# On Increasing Resource Utilization of a High-Performance Computing System

Lung-Pin Chen(✉) and Yu-Shan Cheng

Computer Science and Information Science, Tunghai University, Taichung, Taiwan
{lbchen,g01350061}@thu.edu.tw

**Abstract.** In a render farm, a render host usually requires high I/O bandwidth for reading scene files and writing rendering results. A multi-core render host may only activate partial cores in order to restrict its total throughput to the bandwidth limitations. This paper investigates feasible ways for using resource fragments formed by inactivated cores. We develop a parallel application consisting of single-core malleable tasks with small input/output data, and demonstrate that it can be scheduled to adaptive to the unstable resource fragments of the render farm. We develop a broker to collect idle processor cores and control the execution of the malleable tasks in background mode. The experimental results show a significantly increasing on resource utilization of the render farm without interfering with the original rendering tasks.

**Keywords:** High performance computing · Parallel task scheduling · Render farm

## 1 Introduction

High-performance computing (HPC) refers to the techniques featuring a large number of computing nodes, *e.g.* supercomputers or computer clusters, to solve complex computational problems. It has been used in many areas including biosciences, climate modeling, computer games, *etc*. The parallel jobs supported by HPC systems are usually classified into the following four categories [1]: (1) rigid, (2) moldable, (3) evolving, and (4) malleable. A rigid job has to designate a fixed number of cores before scheduling. In contrast, both moldable and evolving job can designate a flexible range of core number. The difference is that evolving jobs can reconfigure the resources during execution, while moldable jobs cannot. Malleable jobs can change their processor requirements during execution by an *external* job scheduler.

A render farm is a high performance computer cluster that is built to render computer graphics. The input data of a computer graphic rendering task can be divided into a sequence of frames that can be rendered independently. Since computer graphic rendering is both compute and I/O intensive, parallelism has become a crucial tool to building a high performance computer graphic system.

When handling parallel jobs that require multiple simultaneous cores, unused computing resources called *fragments* often occur. For example, many rendering tasks require high I/O bandwidth for reading scene files and writing result data. For this sort


Y. Zhang et al. (Eds.): CloudComp 2015, LNICST 167, pp. 93–101, 2016.
DOI: 10.1007/978-3-319-38904-2_10

of applications, the worker computer only activates partial cores in order to restrict their total throughput to the I/O bandwidth.

Although a render farm can monitor and collect resource fragments, sharing unused resource to other HPC systems seems intractable. This paper demonstrates that the single-core malleable tasks with small input/output data size, such as heuristic search or game tree search applications, are suitable to be executed on the resource fragments of a render farm.

We implement a resource broker in the Qube render management system which can collects idle resources and make use of them to execute the malleable tasks. Our implementation work demonstrates a significantly increasing on resource utilization without interfering with the original rendering tasks.

In Sect. 2, we will introduce the application and the environment we used in the experiments. Section 3 will introduce the system architecture. Section 4 will discuss the scheduling methods for the malleable job scheduling system. Section 5 is the experiments section, which shows the resource utilization of the improved system. Finally, concluding remarks and future work are given in Sect. 6.

## 2 Background

Formosa 3 is a high performance render farm built by the National Center of High-Performance Computing (NCHC) in 2011, which is Taiwan's primary organization supporting supercomputing and high speed networking. Formosa 3 features a cluster of 76-node, each equipped with dual six-core hyperthreaded Intel Xeon 2.8 GHz processors and 48 GB memory. The InfiniBand 40 GB/s fabric serves as the interconnecting backbone of the cluster. Formosa 3 achieves a peak performance of 9 teraflops so far.

Formosa 3 employs Qube [15] as the render farm management system. Figure 1 depicts the architecture of the render farm management system. There are three main components in a Qube system:

- The *Worker* which is a program runs on every computer (called a "host" or a "worker host") on which users execute jobs. There are render daemons run on every worker host to receive instructions from the supervisor.
- The *Supervisor* which runs on a dedicated machine and uses a collection of Render Queues to keep track tasks. It decides when a job runs, and which job runs on which worker.
- The *Client* which is a computer let a user submit jobs via various interfaces, *e.g.* command line, standalone GUI, or in-application submission GUIs.

