Scheduling in Parallel and Distributed Computing Systems

Srishti Srivastava and Ioana Banicescu

Abstract Recent advancements in computing technology have increased the complexity of computational systems and their ability to solve larger and more complex scientific problems. Scientific applications express solutions to complex scientific problems, which often are data-parallel and contain large loops. The execution of such applications in parallel and distributed computing (PDC) environments is computationally intensive and exhibits an irregular behavior, in general due to variations of algorithmic and systemic nature. A parallel and distributed system has a set of defined policies for the use of its computational resources. Distribution of input data onto the PDC resources is dependent on these defined policies. To reduce the overall performance degradation, mapping applications tasks onto PDC resources requires parallelism detection in the application, partitioning of the problem into tasks, distribution of tasks onto parallel and distributed processing resources, and scheduling the task execution on the allocated resources. Most scheduling policies include provisions for minimizing communication among application tasks, minimizing load imbalance, and maximizing fault tolerance. Often these techniques minimize idle time, overloading resources with jobs and control overheads. Over the years, a number of scheduling techniques have been developed and exploited to address the challenges in parallel and distributed computing. In addition, these scheduling algorithms have been classified based on a taxonomy for an understanding and comparison of the different schemes. These techniques have broadly been classified into static and dynamic techniques. The static techniques are helpful in minimizing the individual task's response time and do not have an overhead for information gathering. However, they require prior knowledge of the system and they cannot address unpredictable changes during runtime. On the other hand, the dynamic techniques have been developed to address unpredictable changes, and maximize resource utilization at the cost of information gathering overhead. Furthermore, the scheduling algorithms have also been characterized as optimal or sub-optimal,

S. Srivastava (✉) · I. Banicescu
University of Southern Indiana, Evansville, IN, USA

Mississippi State University, Starkville, MS, USA
e-mail: fsrishti@usi.edu; ioana@cse.msstate.edu

S. K. Prasad et al. (eds.), *Topics in Parallel and Distributed Computing*,
https://doi.org/10.1007/978-3-319-93109-8_11

cooperative or non-cooperative, and approximate or heuristic. This chapter provides content on scheduling in parallel and distributed computing, and a taxonomy of existing (early and recent) scheduling methodologies.

- **Relevant core courses:** DS/A, ParAlgo, DistSystems.
- **Relevant PDC topics:** shared memory (C), distributed memory (C), data parallel (C), parallel tasks and jobs (K), scheduling and mapping (C), load balancing (C), performance metrics (C), concurrency (K), dependencies (K), task graphs (K).
- **Learning outcomes:** The chapter provides an introduction of scheduling in PDC systems such that it can be easily understood by undergraduate students, who are exposed to this topic for the first time. The chapter is intended to provide learning to undergraduate students, who are beginners in the field of high performance computing. Therefore, the goal of this book chapter is to present an overview of scheduling in parallel and distributed computing. Using the knowledge from this chapter, students are expected to understand the basics and importance of scheduling in parallel and distributed computing, understand the difference between different classes of scheduling algorithms and the computational scenarios for their application, and be able to compare different scheduling strategies based on various performance metrics, such as execution time, overhead, speedup, efficiency, energy consumption, and others. In addition, a number of useful resources related to scheduling in PDC systems have been provided for instructors.
- **Context for use:** The material is designed for being incorporated into core courses such as, data structures and algorithms (DS/A), or advanced courses such as, parallel algorithms (ParAlgo), and distributed systems (DistSystems). The material is intended for students who already have an understanding of the basic concepts and terminology of parallel and distributed computing systems.

Introduction

The scheduling problem has been formulated with several definitions across many different fields of application. The problem of job sequencing in manufacturing systems forms the basis for scheduling in parallel and distributed computing systems, and is also recognized as one of the original scheduling problems. Similar to the job sequencing problem in a manufacturing process, a scheduling system is comprised of a set of consumers, a set of resources, and a scheduling policy. A basic scheduling system is illustrated in Fig. 1, where a task in a computer program, a bank customer, or a factory job are examples of consumers, and a processing element in a computer system, a bank teller, or a machine in a factory are examples of resources in a scheduling system. A scheduler acts as an intermediary between the consumers and the resources to optimally allocate resources to consumers according to the best available scheduling policy [1].

Fig. 1 A basic scheduling framework

In parallel and distributed computing, multiple computer systems are often connected to form a multi-processor system. The network formed with these multiple processing units can vary from being tightly coupled high speed shared memory systems to relatively slower loosely coupled distributed systems. Often, processors communicate with each other by exchanging information over the interconnection structure. One of the fundamental ideas behind task scheduling is the proper distribution of program tasks among multiple processors, such that the overall performance is maximized by reducing the communication cost. Various task scheduling approaches have a trade-off, between performance and scheduling overhead, associated with them for different applications in parallel and distributed computing. A solution to a scheduling problem determines both the allocation and the execution of order of each task. If there is no precedence relationship among the tasks, then the scheduling problem is known as a task allocation problem [1].

Scheduling is a feature of parallel computing that distinguishes it from sequential computing. The Von Neumann model provides generic execution instructions for a sequential program, where a processor fetches and executes instructions one at a time. As a parallel computing analogy to the sequential model, parallel random access memory (PRAM) was formulated as a shared memory abstract machine [2, 3]. However, no such practical model has yet been defined for parallel computing. Therefore, many different algorithms have been developed for executing parallel programs on different parallel architectures. Scheduling requires allocation of parallel parts of an application program onto available computational resources such that the overall execution time is minimized. In general, the scheduling problem is known to be NP-Complete [4–6]. Therefore, a large number of heuristics have been developed towards approximating an optimal schedule. Different heuristics are applicable in different computational environments depending on various factors, such as, problem size, network topology, available computational power, and others. Based on the heuristics a large number of scheduling algorithms have been developed and the performance of these algorithms also vary with the type of computational environment. One of the goals of this chapter is to clarify the differences among scheduling algorithms, and their application domains. In general, during the scheduling of program tasks on parallel and distributed computing systems, the tasks are often represented using directed graphs called task graphs and the processing elements and their interconnection network is represented using undirected graphs. A schedule is represented using a timing diagram that consists of a list of all processors and all the tasks allocated to every processor. The tasks are ordered on a processor by their starting times [1].

The rest of the chapter is organized as follows. An overview of mapping algorithms onto parallel computing architectures is described in section "Mapping Algorithms onto Architectures". A detailed taxonomy of scheduling in parallel and distributed computing is explained in section "A Scheduling Taxonomy". A discussion of the recent trends in scheduling in parallel and distributed computing systems is given in section "Examples of Recent Trends in Scheduling".

Mapping Algorithms onto Architectures

The mapping problem consists of assigning the subtasks of an application to processors, so that its execution time is minimized. The basic steps involved are: detecting parallelism, partitioning the problem into independent sub tasks, and scheduling these subtasks on processors. Performing any of these steps in isolation can lead to poor mappings, and therefore, low performance. The parallelism in a program depends on the nature of the problem and the algorithm employed by the programmer. To obtain high performance, a problem must contain sufficient parallelism. Parallelism detection is usually independent of the target machine. In contrast, partitioning and scheduling are highly dependent on architectural parameters, such as the number of processors, processor speed, communication overhead, scheduling overhead, etc. Partitioning attempts to match the granularity of the parallel subtasks to that of the target machine. Scheduling assigns subtasks to processors and orders their execution. The goals of scheduling are to spread the load as evenly as possible to processors and to minimize data communication.

