# APPLICATIONS OF QUEUING THEORY IN PARALLEL AND DISTRIBUTED PROCESSING

**Assistant Felician ALECU**
**Economic Informatics Department**
**Academy of Economic Studies Bucharest**

**Abstract**

Queuing theory is the mathematical study of waiting lines and it is very useful in telecommunications, traffic control, determining the sequence of computer operations, predicting computer performance, health services, airport traffic, airline ticket sales, mining industry, manufacturing systems.

According to the dictionary, a queue is a file or line of persons. The etymology is from the Latin coda, which means tail. As a verb, "to queue" means to form a line while waiting for something. According with Saaty, a queue, or a waiting line, involves arriving items that wait to be served at the facility that provides the service they seek.

The execution queue of a parallel system is managed by the scheduler who allocates tasks to available processors as well as implements the queue order and priorities. A grid network is considered like a single center and the service demands are scheduled on the individual computers from the grid. The most common resource used in grid networks is computing cycles provided by the processors of the machines on the grid.

**Keywords**

Queuing theory, queuing system, queuing network, spatially distributed queues, parallel systems efficiency, scheduling, parallel and processing.

## 1. INTRODUCTION TO QUEUING THEORY

Queuing theory, also known as the theory of congestion, is the branch of operational research that explores the relationship between demand on a service system and the delays suffered by the users of that system.

A queuing system is a generic model that comprises three elements: a user source, a queue and a service facility that contains one or more (possibly an infinite number of) identical servers in parallel. Each user of the queuing system passes through the queue where he may remain for a period of time (positive, possibly zero) and than is processed by a single server because of the parallel arrangement of the servers. Once a user has left the server, after obtaining the service, the user is considered to have left the queuing system as well.

A queuing system is formed from three generic elements (Figure 1):

1. The arrival process of users in the system;
2. The order in which users obtain access to the service facility, once they join the queue;
3. The service process.

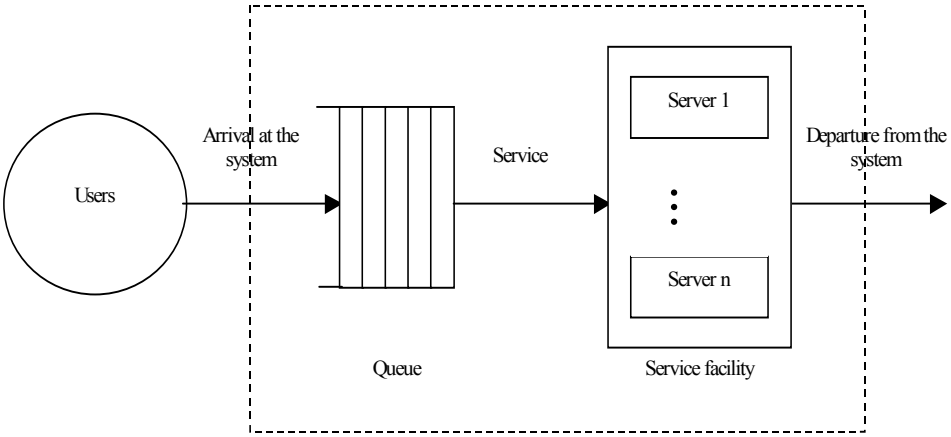


**Figure 1 – Queuing system**

A queuing network is a set of interconnected queuing systems. The user sources for some of the queuing systems in the network may be other queuing systems in the same network (Figure 2). To describe a queuing network, further information must be provided on how the queuing systems are interconnected, how they interact and how users are assigned to the queuing systems.

Busy period service delays must occur in case of the services that respond to unpredictable demands whose time and location of occurrence are governed by probabilistic laws. The cost of providing sufficient capacity to avoid all delays under all circumstances would be impossible. The role of the analysis is to design service systems that achieve an acceptable balance between system operating costs and the delays suffered by the users of that system.

The system capacity is another important parameter in the description of a queuing system. It indicates the maximum number of users that can be in the service facility and in the queue at any time. On the other hand, the queue capacity indicates the maximum number of users that can be in the queue alone.

