Synchronous Queuing: A Co-allocation Mechanism for Multimedia Enabled Grids

by

Farag Azzedin

A thesis

Submitted to the Faculty of Graduate Studies

in Partial Fulfillment of the Requirements

for the degree of

MASTER OF SCIENCE

Department of Computer Science
University of Manitoba
Winnipeg, Manitoba, Canada

Acknowledgement

In the name of Allah, the Merciful, the Compassionate. Praise be to Him for His creation and making me submissive to Him. Thanks to Him that He sent us the prophets to guide us to the straight path.

I express my gratefulness to my supervisor Prof. M. Maheswaran, whose amazing patience, infinite help in all aspects, and non-stop support made me achieve what I did not know I had in me. I can never thank him enough. I am also thankful to the thesis committee members, Prof. N. Arnason and Dr. J. Rueda, for being on my thesis committee. Also, I would like to thank Prof. N. Arnason for his useful suggestions and help during the simulation phase.

All the love to my mother (Fatima), my father (Ahmed), who are always in my heart and mind, and whose my accomplishment in life is none but the result of Allah's answering to their prayers and supplication to Him for my success.

Finally, I express my heartfelt gratitude to my wife (Ameena), whom whatever words of thanks I say, they would not be enough to do justice to her. And to our children Muhammad, Ahmed, and Hala whom just remembering them makes me know what I want to do in life.

Abstract

Grid computing systems are being positioned as a computing infrastructure of the future that will enable the usage of wide-area network computing systems for a variety of challenging applications. The multimedia enabled Grid (MEG) is an extension of the Grid concept to support the deployment of multimedia services to meet the ever increasing demand for multimedia from users engaging in a wide range of activities such as scientific research, education, commerce, and entertainment. The MEG will provide several new services and sustain several enabling technologies to support multimedia.

To provide an adequate level of service to multimedia applications, it is often necessary to simultaneously allocate the resources including predetermined capacities from the interconnecting networks to the applications. The simultaneous allocation of resources is often referred to as co-allocation in the Grid literature.

In this thesis, I propose a novel scheme called synchronous queuing (SQ) for implementing co-allocation with quality of service (QoS) assurances in Grids. The SQ does not require advance reservation capabilities at the resources, which is a fundamental difference between SQ and the other existing schemes. I formally define the co-allocation problem and classify existing approaches based on a taxonomy that is presented here. Based on the taxonomy, I discuss the situations under which SQ can be used for co-allocation in MEGs. The SQ scheduler introduces new scheduling concepts *such as the notion of accounting for the previous work, the notion of introducing intraQueue and interQueue schedulers and the notion of calculating the co-allocation skew. Simulation studies performed to evaluate SQ indicate that it outperforms admission control-based scheme by a significant margin.*

Table of Contents

List of Figures

List of Tables

1 INTRODUCTION

The deployment of faster networking infrastructures and the availability of powerful microprocessors have positioned network computing as a cost-effective alternative to the traditional computing approaches. The network computing systems can be grouped into various categories depending on the extent of the system and the performance of the interconnection media. For example, clusters of workstations are network computing systems that use commodity networks to create very tight and dedicated coupling among the nodes. Another example of network computing is the metacomputing initiatives on the Internet that attempt to harness the available resources to perform complex parallel applications such as prime number sieves. Motivated by the successes of such specialized efforts, researchers have started examining a more generalized resource/information sharing and integration infrastructure called the Grid [FoK99]. The Grid is defined as a generalized, large-scale computing and data handling virtual system that is formed by aggregating the services provided by several distributed resources [BaB00, FoK98, KrM00, JoG99, MaK00]. A Grid can potentially provide pervasive, dependable, consistent, and cost-effective access to the diverse services provided by the distributed resources and support problem solving environments that may be constructed using such resources.

The *multimedia enabled Grid* (MEG) is an extension of the Grid concept to support the deployment of multimedia services to meet the ever-increasing demand for multimedia from users engaging in a wide range of activities such as scientific research, education, commerce, and entertainment. The MEG will provide various new services and sustain several enabling technologies to support multimedia. Some of the new services include transparent user profile location and access supporting “upcalls” so that adaptive applications can be developed. A user in such an environment is not tied to a specific machine but rather is a machine independent entity that exists in the Grid and can transparently carry its profile across the different platforms constituting the Grid. Some of the enabling technologies that will be supported include: (a) *quality of service* (QoS), (b) multicast, (c) streaming data, (d) co-allocation of resources, and (e) resource discovery.

Multimedia applications (e.g., digital audio or video) are known to impose real-time requirements on the underlying computing and communication systems [NiL97, AzM00]. Some of these applications require multiple networked resources for their execution. To provide an adequate level of service to the users, it is often necessary to allocate these resources including predetermined capacities from the interconnecting networks simultaneously to the particular applications. Examples of applications that require simultaneous allocation of resources include multimedia conferencing, virtual reality based distributed interactive simulation, distance learning, etc. The simultaneous allocation of resources is referred to as *co-allocation* in the Grid literature.

The co-allocation in a MEG environment is a much more general problem than that in traditional distributed multimedia systems. This is due to various issues including: (a) location independent access and management of resources, (b) resource heterogeneity both in terms of capability and policy, and (c) geographically distributed location of the resources. These issues call for a resource management model with a hierarchical scheduling structure. The hierarchical scheduling structure introduces “hidden” scheduling [YaL96] problems rendering the overall resource management and particularly the co-allocation of resources a challenging task.

Most existing approaches [FeG97, FoK99] to co-allocation in wide-area distributed systems depend on the ability of the resources to support advance reservations. While performing co-allocation via advance reservation simplifies the problem, this approach has several drawbacks. One of the drawbacks is that this model does not allow the over subscription of the resources and thus leading to under utilization of the overall system. Another drawback is that the advance reservation-based approach imposes strict timing constraints on the client side.

This thesis presents *synchronous queuing* (SQ) a novel scheme for co-allocation that does not require advance reservation capabilities at the resources. The scheme provides co-allocation with QoS constraints, i.e., it is possible to perform co-allocation with hard QoS guarantees as well as co-allocation with best-effort guarantees.

The thesis is organized as follows: Section 2 presents the notation and mathematically defines the co-allocation problem. A taxonomy of existing approaches to perform co-allocation in distributed systems is presented in Section 3. Section 4 examines the related work. Section 5 sketches in detail the *synchronous queuing* (SQ) scheme to solve the co-allocation problem in MEGs. This is followed by simulation results in Section 6. Section 7 summarizes the thesis and presents directions for future work.

2 NOTATION AND PROBLEM DEFINITION

One of the distinguishing features of the Grid concept is the recognition of the heterogeneity [ScN99] and site autonomy issues that are faced by the ultra-large scale distributed systems. One of the ways Grids handle these issues is to use a scheduling hierarchy. Figure 1 shows a basic block diagram for the resource management architecture for the Grid system with a two-level scheduling hierarchy consisting of local schedulers and Grid-level schedulers.

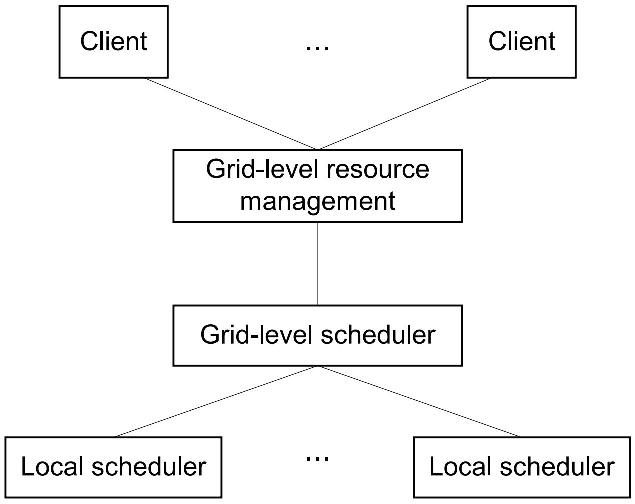


Figure 1: A block diagram for an overall resource management architecture for the Grids.

Let t denote a task submitted by a client to the Grid for processing and let this task t be composed of n subtasks $s_0,...,s_{n-1}$. Consider the situation where a Grid-level scheduler maps the different subtasks to different machines in the Grid. The Grid-level schedulers assign to a particular machine various tasks and subtasks, which are further scheduled by the local scheduler that controls the machine in a timeline fashion as illustrated in Figure

2. Some of these tasks and subtasks might have co-allocation requirements and others may not. t_0

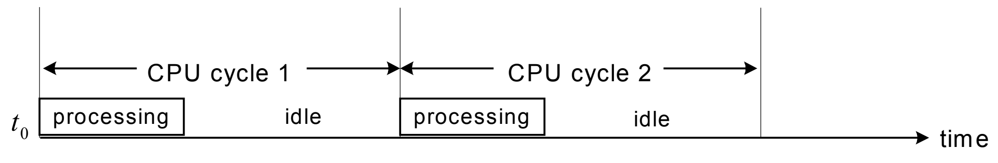


Figure 2: A local scheduler's periodic timing diagram.

Once the subtasks $s_0,...,s_{n-1}$ of task t are assigned to the different local schedulers, it the responsibility of the local schedulers to allocate sufficient machine resources (e.g., CPU cycles) to execute each subtask. Furthermore, let $a_0, a_1,..., a_{n-1}$ be the threads that are instantiated at the local machines for the subtasks $s_0,...,s_{n-1}$, respectively. Because the different local schedulers will have different mix of tasks and subtasks their behavior will be different. Note that because task t has co-allocation requirements all its subtasks *must*

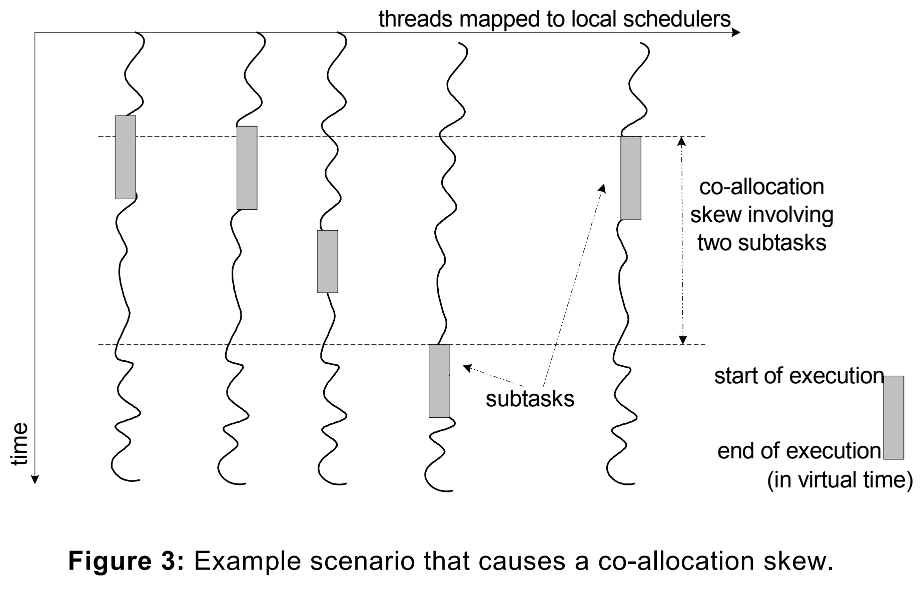

threads mapped to local schedulers
co-allocation skew involving two subtasks
subtasks
start of execution
end of execution (in virtual time)
time

Figure 3: Example scenario that causes a co-allocation skew.

proceed with their execution simultaneously. Some of these subtasks might be delayed before they are allocated sufficient resources. This delay is referred to as *co-allocation skew*. The *co-allocation skew* involving two subtasks is illustrated in Figure 3. The goal of the synchronous queuing algorithm is to minimize this *co-allocation skew* for all applications that require co-allocation.

Consider two subtasks s_i and s_j. Assume that they become runnable at the 1^{st} schedule cycle. For the rest of the thesis, the term "CPU bandwidth" means the total CPU cycles per second available. Each thread a_i will be asking of a *share* of the local machine's CPU bandwidth. This *share* is expressed as a weight (explained in more detail in Section 5.5) assigned to thread a_i. Let r_{a_i} be the weight of thread a_i, m be the number of schedule cycles to date, and $W_k^{a_i}$ be the work done by thread a_i at the k^{th} schedule cycle. Then, threads a_i and a_j are said to be synchronized if, for any k^{th} schedule cycle the aggregate work done normalized by weight since the two threads a_i and a_j became runnable are identical (i.e., $\left| \frac{\sum_{k=1}^{m} W_k^{a_i}}{r_{a_i}} - \frac{\sum_{k=1}^{m} W_k^{a_j}}{r_{a_j}} \right| = 0$). Clearly, this is an idealized definition of synchronization that assumes infinitely divisible subtasks. Hence, the objective of synchronous queuing is to minimize the difference as close to zero as possible (i.e., $\left| \frac{\sum_{k=1}^{m} W_k^{a_i}}{r_{a_i}} - \frac{\sum_{k=1}^{m} W_k^{a_j}}{r_{a_j}} \right| \leq \mathit{threshold}$, for all $i, j,\ i \neq j$).

3 A TAXONOMY OF EXISTING APPROACHES

Traditionally, the Internet was used for running elastic applications for which it was sufficient to provide one single service class known as "best-effort" service. Elastic applications are those that can adjust, over wide ranges, to changes in delay and throughput across an internet and still meet their needs [Sta97]. In the MEG, there will be different types of applications from various application domains for which the "best-effort" service is inadequate. Because MEG is a specialization of the Grid concept [FoK99] to the multimedia applications, supporting various classes of quality of service is essential.

Classifying these wide ranges of applications that might co-exist in the MEG environment is a key element in determining a suitable algorithm to solve the co-allocation problem. In the MEG system, it is not uncommon for a user to run non-real and real time jobs simultaneously. Figure 4 shows a classification of the various applications that might co-exist in a MEG system.