Scheduling schemes can be static or dynamic. In static schemes, subtasks are assigned to processors at compile time either by the programmer or by the compiler. There is no runtime overhead. The disadvantage of static allocation is that the unpredictable runtime execution of subtasks can lead to load imbalance. Dynamic scheduling schemes assign subtasks to processors at runtime. Dynamic assignment of tasks can improve processor utilization, with a trade-off for an additional allocation overhead. Dynamic assignments can be distributed or centralized. In a centralized allocation scheme, there is a pool of tasks that is accessible by all idle processors. Accessing the central pool may be a bottleneck when the number of processors is large. In a distributed allocation scheme, tasks are allocated on the basis of processor negotiation. Distributed allocation may result in sub-optimal load balancing, as scheduling decisions are mainly based on local information.

For some applications, it may be necessary to order the execution of tasks with data dependencies. Executing data dependent tasks on different processors requires costly synchronization and communication. Therefore tasks allocated to different processors should be made as independent of each other as possible. Synchronization and communication overhead depend upon several factors, such as, the algorithm, the subdomain size, and the machine characteristics. An effective scheduling algorithm must ensure that computational tasks with dependencies are mapped onto processors that can communicate with low latency. Therefore, work

allocation is not independent of work partitioning. Mapping should, thus, consider the communication topology during the partitioning step. This leads to a need for a close match between the topology of the dependency graph of the tasks and the communication topology of the machine.

Parallelism Detection

An important component for parallel and distributed computing is a technique that detects and schedules the parallelism in a sequential program, possibly by applying code transformations to effectively utilize the system resources. This process of detecting parallelism is done by examining the code for fine grain operations (such as, parallel operations in program statements) and/or coarse grain operations (such as, vector operations or loop parallelization), depending on the target architecture. Coarse grain parallelism is best detected using the program source code while the detection of fine grain parallelism usually requires an intermediate level program representation. Techniques for the detection of both coarse and fine grain parallel operations have been developed to take advantage of various parallel architectures [7].

Coarse grain parallelism found in sequential programs is mainly in the form of vectorizable computations. Considerable research attention has been devoted to the detection of vectorizable loops in Fortran programs. The techniques include the detection of coarse grain parallelism useful in generation of code for loosely coupled multiprocessor systems. Research in the detection and utilization of fine grain parallelism has also received some attention. A technique that has effectively tackled the problem of detecting fine grain parallelism across basic blocks is trace scheduling which uses a control flow graph representation of a program [8].

In general, algorithms for parallelism detection transform the code so that each statement is surrounded by the same number of loops before and after the transformation. Parallelism detection is optimal if, after transformation, each statement is surrounded by a maximal number of parallel loops. The only constraint that a parallelism detection algorithm must respect is that the partial order of operations defined by the dependencies in the program are preserved. Parallelism detection is a wide topic and has been a research topic in the area of compiler optimization [7].

Partitioning

A process or a task is the basic unit of program execution, and a parallel application is one that has multiple processes or tasks actively performing computation at one time. Partitioning is the process of decomposing a serial application into multiple concurrently executing parts. In parallel and distributed computing applications,

task and data parallelism are two of the most commonly referenced parallel patterns [9]. A task parallel application is decomposed into concurrent units that execute separate instructions simultaneously. On the other hand, a data parallel application is decomposed into concurrent units that execute the same instructions on distinct data sets. Moreover, applications in parallel and distributed computing exhibit spatial and temporal patterns indicating their execution in time and space. For instance, the location of a data point in memory represents the spatial index for that application, and the order in which the data points are accessed for application execution represents the temporal index of that application. Different partitioning strategies are developed to distinguish parallel patterns in an application and further employ temporal and spatial partitioning as required. A generic procedure for determining the dimensionality of the instructions and data of an application to prepare it for partitioning, is summarized as follows [10]:

1. Determine what constitutes a single input to define the temporal dimension of the program's data. For some programs an input might be a single reading from a sensor. In other cases an input might be a file, data from a keyboard or a value internally generated by the program.
2. Determine the distinct components of an input to define the spatial dimension of the program's data.
3. Determine the distinct functions required to process an input to define the spatial dimension of the program's instructions.
4. Determine the partial ordering of functions using topological sort on the program dependence graph to define the temporal dimension of the program's instructions.

The problem of building a partition with the smallest partitioning cost is known to be intractable [11]. Therefore, research in this area has been focused on developing approximation algorithms to provide a solution to the partitioning problem.

Task Allocation and Scheduling

Task allocation is a relevant concept in distributed systems. Given a distributed system made up of a number of processing elements connected together using an interconnection network and a distributed application consisting of communicating tasks, allocation techniques assign tasks to processing elements, to optimize the execution of the application as a whole. Task allocation is considered when there is no precedence among the tasks forming a program or an application [1]. Scheduling is an ordering of the execution of the application tasks on the available processing elements. Often, task allocation and scheduling are used interchangeably and are considered to be performed together. Moreover, scheduling is considered to encompass the previous steps of parallelism detection, partitioning, and task allocation.

There are four components in any scheduling system: the target machines, the parallel tasks (defined as a set of sequential tasks, where different tasks can be executed in parallel if there are no dependencies), the generated schedule, and a performance criterion. The following mathematical description, of these four components of a scheduling system, has been adopted from [1].

Target Machine

The target machine is assumed to be made up of m heterogeneous processing elements connected using an arbitrary interconnection network. Each processing element can run one task at a time and all tasks can be processed by any processing element. Formally, the target machine characteristics can be described as a system $(P, [P_{ij}], [S_i], [I_i], [B_i], [R_{ij}])$ as follows:

1. $P = \{P_1, \cdots, P_m\}$ is a set of processors forming the parallel architecture. P_{ij} is an $m \times m$ interconnection topology matrix of processors as its rows and columns, and each matrix element represents a link between corresponding processors.
2. $S_i, 1 \leq i \leq m$, is the speed of processor P_i.
3. $I_i, 1 \leq i \leq m$, is the startup cost of initiating a message on processor P_i.
4. $B_i, 1 \leq i \leq m$, is the startup cost of initiating a process on processor P_i.
5. R_{ij} is the transmission rate over the link connecting two adjacent processors P_i and P_j.

The connectivity of the processing elements can be represented using an undirected graph called the target machine graph as illustrated in Fig. 2.

