Taxonomy of Scheduling Algorithms

Abhilash Sharma[1], Anjali Sen[2]
[1]*Assistant.Professor, Vaishno College of Engineering, Thapkour, Kangra, HP, India*
[2]*Lecturer, Vaishno College of Polytech and Engineering, Thapkour, Kangra, HP, India*

Abstract- The scheduling algorithm is the last component of a scheduling system. It has the task to generate a valid schedule for the actual stream of submission data in an online fashion. A good scheduling algorithm is expected to produce very good if not optimal schedule with respect to the objective function while not taking 'too much' time resources to determine the schedule. Unfortunately, most scheduling problems are computationally very hard. This is even true for online problems with simple objective functions and few additional requirements see from instance. Therefore, it is not reasonable to hope in general for an algorithm that always guarantees the best possible schedule. In addition, some job data may not be immediately available or may be incorrect which makes the task for the algorithm even harder. In order to obtain good schedule the administrator of a parallel machine is therefore faced with the problem to pick an appropriate algorithm among a variety of suboptimal ones. She may even decide to design an entirely new method if the available ones do not yield satisfactory result. The selection of the algorithm is highly dependent on a variety of constraints.

Keywords- *Taxonomy, Scheduling Algorithms, Time Sharing Policies, Space Sharing Policies.*

I. INTRODUCTION

A task is the unit of computation in our computing systems, and several tasks working towards a common goal are called a job. There are two levels of scheduling in multiprocessor system: global scheduling and local scheduling [casavant and kuhi, 1988]. Local scheduling determines which of the set of variables tasks at a processor runs next on that processor. Global scheduling involves assigning a task to a particular processor within the system. This is also known as mapping, task placement, and matching. Global scheduling takes places before local scheduling, although task migration, or dynamic reassignment, can change the global mapping by moving a task to a new processor. To migrate a task, the system freezes the task, save its state, transfer the saved state to a new processor, and restart the task. There is substantial overhead involved in migration a running task.

One of the main uses for global scheduling is to perform load sharing between processors. Load sharing allows busy processors to offload some of their work to less busy, or even idle, processors. Load balancing is a special case of load sharing, in which the goal of the global scheduling algorithm is to keep the load even (or balanced) across all processor. Sender-initiated load sharing occurs when busy processors try to find idle processors seek busy processors. It is now accepted wisdom that load balancing is generally not worth doing, as the small gain in execution time of the tasks is more than offset by the effort expended in maintaining the balanced load. A global scheduling policy may be thought of as having four distinct parts:

- The transfer policy
- The selection policy
- The location policy, and
- The information policy.

The transfer policy decides when a node should migrate a task, and the selection policy decides which task to migrate. The location policy determines a partner node for the task migration and the information policy determines how node state information is disseminated among the processor in the systems. An important feature of the selection policy is whether it restricts the candidates set of task to new tasks which have not yet run, or allow the transfer of tasks that have begun execution[1].

Non- preemptive policies only transfer new jobs, while preemptive policies will transfer running jobs as well. preemptive policies have a larger set of candidates for transfer , but the overhead of migration a job that has begun execution is higher than for a new job because of the accumulated state of the running job (such as open descriptors, allocated memory , etc). As the system runs, new tasks arrives while the old tasks complete execution (or served). It arrival rate is greater than the service rate , the process waiting queues within the system will grow without bound and the system is said to be unstable, if , however , tasks are serviced as least as fast as they arrive, the queues in the system will have bounded length and the system is said to be stable. If the arrival rate is just slightly less than the service rate for a system, it is possible for the additional overhead of load sharing to push the system into instability. A stable scheduling policy does not have this property, and will never make a stable system unstable. In most cases, work in distributed

scheduling concentrates on global scheduling because of the architecture of the underlying system. Casavant and kuhl [casavant and kuhi, 1988] [12] defines a taxonomy of task placement algorithms for distributed systems. The two major categories of global algorithms are static and dynamic.