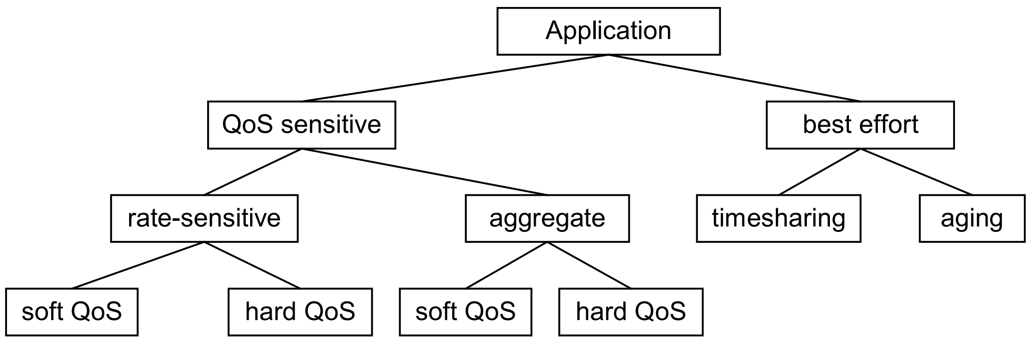


Figure 4: Different classes of applications.

- *rate-sensitive applications:* are applications that depend on accomplishing (finishing) a consistent x amount of work per t amount of time throughout the application's lifetime.
- *aggregate applications:* are applications that depend on accomplishing (finishing) x amount of work per t amount of time throughout the application's lifetime. The amount of work established per period of time t varies (i.e. is not consistent). The emphasis here is on finishing the task by its deadline.
- *hard QoS applications:* are application with stringent progress constraints [YaD99], for which missing a deadline for these applications leads to catastrophic failures. These applications require a deterministic guarantee for their QoS parameters.
- *soft QoS applications:* missing a deadline for these applications only diminishes the quality of the results and does not lead to catastrophic failures. These applications require a statistical or probabilistic guarantee for their QoS parameters.

Different approaches exist to solve the allocation and co-allocation of resources to this wide range of applications. An allocation/co-allocation approach can be summarized based on the following properties: a) the specific types (i.e. best effort and/or QoS sensitive) of applications the scheme can accommodate, b) the environment (i.e., single or distributed) in which the scheme can be implemented, c) the scheme's technique can be specific to one or particular mixes of resources, and do not extend easily to other resources. Or it can be generalized to manage many diverse resources, and d) the allocation/co-allocation scheme can schedule all the tasks in the system based on proportional sharing, priorities, or support such a generality by introducing hierarchical schedulers as a tool to support a variety of QoS sensitive as well as best-effort applications.

4 Related Work

4.1 Overview

In recent years, a lot of research has been done on finding ways for resource reservation schemes to accommodate the increasing demand for deploying real-time (multimedia) applications. The reservation models are mostly concentrated on *immediate reservation* where the reserved resources are scheduled immediately. However, providing QoS guarantees to *immediate reservation* schemes is a difficult task simply because tasks are competing for resources' availability. Furthermore, *immediate reservation* scheme's admission decision is made based on the resource availability at hand and hence under utilizes the overall system. For compensating *immediate reservation* scheme, *advance reservation* scheme has been introduced, for the client to make a reservation for resources for future resource usage. The client specifies time parameters to request advance reservation: START TIME and DURATION. Once admitted, the reserved resources will be effective after the start time for the duration. While those researches are well developed, they mostly treat *advance reservation* separately from *immediate reservation* so that they tend to give best-effort guarantees to *immediate reservation* and treat *advance reservation* as QoS guarantee scheme.

Without advance reservation, providing hard-QoS and soft-QoS to real-time (multimedia) applications running in a heterogeneous environment further complicates the problem of co-allocation. The objective of this section is to briefly touch upon the more important contributions that are directly relevant to this thesis. Therefore, this literature review includes allocation and co-allocation schemes in a single environment as well as in a distributed environment.

4.2 Globus Architecture for Reservation and Allocation

There have been several attempts to solve the co-allocation problem in a Grid-like setting. Globus Architecture for Reservation and Allocation (GARA) system was proposed in [FoK99] to extend the Globus resource management architecture [CzF98]. The Globus system is a software infrastructure for sharing geographically distributed computational and information resources.

The Globus resource management architecture supports the co-allocation of heterogeneous compute resources to provide end-to-end computational QoS. Two issues that the Globus resource management architecture does not address: a) advance reservation which means that the required QoS cannot be guaranteed. Hence, the ability to perform co-allocation will be drastically restricted; and b) heterogeneous resource types. The absence of support for heterogonous resource types like network, disk, and others makes it impossible to provide end-to-end QoS guarantees when an application involves more than just computation [FoK99].

GARA extends this limitation by introducing the generic resource object where it reformulates computation-specific allocation functions in terms of general resource objects. This allows different application components to be manipulated in common ways. GARA also introduces reservation, which will provide some confidence that subsequent allocation requests will succeed.

Since resources are independently controlled and administered, the GARA scheme will not work well because co-allocation requests can be rejected anytime in favor of internal requests. That is, there is no commitment from the GRAMs for the co-allocation classes. Furthermore, advance reservation is a requirement for co-allocation requests in GARA, which (as mentioned earlier) imposes strict timing constraints on the client side and does not allow the over subscription of the resources and thus leads to under utilization of the overall system.

4.3 Implicit Co-scheduling

Implicit co-scheduling [ArC98] is a new time-sharing approach for scheduling parallel applications. Implicit co-scheduling uses communication and synchronization occurring naturally within the application to coordinate scheduling across workstations. Here, two events *response time* and *message arrival* are used to decide whether to continue with the executing process or to block and schedule another process.

The basic idea is that, if a response to a request arrives, or a message arrives from a cooperating process executing on a different processor, it means that the other process was scheduled at that time. So, it is a better idea to continue executing the local process. On the other hand, if there is some delay in these events, then, the executing process can use the two-phase spin locking mechanism because it probably might be a better idea to wait for some time rather than pay the penalty of context switching to another process.

While implicit co-scheduling presents a new approach for improving the global performances for parallel scheduling, it addresses time-sharing applications and comes short of addressing real-time multimedia applications. Implicit co-scheduling also approaches the co-scheduling problem from the application layer itself (i.e. the application has to sense its own progress and adapt accordingly) whereas SQ tackles the problem at the scheduler level relieving the application from the overhead.

4.4 Tenet Real-Time Protocol Suite 2

Another approach to the co-allocation problem is the Tenet Real-Time Protocol Suite 2. This system is a suite being developed for multi-party communications and it offers advance reservation capabilities to its network clients [FeG97]. The fundamental requirement that networks clients, with performance-oriented network-based applications, have is to be allowed to specify in advance their needs in terms of real-time channels. In addition, those needs must be guaranteed (*reserved*) and immediately or later be created (*allocated*).

In [FeG97], the client will be requested to provide, in addition to the parameters that define end-to-end provision of high QoS, (a) the starting time, and (b) the duration of the real time channel. This way, the advance reservation provider will be able to plan for future allocations of resources; more clients can reserve in advance; sharing and utilization of network resources are increased; and hence better resource management. Since resources are independently controlled and administered, the Tenet scheme will not work well since co-allocation requests can be rejected anytime in favor of internal requests. Furthermore, advance reservation is a requirement for co-allocation requests, hence different from the proposed scheme, which does not require the client(s) to reserve resources in advance. Advance reservations, while simplifying the problem of co-allocation, have several drawbacks as mentioned in the introduction section.

4.5 Scheduler for Multimedia And Real-Time applications

A proposed scheme in [NiL97], a scheduler for multimedia applications, Scheduler for Multimedia And Real-Time applications (SMART) is designed to support QoS sensitive and best effort applications as well as providing dynamic feedback to applications to allow them to adapt to the current load. In this way, SMART allows the user to prioritize across different classes of applications and dictate how resource(s) to be shared between applications with same priority. SMART is unique in regulating QoS sensitive tasks when the system is overloaded, while providing better value in under load conditions.

This suggests that SMART has a fairly complex resource management, which is fulfilled by basing the resource management decisions on two features; importance and urgency. Importance is used to determine the overall resource allocation for each task. After the importance of each task is determined, SMART uses urgency based on earliest deadline scheduling to settle when each task is given its allocation.

Although SMART is effective in supporting multimedia application in a single system, it comes short of addressing the co-allocation problem in a Grid-like system, where resources are geographically distributed and administered by separate systems.

4.6 Proportional Share Algorithms

An alternative is to allocate each task some share of the CPU's capacity. Such algorithms are known as *proportional share algorithms* [WaW94, WaW95]. In these approaches, a machine will be shared by a real time scheduling policy and a conventional scheduling policy. An underlying proportional sharing mechanism will manage and therefore time slice between them. However, real time applications cannot effectively meet their deadlines as a result of conventional scheduler taking away resources at an inopportune and unexpected time in the name of fairness [NiL97].

The fundamental problem with proportional share or priority-based scheduling algorithms is the lack of *generality*. Proportional sharing is not suitable for QoS sensitive tasks because they cannot meet their time constraints effectively. With priority-based, QoS sensitive tasks are assigned higher priority than best-effort tasks. This is done, whether QoS sensitive tasks are important or not and causes all QoS sensitive tasks to run ahead of best effort-tasks. To support such *generality*, hierarchical schedulers are proposed as a tool to support a variety of QoS sensitive as well as best effort applications.

A local machine's resource (e.g. CPU) is partitioned among various application classes. This partitioning is referred to as *hierarchal partitioning* and will be explained in more detail in Section 5.6

4.7 Start-time Fair Queuing

The level of *generality* described above is achieved in Qlinux[1] by enabling hierarchical scheduling of applications and fairly allocating CPU bandwidth to individual applications and application classes. Qlinux accomplishes this by exploiting features such as Hierarchical Start-time Fair Queuing (H-SFQ) [GoG96].

Start-time Fair Queuing (SFQ) is a hierarchical scheduling algorithm that was proposed in [GoV96]. The work done by the CPU for a task is measured by the number instructions executed for that task. Then, the allocation of CPU is considered fair if, for all intervals $[t_1, t_2]$ in which two tasks are runnable, the difference of the normalized work (by the tasks' weight) received by them is as close to zero as possible.

Although SFQ supports different application classes, it works in a single environment. That is, it does not tackle the problem of resources co-allocation where a task requires different types of resources that are independently managed and reside on heterogeneous systems.

[1] Qlinux is a QoS enhanced Linux Kernel for Multimedia Computing: http//www.cs.umass.edu/~lass/software/qlinux/

To conclude this section, the existing schemes to tackle allocation and/or co-allocation of resources are summarized. The schemes are presented along with their properties which can be classifies into 4 categories as explained at the end of section 3. The summary is provided in Table 1.

Table 1: Summary of existing allocation and co-allocation schemes.

Properties of Existing Allocation/Co-Allocation Schemes				**Existing Scheme(s)**
Supported application(s)	**Environment**	**Supported resource(s)**	**Used scheme**	
best-effort and QoS sensitive	Single	CPU	hierarchical	**H-SFQ**
best-effort and QoS sensitive	Single	CPU	hierarchical	**SMART**
best-effort	Single	diverse resources	proportional	**Lottery and Stride scheduling**
best-effort and QoS sensitive	distributed	diverse resources	hierarchical	**Tenet Suite 2**
best-effort	distributed	CPU	Application based	**Implicit co-scheduling**
best-effort and QoS sensitive	distributed	diverse resources	hierarchical	**GARA**

5 SYNCHRONOUS QUEUING

5.1 Overview

Grid-based multimedia applications have a variety of QoS constraints that have to be met locally at each machine as well as globally across the Grid computing systems. A mechanism is needed to be positioned as a co-allocation infrastructure that will enable the usage of MEG environment resources for a variety of these challenging multimedia applications. The co-allocation issue that is addressed is concerned with ensuring that an application that has several sub components would be allocated sufficient resources so that all sub components of the application can make satisfactory progress with their execution. Some of these challenging multimedia applications include multimedia conferencing, virtual reality based distributed interactive simulation, distance learning, etc. The co-allocation is an essential feature for several important classes of multimedia applications. *Synchronous queuing* (SQ) is such a co-allocation scheme infrastructure that is capable of meeting those constraints for a satisfactory deployment of these applications. SQ algorithm is essentially a detection of asynchrony that can signal corrective action. Detection of asynchrony can be done at every schedule cycle or at a much larger interval (e.g. a group of schedule cycles). Corrective action is local to some extent and is done more often whereas global corrective action is done less frequently and is needed to handle heavy loading situations.

A detailed description and analysis of the SQ co-allocation scheme for MEGs are presented in this section. Discussion of the situations under which SQ can be used for co-allocation in MEGs and the performance of the SQ co-allocation mechanism are also detailed.

5.2 Traditional QoS versus Synchronous Queuing QoS

SQ is a co-allocation scheme providing co-allocation with QoS constraints without advance reservation of resources. Hence, it is possible to perform co-allocation with hard QoS guarantees, as well as co-allocation with best-effort guarantees. That means all the local machines are capable of providing QoS which raises the following question: if there is a QoS guarantee from all the local machines, then a Grid-level QoS subtask s_i will be guaranteed its share of the local resource and thus, there is no need for either local or global synchronization and consequently there is no need for SQ.

To answer the above question, three points are considered; First, traditional QoS (admission control-based) guarantees are probabilistic in the sense that a subtask s_i requiring $m\%$ of a local machine's resource might get for each schedule cycle a different value x in the neighborhood of m depending on the machine's load. Note that, the application or task t has no control on the acceleration (upper) or retardation (lower) value of x. Second, traditional QoS guarantees are for the current schedule cycle and thus, it is an instantaneous guarantee and requires the application to be adaptive and sense its own progress. The traditional QoS scheme does not know or remember how much of the local machine's resource (e.g. CPU bandwidth) has been given to subtask s_i in the previous schedule cycles. Third, traditional QoS guarantees are environment-unaware and do not know about other subtasks' QoS status in order to assure that the co-allocation skew is minimized for all subtasks belonging to the same task or application.