When rendering, the user first need to upload the scene files and all the necessary data to the NAS (network attached storage) server. After the job starts, the supervisor controls the execution of the tasks according to the *agenda* which specifies the sequence numbers of the frames and the number of cores required for parallel rendering. Upon receiving the job from the supervisor, the worker renders out individual frames. The rendering results are sent and stored to the NAS server.

Desktop grid is a network computing model that can harvest unused computational power from desktop level computers. In this paper we employ the CGDG which is a desktop grid platform designed for parallel game tree search applications [5].

## 3 System Design

### 3.1 Architecture

In this paper, the *resource stealing* approach is developed intended to fully utilize idle cores in the render farm without not interfering the rendering tasks. That is, the render farm will not sense the existence of the extra foreign tasks.

The proposed system involves a render farm (RF), a desktop grid (DG) and a component named ResourceMonitor, as illustrated in Fig. 1. In the system architecture, the ResourceMonitor is a process that continually monitors the resource usage of the render farm and then notifies the desktop grid server.

To avoid confusing, the workers of the desktop grid and the render farm are called *DG-worker* and *RF-workers*, respectively. To achieve resource stealing, the client-side software of the desktop grid have to be installed in the render host in order to turn it to be a DG-worker. A DG-worker that run in a render host is called a "*RFDG-workers*" hereinafter. We assume that the desktop grid establishes connection-oriented control links between the desktop grid servers and the workers, enabling direct control to the workers in a timely manner.

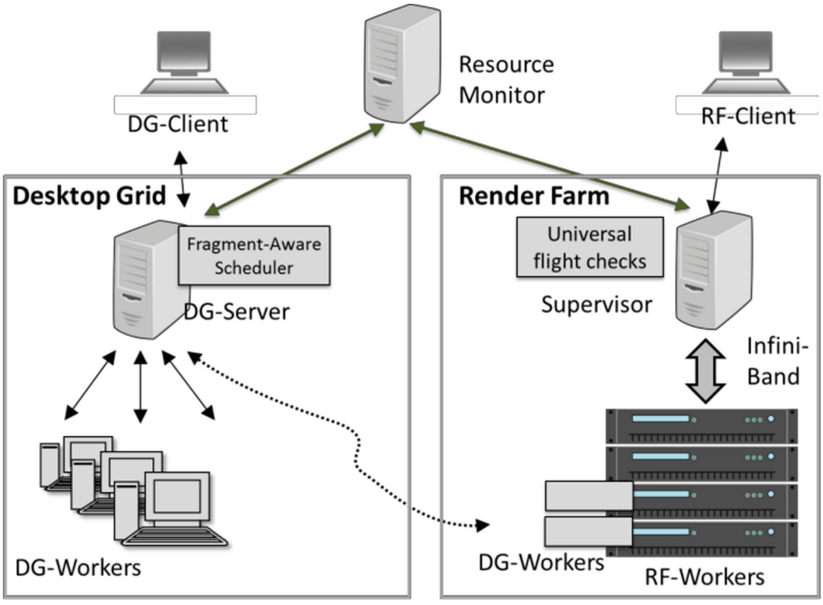


**Fig. 1.** The architecture of the fragment-aware scheduler ("RF" refers to render farm; "DG" refers to desktop grid).

It is possible that a render host can execute tasks sent from render farm and desktop grid at the same time. In this case, both DG-task and RF-task share the same resources (such as CPU, storage space, and I/O bandwidth) on a same computer that are managed by the multitasking operating system.

However, in the system architecture shown in Fig. 1, a render host runs desktop grid tasks only when the Qube rendering system have not allocate tasks. When receiving a new rendering task, an RFDG-worker directly aborts and preempts the running desktop grid task without coordination to the desktop grid server. This abort operation can be done via the connection-oriented TCP/IP links between the DG-server and the workers.

### 3.2 Establishing Workers

The architecture shown in Fig. 1 provides a loosely coupled integration between the render farm and the computer game desktop grid. There are no direct communication between two systems. All communications are done via ResourceMonitor. This approach takes the advantages that no costly modification is required for both systems.