- **Static algorithms** make scheduling decisions based purely on information available at compilation time. For example, the typical input into a static algorithm would include the machine configuration and the number of tasks and estimates of their running time.

- **Dynamic algorithms**, on the other hand, take factor into account such as the current load processor. Adaptive algorithms are subclass of dynamic algorithms, and are important enough that they are often discussed separately. Adaptive algorithms go one step further algorithms, in that they may change the policy based on dynamic information. A dynamic load sharing algorithm might use the current system state information to seek out a lightly-loaded host, while an adaptive algorithm might switch from sender-initiated to receiver–initiated load rises above a threshold.

- In **physically non- distributed, or centralized,** scheduling policies, as single processor makes all decisions regarding task placement. Under physically distributed algorithm, the logical authority for the decision- making process is distributed among the processors that constitute the system. [12]

- Under **non-cooperative distributed scheduling policies**, individual processors make scheduling choices independent of the choices made by other processors. With cooperative scheduling, the processor subordinates local autonomy to the achievement of a common goal. [12]

- Both **static and cooperative distributed scheduling** have **optimal and sub optimal branches**. Optimal assignments can be reached if complete information describing the system and the task force is available. Suboptimal information is either approximate or heuristic. Heuristic algorithm use guiding principle s, such as assigning tasks with heavy inter-task communication to the same processor, or placing large job first. [12]

Approximate solutions use the same computational methods as optimal solutions that are within an acceptable range, according to an algorithm-dependent metric. Approximate and optimal algorithms employ techniques based on one of four computational approaches: enumeration of all possible solutions, graph theory, mathematical programming, or queuing theory. In the taxonomy, the sub tree appearing below optimal and approximate in static branch is also present under the optimal and approximate nodes on the dynamic branch, [12]

II. COMPARISON BETWEEN DIFFERENT CLASSES OF ALGORITHMS

A. Local versus Global

At the highest level, we may distinguish between local and global scheduling. Local scheduling in involved with the assignment of processes to the time-slices of a single processor. Global scheduling is the problem of deciding where to execute a process, and the job of local scheduling is left to the operating system of the processor to which the process is ultimately allocated. This allow the processors in a multiprocessor increased autonomy while reducing the responsibility (and consequently overhead) of the global scheduling mechanism. Note that this does not imply that global scheduling must be done by single central authority , but rather , we view the problems of local and global scheduling as separate issues , and (at least logically) separate mechanism are not work solving each[1;12]

B. Static versus Dynamic

The next level in the hierarchy is a choice between static and dynamic scheduling. This choice indicates the time at which the scheduling or assignment decision is made. In the case of static scheduling, information regarding are made. In the case of static scheduling, information regarding the total mix of processes in the system as well as the independent sub tasks involved in job or task force is assumed to be available by the time the program object modules are linked into load modules. Hence, each executable image in as system has a static assignment to a particular processor, and each time that process image is submitted for execution, it is assigned to that processor. A more relaxed definition of static scheduling may include algorithms that schedule task forces for a particular hardware configuration. Over a period of time, the topology of the system may change, but characteristics describing the task force remain the same. Hence, the scheduler may generate a new assignment of processes to processors to serve as the schedule until the topology changes again. Note here that the static scheduling as used in this paper has the same meaning as deterministic scheduling in and task scheduling. These alternatives terms will not be used, however, in an attempt to develop a consistent set of terms and taxonomy. [12]

C. Optimal versus sub optimal

In the case that all information regarding the state of the system as well as the resources need of a processes are known, an optimal assignment can be made based on some criteria function. Examples of optimizing measures are minimizing total process completion time, maximizing utilization of resources in the system, or maximizing system throughput, In the event that these problems are computationally infeasible,

suboptimal solutions may be tired. Within the realm of suboptimal solutions to the scheduling problem, we may think two general categories [12]