The above-mentioned points are fundamental differences between traditional QoS guarantees and SQ QoS guarantees. SQ is an aggregated scheme that assures the total work accomplished by each subtask s_i does not fall behind its agreed QoS. In addition, it is also an environment-aware scheme that assures the aggregated work accomplished by each subtask s_i does not fall behind the other subtasks belonging to task t.

5.3 Applications Suitable for Synchronous Queuing

Because the MEG is a specialization of the Grid concept to the multimedia applications, supporting wide ranges of applications is essential. In addition, classifying these challenging applications that might co-exist in the MEG environment is a key element in solving the co-allocation problem. In such a heterogeneous environment it becomes essential to subtasks $s_0,...,s_{n-1}$ (belonging to task t) to be synchronized in such a way that renders task t useful to the client or the end-user. This can be established by having the capability to assist each subtask to become environment-aware as well as providing aggregate rather than instantaneous QoS guarantees. Aggregate QoS guarantees provide the mean for controlling a subtask's consumption of a local machine's resources. The subtask is disciplined and accounted for the consumption since the time of execution up to the current schedule cycle. Environment-awareness is another essential capability for a subtask to have in such a distributed heterogeneous setting. This capability ensures that a subtask is globally aware of the progress of all subtasks belonging to the same task t so that global synchronization can be signaled. To visualize these emerging synchronous-oriented MEG applications, consider the following example of a continuous multimedia application.

Consider a slide-show task/application t composed of two subtasks s_1 and s_2 responsible for displaying pictures and their corresponding text respectively. The co-allocation scheme used has to guarantee and monitor the QoS so that asynchronous situations (e.g. subtask s_1 is displaying a new picture, while subtask s_2 is still displaying the text of the previous picture or vise versa) are avoided as much as possible. This can be established by having the capability to assist each subtask to become environment-aware as well as providing aggregate rather than instantaneous QoS guarantees.

5.4 Queuing-based Architecture for Co-allocation

In a MEG environment, a QoS aware client, as shown in Figure 5, can submit a Grid QoS or a Grid best-effort task t simply by contacting a Grid resource broker. The client in this case is unable to execute task t locally due to lack of resources such as computing power, storage devices, etc.

Being part of a MEG environment, while cutting cost and time, allows sharing of resources that would otherwise be unavailable. Over the network, the Grid resource broker is client-Grid middleware that provides a uniform interface to heterogeneous resources in conjunction with the Grid discovery and allocation services.

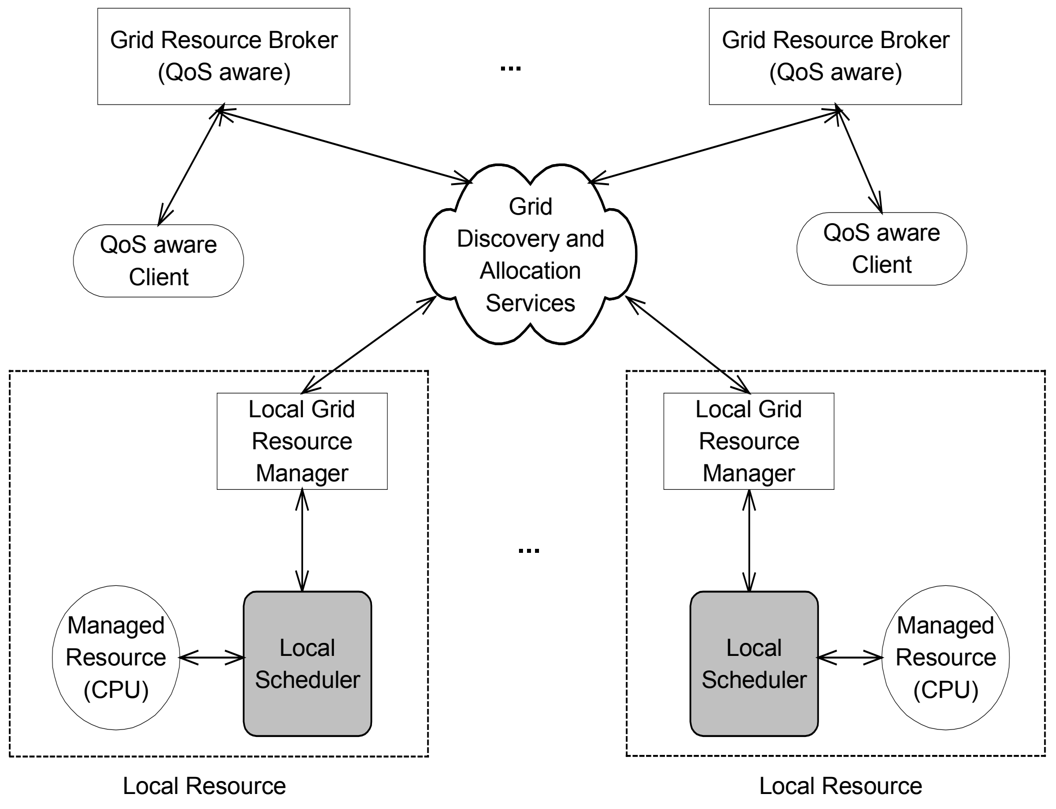

Grid Resource Broker (QoS aware)
...
Grid Resource Broker (QoS aware)
QoS aware Client
Grid Discovery and Allocation Services
QoS aware Client
Local Grid Resource Manager
...
Local Grid Resource Manager
Managed Resource (CPU)
Local Scheduler
Local Scheduler
Managed Resource (CPU)
Local Resource
Local Resource

Figure 5: The overall queuing-based co-allocation architecture.

The Grid discovery and allocation services provides a bridge to the pool of available resources by constructing sets of resources that both match QoS requirements and conform to the local practices and policies of resource providers. Once the resources are discovered and allocated, the Grid-level scheduler, one service provided by the Grid discovery and allocation services, maps the allocation task t or the co-allocation task t's subtasks to the corresponding resource providers. The Grid controller, yet another service provided by the Grid discovery and allocation services, plays a crucial role in the SQ co-allocation scheme by monitoring the progress of the different subtasks of task t and assuring that the QoS guarantee for task t is not violated. The local schedulers on the local resource providers or machines further schedule task t or its subtasks.

The local Grid resource manager, as shown in Figure 5, is a communication channel between the Grid discovery and allocation services and the local resources. One responsibility of the local Grid resource manager is to convey the local practices and policies of the local machine. Such practices and policies is partitioning of a local resource between local tasks, Grid QoS tasks, and Grid best-effort tasks. The local Grid resource manager is also responsible for monitoring and adaptively reporting the progress of the various tasks/subtasks executing on its local machine environment to the Grid controller. With limited local resource partition for each class (local, Grid QoS, and Grid best-effort) and depending on the QoS assurances sought, the local Grid resource manager may perform an admission test before admitting a task or a subtask. The Grid QoS tasks will have a mixture of tasks and subtasks some of which have hard QoS and others have soft QoS requirements. Providing QoS guarantees for these types of tasks or subtasks ensures that the requirements of admitted tasks and subtasks do not exceed the allocated resources assigned by the resources provider. Having strict admission control assures that the load of tasks/subtasks, in competing for a local resource, does not exceed the bandwidth of that resource. Traditional QoS has a strict admission control. In SQ, such strict admission control is relaxed in the sense that the load of admitted tasks/subtasks competing for a local resource could exceed the bandwidth of the local resource. Hence, SQ will accommodate more tasks/subtasks and yet provide better QoS guarantees. The trades-offs that allow relaxed admission control are explained in detail in Section 5.10.

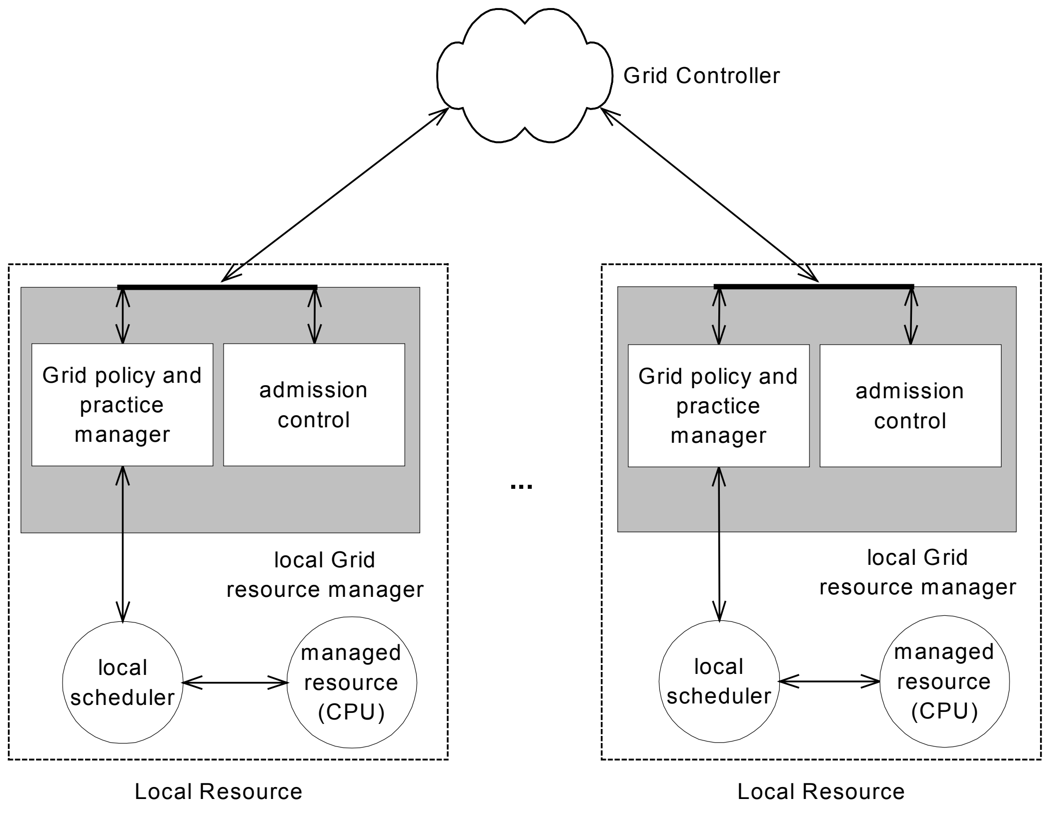


Figure 6: Architecture of the local resource.

The admission control unit as illustrated in Figure 6 performs the admission test. Once the admission control test is performed at the local node, it is the responsibility of the local Grid resource manager to convey the result, especially in case of rejection, to the Grid discovery and allocation services. The flow of tasks and subtasks coming to the local resource can be generated either locally or globally from the MEG environment. In turn the MEG traffic is further classified into two classes (Grid QoS and Grid best-effort), as explained earlier. The hierarchical partitioning of a local managed resource (e.g. CPU) to accommodate this flow of tasks and the different components of the local scheduler are illustrated in Figure 7. The local scheduler can be viewed as the implementation of the policies and practices drawn by the Grid policy and practice manager. These implementations are used to manage and control local resources such as CPU.

5.5 Simplified Example

In this thesis, a queuing-based mechanism is presented to solve the co-allocation problem in MEGs. Unlike most of the previous approaches to co-allocation, this scheme does not require the target resources to support advance reservations. This allows for a flexible resource management scheme and also co-allocations with varying levels of QoS assurances. The basic idea is to adjust the resource allocations given to the different threads of the same application in an adaptive fashion so that the co-allocation skew is minimized among the threads that belong to the same application. Next, I sketch the overall synchronous queuing idea using a simplified example whereas subsequent sections will provide in detail SQ and the associated algorithms.

For brevity, consider that task t is subdivided into two subtasks s_0 and s_1. Let a_0 and a_1, with equal weights (i.e., $r_{a0} = r_{a1}$), be the two threads generated by the two local schedulers. The work done by a_0 and a_1 (i.e., the CPU quantum allocated to each) should be monitored to assure that aggregated work accomplished by each thread does not fall behind or exceed the other thread belonging to task t. One approach that can be taken is to use real time (RT) and virtual time (VT) clocks [Zha91]. Let t_0 be the starting RT when a_0 starts execution. Initially $RT = pRT = VT = 0$, where pRT is the previous RT. For each schedule cycle (y), RT will be advanced by y. However, for the same schedule cycle, VT will be advanced by $VT + \frac{(RT - pRT)}{x} * x'$, where x is the agreed quantum allocated for a_0, and x' is the actual quantum a_0 gets. After VT is computed, pRT is set to RT.

Let us monitor thread a_0 after the j^{th} schedule cycle. For simplicity, let $VT = RT = pRT = 0$. Two scenarios can occur: first, a_0 is getting its quantum x in each schedule cycle, then $x = x'$ and hence VT will be advanced by $\frac{y}{x} * x' = \frac{y}{x} * x = y$. That is, in virtual time, the aggregate work done by a_0 is $\sum_{k=1}^{j} W_k^{a_0} = \sum_{k=1}^{j} y = j * y$ which is the same aggregate work expected in real time (RT). Second, a_0 is not getting its quantum x in each schedule cycle, then $x \neq x'$. Let us assume that a_0 is getting 90% of its agreed quantum (i.e., $x' = 90\% * x = 0.9x$). Hence VT will be advanced by $\frac{y}{x} * x' = \frac{y}{x} * 0.9x = 0.9y$. That is, in virtual time, the aggregate work done by a_0 is $\sum_{k=1}^{j} W_k^{a_0} = \sum_{k=1}^{j} 0.9y = j * 0.9y$ which is less than the aggregate work expected in real time (RT). If $j = 4$, then in virtual time, the aggregate work done by a_0 is $3.6y$, whereas the aggregate work expected in real time is $4y$. From the second scenario, we know that a_0 is running behind its agreed schedule and some control mechanism has to be done. In this case, local synchronization is attempted to bring a_0 up to speed. If local synchronization fails, global synchronization is signaled. This is accomplished by sending messages to the other threads so that synchronization can be accomplished again for task t.

5.6 Tasks Flow Within Synchronous Queuing

Each local machine's load is a combination of the three flows of tasks; Grid QoS, Grid best-effort, and local tasks; which is assigned to the appropriate local queue waiting for execution as shown in Figure 7. A hierarchy of schedulers is used within each local scheduler.