An important issue is establishing RFDG-workers. Since a render farm can contain dozens or even hundreds of render hosts, installing desktop grid software to all render hosts one by one is apparently intractable. We simply use a batch script to copy and install the software in the render hosts. Automatically deploying the software packages to a large number of render hosts is considered beyond the scope of this paper.

### 3.3 Resource Fragment Monitoring

This subsection discusses a new scheduling algorithm that tries to maximize the effect of using the high-capability RF-workers. In the new Fragment-aware scheduling algorithm, the Result tasks of the same workunit are grouped and assigned to either the render farm or the desktop grid.

In our system, there is a server named ResourceMonitor which continually detects the resource fragments on the Qube and tries to notify the DG server when useful idle resources are detected. ResourceMonitor can be implemented in polling mode or event-driven mode, as shown in Fig. 2.

The most up-to-date version of Qube supports the universal *preflight* and *postflight* events (called *universal flight checks* in Fig. 1) which are triggered immediately before and after a collection of render hosts is allocated or deallocated for a rendering task. The preflight and postflight events are developed intended to monitoring resource changes for the render system. For old versions of Qube system, a polling mode, also inefficient, can be adopted.

The event-driven ResourceMonitor works as follows. Qube system triggers the pre-flight and post-flight events before and after a task execution. Upon receiving a pre-flight message, ResourceMonitor notifies the DG-server that the render farm worker is going to be activated for some rendering task. The DG-server then immediately aborts the current tasks and release the resources.

On the other hand, upon receiving a post-flight message, ResourceMonitor knows that the render farm releases some computing resources. ResourceMonitor has to wait

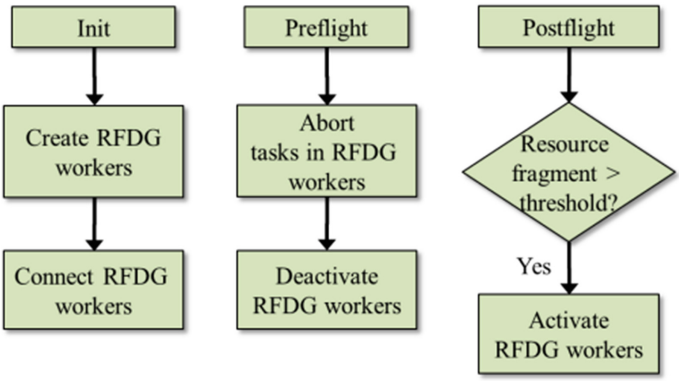


**Fig. 2.** Workflows of task executions in the MJS system.

for a period of time (*e.g.* 60 s) to ensure that there are no upcoming rendering tasks. After detecting idle resources, ResourceMonitor notifies the DG-server, which in turn activates the RFDG-worker via the connection-oriented control link. Note that the DG-server does not push tasks to a worker; instead, it passively waits for the worker to issue the task request.

## 4 Fragment-Aware Scheduling Algorithm

In the parallel game tree search applications, a tree node is implemented as a lightweight, single-code and malleable task. Unlike traditional parallel programs, such malleable task can be interrupted without server coordination. The server will simply discard or redispatch a task when it expires. Owing to this property, the game tree application can generate enough number of tree nodes to adaptive to the unstable idle resources in the render farm.

The DG-server relies on the inherently unstable computer CPU cycles that are stolen from the render farm. In this way, the resource availability must be taken into consideration before operating on these resources. In the DG platform, a task, also called a *Workunit* (WU), is replicated to several instances, called *Results*. A scheduled workunit is complete when its Results are reported and a certain agreement policy, such as majority voting threshold, is reached. For a same Workunit, its Result tasks are called in the same *group*. Based on the statistics principle, the Result tasks in a same group should be assigned to different unrelated workers in order to mitigate the effects of unstable and malicious workers.

In the above Result-based mechanism, the Result tasks in a same group may be assigned to different systems (*i.e.* the render farm and the desktop grid). For example, for a same WU, the replica Result 1 is assigned to an RF-worker, while replica Result 2 is assigned to some DG-worker. However, often a DG-worker disconnects with unsolved Result tasks, halting them until timeout and getting rescheduled. Therefore, although the high capability RF-worker can speeding up the

Result 1, the global performance of the WU it may not be improved since the WU's completion time is determined by the Result instance that finishes last.