D. Approximate versus Heuristic

The first is to use the same formal computational model for the algorithm, but instead of searching the entire solution space for optimal solution, we are satisfied when we find a 'good' one. We will categorize these solutions as suboptimal-approximate. The assumption that a good solution can be recognized may not be so insignificant, but in the cases where a metric is available for evaluating a solution , this technique can be used to decrease the time taken to find an acceptable solution schedule[12], the factors which determine whether this approach is worthy pursuit include:

- The time required to evaluate a solution
- The ability to judge according to some metric the value of an optimal solution.
- Availability of a mechanism for intelligently pruning the solution space.

The second branch beneath the suboptimal category is labeled heuristic. This branch represents the category of static algorithm which makes the most realistic assumptions about prior knowledge concerning process and system loading characteristics. It also represents the solution to the static scheduling problem which requires the most reasonable amount of time and other system resources to perform their function. The most distinguishing feature of heuristic scheduler is that they make use of special parameters which affect the system in indirect ways. Often, the parameters being monitored are correlated to system performance in an indirect instead of a direct way, and this alternate parameter is much simpler to monitor or calculate. For example, clustering groups of processes which communicate heavily on the same processor and physically separating processes which would benefit from parallelism directly decreases the overhead involved in passing information between processors, while reducing the interface among processes which may run without synchronization with one another. This result has impact on the overall service that user receive, but cannot be directly related to system performance as the user sees it. Hence, our intuition, if nothing else, leads us to believe that talking the aforementioned action when possible will improve system performance. However we may not be able to prove that a first order relationship between the mechanism employed and the desired result exists.

E. Optimal and suboptimal Approximate Techniques

Regardless of whether a static solution is optimal or suboptimal – approximate, there are four basic categories of task allocation algorithms which can be used to arrive at an assignment of processes to processors.

- Solution space enumeration and search
- Graph theoretic
- Mathematical programming
- Queuing theoretic

F. Distributed versus Non –distributed

The next issue involves whether the responsibility for the t5ask of global dynamic scheduling physically reside in a single processor (physically non-distributed) or whether the work involved decisions should be physically distributed among the processors. Here the concern is with the logical authority of the decision – making process [12].

G. Cooperative versus Non cooperative

In non-cooperative case, individual processors act along as autonomous entities and arrive decisions regarding the use of their resources independent of the effect of their decision on the rest of the system. In the cooperative case each processor has the responsibility to carry out his own portion of the scheduling task, but all processors work together to achieve a system wide goal. [12]

H. Adaptive versus Non-adaptive Algorithm

An adaptive allocation algorithm [1, 12] is one in which algorithm and parameters used to implement allocation policy changes dynamically according to the previous and current system behavior. An example of such an adaptive algorithm would be one which curtails all allocation activities as the system load increases above some threshold value.

III. TIME SHARING AND SPACE SHARING POLICIES

An alternate classification of multiprocessor scheduling algorithms is described below.

Job scheduling policies decide how many processors to allocate to each job. At this level of decision, the processor scheduler can implement three types of policies: timesharing, space sharing, and gang scheduling. In time sharing, the processor scheduler considers the kernel thread as the unit of scheduling. Space sharing policies consider the job as the unit of scheduling. They divide the processor scheduling policy into two levels: processors to jobs, and processors allocated to jobs to kernel; threads, gang scheduling is a combination of time –sharing and space sharing. It implements a time sharing among jobs, where each job implements a one to one mapping between processors and kernel threads. Fig.1 shows the three processor scheduling options.