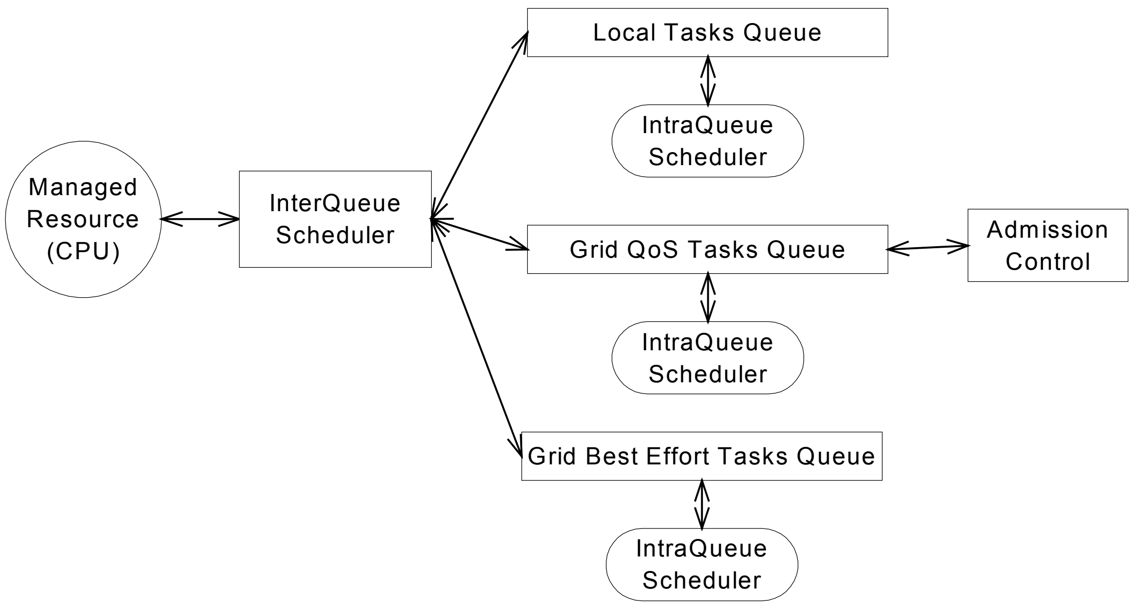


Figure 7: The different components of a local scheduler.

Each machine's resource (e.g. CPU) is hierarchically partitioned amongst the three task flows; local, Grid QoS, and Grid best effort. Statically assigned by the local resource, let r_{q1}, r_{q2} and r_{q3} denote the partition weights given to these class flows respectively such that $r_{q1}\% + r_{q2}\% + r_{q3}\% = 100\%$ of the machine's resource. As mentioned earlier, each flow will be assigned to its appropriate queue and hence each of the local queue (LQ), the Grid QoS queue (QoSQ), and the Grid best-effort queue (BEQ) will have associated weights of r_{q1}, r_{q2} and r_{q3} respectively.

Locally generated tasks require allocation of local resources and are assigned to the LQ. As a resource provider to MEG, a local machine is expected to accommodate Grid flow tasks or subtasks as well. A Grid task may have hard QoS, soft QoS, or best-effort requirements. Some of these Grid QoS tasks and subtasks might have co-allocation requirements and others may not. The Grid QoS tasks and subtasks are assigned to the QoSQ while the Grid best-effort tasks and subtasks are assigned to the BEQ. The interQueue scheduler determines which queue should be selected whereas the intraQueue scheduler decides which task or subtask should be scheduled from the selected queue.

5.6.1 Grid Task's Weight Assignment

When the Grid-level scheduler assigns Grid tasks or subtasks, a standard CPU speed of one GHz is assumed. Let t denote a Grid QoS task that requires $m\%$ of CPU bandwidth and let this task t be composed of n subtasks $s_0,...,s_{n-1}$. Furthermore, let $a_0, a_1,..., a_{n-1}$ be the threads that are instantiated at the local machines for the subtasks $s_0,...,s_{n-1}$, respectively. Each of these threads will have a weight, r_{ai}, assigned by the Grid-level scheduler. Thus, each thread will be asking for m_{ai}% of the local machine's CPU bandwidth given by the following equation $m_{ai} = \left(\frac{d * m * r_{ai}}{r_a} \right)$, for all $0 \le i \le (n-1)$, where $d = \frac{1GHz}{CPU_l}$, i.e. d is the standard machine CPU speed divided by the local machine's CPU speed, and $r_a = \overset{n-1}{\underset{i=0}{}} r_{ai}$, i.e. r_a is the sum of the weights for all the subtasks that belong to task t.

Now, m_{ai} is calculated based on 100% of CPU availability, so we have to map it to r_{q2} (partition weight of CPU allocated to the Grid QoS flow). This is accomplished using the following scaling $m_{ai} \Leftarrow \frac{r_{q2} * m_{ai}}{100}$.

As an illustrated example, let 100 CPU quanta represent a schedule cycle and consider a Grid QoS level task t asking for 5% of a resource (e.g. CPU) every schedule cycle and specifically 5 quanta of the local CPU every schedule cycle on a standard machine running at 1 GHz. Suppose that task t is composed of 4 subtasks $s_0, s_1, s_2,$ and s_3. Furthermore, let $a_0, a_1, a_2,$ and a_3 be the threads that are instantiated at the local machines for the subtasks $s_0, s_1, s_2,$ and s_3 respectively. Each of these threads will have a weight, r_{ai}, assigned by the Grid-level scheduler as shown in Figure 8. Let thread a_0 be given a weight r_{a0} of 1 and assigned to a local machine running at 100 MHz and let $r_a = 10$. Thus, thread a_0 will be asking for m_{a0} share of its local machine given by as follows: $m_{a0} = \left(\frac{1}{10} * 5 * \frac{1{,}000{,}000{,}000}{100{,}000{,}000}\right) = 5$. Therefore, thread a_0 requires 5 CPU quanta every schedule cycle on this local machine. Now, keep in mind that the resource (CPU) of the local machine is partitioned amongst three task classes, so thread a_0 CPU requirement needs to be mapped to the QoSQ weight, which is r_{q2}. If we let r_{q2} = 40%, then the weights for thread a_0 can be mapped as follows: $m_{a0} = \left(\frac{40}{100} * 5\right) = 2$ and thus a_0 is asking for 2 CPU quanta every schedule cycle on its local machine. In a similar fashion, the weights for the other threads a_1, a_2, and a_3 can be computed following the same procedure.

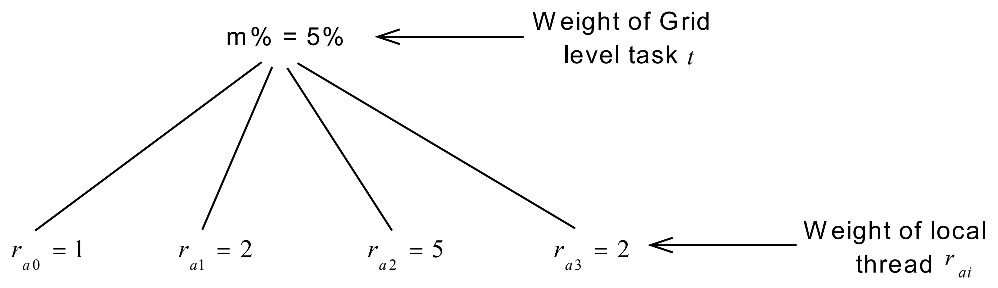

m% = 5%
Weight of Grid level task t
$r_{a0} = 1$
$r_{a1} = 2$
$r_{a2} = 5$
$r_{a3} = 2$
Weight of local thread r_{ai}

Figure 8: Assigning weights for a Grid level task and its four local subtasks.

5.7 Hierarchy of Local Schedulers

The interQueue scheduler shown in Figure 7 uses SFQ [GoV96], which enables the co-existing of resource allocation algorithms, achieves fair resource allocation among the three local queues, and requires only relative importance of tasks expressed by weights to be known. SFQ achieves CPU fairness allocation amongst the threads based on their associated weights.

The objective of SFQ is to allocate CPU quantum/quanta to threads proportional to their weight. To achieve this objective, SFQ assigns a *start tag,* and *finish tag* to each thread and also assigns a common *virtual time.* SFQ schedules the threads in the increasing order of start tags and ties are broken arbitrarily. *Start tags, finish tags,* and *virtual time* are initially 0. When the CPU is idle, the *virtual time* is set to the maximum of *finish tag* assigned to any thread. On the other hand, when the CPU is not idle, *virtual time* is set to the *start tag* of the queue in service. When the scheduling quantum for the thread finishes execution, two things happen:

- the thread's *finish tag* is incremented by the following equation:

  $finishtag = starttag + \left(\frac{b}{r_{ai}} \right)$, where b is the length of the scheduling quantum for thread a_i, and r_{ai} is the weight for thread a_i.

- the thread's *start tag* is computed as the maximum of the *virtual time* or its *finish tag*

Since each machine's local resource (e.g. CPU) is hierarchically partitioned amongst the three queues; LQ, QoSQ, and BEQ, the objective of SFQ is to assure that the allocation of machine's local resource to the three queues is proportional to their respective associated weights (r_{q1}, r_{q2} and r_{q3}). The machine's local resource of each machine is statically partitioned amongst the three queues (i.e. flows of tasks/subtasks). The pseudo-code of the interQueue selection scheme is presented in Figure 9.

```
// queue is the queue that will be selected by the interQueue scheduler.
// LQ is the local queue.
// QoSQ is the Grid QoS queue.
// BEQ is the Grid best effort queue.
SelectQueue()
{
        queue = FindMinStartTag(LQ, QoSQ, BEQ)

        if ( queue is empty ) //equivalent to CPU idle
                virtual time = FindMaxFinishTag(LQ, QoSQ, BEQ)
        else
                virtual time = queue start tag
        endif

        queue finish tag = queue start tag  +  ( scheduling quantum length / queue weight )
        queue start tag =  max(queue finish tag, virtual time)

        return(queue)
}
```

Figure 9: The interQueue SFQ pseudo-code for selecting a queue.

As an illustrated example of how SFQ works, assume that the local queues LQ, QoSQ, and BEQ are given 40, 40, and 20 weights respectively. Each local queue will be given *starttag* and *finishtag.* Furthermore, a common *virtual time* will be assigned. Initially, *starttag, finishtag,* and *virtual time* are all set to zero. Let the scheduling quantum length for each queue be one second. Since ties are broken arbitrarily, assume that LQ is scheduled first. Since, *virtual time* is defined to be the *starttag* of the thread in service, *virtual time* is set to zero and the *finishtag* for LQ is set to $0+\frac{1}{40}=0.025$. In addition, the *starttag* of LQ is set to $\max\{0,0.025\}=0.025$. At this time, SFQ will schedule QoSQ or BEQ because their *starttags* are smaller that LQ's *starttag.* In the same manner SFQ will continue to schedule these three queues and if we complete this for 10 scheduling quanta, we will find that the 10 scheduling quanta is proportionally assigned as 4, 4, 2 to LQ, QoSQ, and BEQ respectively.

After determining the queue to schedule next, the intraQueue scheduler of the selected queue determines which task or subtask should be executed from the selected queue. Depending on the service offered by each queue, a particular scheduling algorithm for the selected queue is exploited. As the intraQueue scheduler, *round robin* (RR) scheduler is used for the LQ and BEQ whereas SFQ is used for QoSQ because fairness in resource allocation is sought in order to provide QoS guarantees.

RR is designed especially for time-sharing systems. A small unit of time, called timeslice or quantum, is defined. All tasks/subtasks are kept in a circular queue. The CPU scheduler goes around this queue, allocating the CPU to each task/subtask for a time interval of one quantum. New tasks/subtasks are added to the tail of the queue. The CPU scheduler picks the first task/subtask from the front of the queue, sets a timer to interrupt after one quantum, and dispatches the task/subtask. If the task/subtask is still running

```
// queue is the queue selected by the interQueue scheduler.
// intraAlgorithm is the scheduling algorithm (e.g. round robin, SFQ) that is used by
// the intraQueue scheduler to schedule a task or subtask from the selected queue.

ScheduleTask ()
{
        // the tasks and subtasks are being generated by another process and
        // assigned to one of  the three different queues accordingly.
        // SelectQueue() is defined in Figure 8.
        queue = SelectQueue ()

        if (queue.id == QoS)
                intraAlgorithm = SFQ
        else
                intraAlgorithm = RR
        endif

        task = SelectTask (queue, intraAlgorithm)

        //start to execute the task
        // schedule the task and put it back in the queue

        if (task execution time > machine service time)
                        task execution time = task execution time – machine service time
                        increment the machine real time by machine service time
                        enqueue(task, queue)

        // schedule the task and remove it from the queue
        else
                        task execution time = 0
                        increment the machine real time by machine service time
                        dequeue(task, queue)
        end if
}
```

Figure 10: Pseudo-code for selecting and executing a task.

at the end of the quantum, the CPU is preempted and the process is added to the tail of the queue. If the task/subtask finishes before the end of the quantum, the process itself releases the CPU voluntarily. In either case, the CPU scheduler assigns the CPU to the next task/subtask in the ready queue. Every time a task/subtask is granted the CPU, a context switch occurs, which adds overhead to the task/subtask execution time.

In the simulation, the context switch overhead is neglected. Each time a task/subtask is scheduled, the task/subtask's execution time is subtracted by the machine service time while the machine real time is incremented by the machine service time. If the scheduled task/subtask's execution time is less than or equal to the machines service time, then its execution time is set to zero and it is removed from the queue. The pseudo-code to schedule tasks and subtasks is presented in Figure 10.