It is obvious that there are countless variations of queuing systems. This is why a code has been used to describe the best-understood queuing systems. The code has the form *A/B/m*, where *A* and *B* are letter symbols that indicate the probability distribution of arrival and service times and *m* is the number of identical parallel servers from the queuing system ( $m \in [1, \infty)$ ).

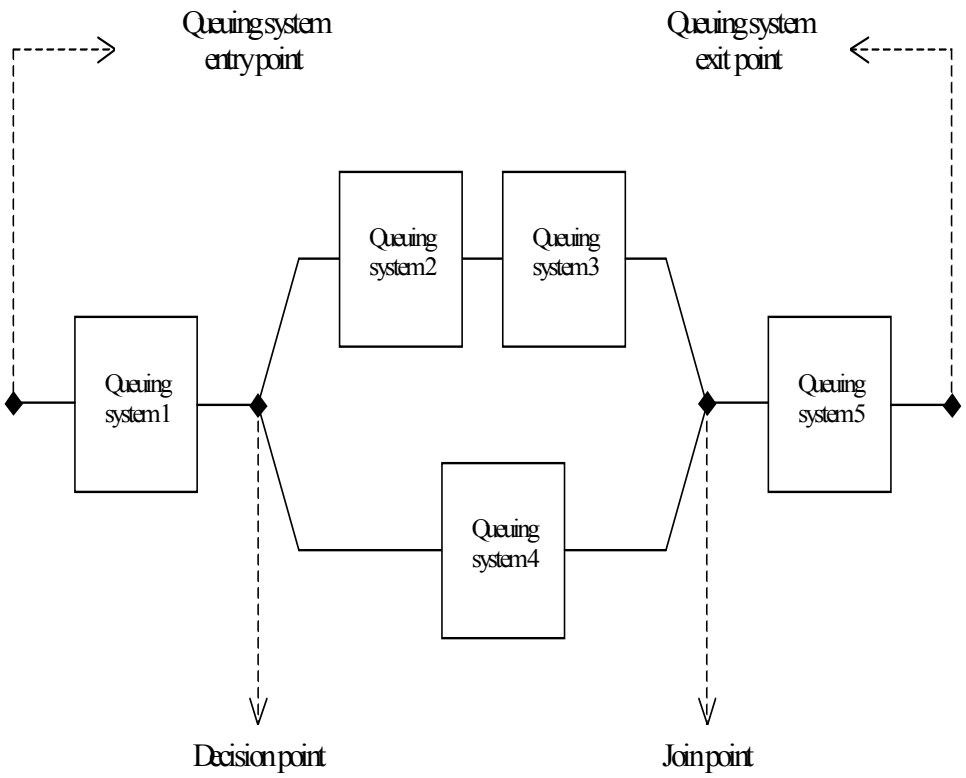


**Figure 2 – Queuing network**

Table 1 shows the standard code letters used to express the probability distributions in queuing theory.

| Symbol | Description |
|---|---|
| M | Poisson distribution |
| D | Deterministic distribution |
| $E_k$ | Erlang distribution |
| $H_k$ | Hyper exponential distribution |
| G | General distribution |

**Table 1– Probability distributions**

Table 2 contains the abbreviations of the most common queue disciplines.

| Abbreviation | Description |
|---|---|
| FIFO | First in, first out |
| FCFS | First come, first served |
| LIFO | Last in, last out |
| LCFS | Last come, first served |
| SIRO | Service in random order |

**Table 2 – Queue disciplines**

A stationary located server (like an ATM) where the users queue up to be served represents the classical perspective of a queuing system. In our days, it is very common to have systems where users remain stationary at separated locations while the servers visit them and provide the required service. Such a system is called spatially distributed queue. An example of a spatially distributed queuing system is the ambulance service.

## 2. WAITING QUEUES APPLIED IN PARALLEL PROCESSING

To obtain a faster execution time, a parallel program is usually divided into independent tasks that will be executed concurrently. Two tasks are independent each other if the same result is obtained if the tasks are executed sequentially in any order or in parallel.

