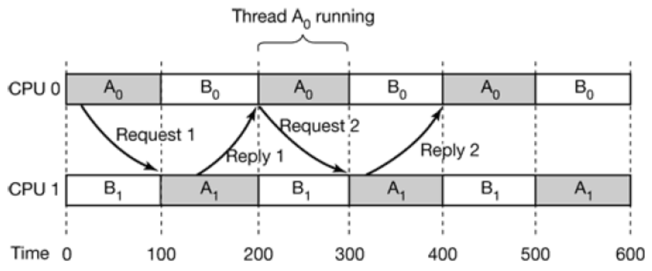


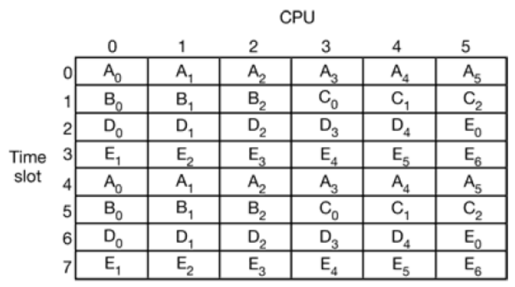
*Fig. 3: Communication between two threads belonging to process A that are running out of phase. [24]*

In time slice 0, $A_0$ sends $A_1$ a request, but $A_1$ does not get it until it runs in time slice 1 starting at 100 msec. It sends the reply immediately, but $A_0$ does not get the reply until it runs again at 200 msec. The net result is one request-reply sequence every 200 msec. The solution to this problem is gang scheduling, which is an outgrowth of co-scheduling**.** Gang scheduling has three parts:

1. Groups of related threads are scheduled as a unit, a gang.
2. All members of a gang run simultaneously, on different timeshared CPUs.
3. All gang members start and end their time slices together.

The trick that makes gang scheduling work is that all CPUs are scheduled synchronously. This means that time is divided into discrete quanta as we had in Fig. 3. At the start of each new quantum, *all* the CPUs are rescheduled, with a new thread being started on each one. At the start of the following quantum, another scheduling event happens. In between, no scheduling is done. If a thread blocks, its CPU stays idle until the end of the quantum.

An example of how gang scheduling works is given in Fig.4. Here we have a multiprocessor with six CPUs being used by five processes, *A* through *E*, with a total of 24 ready threads. During time slot 0, threads $A_0$ through $A_6$ are scheduled and run. During time slot 1, Threads $B_0$, $B_1$, $B_2$, $C_0$, $C_1$, and $C_2$ are scheduled and run. During time slot 2, *D*'s five threads and $E_0$ get to run. The remaining six threads belonging to process *E* run in time slot 3. Then the cycle repeats, with slot 4 being the same as slot 0 and so on.



| Time slot | CPU 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | $A_0$ | $A_1$ | $A_2$ | $A_3$ | $A_4$ | $A_5$ |
| 1 | $B_0$ | $B_1$ | $B_2$ | $C_0$ | $C_1$ | $C_2$ |
| 2 | $D_0$ | $D_1$ | $D_2$ | $D_3$ | $D_4$ | $E_0$ |
| 3 | $E_1$ | $E_2$ | $E_3$ | $E_4$ | $E_5$ | $E_6$ |
| 4 | $A_0$ | $A_1$ | $A_2$ | $A_3$ | $A_4$ | $A_5$ |
| 5 | $B_0$ | $B_1$ | $B_2$ | $C_0$ | $C_1$ | $C_2$ |
| 6 | $D_0$ | $D_1$ | $D_2$ | $D_3$ | $D_4$ | $E_0$ |
| 7 | $E_1$ | $E_2$ | $E_3$ | $E_4$ | $E_5$ | $E_6$ |

*Fig. 4: Gang scheduling [24]*

The idea of gang scheduling is to have all the threads of a process run together, so that if one of them sends a request to another one, it will get the message almost immediately and be able to reply almost immediately. In Fig. 4, since all the *A* threads are running together, during one quantum, they may send and receive a very large number of messages in one quantum, thus eliminating the problem of Fig. 3.