5.8 Strict Versus Relaxed Admission Control

In traditional QoS admission-based algorithms, QoS is provided by having strict admission control assuring the load competing for a local machine's resource does not exceed the upper limit availability of the local resource. Hence, the system will never be overloaded and the following condition will hold true $w - r_{q2} \geq 0$, where $w = \sum_{i=0}^{n-1} r_i$, i.e. w is the sum of the weights for all the tasks and subtasks that are in QoSQ and competing for the local resource, and r_{q2} is the available weight associated with the QoSQ. Having strict admission control does not assure QoS guarantees especially under situations where $w - r_{q2} > 0$ meaning that the local machine in under loaded. In this case, using a scheduler such that SFQ will assure that each task and subtask gets its share of the local resource (e.g. CPU) proportional to its weight. So, if the machine is under loaded, tasks or subtasks will get more than their agreed share of the local resource (i.e. task's virtual time > task's real-time) and co-allocation skew situations will occur.

Relaxed admission control accommodates more QoS demands of a local resource than is provided by the resource provider. In this case, the system can be overloaded $w - r_{q2} < 0$) or under loaded ($w - r_{q2} > 0$). So, the probability of having co-allocation skew is higher with relaxed admission control. In spite of that, SQ scheme, which uses relaxed admission control, outperforms strict admission control-based scheme by a significant margin (refer to Section 6). For the under loaded situation, the scenario will be as explained with strict admission control. With the overloaded situation, we have $w > r_{q2}$ and some of the subtasks will be getting less than the agreed weight causing a co-allocation skew to occur.

5.9 Basic SQ Co-allocation Algorithm

After each schedule cycle (monitor cycle) or a much larger interval (e.g. a group of schedule cycles), the local scheduler, through the Grid policy and practice manager, reports the progress of the co-allocation subtasks to the Grid controller. For each schedule cycle (y), real time (RT) is advanced by y and the local scheduler calculates virtual time (VT) for each of its subtasks. As explained in section 5.5, the calculation of virtual time will be by following the equation: $VT + \frac{(RT - pRT)}{x} * x'$, where x is the agreed quantum allocated for s_0, and x' is the actual quantum s_0 gets. It can be noticed that as x' approaches x, VT approaches RT and hence the finishing time for the subtask is approaching the expected finish time. Once, the Grid controller receives the subtasks' execution progress report from the local schedules, it calculates a pivotal point for each task and then performs detection of asynchrony test. Upon the outcome of this test, the Grid controller might take a corrective action.

SQ is basically applied to hard QoS tasks for which missing a deadline leads to catastrophic failures. These applications require a deterministic guarantee for their QoS parameters and thus the Grid controller has to adaptively monitor their execution and accordingly signal the appropriate corrective action. The next few subsections discuss in detail the steps taken to perform the SQ co-allocation scheme. These steps involve selection of a pivotal point, performing the detection of asynchrony, and signaling the corrective action. Detection of asynchrony involves performing the asynchrony and the overall deviation tests. The overall deviation test can be further classified into two steps: overall retardation test and overall acceleration test.

5.9.1 Selecting a Pivotal Point

Upon receiving the information on the progress of the co-allocation subtasks from the local machines, the Grid controller selects a pivotal point (pp) that is calculated as follows:

$pp = \frac{{}_{i=0}^{n-1} VT_i}{n}$, where n is the number of subtasks belonging to task t, and VT_i is the virtual time for subtask s_i. So, the pivotal point is essentially the average of virtual time for the n subtasks that belong to task t.

5.9.2 Detection of Asynchrony

For each Grid QoS task, the clients have to provide two QoS attributes: *asynchrony*, and *overall deviation*. *Asynchrony* is the acceptable *async* that a task t can tolerate and is calculated as: $async = VT_f - VT_s$, where VT_f is the virtual time of the fastest subtask, and VT_s is the virtual time of the slowest subtask among all subtasks belonging to task t. O*verall deviation* is the acceptable retardation or acceleration that a task t can tolerate for its subtasks. Retardation puts a lower bound on how much a subtask's virtual time can be behind its real time, whereas acceleration puts an upper bound on how much a subtask's virtual time can be a head of its real time.

Suppose that a task t composed of 5 subtasks $s_0, ..., s_4$, where d_i represents the deviation of a subtask from its expected finish time. The *asynchrony* window and the *overall deviation* window are illustrated in Figure 11.

For each task t, its pivotal point is checked whether it falls within the *overall deviation* window. As shown in Figure 12, the outcome from the overall deviation test can be one of the following:

- Yes, the pivotal point falls within the *overall deviation* window. If this is the case, then the asynchrony test is performed to assure that $Async$ (the difference between virtual time of the fastest subtask and the virtual time of the slowest subtask among all subtasks belonging to task t) is within the asynchrony window.
- No, the pivotal point falls outside of the *overall deviation* window. In this case or in the case where the asynchrony test fails, corrective action is required.

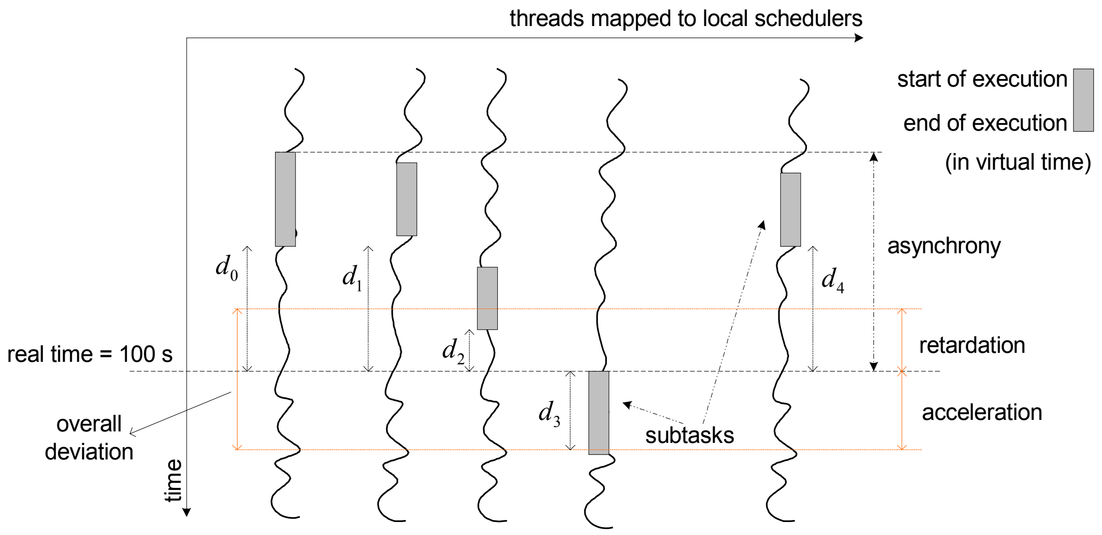


Figure 11: Progress of subtasks in the first schedule cycle.

```
// The Grid controller executes this code.
// queue containing all subtasks belonging to a hard QoS task $t$.
// QoSQ is the queue containing all hard and soft QoS tasks and subtasks.
// async is as defined in subsection 5.8.2 by the equation $async = VT_f - VT_s$

Monitor(queue) {
    while (QoSQ is not empty)
        //dequeue all subtasks belonging to a hard QoS task $t$
        queue = dequeue(QoSQ)

        //calculate pivotal point of task $t$ where its subtasks are in queue
        pp = calculate_pp(queue)

        if (pp is within the overall window)
            if (async > asynchrony window)
                corrective_action(queue)
            endif
        else
            corrective_action(queue)
        endif
    endwhile
}
```

Figure 12: Pseudo-code for detection of asynchrony.

5.9.3 Corrective Action

At this point, the Grid controller (based on information collected globally) signals a local machine for a corrective action. The corrective action can be to speedup or slowdown a subtask s_i. The local machine might succeed or fail in carrying out the corrective action locally.

Failure can happen in situations where subtask s_i needs to speed up and the local machine is overloaded. In other words, the local machine has no extra CPU quanta to spare. In this case, the local machine reports back to the Grid controller for a global corrective action to take place.

On the other hand, success can happen in situations where subtask s_i needs to slow down, which means that the local machine subtask s_i is running on is under loaded. In this case there are extra CPU quanta that are given to subtask s_i. One way, the extra CPU quanta can be absorbed is to create *an idle* task and assigns it a weight equal to the extra CPU quanta. Care has to be taken of whether to penalize subtask s_i and lower its weight to compensate for the extra CPU quanta it absorbed.

Since SQ is an ongoing feedback process, its effectiveness might take a few schedule cycles before satisfying the QoS attributes given by the client (*asynchrony*, and *overall deviation*). Figure 13 presents the monitor module within SQ the co-allocation scheme.

5.10 Isolation Guarantee

By using relaxed admission control, SQ admits more load (tasks/subtasks competing for a local resource) than the available local resource bandwidth. As mentioned at the end of Section 5.6, Grid QoS task may have hard QoS or soft QoS requirements. Allowing more demand than what is available (i.e. implementing relaxed admission control) assures that some of the QoS tasks/subtasks will be getting less CPU quantum/quanta than what they expressed in their weights. Since missing a deadline for hard QoS tasks/subtask will result in a catastrophic failure, the trade off that SQ makes is to borrow the needed CPU quantum/quanta from soft QoS tasks/subtasks by reducing their weights and lending the borrowed weights to the needy hard QoS tasks/subtask.

```
// The Grid controller executes this code.
// queue containing all subtasks belonging to a hard QoS task t .
// subtask is one of the subtasks of the hard QoS task t .
corrective_action(queue) {
        while (queue is not empty)
                //dequeue a subtask
                subtask  = dequeue(queue)
                //determine the appropriate action to be taken.
                // the action can be speeding up or slowing down the subtask
                action = determine_action(subtask)
                //signal the subtask's local machine to carry the action
                //if the action can not be carried locally
                        //mark this subtask's action to be carried globally
        endwhile
}
```

Figure 13: The Global controller corrective action module in SQ.

A needy hard QoS task/subtask will have its weight increased and this will not affect any other hard QoS tasks/subtask because their weights are not affected and hence SFQ will assure their share of the CPU remains the same. Therefore, SQ guarantees a total isolation between the hard QoS tasks/subtasks.

Furthermore, whatever happens (increasing or decreasing tasks/subtasks' weights) in QoSQ does not affect the other two queues (LQ and BEQ) because each weight associated with LQ and BEQ is not affected and thus the interQueue scheduler (SFQ) assures LQ and BEQ their share of CPU remains the same. In conclusion, SQ guarantees a total isolation between the tasks/subtasks in QoSQ as well as a total isolation between the three different queues (LQ, QoSQ, and BEQ).

5.11 Scheduling Concepts With SQ

Introducing the hierarchy of schedulers such as interQueue and intraQueue schedulers SQ uses in a new concept introduced. The concept of *real* and *virtual* time [Zha91] is used by Lixia Zhang as a data traffic control in high-speed networks. The concept is used by SQ but is extensively altered to account for the previous work done by each task/subtask so SQ can provide an aggregated work. Also, the scheme of calculating the *co-allocation skew* detailed in Section 5.9 is original with this thesis

6 SIMULATION RESULTS AND DISCUSSION

6.1 Overview

The Grid topology model used in the simulation is discussed in detail in section 6.2. The simulation model is written using Java base classes [ArN99] and further extensively modified to fit our purpose. The effectiveness and performance of SQ are assessed by writing a discrete event simulation modeling the Grid topology shown in Figure 14. The proposed SQ algorithm for synchronizing multimedia applications was simulated using the Advanced Networking Research Laboratory (ANRL) facilities.

The performance of SQ was compared to the traditional QoS with strict admission control to assess the advantages and to show the benefits of SQ. The next sections describe the simulation model, the performance measures, and parameters used in the simulation. This is followed by the simulation results and discussions.

6.2 Goals of The Simulation

The goal of the simulation is to investigate and examine the co-allocation problem. The co-allocation is defined as simultaneous allocation of resources to subtasks belonging to a task running on geographically distributed machines. The goal of the simulation is to look into and focus on the co-allocation problem and explore the effectiveness of SQ in reducing (minimizing) the co-allocation skew among the different subtasks belonging to task t.

Let us now think about how the co-allocation skew occur? Let task t composed of 2 subtasks s_0 and s_1. Furthermore, let s_0 be asking for 2 and s_1 be asking for 1 CPU quanta every schedule cycle. The co-allocation skew occurs when the scheduler start giving the subtasks CPU quanta different from what they asked for. But if, for every schedule cycle for the life time of the subtasks, the scheduler gives 2 and 1 CPU quanta to subtask s_0 and s_1 respectively, there will be no co-allocation skew and both of the subtasks will be executing in synchronization. This is the optimal scenario, but in real system this is not the case because machines can be underloaded or overload? If the machine is underloaded, then the subtasks will get more than what they asked for, and if the machine is overloaded the tasks will get less than what they asked for. In both cases, co-allocation problem will occur. Underloaded situations can happen with having strict admission control, but for overloaded situations one might ask: QoS is provided by having admission control. Hence, the system will never be overloaded and the following condition will hold true $w - r_{q2} \geq 0$, where $w = \sum_{i=0}^{n-1} r_i$, i.e. w is the sum of the weights for all the tasks and subtasks in QoSQ competing for the local resource, and r_{q2} is the available weight associated with the QoSQ. The answer to such question is yes. This is true if strict admission control is used, but SQ uses relaxed admission where the situation $w - r_{q2} < 0$ will occur for sure because we are admitting more total CPU demand (expressed in tasks/subtask weight) for a local resource than the availability of CPU cycles at hand. Hence, the probability of co-allocation skew situations will occur more often for SQ than a strict admission control-scheme.

Now, let us think about what contributes to the complexity of the co-allocation skew? That is what makes it a harder problem to manage? When we look at a co-allocation skew, the number of subtasks plays the primary role in the complexity of the problem. In the simulation study, I looked at situations where co-allocation skew is somewhat an easy problem to manage and at the same time I compared and investigated the situations where the co-allocation skew problem is much harder. In particular, I increased the offered load with the increased number of local machines. What can be learned from this as opposed to a study in which job resource requirement is fixed while number of machines is increased? I can purely test the effectiveness of SQ as well as QoS schemes as the co-allocation gets harder and harder to manage (i.e. as the number of subtasks increase).