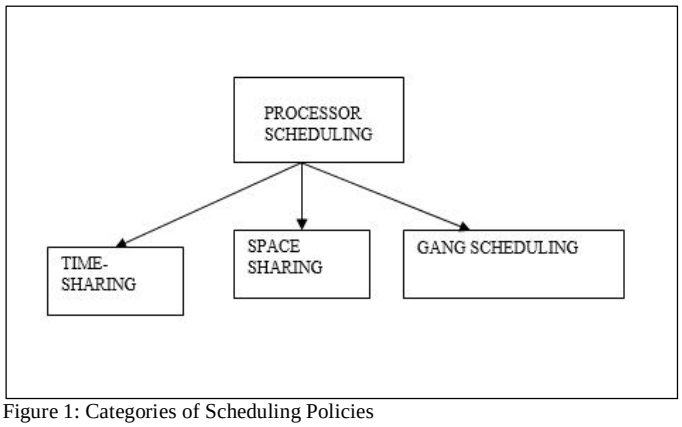

Figure 1: Categories of Scheduling Policies

A. *Time-sharing policies*

Time-sharing policies are the automatic extension from uni processor policies to multiprocessor policies. Time sharing policies do not consider grouping of threads. These policies are usually designed to deal with the problem of many-to-few mapping of threads to processors. There are two main approaches to deal with this problem: local queues (typically one per processor) or a global queue (shared by all the processors.)

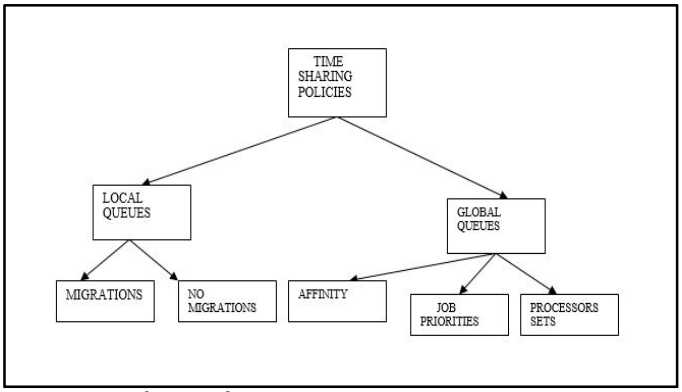

Figure 2: Time-sharing policies

B. *Gang Scheduling*

Gang scheduling is a technique that combines space and time sharing and it was presented as the solution to the problem of static space sharing policies. We define gang scheduling as a scheme that combines three features:

- Application threads are grouped into gang(typically all the threads of the application in the same gang)
- Threads in each gang are executed simultaneously, using a one to one mapping.
- Time slicing is used among gangs (all the threads in the gang are preempted at the same time)

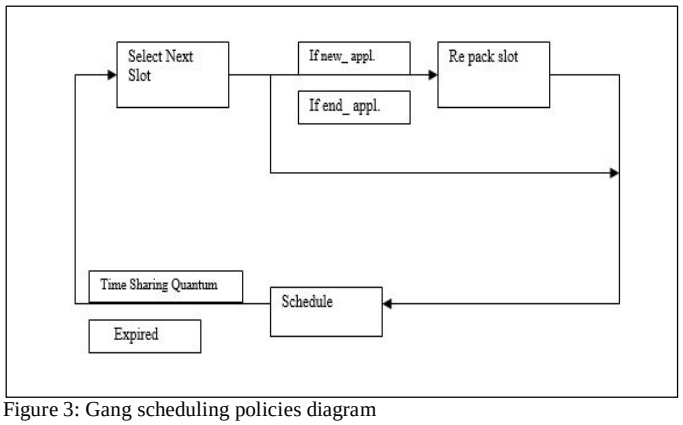

Figure 3: Gang scheduling policies diagram

Fig.3 show the diagram that represents the gang scheduling policies behavior. Periodically, at each time-sharing quantum expiration, the system selects a new slot to execute. A slot is a set of applications that will be concurrently executed. If during the last quantum, any new job has arrived to the system, or any job has finished, the slot will be re-organized. The algorithm to re organize one slot is called re-packing algorithm. The re-packing algorithm is applied to migrate some job from ant other slot to the currently selected. Traditionally, the scheduling phase is traduced by a single dispatch, to allocate as many processors to jobs as they request. Gang scheduling policies mainly differs in the re-packing algorithms applied.