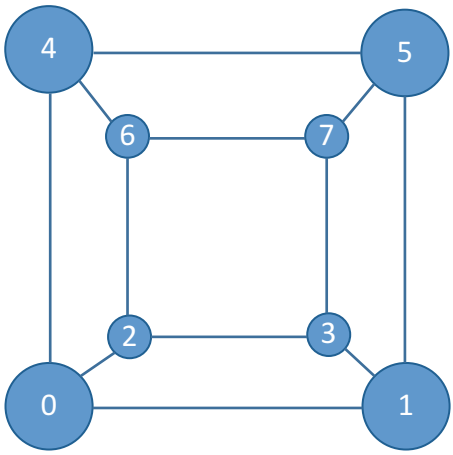

Fig. 2 An example of a target machine with eight processors (m = 8) forming a three dimensional hypercube network.The nodes are labeled with integers indicating the processor numbers

Parallel Application Tasks

A parallel program is modeled as a partially ordered set (poset) $(T, <)$, where T is a set of tasks. The relation $u < v$ implies that the computation of task v depends on the results of the computation of task U, and therefore, task u must be computed for delivering the result to the processor computing the task v. The characteristics of a parallel program can be defined as the system $(T, <, [D_{ij}], [A_i])$ as follows [1]:

1. $T = \{t_1, \cdots, t_n\}$ is a set of application tasks to be executed.
2. $<$ is a partial order defined on T, which specifies the operational precedence constraints.
3. $[D_{ij}]$ is an $n \times n$ communication data matrix, where $D_{ij} \geq 0$ is the amount of data required to be transmitted from task t_i to task t_j.
4. $[A_i]$ is an n-length vector specifying the computational requirements of a task t_i in terms of number of instructions.

The ordered tasks are represented using a directed acyclic graph, which is called a task graph. A directed edge, (i, j), between two tasks t_i and t_j indicates that t_i must be completed before a processor starts executing t_j. An example of a task graph is illustrated in Fig. 3.

Given a parallel program model in the form of a task graph and a description of the target machine, task execution time (T_{ij}) and communication delay $(C(i_1, i_2, j_1, j_2))$, between two processors j_1, j_2 executing tasks i_1, i_2 respectively, can be calculated as follows [1]:

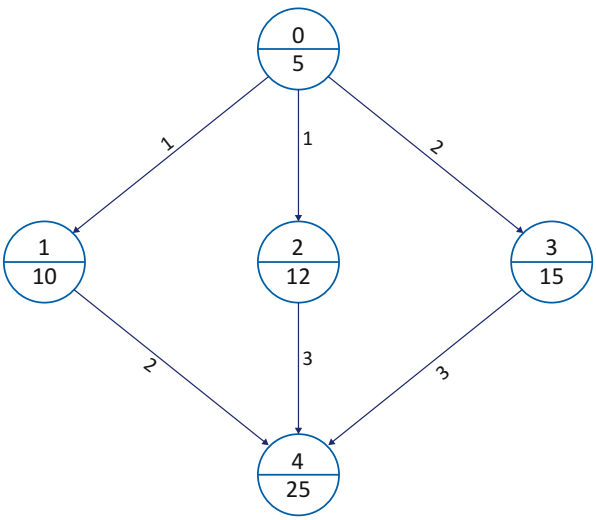

Fig. 3 A task graph with five tasks represented as nodes showing task numbers and task execution times (for example, milliseconds), and directed edges, indicating the order of execution of tasks, labeled with communication costs

$$T_{ij} = \frac{A_i}{S_j} + B_j \tag{1}$$

$$C(i_1, i_2, j_1, j_2) = \frac{D_{i_1 i_2}}{R_{j_1 j_2}} + I_{j_1} \tag{2}$$

The Schedule

Given a task graph $G = (T, A)$ for a target machine consisting of m processors, a schedule is a function f that maps each task to a processor at a specific starting time. A schedule $f(v) = (i, t)$, indicates that a task $v \in T$ is scheduled to be processed by processor p_i starting at time $= t$ units. No two tasks can have equal scheduling function. If $v < u$, where $v, u \in T$ and $f(v) = (i, t_1)$, $f(u) = (j, t_2)$, then $t_1 < t_2$. A schedule is considered feasible if it preserves all task precedence relations and communication restrictions. A Gantt chart is used to represent a schedule with task start and finishing times [1]. An example of a system that takes as input the task graph and the target machine representation, and gives out a Gantt chart representing the schedule as an output is illustrated in Fig. 4.

Performance Measures

The primary goal for scheduling in parallel and distributed computing systems is to achieve load balancing and to minimize the overall application execution time. The performance measure used to achieve this goal is the parallel execution time. The scheduling objective then is to minimize the parallel execution time for minimizing the overall completion time of an application. This, in turn, requires the minimization of the overall schedule length. Given a task graph $G = (T, A)$, the length of a schedule is the maximum finishing time of any task belonging to G. Formally [1],

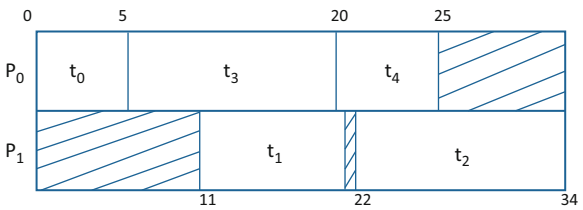

Fig. 4 A Gantt chart representing a schedule for the task graph shown in Fig. 3 on a machine with two processors P_0 and P_1. The shaded area represents the waiting time for each processor based on the task communication delays, assuming the tasks are initially located at processor P_1

$$length(f) = t_{max}, \text{ where } t_{max} = max\{t + T_{ij}\} \text{ and } f(i) = (j, t) \quad \forall i \in T, 1 \leq j \leq m \tag{3}$$

A Scheduling Taxonomy

Parallel and distributed computing has increasingly gained capacity to include a large range of applications. However, the power of a parallel and distributed computation can only be exploited to its full potential with efficient management and allocation of system resources relative to the computational load of the system. This motivation led to a large number of research, which focused on proposing solutions, in the form of scheduling techniques, to solve the problem of resource management in parallel and distributed computing systems. However, this has resulted in the development of various scheduling methodologies leading to the use of inconsistent terminology, problem formulations, and assumptions. Different techniques have been developed for optimizing different performance goals that used different performance metrics. Therefore, to unify the vast number of available scheduling methodologies for parallel and distributed computing, under a common, uniform set of terminology, Casavant and Kuhl [12] proposed a taxonomy that allows the classification of distributed scheduling algorithms according to a common and manageable set of salient features. This section details upon the proposed taxonomy along with a discussion on scheduling at global or system level, and at local or operating system level.

As already described in the previous section, the scheduling problem consists of three main components: (i) consumer(s), (ii) resource(s), and (iii) scheduling policy. Often, there is an assumption in parallel and distributed computing that considers a slight difference in the terms scheduling and allocation. Allocation is viewed in terms of resource allocation from the perspective of a resource, and scheduling is viewed from the perspective of a consumer in a computing system. Therefore, it is often assumed that allocation and scheduling are terms that exhibit a similar general mechanism from different viewpoints. Considering the three components, a scheduling system is evaluated via (1) performance, and (2) efficiency. Performance in a scheduling system is directly related to consumer satisfaction, which depends on how the scheduler allocates resources to process the consumer demands. Efficiency is measured in terms of the overhead and the cost to access the required resource.

There are two kinds of classification schemes for categorizing the scheduling algorithms: (i) hierarchical classification, and (ii) flat classification. The taxonomy presented in [12] is based on a hierarchical classification. However, a hierarchical classification does not capture all the issues in a scheduling system. Therefore, a flat classification that covers a number of scheduling parameters, which are not considered in a hierarchical scheme.