6.3 Simulation Model

The Grid topology model used in the simulation is shown in Figure 14. Each tasks generator model generates a Poisson stream of tasks/subtasks with specified Mean InterArrival Time (λ) until a specific number of tasks have been generated. The local machines are heterogeneous and each reports the execution progress of its co-allocation subtasks to the Grid controller. Each machine has a local generator generating best effort local tasks and assigns them to the local queue (LQ). Two global generators are at the Grid level and they are responsible of generating Grid QoS and Grid best effort tasks/subtasks. The global tasks/subtasks are assigned to the Grid QoS queue (QoSQ) and the Grid best effort queue (BEQ) accordingly. Furthermore, the Grid QoS tasks are stochastically divided into hard and soft QoS tasks.

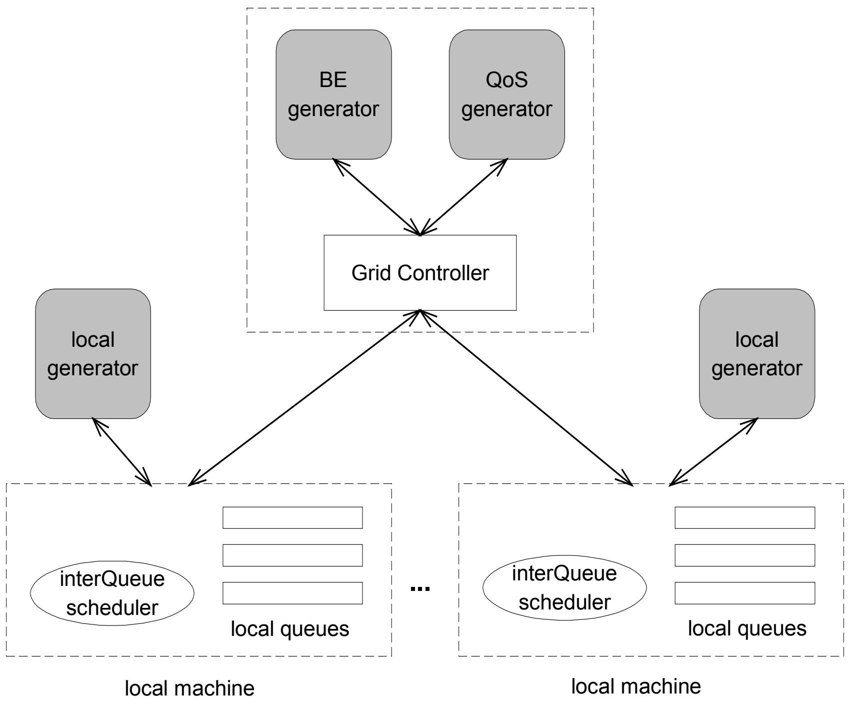


Figure 14: Grid topology used in the simulation.

To sufficiently assess the performance of SQ co-allocation algorithm, various performance metrics need to be explored. Even though the co-allocation skew is one of the quantities that the SQ algorithm is trying to minimize, the trade-offs and their impact on the system should be examined. The trade-offs that are examined in the simulation are acceptance ratio, QoS conformance, and effective machine usage.

The four performance metrics used to assess the performance of SQ are acceptance ratio, effective machine usage, QoS conformance, and average co-allocation skew and they are defined as:

$$\text{acceptance ratio} = \frac{\text{number of QoS tasks accepted}}{\text{total number of QoS tasks generated}}$$

$$\text{effective machine usage} = \frac{\text{number of hard QoS tasks conforming to } \textit{asynchrony} \text{ window}}{\text{total number of hard QoS tasks accepted}}$$

$$\text{QoS conformance} = \frac{\text{number of hard QoS tasks confirming to } \textit{overall deviation} \text{ window}}{\text{total number of hard QoS tasks accepted}}$$

$$\text{average co-allocation skew} = \frac{\sum_{i=1}^{i=m}\left(\sum_{j=0}^{j=n-2}\left(VT_{ij} - VT_{is}\right)/(n-1)\right)}{\text{total number of hard QoS tasks accepted}}$$

, where VT_{ij} is the virtual time of subtask j of task i, and VT_{is} is the virtual time of the slowest subtask belonging to task i.

The term randomly generated over a range [a, b] means that the number is generated using a discrete (integer-valued) uniform distribution over $a, a+1, ..., b$ inclusive. That is written as U[a, b]. The Grid topology used in the simulation consists of local machines ($Nloc$) set deterministically at [5,10,15, 25] and 3 generators each generating tasks (t) randomly generated over a range [1000, 2000]. For each simulation run, the generators generate a Poisson stream of tasks with specified λ set deterministically at [10, 100, 200, 500] seconds. For each QoS task, the two QoS attributes provided by the user are *asynchrony* and *overall deviation*, which are randomly generated over a range [100, 500] seconds. Furthermore, each Grid task is composed of subtasks (n) randomly generated over a range [1, # of local machines] and each of these subtasks is assigned an execution time (μ) randomly generated over a range [1500, 2000] seconds. The value μ is chosen to be large enough to resemble a continuous media application so SQ (being a feed back scheme) will have enough time to "kick in" and carry its corrective action and hence be effective. The CPU speed for each local machine ($LCPU$) is randomly generated over a range [100, 600] MHz. A Grid level CPU bandwidth ($GCPU$) of one GHz is assumed when assigning Grid tasks/subtasks to local machines as explained in section 5.6.1 and the CPU bandwidth of each local machine is statically partitioned among the 3 flows of tasks/subtasks. Furthermore, a weight (m) is assigned to a Grid QoS task t and a weight (r_{ai}) is assigned to a thread representing subtask s_i belonging to task t (refer to Subsection 5.6.1). Also, r_a is the sum of the weights for all the subtasks belonging to task t. The two weights *m,* and r_{ai} are randomly generated over a range [1,5] of CPU quanta. For the sake of computing mean value analysis, I will refer to *m,* r_{ai}, and r_a as representing the mean value of their associated weights respectively. Table 2 and Table 3 show the design and exogenous parameters used in the simulation. In addition, Table 4 shows the two algorithms and their parameters used in the simulation.

Table 2: Design parameters used in the simulation.

Symbol	Definition	Design Parameters Values
λ	mean inter-arrival time (second)	$\lambda = (10,100,200,500)$
Nloc	Number of machines	$Nloc = (5,10,15,25)$
GCPU	the assumed Grid level standard CPU speed	$GCPU = 1\ GHz$
Reps	how many times the simulation run is repeated for each point in the graphs	$\mathrm{Re}\,ps = 50$

Table 3: Exogenous parameters used in the simulation.

Label	Definition	Exogenous Parameters Distribution
assynchrony	QoS attribute specified by user (second)	$assynchrony = U[100,500]$
overall deviation	QoS attribute specified by user (second)	$overall\ deviation = U[100,500]$
n	Number of subtasks	$n = U[1, Nloc]$
μ	Number of the execution time (second)	$\mu = U[1500,2000]$
m	task's weight (CPU quanta)	$m = U[1,5]$
r_{ai}	Subtask's weight (CPU quanta)	$r_{ai} = U[1,5]$
LCPU	CPU speed of local machine	$LCPU = U[100,600]\ MHz$
t	Number of tasks	$t = U[1000,2000]$

Table 4: Different classes of algorithms used in the simulation.

Parameter	Algorithm Used	
	Traditional QoS guarantees	SQ
IntraAlgorithm	SFQ	SFQ
Admission Control	Strict	Relaxed
Queue Used	Grid QoS	Grid QoS
Performance Metric Used	All	All

6.4 Mean Value Utilization Analysis

In this subsection, mean value utilization analysis is performed to compute total resource demand and compare it to available resources (CPU cycles). After that I relate to admission control to help in explaining the results of the simulation. For each entry in Table 5, ρ is calculated as *total CPU demand* over *CPU cycles available*. *Total CPU demand* and *CPU cycles available* are computed over an interval of 100 seconds.

Total CPU demand **is calculated as:**

mean number of tasks generated every 100 seconds * *n* * mean subtask's weight. Tasks generated every 100 seconds $= \frac{100}{\lambda}$ and mean subtask's weight (based on 100% CPU availability) = $\frac{r_{ai} * m * GCPU}{r_a * LCPU}$, where $r_a = n * r_{ai}$

Therefore, *total CPU demand* is given by $\frac{100}{\lambda} * n * \frac{r_{ai} * m * GCPU}{(n * r_{ai}) * LCPU}$. Furthermore, *total CPU demand* (based on the weight associated with QoSQ which is r_{q2}) is $\frac{100}{\lambda} * n * \frac{r_{ai} * m * GCPU}{(n * r_{ai}) * LCPU} * \frac{r_{q2}}{100} = \frac{100}{\lambda} * \frac{m * GCPU}{LCPU} * \frac{r_{q2}}{100}$. Refer to subsection 5.6.1. for more detail.

CPU cycles available **is calculated as:**

CPU cycles available every 100 seconds for QoSQ * *Nloc* = r_{q2} * *Nloc*.

Finally, $\rho = \left(\frac{100}{\lambda} * \frac{m * GCPU}{LCPU} * \frac{r_{q2}}{100} \middle/ r_{q2} * Nloc \right)$. Since *GCPU*, *LCPU*, *m*, and r_{q2} are constants, the above equation can be simplified as follows:

$$\rho = \left(\frac{100}{\lambda} * \frac{3 * 1{,}000{,}000{,}000}{350{,}000{,}000} * \frac{40}{100} \middle/ 40 * Nloc \right) = \frac{60}{7 * \lambda * Nloc}.$$

Table 5: Mean value utilization of the different number of machines as λ increase.

λ	Number of machines			
	5	10	15	25
10	$\rho = 17.1\%$	$\rho = 8.57\%$	$\rho = 5.71\%$	$\rho = 3.43\%$
100	$\rho = 1.71\%$	$\rho = 0.86\%$	$\rho = 0.57\%$	$\rho = 0.34\%$
200	$\rho = 0.86\%$	$\rho = .43\%$	$\rho = 0.29\%$	$\rho = 0.17\%$
500	$\rho = 0.34\%$	$\rho = 0.17\%$	$\rho = 0.11\%$	$\rho = 0.07\%$

6.5 Simulation Results

The two scheduling techniques: traditional QoS algorithm (QoS) and SQ algorithm (refer to Table 4) were implemented and compared. Performance measures are presented for different number of machines and different values of λ. Each point in the graphs below is the result of 50 simulation runs. In each simulation run, a random number of tasks/subtasks for each of the three types of traffic was generated.

Simulation results are presented separately for each of the performance metrics. While section 6.3.1 shows that the SQ algorithm is working as intended by minimizing the co-allocation skew quantity, subsequent sections show how the SQ algorithm is working and what are the trade-offs, which are examined by a) determining the machine utilization in terms of the effective cycle usage, b) showing how more tasks are included with relaxed

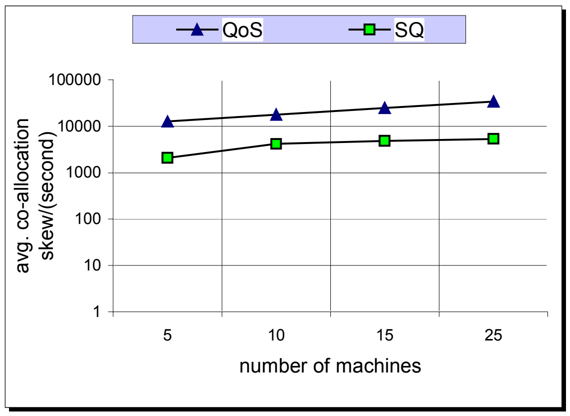
SQ admission control and what is their impact, and c) showing the ratio of the QoS conformance. The simulation results and discussion are presented below.

6.5.1 Co-allocation Skew Average

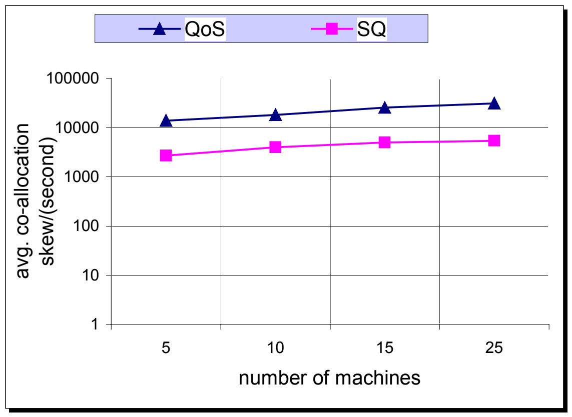
The co-allocation skew average for different number of machines and different values of λ is presented in Figure 15. This scenario is simulated for the two different algorithms presented in Table 5. It can be noted from Figure 15 that the average co-allocation skew is the highest for QoS.

Since the number of subtasks generated to form each of the GridQoS and GridBE is randomly chosen in the range of [1, # of machines], the number of subtasks will, on average, increase as the number of machines increases. Hence, the co-allocation skew for the two algorithms tends to increase with an increase in the number of machines. I chose to increase offered load with the increased number of local machines because I wanted to test SQ under both situations that cause the co-allocation problem. Particularly under the situation where the co-allocation problem is somehow an easy problem to manage (i.e. number of subtasks is small) as well as under the situation where the co-allocation problem is a much harder problem to manage (number of subtasks is much larger).

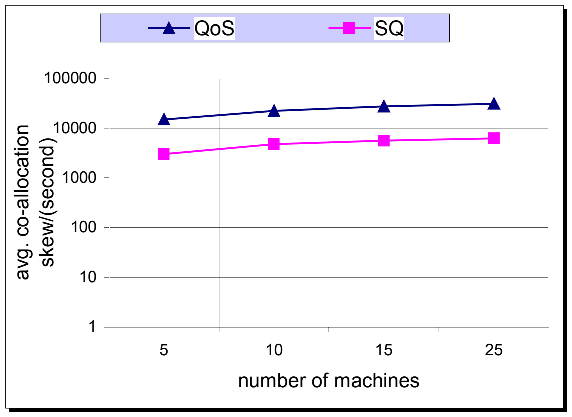
SQ has the lowest average allocation skew amongst overall. Especially compared with QoS, which uses strict admission control, SQ outperforms QoS for the different number of machines as well as for different values of λ.