Any computer, sequential or parallel, implements waiting queues to properly manage the access to the system shared resources like processor, memory, peripheral devices and so on. Usually, there is a queuing system for each shared resource from the system. The resource represents the server and the tasks that try to access the resource concurrently form the users of the waiting system. If the shared resource is the processor, the waiting queue is known as execution queue. Execution queues make the transition from the sequential programming to the parallel one.

Basically, an execution queue is a list containing ready to be executed processes. If this list is located in the system shared memory, the access to the list has to be done in a critical section in order to avoid the possibility of an execution of a process on two different processors in the same time. The mutual exclusion is used to access the shared resources and it is implemented using the classical mechanisms from uniprocesor systems like barriers, semaphores, monitors, etc. If a processor enters the critical section, it has exclusive access to the list of the ready to be executed processes. Based on the queue discipline and priorities, the processor will pick up a process from the list. Than, it will leave the critical section and will execute the selected process.

A parallel computer is a set of processors that are able to work cooperatively to solve a computational problem. Based on this definition, a parallel computer could be a supercomputer with hundreds or thousands of processors or could be a network of workstations.

According to Tanenbaum, a distributed system is a set of independent and interconnected computers that appear to the user as a single one. The computers can communicate and collaborate each other using software and hardware interconnecting components. The computers have to be independent and the software has to hide individual computers to the users. Multiprocessors (MIMD computers using shared memory architecture), multicomputers connected through static or dynamic interconnection networks (MIMD computers using message passing architecture) and workstations connected through local area network are examples of such distributed systems.

A distributed operating system is an operating system used on a distributed system. It is the extension for multiprocessor architectures of multitasking and multiprogramming operating systems. A distributed operating system is a special kind of software used on a distributed system. It manages the system-shared resources used by multiple processes, the process scheduling activity (how processes are allocated on available processors), the communication and synchronization between running processes and so on.

Multiprocessors are known as tightly coupled systems and multicomputers as loosely coupled systems. The software for parallel computers could be also tightly coupled or loosely coupled. Combining loosely and tightly coupled hardware and software we can identify four distributed operating systems categories but the loosely coupled software and tightly coupled hardware case is not met in practice (Table 3).

| | | Software Type | |
|---|---|---|---|
| | | loosely coupled | tightly coupled |
| Hardware Type | loosely coupled | Network operating systems | Real distributed operating systems |
| | tightly coupled | - | Multiprocessing operating systems |

**Table 3 – Types of distributed operating systems**

A multiprocessing operating system is a multitasking operating system running on a multiple processor system. There is a single list of ready to run processes. When a new process is ready to run it is added to the list located in the

shared memory area. Any process can access the list. When a processor is free, it extracts a process from the list and executes it. The operating system has to implement mutual exclusion mechanisms (semaphores, monitors, locks or events using various protocols) in order to protect the concurrent accesses to the ready to run processes list. Using these mechanisms, a processor has exclusive access to the processes list and it can extract the first list entry. Then, the list is released and the process is executed. Because the running time of a parallel program on such a system is finite, we can presume that the program will be divided into a finite number of processes. The queuing model associated to a parallel system running a multiprocessing operating system will be an infinite capacity one. The service facility is formed from m servers running in parallel, where m is the number of the processors from the system.

On the opposite side there are network operating systems and real distributed operating systems. A copy of the operating system is running in every processing node so the distributed system has as many waiting queues as the number of the processors. In every processing node there is a copy of the operating system that manages the processor execution queue. Assuming that the execution time of a parallel program is finite, we can presume that the program will be divided into a finite number of processes too. This is why we can associate to each processing node a queuing model with a single server and arrivals from a finite population. The queuing systems at the nodes level are interconnected together into a queuing network because the processors have to communicate and synchronize each other. The processes may have specific dependencies that may prevent them from executing in parallel in all cases. For example, a process may require the output produced by certain task and it cannot be executed until that prerequisite task has completed executing.

The waiting systems presented above could become queuing models with arrivals from an infinite population if the parallel program needs a very long time to complete its execution and the number of the generated processes will increase substantially.

The effective type of the queuing systems used to describe the execution of a parallel program depends by the probability distribution of the service facility servers. The most common model used is the M/M/m one because usually the service distribution is a Poisson one. An M/G/m model has to be implemented if processes can use unlimited time to complete their execution. For a data-parallel program, the same instructions will be executed on different data sets and the final result will be obtained by combining the partial ones. This is why we can assume that the services are distributed exponentially negative so an M/M/m model could be successfully used.