Hierarchical Classification

A tree based hierarchical classification of the taxonomy in [12] is illustrated in Fig. 5.

(a) *Local and global scheduling*: Local scheduling is performed at the operating system (OS) level and manages the assignment of tasks or processes to the time-slices of a single processor. Global scheduling is done at system level and provides a mechanism for allocating application tasks onto available processing elements. The classification discussed below has been developed for global scheduling techniques. Local scheduling will be discussed in more detail later in this section.

(b) *Static versus dynamic*: a choice between static and dynamic scheduling indicates the time at which the scheduling or allocation decisions are to be determined. Static scheduling algorithms are based on the assumption that the information regarding the application tasks, processes within these tasks, and the characteristics of the processing elements are available before the scheduling decision is made. Hence, each application task has a static assignment to a specific processor. Moreover, every time the scheduler encounters the same task, it assigns the task to that specific processor. Therefore, static scheduling algorithms are developed for a particular system configuration. Further, the scheduler may generate a new static assignment of tasks to processors, if the system topology or the task configurations change over a period of time. Static scheduling algorithms are also referred to as deterministic scheduling algorithms. Dynamic scheduling algorithms are based on a more realistic assumption that little or no a priori knowledge is available about the resource requirements of an application task, or about the computational environment in which the application will execute during its lifetime. In dynamic scheduling, an allocation decision is not made until the application tasks begin execution in the dynamic computational environment.

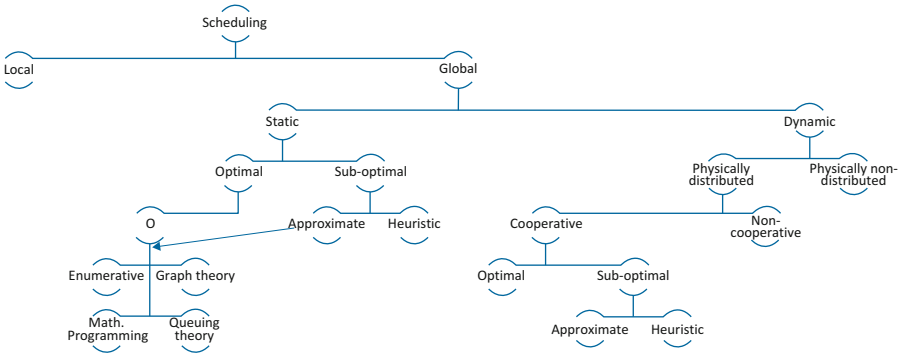

Fig. 5 Hierarchical classification based taxonomy for distributed scheduling algorithms [12]

(c) *Optimal versus sub-optimal*: In static scheduling, where complete information regarding the state of the computational system, and the resource requirements of application tasks are known a priori, optimal scheduling can be achieved based on some optimization function, such as, a function for minimizing the parallel execution time, a function for maximizing resource utilization, or a function for maximizing system throughput. However, for a different case of static scheduling, where some system parameters are computationally infeasible, suboptimal scheduling algorithms are more useful. Suboptimal scheduling algorithms are further categorized as approximate and heuristic algorithms, which are discussed next. Further, static optimal and static suboptimal-approximate scheduling is further categorized to employ the following techniques:

- Solution space enumeration and search.
- Graph theory
- Mathematical programming
- Queuing theory

(d) *Approximate versus heuristic*: Approximate solutions settle for a "good enough" solution as soon as it can be obtained, instead of searching the entire solution space for an optimal solution. Such solutions are often based on the assumption that a good solution can be recognized with minimal overhead. Moreover, in cases, where a metric is available for evaluating a solution that is obtained using approximate algorithms, result in decreased overhead time that is required to obtain the first acceptable schedule. The factors determining when an approximate algorithm should be used are: (i) availability of a function to evaluate a solution, (ii) time required to evaluate a solution using the function, (iii) availability of a metric to calculate the value of a solution, and (iv) availability of a mechanism for efficiently reducing the search space. The other suboptimal category belongs to heuristic-based algorithms. These are static algorithms, which are based on realistic assumptions regarding prior knowledge about the application and system characteristics. Unlike approximate algorithms, heuristic algorithms provide solutions to static scheduling problems, which require an exhaustive search of the solution space and obtain a solution in a reasonable amount of time. Often, the parameter being monitored for obtaining a solution is correlated to system performance in an indirect manner, and is easier to calculate than the actual performance of the system. Tuning the monitored parameter results in an impact on the overall application performance. However, quantitatively, the parameter tuning can not be directly related to system performance from an application viewpoint. Therefore, heuristic algorithms are based on the assumption that certain actions, on a system parameter, could result to an improved system performance. Although, a first-order relationship between the algorithm actions and the desired results may not be proved for existence.

(e) *Distributed versus non-distributed*: This classification has been categorized under dynamic scheduling algorithms. In dynamic scheduling, the decision

for assigning tasks to processors is made during runtime. This classification categorizes dynamic scheduling techniques that either distribute the responsibility of assignment decisions among several processors (physically distributed approach), or that use a single processor for the work involved in making scheduling decisions (physically non-distributed approach). Therefore, this classification distinguishes between dynamic scheduling techniques, based on the logical authority of the decision-making process for task allocation.

(f) *Cooperative versus non-cooperative*: this classification distinguish between dynamic scheduling techniques, which target cooperation between the distributed components (cooperative), or the techniques that are developed for systems, where individual processors make decisions independent of the actions of the other processors (non-cooperative). In the non-cooperative case, individual processors are autonomous entities that make decisions for the use of their resources independently, disregarding the effect of their decision on the other processors in the system. In the cooperative case, every processor, in addition to delivering its own scheduling task, is responsible for working with the other processors to achieve a common system-wide goal.

In addition to the attributes that have been categorized using the hierarchical classification, there are a number of other distinguishing characteristics of scheduling in parallel and distributed systems that are not captured under any branch of the tree-structured taxonomy [12]. These attributes of a scheduling system could be sub categorized under several nodes of the hierarchical structure. Therefore, for the sake of clarity, these characteristics of a scheduling system are represented as a flat classification providing an extension to the existing hierarchical taxonomy.

Flat Classification

(a) *Adaptive versus nonadaptive*: An adaptive scheduling algorithm provides a solution for mapping application tasks to processing elements in the presence of runtime variations in application, algorithm, and system parameters. Such an adaptive scheduler is capable of taking multiple parameters into consideration while formulating a scheduling decisions. An adaptive scheduler modifies the value of a parameter in response to the behavior of the system. Often, such a system is known as a reward based system, where the scheduler receives reward, in the form of system performance, upon an action that it executes in the form of a specific resource assignment. Based on the reward, the scheduler may reformulate its allocation policy by tuning certain system parameters, if those parameters are inconsistent with the desired execution performance. On the other hand, a nonadaptive scheduler does not modify its basic scheduling mechanism due to variations in system activity. A non-adaptive scheduler manipulates the input parameters in the same way regardless of the system behavior.