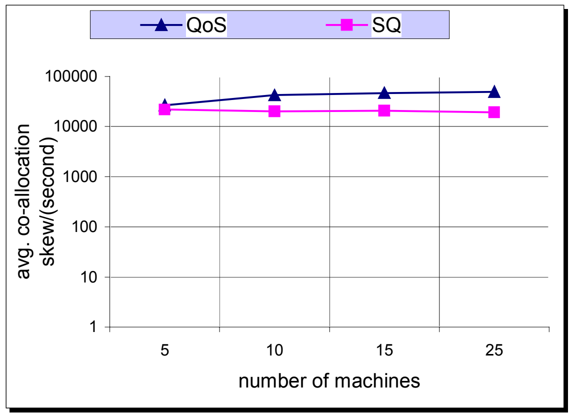
(a) Co - allocation skew variation with λ of 10 seconds.



(b) Co - allocation skew variation with λ of 100 seconds.



(c) Co - allocation skew variation with λ of 200 seconds.



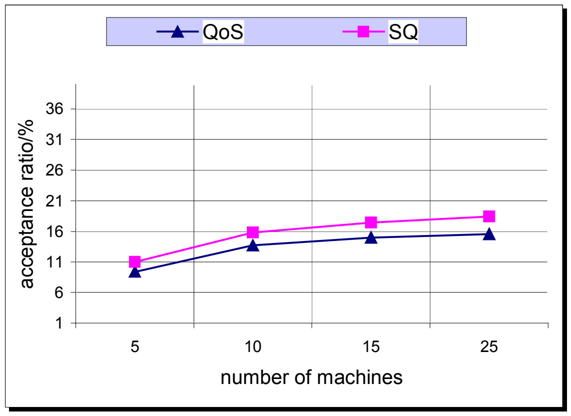
(d) Co - allocation skew variation with λ of 500 seconds.

Figure 15: co-allocation skew for different number of machines.

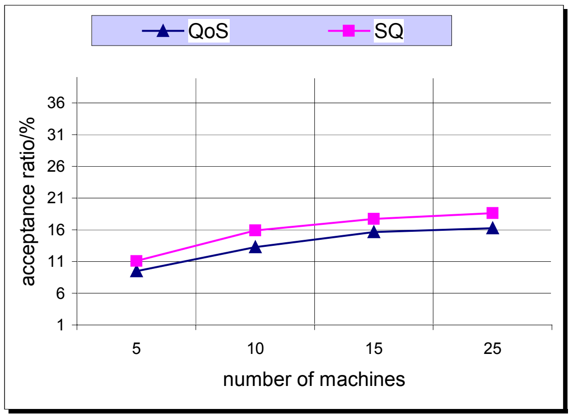
6.5.2 Acceptance Ratio

The acceptance ratio for different number of machines for the QoS and SQ algorithms with different values of λ is presented in Figure 16. Every time a QoS task/subtask is admitted to a local machine, the admission control quantity x of the local machine is decreased by the task/subtask's weight. It should be noted that admission control helps you to maintain a desired level of QoS by limiting the number of the tasks/subtasks competing for a local resource but admission control does not guarantee QoS as illustrated from the simulation study as illustrated in Figure 15. Once the quantity x is zero, no tasks/subtasks are admitted to the local machine. Once a task/subtask is finished execution, its weight is added to the quantity x, so more tasks/subtask can be admitted.

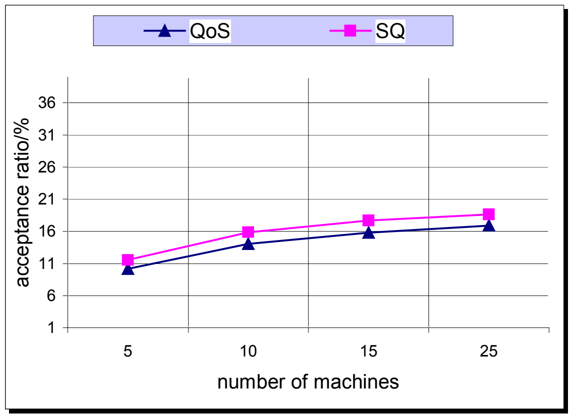
Therefore, for each algorithm, the acceptance ratio depends on the length or the lifetime of tasks/subtask assigned to the QoSQ. The length of tasks/subtasks is basically the execution time. Once the first patch of tasks/subtasks are accepted, which means that the quantity x is zero, then future tasks/subtasks can be admitted if a current task/subtask finishes execution and leaves the QoSQ. For $\lambda = 10,100,$ and 200, the Grid QoS generator is generating tasks/subtask much faster than the lifetime of theses tasks/subtasks already admitted to the QoSQ. Once the intraQueue scheduler (SFQ) for the QoSQ finishes executing a task/subtask and hence the admission control unit is in a position to admit more Grid QoS tasks/subtask, the Grid QoS generator would have finished generating tasks/subtasks. That is the reason why the results for $\lambda = 10,100,$ and 200 are somehow identical. Whereas for $\lambda = 500$ once the intraQueue scheduler (SFQ) for the QoSQ finishes executing a task/subtask and hence the admission control unit is in a position to admit more Grid QoS tasks/subtask, the Grid QoS generator is still generating tasks/subtasks because the generation process is a lot slower than in the cases where $\lambda = 10,100,$ and 200. But overall, the acceptance ratio for both of the algorithms increases as the number of machines increase. Also since SQ uses relaxed admission, its ratio acceptance is overall higher than QoS.



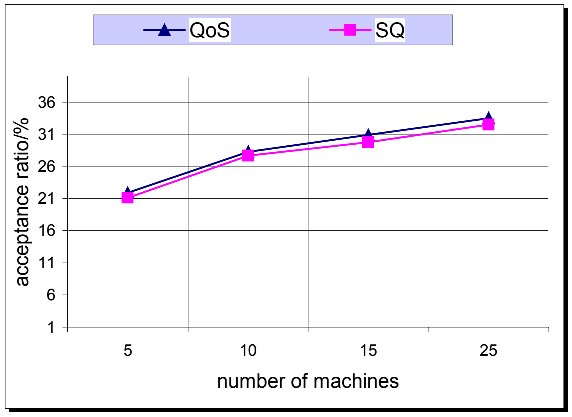
(a) Acceptance ratio with λ of 10 seconds.



(b) Acceptance ratio with λ of 100 seconds.



(c) Acceptance ratio with λ of 200 seconds.



(d) Acceptance ratio with λ of 500 seconds.

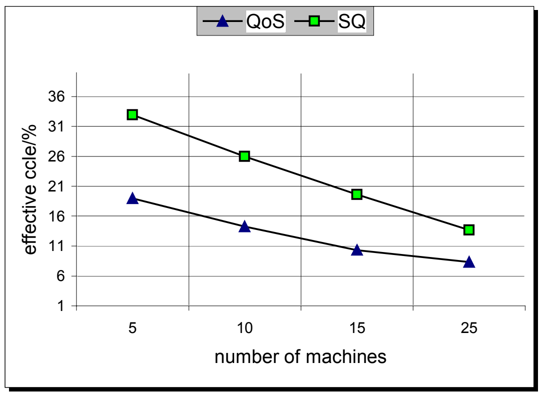
Figure 16: Variation of acceptance ratio with number of machines.

6.5.3 Effective machine usage

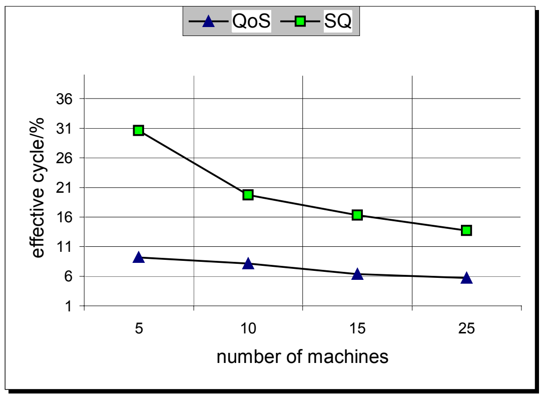
One of the two windows that SQ uses to synchronize hard QoS tasks is the *asynchrony* window. *Asynchrony* is one of the two QoS attributes the client provides (refer to Section 5.8). Effective machine usage is calculated as the ratio between the number of hard QoS tasks confirming to the *asynchrony* window and the total number of of hard QoS accepted. From Figure 17, it can be noticed that overall SQ has a better effective machine usage than QoS because of the corrective action taken by SQ. The fastest and the slowest subtasks of each hard QoS task are the only two subtasks from each hard QoS task that are of concern to us here.

The corrective action can be slowing down or speeding up a subtask. Consider a situation where subtask s_i needs to slow down, meaning that the local machine is under loaded. In this case there are extra CPU quanta given to subtask s_i. Therefore, the extra CPU quanta can be absorbed by creating an *idle* task and that can be thought of as forcing the local machine to be idle and as a consequence wasting some of its CPU quanta. On the other hand, speeding up subtask s_i means taking back the CPU quanta needed by s_i from the *idle* task. If there is no *idle* task in the QoSQ, then the needed CPU quanta is taken from a soft QoS task/subtask if one exists.

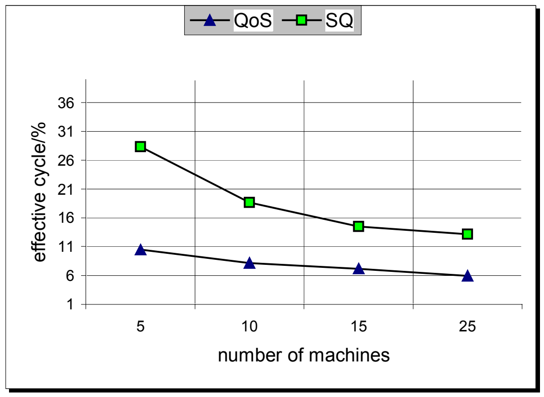
As part of the simulation, this trade-off of forcing a local machine sometimes to have ineffective (idle) schedule cycles has been explored as shown in Figure 17. For each schedule cycle (monitor cycle), the asynchrony test for each hard QoS task t is performed. If it succeeds, the schedule cycle is counted as an effective cycle for task t otherwise the schedule cycle is counted as ineffective cycle for task t. As the number of the machines increase, the number of subtasks for task t increase as well and the ability to synchronize all these subtasks becomes more difficult.



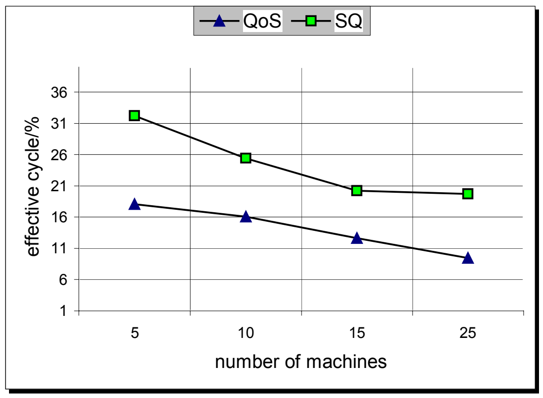
(a) Effective machine usage with λ of 10 seconds.



(b) Effective machine usage with λ of 100 seconds.



(c) Effective machine usage with λ of 200 seconds.



(d) Effective machine usage with λ of 500 seconds.

Figure 17: Effective machine usage for different number of machines.

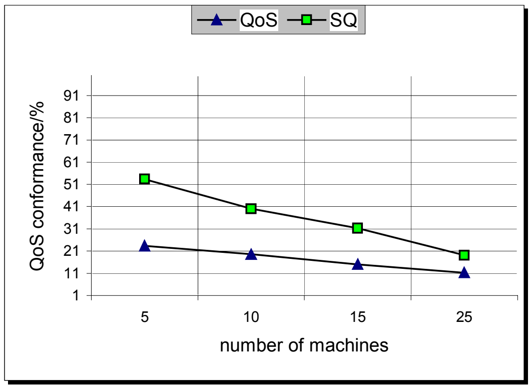
Therefore, as presented in Figure 17, the ability of confirming subtasks to the *asynchrony* window increases as the number of machines decrease. SQ has a higher success ratio than QoS of confirming the subtasks to the *asynchrony* window, while at the same time maintaining less co-allocation skew and accepting more QoS tasks.

6.5.4 QoS conformance

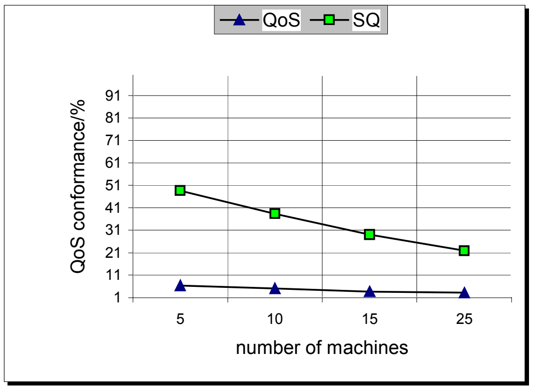
The other window that SQ uses to synchronize hard QoS tasks is the *overall deviation* window. *Overall deviation* is one of the two QoS attributes the client provides (refer to Section 5.8). QoS conformance is calculated as the ratio between the hard QoS tasks confirming to the *overall deviation* window and the total of hard QoS accepted. From Figure 18, it can be noticed that overall SQ has a better QoS conformance than QoS because of the corrective action taken by SQ.

The corrective action can be slowing down or speeding up a subtask as explained in the previous subsection. From Figure 18, we observe that QoS conformance decrease with the increase of machines numbers. As the machines number increase, the number of subtasks for task t increase as well and the task of synchronization becomes more difficult due to the increase of subtasks. This phenomenon affects the co-allocation skew and the effective machine usage as well.