The queuing theory is a very useful tool to predict the performances of computer systems in general and of the parallel ones in particular. The average quantities of interest (expected waiting time in the queue and in the system,

expected total number of users in the queue and in the system) offer us a clear picture about the system performances and the ways to improve them. The queuing theory will help us to find out how to tune the system parameters in order to increase the system efficiency.

Usually, the system scheduler manages the queuing system. Its main responsibility is to scheduled tasks to the system processors but it also implements the queue discipline and the priority classes.

The total turnaround time of a process is formed by the queue wait time and the elapsed execution time. The queuing theory teaches us that it is important to not only reduce the job elapsed time by using faster processors but also the queue time through effective scheduling and management resources.

## 3. APPLICATIONS OF QUEUING THEORY IN GRID PROCESSING

A grid is a collection of machines that contribute any combination of resources as a whole. Basically, grid computing represents a new evolutionary level of distributed computing. It tries to create the illusion of a virtual single powerful computer instead of a large collection of individual systems connected together. These systems are sharing various resources like computing cycles, data storage capacity using unifying file systems over multiple machines, communications, software and licenses, special equipments and capacities.

The use of the grid is often born from a need for increased resources of some type. Grids can be built in all sizes, ranging from just a few machines in a department to groups of machines organized in hierarchy spanning the world. The simplest grid consists of just few machines, all of the same hardware architecture and same operating system, connected on a local network. Some people would call this a cluster implementation rather than a grid. The next step is to include heterogeneous machines but within the same organization. Such a grid is also referred to as an *intragrid*. Security becomes more important as more organizations are involved. Sensitive data in one department may need to be protected from access by jobs running for other departments. Dedicated grid machines may be added to increase the service quality. Over time, a grid may grow to cross organization boundaries and may be used for common interest projects. This is known as an *intergrid*.

The easiest way to use a grid is to remotely run an application on a different computer than the one on it is usually executed. If the computer that usually runs the job is busy, it can execute the application on another idle machine from the grid network. The remote machine must meet hardware, software and resource requirements of the application. Desktop machines from most organizations are underutilized because they are busy less then 5% of time. Grid computing is able to increase the resource usage efficiency because it could be obtained a better balance of resource utilization. If an application is grid enabled it

could be moved to an idle computer from the grid whenever the host computer is busy.

If an application is written to use algorithms that can be divided into independent parts than each part could be executed on a different machine in the grid. This is why the grid computing offers a high potential for massive parallel CPU capacity. This huge computing power obtained by the use of the grid is driving a new evolution in various industries like financial modeling, oil exploration, bio-medical field and so on. Scalability is the measure of how efficiently the grid processors are used. A perfect scalable application will finish *n* times faster when it uses *n* times the number of processors. It is very hard to achieve the perfect scalability. The limits of scalability are called barriers. An example of such a barrier is the situation when an application can be split only into a limited number of independent parts. Not completely independent parts and communications between jobs are another barrier examples because they limit the scalability. Figure 3 shows how the application parts are executed concurrently in a grid network.