## 2. **DETAILED SURVEY OF MULTIPROCESSOR SCHEDULING ALGORITHMS**

Nirmeen et. al[4] has defined many existing task scheduling algorithms like Heterogeneous Earliest Finish Time (HEFT) [7], Critical Path On a Processor (CPOP) [8], Critical Path On a Cluster (CPOC) [8], Dynamic Level Scheduling (DLS) [8], Modified Critical Path (MCP) [6], Mapping Heuristic (MH) [7] and Dynamic Critical Path (DCP) [5].

Dan et. al[9] have discussed the three multiprocessor scheduling algorithms classic and simple global EDF algorithm , the optimal P-fair algorithm PF, and an optimal algorithm LLREF that intends to improve on existing P-fair algorithms and global EDF variants

### **2.1 Global EDF**

EDF is one of the oldest and most eminent scheduling algorithms, demarcated by Liu and Layland [10]. In this algorithm, the priority of each instance of a task is determined by its outright deadline; the task with the most primitive absolute deadline will always have the highest priority. EDF is work-saving; that is, if there is any dynamic work, the processor will execute it instead of being unmoving. It has been demonstrated that EDF is optimal for intermittent undertakings, and that having an usage less than or equivalent to1 is a fundamental and sufficient condition for EDF to have the ability to timetable undertakings.

Global EDF is an extension of EDF for multiple processors [11]. Like PF and LLREF, it is a dynamic priority algorithm. Similar to EDF, the jobs are ordered by earliest absolute deadline, but the P highest priority processes are executed by the P processors in every time step. Scheduling events occur only when new jobs are introduced or when a job completes. The likelihood of migrations in global EDF depends first on how often preemptions happen. Migrations can only occur due to unfavorable scheduling events. The introduction of new jobs causes only active →idle transitions. If the introduced job has an earlier deadline than an active job, it is swapped in for the lowest priority active job. Job completion causes only idle→active transitions. If a job has completed, the highest priority idle job is chosen to execute, regardless of what CPU the job may have been executing on previously. If a job's execution state has an active → idle → active sequence, it is possible for the job to migrate between CPUs. The other factor in the likelihood of migrations in global EDF is the number of CPUs. The only reason a preempted task might not be migrated is if the preempting task completed before the other active jobs. Suppose completion time is uniformly random. If we consider a uniprocessor system, migration is not possible. For P = 2, preempted jobs are migrated with 50% probability; for P= 10, 90%. As CPU counts increase, the frequency of migrations made by global EDF increases.

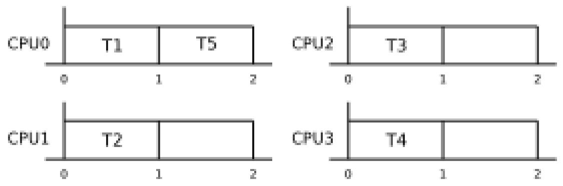


*Fig. 5: Global EDF schedule for 5 identical tasks [24]*

## 2.2 P-FAIR (PF) ALGORITHM

The PF algorithm, presented in [12, 13], is optimal for scheduling periodic tasks on multiple processors. In this regard, PF improves on global EDF because global EDF is not optimal. PF uses a "time-slicing" approach, which means it splits up tasks into unit Intervals of execution, termed "slots", or ticks. PF uses this approach to approximate a "fluid scheduling", or constant rate scheduling[14].The basic idea of the PF algorithm is to assign slots to each task such that it is always scheduled proportionally to the utilization of the task. That is, if a task has a utilization of U , at any slot t , the task will have been scheduled for U.t slots between 0 and t . The authors of [12,13] define any schedule that satisfies this requirement as proportionately fair or P-fair. They also prove that any schedule that is P-fair will meet all the deadlines for its tasks. The goal of the PF algorithm is to maintain the P-fair requirement for all tasks. One important point about the formulation of P-fairness is that the value U.t is not always an integer, and tasks cannot be scheduled to partial numbers of slots.