(b) *Load balancing*: Runtime variations in application, algorithm, or system characteristics, along with poor scheduling decisions, lead to load imbalance among the executing processors in a parallel and distributed computing system. Often, load imbalance is one of the major reasons for performance degradation causing poor resource utilization, increased execution time and decreased system throughput. Recently, scheduling algorithms, focusing on load balancing, have received a great deal of attention. The goal of such scheduling algorithms is to allow processes on all nodes to finish execution at the same rate. A homogeneous system configuration facilitates this approach due to similar characteristics of all the processing elements. A load balancing scheduling system can further be categorized as a centralized system, or a distributed system. In a centralized system, a single master node is responsible for maintaining the information about the workload on the other processing elements. Further, in case of a load imbalance, the central node is responsible for transferring work from a heavily loaded processor to an idle or lightly loaded processor. However, in case of a highly imbalanced environment, the centralized node can become a bottleneck generating a large overhead leading to performance degradation. In a distributed scheduling system, each processor is responsible for maintaining the current state of information about the workload of other processors. In such a system, workload information is circulated over the network at regular time intervals, or as demanded by a processor. The processors are responsible for cooperating such that work can be transferred from a heavily loaded processor to a lightly loaded processor. However, with an increase in the skewed distribution of heavily loaded and idle processors, a distributed approach can generate large communication overhead where processors spend more time transferring work over the network than performing any useful work leading to a degraded performance. Often, load balancing scheduling algorithms are based on the assumption that the workload information, available for making load balancing decisions, is always accurate.

(c) *Bidding*: Scheduling techniques that utilize a bidding approach for assigning tasks to processors, deliver a cooperative scheduler such that enough information is exchanged between task nodes and processor nodes to facilitate an efficient allocation to optimize the overall performance of the system. As a basic mechanism of bidding, each processor node behaves as a manager and a contractor. The manager represents a task in a waiting state which is waiting to be allocated some computational resources. The contractor represents a processor node that is waiting to be allocated to a task node for execution. The manager announces the state of the task waiting for a computational resource. Further, the manager node receives bids from the potential contractor nodes. The amount and type of information exchanged, between the manager and the contractor, are the major factors in determining the efficiency of the bidding-based scheduler. Such a scheduling system is based on the notion of a fully autonomous collection of nodes, such that the manager has the freedom to select autonomously from a collection of bidding computational nodes, and the contractors are allowed to reject any assigned work if it leads to violation

of local performance goals. Cloud brokers are an example of a bidding based scheduling system in cloud computing environments [13].

(d) *Probabilistic*: Probabilistic scheduling algorithms employ random selection of task to processor mapping from a large number of permutations of the available mappings, to reduce the prohibitive amount of time that would otherwise be required for analytically examining the entire solution space. The methodology generates a large number of different schedules via iteratively using the random selection process. Further, the generated set of randomly selected schedules is analyzed for selecting the best schedule from this set. Probabilistic scheduling is based on the assumption that enough variation is introduced by the random selection (using a certain probability distribution) to allow at least one good solution to enter into the randomly chosen set.

(e) *One-time assignment versus dynamic reassignment*: Scheduling methodologies that use one-time assignment technique are often used for jobs in the traditional batch processing environment in a parallel and distributed system. Such techniques generate a fixed schedule at a single point in time. Although many dynamic scheduling techniques use one-time assignment approach, they are considered static such that once a schedule has been generated for task allocation at runtime, no further changes can be made to that schedule. The scheduler generates a mapping of tasks to resources based on the information (in the form of estimated execution times or other system resource demands) provided by the application user. However, the variations that occur in the application and the system parameters at runtime are not considered by the generated schedule. Moreover, a user that understands the characteristics of the underlying computational system and the application, may provide false information to the system for manipulating the system to achieve better results.

Scheduling techniques that employ dynamic reassignment iteratively improve on earlier scheduling decisions. Dynamic reassignment is based on information on smaller computation units that are monitored over a time interval. Such techniques use dynamically created information, available from monitoring resources, to adapt to variations in application and system parameters. Therefore, dynamic reassignment can also be viewed as an adaptive approach for scheduling. Often, such an approach requires migrating tasks among processors generating an overhead. Thus, the use of such techniques should be weighed for trade-off between the generated overhead and the performance gain.

Operating System Scheduling

The classification of the scheduling strategies that have been discussed so far have been designed for global scheduling at system level. However, once the tasks are mapped to a processor, there is a need for a local scheduling mechanism that manages the execution of processes mapped to that processor. Scheduling

at operating system level, also known as process scheduling, is an activity of a process manager that manages process selection, mapping, and removal of a process for a processor, according to a particular local scheduling methodology. Process scheduling is an integral part of operating systems running in the processing elements of parallel and distributed computing systems. A good process scheduling scheme allows multiple processes to be loaded simultaneously into the executable memory and share the CPU using time multiplexing.

During the lifetime of a process, it spends some time executing instructions (computing) and then makes some I/O request, for example, to read or write data to a file or to get input from a user. The period of computation between I/O requests is called a CPU burst. *Interactive processes* spend more time waiting for I/O and generally experience short CPU bursts. A text editor is an example of an interactive process with short CPU bursts. *Compute-intensive processes*, conversely, spend more time running instructions and less time on I/O. They exhibit long CPU bursts. A video transcoder is an example of a process with long CPU bursts. Even though it reads and writes data, it spends most of its time processing that data. A comparative example of an interactive process and a compute-intensive process switching between I/O and CPU burst cycles is shown in Fig. 6.

Almost all programs have some alternating cycle of CPU number crunching and waiting for I/O of some kind. In a simple system running a single process, the time spent waiting for I/O is wasted, and those CPU cycles are lost forever. A scheduling system allows one process to use the CPU while another is waiting for I/O, thereby making full use of otherwise lost CPU cycles. The challenge is to optimize the overall system performance and efficiency, subject to dynamically varying conditions. When the process enters into the system, then this process is put into a job queue. This queue consists of all processes in the system. The operating system also maintains other queues such as device queues. A device queue contains multiple processes waiting for a particular I/O device. Each device has its own device queue. A newly arrived process is put in the ready queue. Processes wait in ready queue for allocating the CPU. Once the CPU is assigned to a process, then that process will execute. To provide good time-sharing performance, the scheduler preempts a running process to let another one run. When an I/O request for a process is complete, the process moves from the waiting state to the ready state and gets placed on the ready queue. The process scheduler is the component of the operating system that is responsible for deciding whether the currently running process should continue running and, if not, which process should run next. There are four events that may occur where the scheduler needs to step in and make this decision:

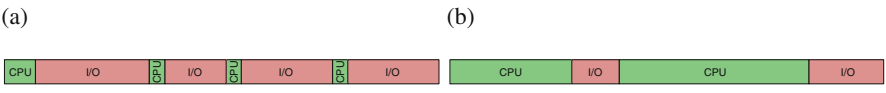

Fig. 6 A comparative example of differences between the I/O and CPU burst cycles of an interactive process versus a compute-intensive process. (**a**) Interactive process. (**b**) Compute-intensive process

1. The current process goes from the running to the waiting state because it issues an I/O request or some operating system request that cannot be satisfied immediately.
2. The current process terminates.
3. A timer interrupt causes the scheduler to run and decide that a process has run for its allotted interval of time and it is time to move it from the running to the ready state.
4. An I/O operation is complete for a process that requested it and the process now moves from the waiting to the ready state. The scheduler may then decide to preempt the currently-running process and move this newly-ready process into the running state.
   A scheduler is a preemptive scheduler if it has the ability to get invoked by an interrupt and move a process out of a running state to let another process run. The last two events in the above list may cause this to happen. If a scheduler cannot take the CPU away from a process then it is a cooperative, or non-preemptive scheduler. Older operating systems, such as Microsoft Windows 3.1 or Apple MacOS prior to OS X, are examples of cooperative schedulers.