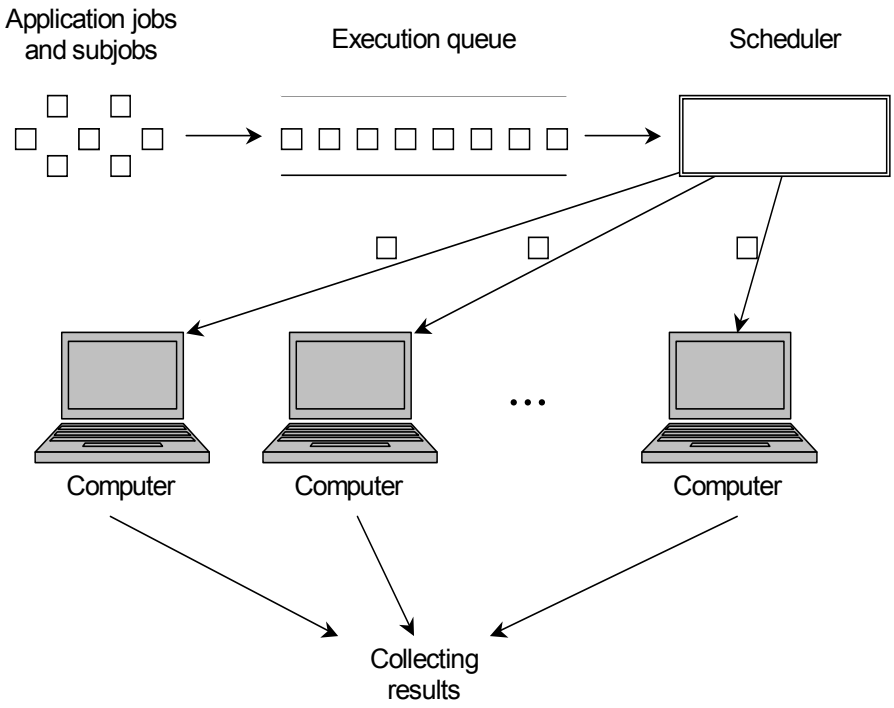


**Figure 3 – Running an application in a grid network**

Dependencies prevent jobs to be executed in parallel in all cases. Jobs may spawn additional subjobs. The result is a hierarchy of jobs and subjobs. The results of all jobs must be collected in order to obtain the final result of the application.

The grid network is considered like a single center and the service demands are scheduled on the individual computers from the grid. There is a single queue of jobs for all processors from the network. The total service rate of *n* processors from the grid can be significantly les then *n* times the rate of a single processor because of competition for software locks (such of those controlling access to the shared queue of jobs) and interference in accessing shared resources. On the other hand, the effective service rate of the grid network is not constant but depends on the number of jobs queued at center. Consider a four processor system. Ideally, if four or more jobs desire service at the center, all four processors can be kept busy, and the effective service rate of the center is its maximum rate. However, if less than four jobs are queued at the center, some of the machines will be idle, and so the effective service rate will de reduced correspondingly. Figure 4 graphs effective service rate as a function of the queue length for a four-processor system. Service rates increase with queue length until all four systems are busy, after which increasing the number of jobs contending for the processors does not result in any increase in effective service rate. The dashed line illustrates the ideal growth in service rate and the other one represents the effect of contention.

Grid computing could be used also to run an application that needs to be executed many times on the computers from the grid network. In such a way the results could be obtained faster using the grid.

There are many factors to consider writing a grid enabled application. Application designers can use tools to write parallel grid applications. There are no tools for transforming applications to use the parallel capabilities of a grid. Some already existing applications could not be transformed to run in parallel on a grid. By scheduling jobs on underutilized machines the grid offers a resource balancing effect for grid enabled applications. The scheduler can migrate jobs to less busy parts of the grid to balance resource loads and absorb unexpected peaks of organization activity. The scheduler could schedule jobs to reduce the communications traffic or to minimize the distance of the communications.

The most common resource used in grid networks is computing cycles provided by the processors of the machines on the grid. Scalability is used to measure how efficiently the multiple processors on a grid are used. The second most common resource is data storage and it could be memory attached to a processor (very fast but volatile) or a permanent storage media like a hard disk. A large file can be stored on multiple machines using a unifying file system that provides a single uniform name space for grid storage. The users don't need to know the exact location of referenced data. The jobs that are using data have to be scheduled closer to the data, preferably on the node where the data resides. The grid file system has to

implement synchronization mechanism in order to avoid contention when many users concurrently update shared data.