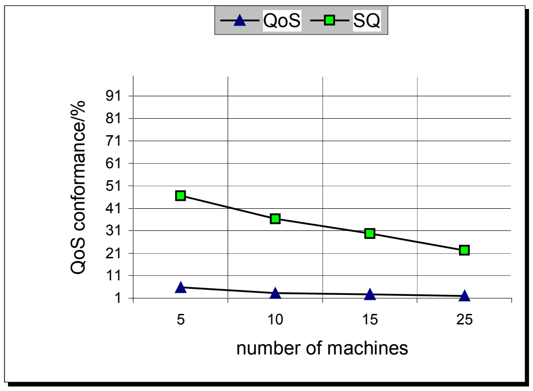
Therefore, as presented in Figure 18, the ability of confirming subtasks to the *overall deviation* window increases as the number of machines decrease. SQ has a higher success ratio than QoS of confirming the subtasks to the *overall deviation* window, while at the same time maintaining lower co-allocation skew and accepting more QoS tasks.



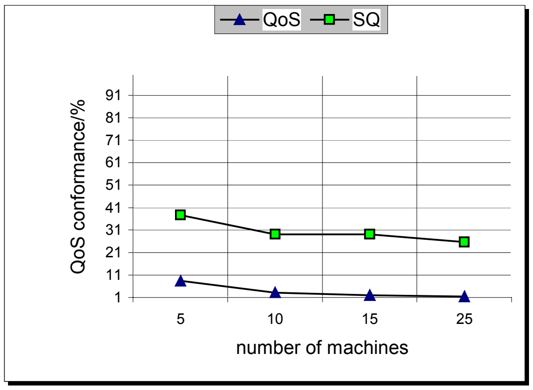
(a) QoS conformance with λ of 10 seconds.



(b) QoS conformanc e with λ of 100 seconds.

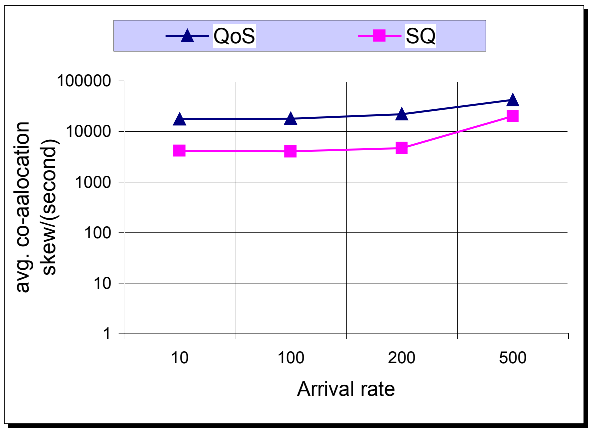


(c) QoS conformance with λ of 200 seconds.

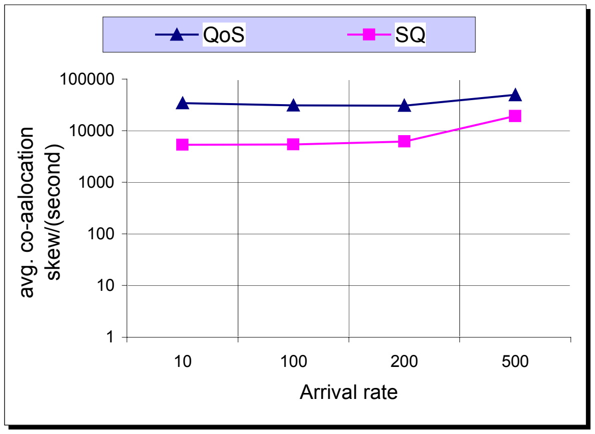


(d) QoS conformance with λ of 500 seconds.

Figure 18: QoS conformance for different number of machines.



(a) Avg. co - allocation skew for number of machines = 10.



(b) Avg. co - allocation skew for number of machines = 25.

Figure 19: Average co-allocation skew for different values of λ.

6.6 Simulation Discussion

Table 5 shows the mean value utilization analysis for the different machines with the various values of λ. From this analysis it can be concluded that the simulation performed represents underload situations and therefore are not subject to the QoS acceptance bottleneck (admission control).

As stated in Section 6.2, the goals of the simulation are to investigate and examine the co-allocation problem under different situation. Since the co-allocation problem gets more harder to manage with the increase of the number of subtasks, I chose to increase the offered load (i.e. number of subtasks) with the increased number of local machines. Also, since SQ is a feed back process, tasks/subtasks must stay in the system long enough for SQ to "kick in". Therefore, the execution time μ is chosen to be large enough to resemble a continuous media application so SQ, being a feed back scheme, will have enough time to "kick in" and carry its corrective action and hence be effective.

Using admission control whether strict or relaxed will allow the first batch or group of tasks/subtasks into the QoSQ and then do not accept any more tasks/subtask until a task/subtask or a group of tasks/subtasks in the QoSQ finish execution and leave the QoSQ. Once, some tasks/subtasks leave the QoSQ, more Grid QoS tasks/subtasks can be admitted. For $\lambda = 10, 100,$ and 200, the Grid QoS generator is generating tasks/subtask much faster than the lifetime of theses tasks/subtasks already admitted to the QoSQ. Once the intraQueue scheduler (SFQ) for the QoSQ finishes executing a task/subtask and hence the admission control unit is in a position to admit more Grid QoS tasks/subtask, the Grid QoS generator would have finished generating tasks/subtasks. That is the reason why the results for $\lambda = 10,100,$ and 200 are somehow identical. Whereas for $\lambda = 500$ once the intraQueue scheduler (SFQ) for the QoSQ finishes executing a task/subtask and hence the admission control unit is in a position to admit more Grid QoS tasks/subtask, the Grid QoS generator is still generating tasks/subtasks because the generation process is a lot slower than in the cases where $\lambda = 10,100,$ and 200.

The overall goal of SQ is maintained where continuous media applications, expressed in large value of μ, are examined and tested at situations where co-allocation skew is somewhat an easy problem to manage and at the same time I compared and investigated the situations where the co-allocation skew problem is much harder. In particular, I increased the offered load with the increased number of local machines to purely test the effectiveness of SQ and QoS schemes as the co-allocation gets harder and harder to manage (i.e. as the number of subtasks increase) as opposed to a study in which the offered load is fixed while number of machines is increased.

7 Conclusions and Future Work

7.1 Concluding Remarks

Motivated by the successes of network computing, researchers have started examining a more generalized resource/information sharing and integration infrastructure called the Grid which is a generalized, large-scale computing and data handling virtual system that is formed by aggregating the services provided by several distributed resources.

The MEG is a concept to support the deployment of multimedia services and can potentially provide pervasive, dependable, consistent, and cost-effective access to the diverse services provided by the distributed resources and support problem solving environments that may be constructed using such resources. A user in such an environment is not tied to a specific machine but rather is a machine independent entity that exists in the Grid and can transparently carry its profile across the different platforms constituting the Grid. Some of the enabling technologies that will be supported include: (a) *quality of service* (QoS), (b) multicast, (c) streaming data, (d) co-allocation of resources, and (e) resource discovery.

This thesis addressed one of these issues, which is *co-allocation*. The co-allocation issue that is addressed in this thesis is concerned with ensuring that an application that has several subtasks would be allocated sufficient resources so that all subtasks of the application can make satisfactory progress with their execution. The co-allocation is an essential feature for several important classes of multimedia applications. One example of these multimedia applications would be interleaved media streams where co-allocation is needed for each media stream (e.g. audio, video, color, etc) before such multimedia applications can be deployed for widespread use.

The SQ co-allocation scheme is proposed for MEGs. The contributions of this scheme are:

- a memory-oriented QoS capability: SQ is a scheme that assures the total work accomplished by each subtask s_i for each schedule cycle is accounted for. In other words, SQ is an aggregated scheme that remembers the total work accomplished by each subtask s_i in the previous schedule cycles.
- an environment-aware QoS capability: SQ is a scheme that assures the aggregated work accomplished by each subtask s_i does not fall behind the other subtasks belonging to task t. These other subtasks are running in different environments and thus it is a key point of SQ to have an environment-aware QoS capability.
- a framework for co-allocation with the ability to co-allocate heterogeneous resources in a Grid setting without the need for advance reservation.
- a framework for co-allocation with the ability to over subscribe resources and thus leading to a better utilization of the overall system than other schemes.

The algorithm and architecture for implementing SQ are presented. Simulation studies performed to evaluate SQ indicate that it outperforms admission control-based scheme by a significant margin. The simulation studies were performed for various number of machines and inter-arrival times.

7.2 Future Work

It will be interesting to compare SQ with an *advance reservation-based* scheme where resources are reserved in advance where the client specify a start and duration of time to use the resource(s).

Also, the SQ co-allocation scheme minimizes the co-allocation skew for hard QoS tasks Future work can expand this to include Soft QoS tasks as well. . Right now if CPU quanta is needed to speed up a hard QoS subtask and there is no *idle* task in the QoSQ, then the needed CPU quanta is taken from a soft QoS task/subtask if one exists. It should be pointed out that when taking a CPU quantum or quanta from a soft QoS task/subtask, fairness is taken in consideration so that deprivations of one specific soft QoS is minimized as much as possible.

Furthermore, I chose to increase offered load with the increased number of local machines because I wanted to test SQ under both situations that cause the co-allocation problem to happen. Particularly under the situation where the co-allocation problem is somehow an easy problem to manage as well as under the situation where the co-allocation problem is a much harder problem to manage. In future work, a study in which job resource requirement is fixed while number of machines is increased can be performed to examine if SQ is more effective than QoS in finishing the task faster while minimizing the co-allocation skew.

Therefore, in future work: (a) the SQ scheme can be compared to an *advanced reservation-based* scheme to show and compare the strengths and weaknesses of SQ; (b) the SQ scheme can be expanded to assure that the co-allocation skew of Soft QoS tasks is monitored to assure that the co-allocation skew is minimized for all the QoS tasks including hard and soft QoS tasks; and (c) a study in which job resource requirement is fixed while number of machines is increased can be performed to examine if SQ is more effective than QoS in finishing the task faster while minimizing the co-allocation skew.

References

[ArC98] A. C. Arpaci-Dusseau, D. E. Culler, and A. M. Mainwaring, "Scheduling with implicit information in distributed systems," *SIGMETRICS Conference on the Measurement and Modeling of Computer Systems,* June 1998, pp. 233-243.

[ArN99] N. Arnason, *Graduate class notes*, Department of Computer Science, University of Manitoba, ftp://ftp.cs.umanitoba.ca/pub/arnason/simjava, 1999.

[AzM00] F. A. Azzedin, and M. Maheswaran, *Design of a Quality of Service Aware API in Java,* ANRL Research Note ANRL-01-00, Department of Computer Science, University of Manitoba, 2000.

[BaB00] M. A. Baker, R. Buyya, and D. Laforenza,"The Grid: International efforts in global computing," *ACM Computing Surveys,* Oct. 2000.

[CzF98] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke, "A resource management architecture for metacomputing systems," *4th Workshop on Job Scheduling Strategies for Parallel Processing*, Springer-Verlag LNCS 1459, 1998, pp. 62-82.

[GoG96] P. Goyal, X. Guo, and H. Vin, "A hierarchical CPU scheduler for multimedia operating systems," *Proceeding Second Symposium On Operating Systems Design and Implementation*, 1996, pp. 107-122.

[GoV96] P. Goyal, H. M. Vin, and H. Cheng, "Start time fair queuing: A scheduling algorithm for integrated services packet switching networks," *Proceeding of ACM SIGCOMM'96*, Aug. 1996, pp. 157-168.

[FeG97] D. Ferrari, A. Gupta, and G. Ventre, "Distributed advance reservation of real-time connections," *ACM/Springer-Verlag Journal on Multimedia Systems*, Vol. 5, No. 3, 1997.

[FoK98] I. Foster, C. Kesselman, *The Grid: Blueprint for a New Computing Infrastructure*, Morgan-Kaufmann, July 1998.

[FoK99] I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, and A. Roy, "A distributed resource management architecture that supports advance reservations and co-Allocation," *Proceeding of the International Workshop on Quality of Service*, 1999, pp. 27-36.

[JoG99] W. E. Johnston, D. Gannon, and B. Nitzberg, "Information power Grid implementation plan: research, development, and testbeds for high performance, widely distributed, collaborative, computing and information systems supporting science and engineering," *NASA Ames Research Center*, http://www.cs.nas.nasa.gov/IPG, 1999.

[KrM00] K. Krauter and M. Maheswaran, *Architecture for a Grid operating systems, IEEE/ACM International Workshop on Grid Computing (Grid 2000*), Dec. 2000.

[MaK00] M. Maheswaran and K. Krauter, "A Parametric-based approach to resource discovery in Grid computing systems," *1st IEEE/ACM International Workshop on Grid Computing (Grid 2000),* Dec. 2000.

[NiL97] J. Nieh and M. Lam, "The design, implementation and evaluation of SMART: A scheduler for multimedia applications," *Proceeding 16th Symposium on*

Operating System Principles, 1997, pp. 184-197.

[ScN99] O. Schelén, A. Nilsson, J. Norrgård, and S. Pink, "Performance of QoS agents for provisioning network resources," *Proceedings of* IFIP Seventh International Workshop on Quality of Service (IWQoS'99), London, UK, June 1999.

[Sta97] W. Stallings, *High-speed Networks, TCP/IP and ATM Design Principles*, Prentice Hall, New Jersey, 1997.

[WaW94] C. A. Waldspurger and W. E. Weihl, "Lottery scheduling: Flexible proportional-share resource management," *Proceeding First Symposium On Operating System Design and Implementation*, 1994, pp. 1-11.

[WaW95] C. A. Waldspurger and W. E. Weihl, *Stride scheduling: Deterministic proportional-share resource management*, Tech. Report. MIT, Cambridge, June 1995.

[YaD99] D. Yau, "ARC-H: Uniform CPU scheduling for heterogeneous services," *ICMCS,* Vol. 2, 1999, pp. 127-132.

[YaL96] D. Yau and S. S. Lam, "Adaptive rate controlled scheduling for multimedia applications," *ACM Multimedia Conference '96*, Nov. 1996.

[Zha91] L. Zhang, "Virtual Clock: A new traffic control algorithm for packet switching networks," Transactions on Computer Systems, Vol. 9, No. 2, 1991, pp. 101-124.