A number of local scheduling algorithms are being widely used by different operating systems. There are several performance metrics that form the optimization criteria for selecting the most appropriate scheduling algorithm for a specific computing environment. Following is a list of these performance metrics that play an important role in the selection of a particular process scheduling algorithm:

- CPU utilization – percentage of CPU being used for computational work.
- Throughput – number of processes completed per unit time.
- Turnaround time – time required for a particular process to complete, from submission time to completion.
- Waiting time – time spent by a process in the ready queue.
- Response time – The time taken in an interactive program from the issuance of a command to completion a response to that command.

First come first serve (FCFS) is the most straightforward approach to scheduling processes that are stored in a first-in, first-out (FIFO) ready queue. When the scheduler needs to run a process, it picks the process that is at the head of the queue. This scheduler is non-preemptive. *Round robin (RR)* scheduling is a preemptive version of FCFS scheduling. Processes are dispatched in a FIFO sequence, such that each process is allowed to run for a limited amount of time. This time interval is known as a time-slice or quantum. If a process does not complete within the time slice, the process is preempted and placed at the back of the ready queue. The *shortest remaining time first (SRTF)* scheduling algorithm is a preemptive version of an older non-preemptive algorithm known as *shortest job first (SJF)* scheduling. In SJF, the queue of jobs is sorted by estimated job length so that the smaller processes get to run first. This minimizes average response time. In SRTF, the algorithm sorts the ready queue by the estimated CPU burst time of a process. In *priority scheduling*, each process is assigned a priority based on a pre-defined criteria. A process, in the

ready queue, with the highest priority gets to run next (UNIX-derived systems tend to use smaller numbers for high priorities while Microsoft systems tend to use higher numbers for high priorities). If the system uses preemptive scheduling, a process is preempted whenever a higher priority process is available in the ready queue. For a more detailed study on operating system process scheduling, the reader is referred to the literature in [14].

Examples of Recent Trends in Scheduling

With the evolution of the complexity of parallel and distributed computing, there has been a wide range of development of various scheduling algorithms and methodologies that can cater to the growing needs of the modern computing systems. A few examples of the recent trends in the development of scheduling in parallel and distributed computing will be discussed in this section. The examples have been selected such that they cover multiple classification categories of scheduling from the taxonomy described in the previous section. The examples begin with a description of work that have proposed and compared static, dynamic-nonadaptive, and dynamic-adaptive scheduling techniques employed in traditional high performance computing systems for scientific applications, followed by a discussion of a number of heuristic-based scheduling techniques employed in grid computing systems. Further, an example of scheduling strategies for cloud computing systems, which are defined as one of the modern parallel and distributed computing systems, will be discussed.

Dynamic Load Balancing Via Loop Scheduling in High Performance Computing

High performance computing was developed to serve the interests in the accurate modeling and simulation of various complex phenomena from various scientific areas. The scientific applications are often routines that perform varying number of repetitive computations (in the form of DO/FOR loops) over very large data sets. Moreover, these applications may exhibit irregular behavior leading to differing execution times of each iteration. In scientific applications, a loop iteration (or a chunk of loop iterations) with variable execution time is considered to be a task with varying execution time.

Dynamic loop scheduling (DLS) algorithms provide application-level load balancing of loop iterations, with the goal of maximizing application performance on the underlying system. Many DLS methods are based on probabilistic analyses, and therefore possess the capability to be inherently robust against unpredictable variations in application and system characteristics. A number of DLS algorithms

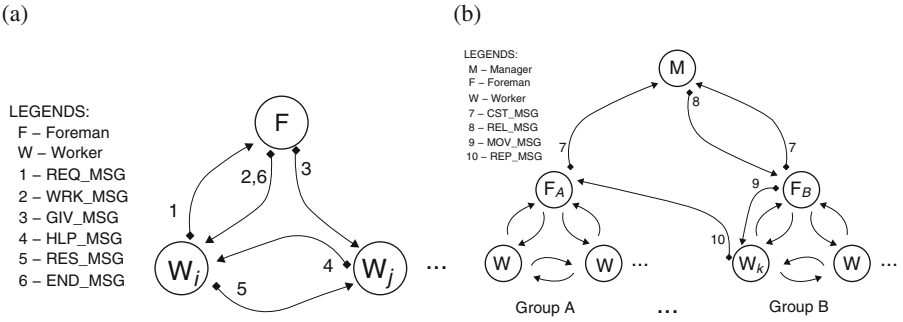

Fig. 7 Dynamic loop scheduling management approaches. (**a**) Centralized management. (**b**) Hierarchical management system

have been proposed in the last decade and have been integrated into several scientific applications, yielding significant performance improvements [15]. The DLS methods are further categorized as *non-adaptive* and *adaptive*. The non-adaptive DLS techniques have been described in a survey presented in [16]. However, the dynamic non-adaptive techniques did not address the unpredictable changes in the computational environment at runtime. Therefore, adaptive DLS techniques were developed to address this problem [17, 18]. Most of the above adaptive methods use a combination of runtime information about the application and the system, to estimate the time the remaining tasks will require to finish execution, in order to achieve the best allocation possible for optimizing application performance via load balancing.

Most loop scheduling methods are developed assuming a central ready work queue of tasks (central management approach), where idle processors obtain chunks of tasks to execute. The scheduling decisions are centralized in the master node, which is also known as the foreman node. However, accessing the foreman may become a bottleneck when a large number of workers attempt to simultaneously communicate with it. To address this bottleneck, a two-level (hierarchical) management strategy is employed, which uses multiple-foremen and partitioned disjoint processor groups of worker nodes. Each processor group executes concurrently independent parts of the problem. Forming processor groups dynamically assists the DLS methods to leverage the best possible application performance on the large-scale platform [19]. Figure 7 illustrates the centralized, and the distributed management approach used in dynamic loop scheduling methods.

Heuristic Scheduling for Grid Computing

Grids computing is a new trend in parallel and distributed computing. Computational grids are distributed systems with independent, and non-interactive compute

intensive workloads. Unlike conventional high performance computing systems such as cluster computing, grid computing is more loosely coupled, heterogeneous, and geographically dispersed. Moreover, scheduling in a grid computing environment is different from scheduling in a traditional computing system, where a scheduler only manages a single local cluster and has control over the cluster resources, whereas a grid scheduler has no control over the distributed resources, and its availability of information about the system state is limited. Scheduling and resource allocation decisions in grid computing systems are approached differently for computational grid versus data grid. The scheduling techniques implemented in a compute grid focuses on managing computational resources, such as, processor compute cycles. In a data grid, the scheduler focuses on managing the distributed data and the related communication over the grid network connecting the distributed geographical locations. The scheduling problem in a grid computing system can be viewed as an optimization problem which is known to be NP-Complete [20]. Therefore, recent research has shown that heuristic techniques are increasingly being used for solving the scheduling optimization problem.