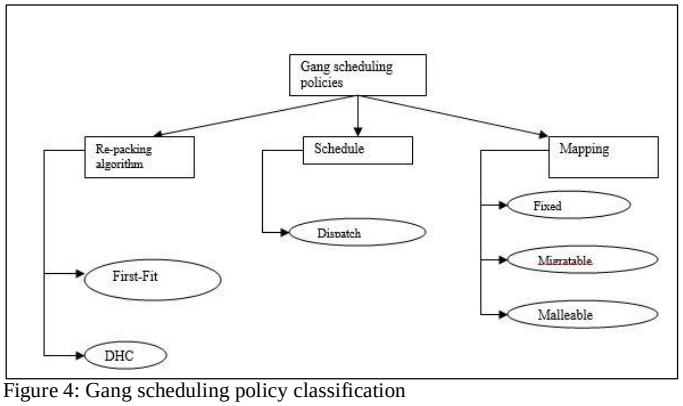

Figure 4: Gang scheduling policy classification

Fig.4 shows taxonomy as a function of the three elements that define a gang scheduling policy: the repacking algorithm, the job mapping, the scheduling algorithm applied.

The simplest version of gang, where threads are always rescheduled in the same set of processors is the most popular one. However there have been proposed more flexible versions one of this is migratable preemptions, where threads are preempted in one set of processors and resumed in another

to reduce the fragmentation. Traditionally, systems that a gang scheduling policy do not include a job scheduling policy because one of the goals of gang scheduling is to reduce the impact of incorrect job scheduling decisions. They show that, even theoretically gang scheduling policies allows the execution of any new job, due resource limitation some jobs remain queued until some running job finishes. Combining backfilling and gang scheduling the queued time can be reduced.

IV. REFRENCES

[1] MANACHER, G.K. Production and stabilization of real-time task schedules. J. ACM 14' 3 (July 1967), 439-465.

[2] McKINNEY, J M. A survey of analytical time-sharing models. Computing Surveys 1, 2 (June 1969), 105-116.

[3] CODD, E.F. Multiprogramming scheduling. Comm. ACM 8, 6, 7 (June, July 1960), 347-350; 413-418.

[4] HELI~ER, J. Sequencing aspects of multiprogramming. J. ACM 8, 3 (July 1961), 426.-439.

[5] GRAHAM, R.L. Bounds for certain multiprocessing anomalies. Bell System Tech. J. 45, 9 (Nov. 1966), 1563-1581.

[6] OSCHNER, B.P. Controlling a multiprocessor system. Bell Labs Record 44, 2 (Feb. 1966), 59-62.

[7] MUNTZ, R. R., AND COFFM~N, E. G., JR. Preemptive scheduling of real-time tasks on multiprocessor systems. J. ACM 17, 2 (Apr 1970), 324-338.

[8] BERNSTEIN, A. J., AND SHARP, J.C. A policy-driven scheduler for a time-sharing system. Comm. ACM 14, 2 (Feb. 1971), 74-78.

[9] LAMPSON, B.W. A scheduling philosophy for multiprocessing systems. Comm. ACM 11, 5 (May, 1968), 347-360.

[10] MARTIN, J. Programming Real-Time Computer Systems, Prentice-Hall, Englewood Cliffs, N.J., 1965.

[11] JIRAUCH, D.H. Software design techniques for automatic checkout. IEEE Trans. AES-$, 6 (Nov. 1967), 9. MARTIN.

[12] Liu, C.L. Scheduling algorithms for hard-real-time multiprogramming of a single processor. JPL Space Programs Summary 37-60, Vol. II, Jet Propulsion Lab., Calif. Inst.of Tech., Pasadena, Calif., Nov. 1969.