The authors of [12, 13] define the characteristic string for a task $T_0$ with utilization $U_0$ at time t over the characters (+0,−) as follows:

$$c(T_0,U_0,t) = \mathrm{sign}(U_{0(t+1)} - \lfloor U_0.t \rfloor - 1) \qquad (1)$$

With the lag computation and the characteristic string in hand, PF classifies each task at a time t in the following way. A task is classified as non-urgent if the lag for the task is strictly less than 0 and the characteristic string at the slot for t is either − or 0. A task is classified as urgent if the lag for the task is strictly greater than 0 and the characteristic string at the slot for t is either + or 0. A task that is neither non-urgent nor urgent is classified as contending. An invariant that falls out of this classification is that at every slot t , all urgent tasks must be scheduled and all non-urgent tasks must not be scheduled. If either of these conditions cannot be met, then the task set cannot be feasibly scheduled using PF or any other algorithm, given that PF is optimal.

For example, let the characteristic string of a task be "− −+0− −+ 0". When PF classifies this task as contending, only the sub string "− −+0" is used to order the task relative to the other contending tasks

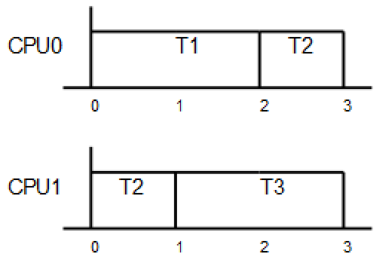


*Fig. 6: PF to schedule three identical tasks on two processors [24]*

## 2.3 LLREF ALGORITHM

The LLREF scheduling algorithm [15] assumes that tasks are preemptible and independent (i.e., that they have no shared resources). The cost of context switches and task migration is ignored and assumed to be negligible. The deadline is assumed to be equal to the period. Like PF, LLREF approximates a fluid scheduling model.

## 3. METRICS FOR MULTIPROCESSOR SCHEDULING

The first metric evaluates the ability of the algorithm to feasibly schedule a given set of tasks. To measure schedulability, task sets are used with utilization less than the number of processors. Both LLREF and PF are able to schedule the task sets in optimal way. However, since global EDF is non-optimal, there is no guarantee that global EDF would be able to schedule the task sets[12]. The second metric measures the total number of task migrations made by a specific schedule for a set of tasks. The third metric measures the total number of scheduler invocations required to produce the final schedule. The task migrations and scheduler invocations characterize the overhead of each algorithm.

## 4. OTHER SCHEDULING ALGORITHMS

## 4.1 LRE-TL ALGORITHM

Shelby et. al[16] have proposed LRE-TL(local remaining execution-TL), a scheduling algorithm based on LLREF, and demonstrated its flexibility and improved running time over existing scheduling algorithms. Unlike LLREF, LRE-TL is optimal for sporadic task sets. While most LLREF events take O(n) time to run, the corresponding LRETL events take O(log n) time. LRE-TL also reduces the number of task preemptions and migrations by a factor of n.

Optimal multiprocessor scheduling algorithms tend to have restrictions that make them less desirable than other non-optimal algorithms. Some common restrictions are that (i) they have high overhead, (ii) they apply only to a restrictive model for jobs or processors, or (iii) the schedule must be quantum based (i.e., the scheduler is invoked every q time units for some constant q). There are two well known optimal multiprocessor scheduling algorithms, Pfair [17] and LLREF [18], each suffer from at least one of these shortcomings.

While Pfair can schedule both periodic [19] and sporadic [20], [21] tasks, it applies only to quantum based systems on identical multiprocessors. On the other hand, LLREF can be scheduled on both identical and uniform multiprocessors, but it has high scheduling overhead and applies only to periodic task systems.