Ant Colony Optimization (ACO) is a heuristic algorithm that employs local search for combinatorial problems. ACO has been used to solve several NP-hard problems such as the traveling salesman problem, graph coloring problem, vehicle routing problem, and others. As a recent study, a modified version of the ACO algorithm, called the Balanced ACO (BACO) algorithm, has been used for grid scheduling to optimize the system makespan [21]. Using this algorithm, the grid scheduler selects a resource for mapping to the job request by finding the largest entry in the Pheromone Indicator (PI) matrix among the available jobs to be executed, where jobs are independent of each other. Another framework that combines the Fuzzy C-Mean clustering ACO algorithm to improve the scheduling in a heterogeneous grid is presented in [22]. Herein, the Fuzzy C-Mean algorithm is used for classification of the jobs into separate classes, and the ACO algorithm maps the jobs to the appropriate resources that are relevant to those classes. A scheduling algorithm for task scheduling using particle swarm optimization (PSO) heuristic for an improved job classification is given in [23]. The heuristic approach is used to map jobs to grid resources based on the calculated task length of a job and the calculated processing power of a grid resource. This method has been developed to optimize resource utilization in a grid environment. Tabu Search (TS) heuristic has also been used in a scheduling technique in grid computing using the GridSim tool in [24]. The basic principle of TS is employ local search techniques after reaching a local optimum and prevent cycling back to previously visited solutions by the use of a storage data structure called Tabu list. Further, TS can be used in conjunction with other heuristic approaches such as genetic algorithm, constraint programming, and integer programming technique, for improved performance results.

Scheduling Advances for Cloud Computing

The advent of cloud computing has revolutionized the concept of parallel and distributed computing. Cloud computing enables the access to computational resources, information, and technology to users as services over the Internet. The services that are provided in a cloud computing environment have been categorized into three main classes: (i) Infrastructure as a Service (IaaS), (ii) Platform as a Service (PaaS), and (iii) Software as a Service (SaaS). These services are provided on demand in a pay-per-use manner via the Internet. Cloud computing differs from traditional computing environments, such as cluster computing and grid computing, as it uses virtualization for resource management. This allows cloud computing resources to be scheduled as cloud services, and are provided to the end-user as a utility [25]. Recently, the concept of a cloud broker has evolved and cloud computing environments are being considered as federated systems that consist of a large number of resources as a federation [13]. However, cloud computing provides a finite pool of virtual on-demand resources, therefore, requiring efficient scheduling and resource allocation techniques that can manage the dynamic and competitive computing environment.

Cloud computing is seen as a three-layered framework consisting of an infrastructure layer, a platform layer, and a software layer. Thus, scheduling methodologies have been proposed for resource management in and between all these layers. A taxonomy of scheduling in the three cloud resource layers has been defined in [26]. The architecture consisting of the three layers, the IaaS, PaaS, and SaaS stacks, and a classification of the scheduling requirements for each of the layers is illustrated in Fig. 8. Scheduling in the software service layer requires delivering software in the form of user applications, tasks, workflows, and others, while optimizing the efficiency and maintaining the QoS requirements. Scheduling in the platform service layer requires mapping virtual resources to physical resources such that there is minimal load balance, and minimized power consumption. Scheduling in the infrastructure service layer requires delivery of physical computational and communication resources to the above two layers for efficient application to resource mapping, with minimal application or virtual machine migration, in a federated cloud computing environment.

Given that cloud computing is still an emerging technology, solutions for scheduling and resource management are fairly recent developments in the field. Some of the solutions to the scheduling problem for different aspects of cloud computing have been proposed as combinatorial solutions in [27–29], and as heuristic approaches in [30–32].

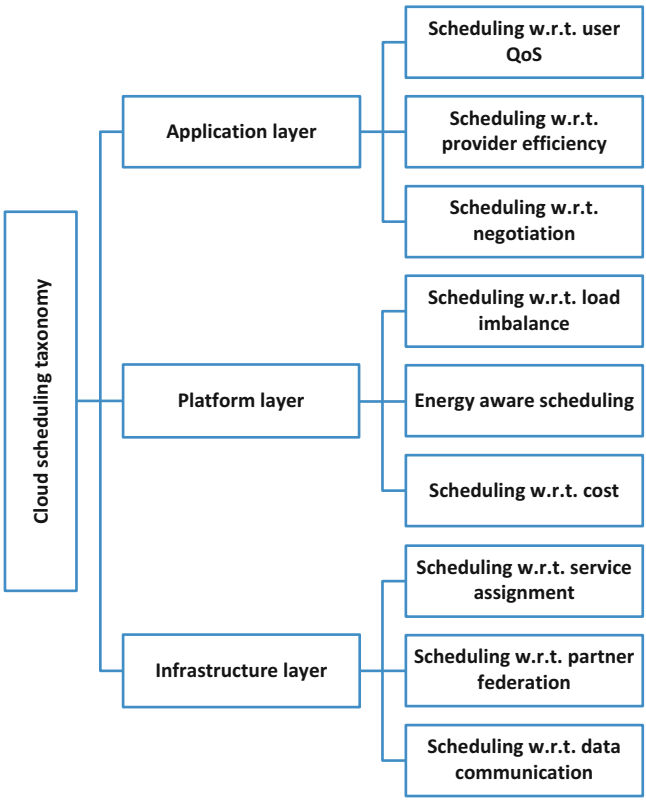

Fig. 8 Taxonomy of the cloud resource scheduling at different service layers with a focus on different scheduling challenges and objectives [26]

Chapter Review

This chapter provides a fundamental description of scheduling in parallel and distributed computing systems. The knowledge presented in here is a result of a survey and collection of information from a number of state-of-the-art work (provided as references) done in this field. Scheduling has been defined as a collective task consisting of the following sub-tasks: detecting parallelism, partitioning the problem into independent sub-tasks, and scheduling these sub-tasks on processors. Often, when scheduling is referred, it is assumed to encompass the afore mentioned sub-tasks. A generic scheduling system is comprised of four components: the target machines, the parallel tasks, the generated schedule, and a performance criterion. Over the years, a number of scheduling techniques have been developed to define the mapping policy for executing applications or tasks in a parallel and distributed computing environment. A taxonomy, proposed in [12], for the classification of various scheduling techniques has been described in section "A

Scheduling Taxonomy". Further, a distinction between application level scheduling and process scheduling at OS level is given via a description of scheduling at global and local level, respectively. A few examples of scheduling in traditional parallel and distributed computing systems, such as clusters and grid, and modern computing systems, such as clouds, have also been discussed to explain the differences in the scheduling approaches and objectives for such systems.