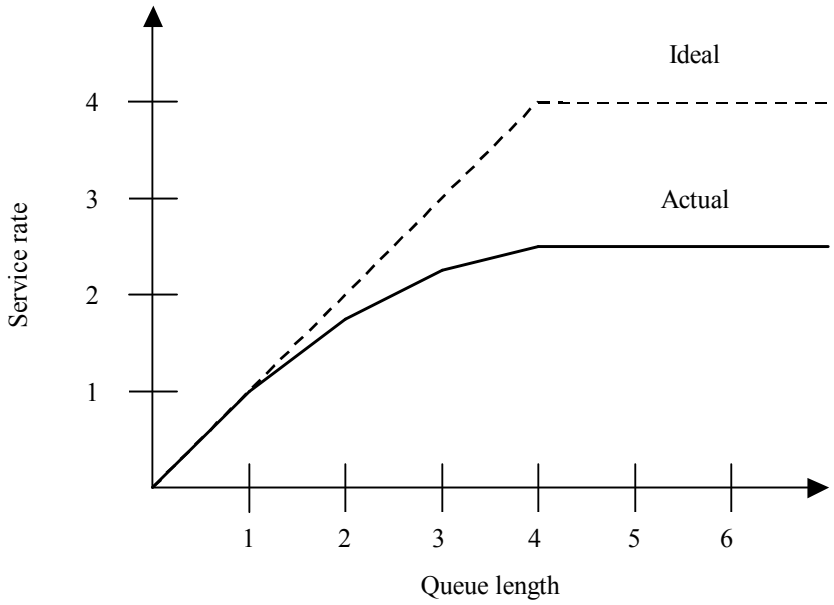


**Figure 4 – Service Rate Function of a Four Processor Grid Network**

Another important resource of a grid is data communication capacity that is very important for sending jobs and their required data to points within the grid. Some jobs require a large amount of data to be processed that could not reside on the machine running the job. In such a case the bandwidth could be a critical resource that can limit utilization of the grid.

If we have software that is very expensive to be installed on every machine from organization, a grid could send the jobs that require this software to the machines on which it is installed. This approach saves significant expenses when the licensing fees are considerable.

The shared resources are usually accessed via an executing application. Applications may be broken down into any number of individual jobs, as illustrated in figure 1. In turn, jobs can be further broken down in subjobs. Independent jobs will be executed in parallel on different computers from the grid network. Some jobs could not be executed in parallel because they have specific dependencies that may

prevent them from executing in parallel in all cases. Finally, the results of all jobs are collected and assembled to produce the ultimate answer for the application.

The scheduler is responsible for sending a job to a given machine to be executed. In the simplest of grid systems, the user selects a machine and executes a grid command that sends the job to that machine. More advanced grid systems include a job scheduler that automatically finds the most appropriate machine for a waiting to be executed task. Gird machines that are not dedicated to the grid are suspending grid tasks when local non-grid work has to be done. The dedicated machines are not preempted by outside work. Schedulers try to assign executing jobs to machines nearest to the data that these jobs require in order to decrease the network traffic and possibly reduce scalability limits. Optimal scheduling is still a difficult mathematics problem.

A few programmers may install a grid in their spare time. The planning become essential as the grid grows and the users will be more dependent on it. Security is much more important factor in planning and maintaining a grid than in conventional distributed computing. Any grid system has some management components used to keep track of the resources available to the grid. The scheduler uses this information in order to decide where grid jobs should be assigned. In the simplest case, the scheduler may assign jobs in a round-robin fashion. Some schedulers implement a priority system using several jobs queues, each with a different priority. When a grid machine become available, the job will be taken from the highest priority queue. Policies of various kinds are also implemented using schedulers.

Not every application is suitable for running in parallel on a grid. Some applications simply cannot be parallelized. For others, it can take a large amount of work to modify them to run concurrently.

**References**


1. 1.G. Boldur-Lățescu, I. Suciu, E. Țigănescu, *Operational Research with Applications in Economy*, A.S.E., 1990
2. Gh. Dodescu, Operating Systems, ASE 1997
3. Gh. Dodescu, B. Oancea, M. Raceanu, *Parallel Processing*, Ed. Economica, Bucharest, 2002
4. D. Gross, C. M. Harris, *Fundamentals of Queuing Theory*, Wiley, New York, 2003
5. H. F. Jordan, H. E. Jordan, *Fundamentals of Parallel Computing*, Prentice Hall, 2002
6. J. Joseph, C. Fellenstein, *Grid Computing*, Prentice Hall, 2003
7. A. S. Tanenbaum, *Distributed Operating Systems*, Prentice Hall, 1995
8. A. S. Tanenbaum, *Computer Networks*, Prentice Hall, 1996