Shelby et. al. [16] have proposed a new scheduling algorithm, LRE-TL, which is based on the LLREF scheduling algorithm and have proved that LRE-TL is optimal for periodic and sporadic task sets and has much lower scheduling overhead than LLREF.
The steps for algorithm presented by Shelby et. al[16] are as under:
The algorithm LRE-TL is comprised of four procedures. The main algorithm determines which type of events have occurred, calls the handlers for those events, and instructs the processors to execute their designated tasks. At each TL-plane boundary, LRETL calls the TL-plane initializer. Within a TL-plane, LRE-TL processes any A, B or C events. The TL-plane initializer sets all parameters for the new TL-plane. The A event handler determines the local remaining execution of a newly arrived sporadic task, and puts the task in one of the heaps (HB or HC). The B and C event handler maintains the correctness of HB and HC.

**4.2 DAG BASED DYTAS ALGORITHM**
D.I. George et. al [22] have proposed a DAG based dynamic tasks scheduling model and a scheduling algorithm DYTAS (DYnamic TAsk Scheduling algorithm) has been proposed with a lower time complexity. The authors have proposed a new dynamic scheduling algorithm based on scheduler model, DYnamic TAsk Schedulling (DYTAS) algorithm. This strategy makes the whole parallel job finished at the possible earliest time viz. the response time of this parallel task is shortest. According to DYTAS, the tasks in ITQ are scheduled by its dependency. The front task in ITQ is always first scheduled and mapped to a processor by the algorithm. While in the static scheduling algorithm, the tasks are sorted by a certain priority rank, because the data of DAG is known in advance. But, the proposed dynamic scheduling algorithm is different from the static scheduling algorithms, by migrating the task during the runtime.

## CONCLUSION

Multiprocessor scheduling is the canonical example of NP-hard optimization problem. The applications of this problem are diverse, but most strongly associated with the scheduling of computational tasks in a multiprocessor environment. The multiprocessor scheduling problem is a generalization of the optimization version of the number partitioning problem that considers the case of partitioning a set of jobs into two processors. On a single CPU system, multithreading (i.e. more than one control flow in a single address space) is a fine, and if performed accurate can greatly avail system’s performance. But on a multiprocessor system, there is even greater assistance to win - while at the same time it gets indurate to achieve them.