Exercises

1. Conduct a comparison between static and dynamic approaches. Exemplify with some cases, where one approach might be better than the other.
2. Suggest a performance metric that would be most appropriate for each of the following scenario:
   - job scheduling in a manufacturing plant
   - management for an aircraft waiting for landing clearance
   - customers waiting for a teller in a banking system
3. Show an example of a case, where load balancing is more important than minimizing the finishing times of every machine.
4. Discuss the differences between scheduling at global and local levels. How does a poor scheduling decision at one of these levels affect the performance at the other level?
5. The Ready queue of an operating system at a particular time instance is given in Table 1. The behavior of each process (if it were to use the CPU exclusively) is as follows. A process runs for the CPU burst given, then requests an I/O operation that takes 10 ms, then runs for another CPU burst of equal duration to its first CPU burst and then terminates. However, the four processes must share the CPU. Assume that the I/O operations can proceed in parallel. Draw a chart showing the execution of these processes under the round robin policy, with time quantum = 2.
6. Discuss the differences in the objectives and the challenges for scheduling in a cluster computing environment, a grid computing environment, and a cloud computing environment.

Table 1 Ready queue of an operating system with process CPU burst in milliseconds

Process	Next CPU burst
P1	2
P2	3
P3	7
P4	18

References

1. H. El-Rewini, T. G. Lewis, and H. H. Ali, *Task Scheduling in Parallel and Distributed Systems*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1994.
2. N. Immerman, "Expressibility and parallel complexity," *SIAM Journal on Computing*, vol. 18, no. 3, pp. 625–638, 1989.
3. J. C. Wyllie, "The complexity of parallel computations," Cornell University, Tech. Rep., 1979.
4. E. G. Coffman and J. L. Bruno, *Computer and job-shop scheduling theory*. John Wiley & Sons, 1976.
5. O. H. Ibarra and C. E. Kim, "Heuristic algorithms for scheduling independent tasks on nonidentical processors," *J. ACM*, vol. 24, no. 2, pp. 280–289, Apr. 1977. [Online]. Available: http://doi.acm.org/10.1145/322003.322011
6. D. Fernandez-Baca, "Allocating modules to processors in a distributed system," *IEEE Transactions on Software Engineering*, vol. 15, no. 11, pp. 1427–1436, Nov 1989.
7. R. Gupta and M. L. Soffa, "Region scheduling: An approach for detecting and redistributing parallelism," *Software Engineering, IEEE Transactions on*, vol. 16, no. 4, pp. 421–431, 1990.
8. J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Transactions on Computers*, vol. C-30, no. 7, pp. 478–490, July 1981.
9. M. D. McCool, A. D. Robison, and J. Reinders, *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012.
10. H. Hoffmann, A. Agarwal, and S. Devadas, "Partitioning strategies for concurrent programming," *MIT Open Access Articles*, 2009.
11. V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors*. Cambridge, MA, USA: MIT Press, 1989.
12. 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, Feb 1988.
13. R. Mehrotra, S. Srivastava, I. Banicescu, and S. Abdelwahed, "Towards an autonomic performance management approach for a cloud broker environment using a decomposition-coordination based methodology," *Future Generation Comp. Syst.*, vol. 54, pp. 195–205, 2016. [Online]. Available: http://dx.doi.org/10.1016/j.future.2015.03.020
14. A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 9th ed. Wiley Publishing, 2009.
15. S. Srivastava, I. Banicescu, F. M. Ciorba, and W. E. Nagel, "Enhancing the functionality of a gridsim-based scheduler for effective use with large-scale scientific applications," in *2011 10th International Symposium on Parallel and Distributed Computing*, July 2011, pp. 86–93.
16. A. R. Hurson, J. T. Lim, K. M. Kavi, and B. Lee, "Parallelization of doall and doacross loops - a survey," *Advances in computers*, vol. 45, pp. 53–103, 1997.
17. I. Banicescu and V. Velusamy, "Load balancing highly irregular computations with the adaptive factoring," in *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*, April 2002, pp. 12 pp–.
18. I. Banicescu, V. Velusamy, and J. Devaprasad, "On the scalability of dynamic scheduling scientific applications with adaptive weighted factoring," *Cluster Computing*, vol. 6, no. 3, pp. 215–226, 2003.
19. R. Cariño, I. Banicescu, T. Rauber, and G. Rünger, "Dynamic loop scheduling with processor groups." in *ISCA PDCS*, 2004, pp. 78–84.
20. J. D. Ullman, "Np-complete scheduling problems," *J. Comput. Syst. Sci.*, vol. 10, no. 3, pp. 384–393, Jun. 1975. [Online]. Available: http://dx.doi.org/10.1016/S0022-0000(75)80008-0
21. R.-S. Chang, J.-S. Chang, and P.-S. Lin, "An ant algorithm for balanced job scheduling in grids," *Future Generation Computer Systems*, vol. 25, no. 1, pp. 20–27, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X08000848

22. T. Helmy and Z. Rasheed, "Independent job scheduling by fuzzy c-mean clustering and an ant optimization algorithm in a computation grid." *IAENG International Journal of Computer Science*, vol. 37, no. 2, 2010.
23. S. Selvarani and G. S. Sadhasivam, "Improved job-grouping based pso algorithm for task scheduling in grid computing," *International Journal of Engineering Science and Technology*, vol. 2, no. 9, pp. 4687–4695, 2010.
24. M. Yusof, K. Badak, and M. Stapa, "Achieving of tabu search algorithm for scheduling technique in grid computing using gridsim simulation tool: multiple jobs on limited resource," *Int J Grid Distributed Comput*, vol. 3, no. 4, pp. 19–32, 2010.
25. R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility," *Future Generation computer systems*, vol. 25, no. 6, pp. 599–616, 2009.
26. Z.-H. Zhan, X.-F. Liu, Y.-J. Gong, J. Zhang, H. S.-H. Chung, and Y. Li, "Cloud computing resource scheduling and a survey of its evolutionary approaches," *ACM Computing Surveys (CSUR)*, vol. 47, no. 4, p. 63, 2015.
27. B. Speitkamp and M. Bichler, "A mathematical programming approach for server consolidation problems in virtualized data centers," *IEEE Transactions on Services Computing*, vol. 3, no. 4, pp. 266–278, Oct 2010.
28. H. N. Van, F. D. Tran, and J. M. Menaud, "Performance and power management for cloud infrastructures," in *2010 IEEE 3rd International Conference on Cloud Computing*, July 2010, pp. 329–336.
29. T. A. L. Genez, L. F. Bittencourt, and E. R. M. Madeira, "Workflow scheduling for saas / paas cloud providers considering two sla levels," in *2012 IEEE Network Operations and Management Symposium*, April 2012, pp. 906–912.
30. V. Roberge, M. Tarbouchi, and G. Labonte, "Comparison of parallel genetic algorithm and particle swarm optimization for real-time uav path planning," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 1, pp. 132–141, Feb 2013.
31. M. A. Rodriguez and R. Buyya, "Deadline based resource provisioning and scheduling algorithm for scientific workflows on clouds," *IEEE Transactions on Cloud Computing*, vol. 2, no. 2, pp. 222–235, April 2014.
32. Y. L. Li, Z. H. Zhan, Y. J. Gong, J. Zhang, Y. Li, and Q. Li, "Fast micro-differential evolution for topological active net optimization," *IEEE Transactions on Cybernetics*, vol. 46, no. 6, pp. 1411–1423, June 2016.