## REFERENCES

1. Yu-Kwong Kwok, Ishfaq Ahmad, "Static Scheduling Algorithm for Allocating Directed Task Graph to multiprocessors", ACM Computing Surveys, Vol. 31, no. 4, December 1999.
2. Poonam Panwar, A.K.Lal and Jugminder Singh, "A Genetic Algorithm based Technique for Efficient Scheduling of Tasks on Multiprocessor System", Advances in Intelligent and Soft Computing, vol 131, pp. 855–861, SPRINGER INDIA, 2012.
3. Garey, Michael R.; Johnson, David S.. Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman and Company. p. 238. ISBN 0716710445. Multiprocessor Operating Systems, By Andrew S. Tanenbaum, Mar 22, 2002.
4. Nirmeen A. Bahnasawy, Fatma Omara, Magdy A. Koutb, Mervat Mosa, "A New Algorithm for Static Task Scheduling For Heterogeneous Distributed Computing Systems", Volume 1 No. 1, May 2011 International Journal of Information and Communication Technology Research, 2010-11.
5. S. Shivle, R. Castain, H. J. Siegel, A. A. Maciejewski, T. Banka, K. Chindam, S. Dussinger, P. Pichumani, P. Satyasekaan, W. Saylor, D.Sendek, J. Sousa, J. Sridharan, P. Sugavanam, and J. Velazco,"Static mapping of subtasks in a heterogeneous ad hoc grid environment," Proc. of Parallel and Distributed Processing Symposium, Apr. 2004.
6. K. Hwang, Advanced Computer Architecture: Parallelism, Scalability, Programmability, New York: McGraw-Hill, Inc., 1993.
7. H. Topcuoglu, S. Hariri, M.Y. Wu," Performance-Effective and Low Complexity Task Scheduling for Heterogeneous Computing," IEEE Trans. Parallel Distributed Systems, Vol. 13 (3), pp.260–274, 2005.
8. Mohammad I. Daoud, Nawwaf Kharma "A High Performance Algorithm For Static Task Scheduling In Heterogeneous Distributed Computing Systems," IEEE Trans. Parallel Distributed Systems, Vol. 28, pp39-49, July 2007.
9. Dan McNulty, Lena Olson, Markus Peloquin, "A Comparison of Scheduling Algorithms for Multiprocessors", December 13, 2010.
10. Liu, C. L., and Layland, J. W. "Scheduling algorithms for multiprogramming in a hard-real -time environment", Journal of ACM 20, (January 1973), 46–61.
11. Brandenburg, B. B., Calandrino, J. M., and Anderson, J. H. "On the scalability of real-time scheduling algorithms on multicore platforms: A case study", In IEEE Real-Time Systems Symposium (2008), pp. 157–169.
12. Baruah, S. K., Cohen, N. K., Plaxton, C. G., and Varvel, D. A. Proportionate progress: a notion of fairness in resource allocation. In Proceedings of the twenty-fifth annual ACM symposium on Theory of computing (New York, NY, USA, 1993), STOC '93, ACM, pp. 345–354.
13. Baruah, S. K., Gehrke, J., and Plaxton, C. G. Fast scheduling of periodic tasks on multiple resources. In Proceedings of the 9th International Symposium on Parallel Processing (Washington, DC, USA, 1995), IPPS '95, IEEE Computer Society, pp. 280–288.
14. Holman, P., and Anderson, J. Adapting Pfair scheduling for symmetric multiprocessors. Journal of Embedded Computing 1 , 4 (2005), 543–564.

15. Cho, H., Ravindran, B., and Jensen, E. D. An optimal real-time scheduling algorithm for multiprocessors. In RTSS ’06: Proceedings of the 27th IEEE International Real-Time Systems Symposium (Washington, DC, USA, 2006), IEEE Computer Society, pp. 101–110.
16. LRE-TL: An Optimal Multiprocessor Scheduling Algorithm for Sporadic Task Sets Shelby Funk and Vijaykant Nadadur.
17. S. K. Baruah, N. Cohen, C. G. Plaxton, and D. Varvel, “Proportionate progress: A notion of fairness in resource allocation,” Algorithmica, vol. 15, no. 6, pp. 600–625, June 1996.
18. H. Cho, B. Ravindran, and E. D. Jensen, “An optimal real-time scheduling algorithm for multiprocessors,” in Proceedings the 27th IEEE Real-Time System Symposium (RTSS), Los Alamitos, CA, 2006, pp. 101 – 110.
19. C. L. Liu and J. W. Layland, “Scheduling algorithms for multiprogramming in a hard real- time environment,” Journal of the ACM, vol. 20, no. 1, pp. 46–61, 1973.
20. M. Dertouzos and A. K. Mok, “Multiprocessor scheduling in a hard real-time environment,” IEEE Transactions on Software Engineering, vol. 15, no. 12, pp.1497–1506, 1989.
21. M. Dertouzos, “Control robotics : the procedural control of physical processors,” in Proceedings of the IFIP Congress, 1974, pp. 807–813.
22. D.I. George Amalarethinam, G.J. Joyce Mary, “A new DAG based Dynamic Task Scheduling Algorithm (DYTAS) for Multiprocessor Systems”, International Journal of Computer Applications (0975 – 8887) Volume 19– No.8, April 2011.
*23.* URL sce.umkc.edu/~cotterr/cs431_sp13
*24. Multiprocessor operating system: Andrew S Tanenbaum*