The traditional scheduler of DG-server does not distinguish DG-workers and RFDG-workers. Restated, Result tasks, even belonging to a same WU, are simply treated as individual tasks. In this paper, we develop a new *fragment-aware scheduling* algorithm that improves the task response times by trying to maximize the effect of high capability RF-workers.

The fragment-aware scheduling algorithm uses three priority queues to store Work-units:

- Qinit: Initially, all the Result tasks, are placed in this queue.
- QDG: stores the Result tasks that are going to assign to DG-workers.
- QRF: stores the Result tasks that are going to assign to RF-workers.

Each Workunit *w* partitions its redundant instances into three lists:

- w.waitResults: records the Result tasks waiting for scheduling.
- w.pendResults: records the Result tasks that are assigned to some worker but not reported yet.
- w.completeResults: records the completed Result tasks.

The order of workunits in the priority queues is determined by the property $w.\text{key} = (|w.\text{waitResults}| + |w.\text{pendResults}|)/\text{replication\_factor}$,

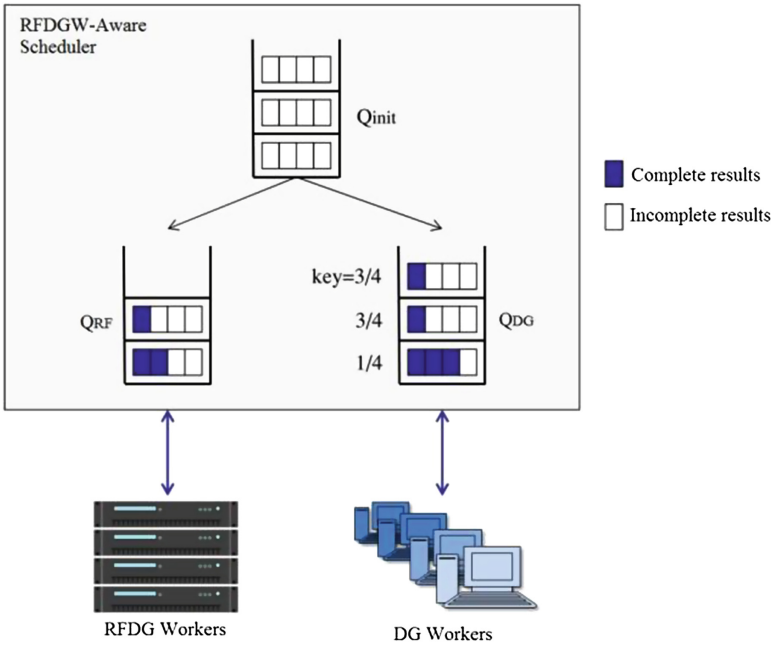


**Fig. 3.** Fragment-aware scheduler.

where the denominator (i.e. replication_factor) is the number of total number of Results of a workunit and the numerator is the number of incomplete tasks. In this paper, we set replication_factor to 4 for all the experiments.

Workunit with less key has higher priority. Thus, performing an extract-min operation on the queues will extract the workunit with the higher ratio of completed tasks. For example, in Fig. 3, the workunit in the front of queue QDG has the key 1/4 and the workunit in the front of queue QRF has the key 2/4.

**Algorithm** 1 Fragment_Aware_Scheduler in DG-server:

**A:** When generating a new workunit *w*:

(1) Generate four copies of replicas (*i.e.* Result tasks) of *w*;

(2) Add all the Result tasks to list w.waitResults;

(3) Add w to Qinit using w.key;

**B:** When receiving a task request from a worker:

(1) If the worker is a DG-worker then

. Q ← QDG;

else //(the worker is an RF-worker)

. Q ← QRF;

End if

(2) If Q is empty then move a Workunit (and the list of its Results) from Qinit to Q ;

(3) w ← extractMin(Q ); //get the workunit with highest priority

(4) If w is not null then

. *r* ← pop(w.waitResults);

. Move *r* from *w*.waitResults to *w*.pendResults;

. Send task *r* to the worker;

End if

➨ **C:** When receiving a report of task *r* :

(1) Find the workunit w containing *r* ;

(2) Move r from w.pendResults to w.completeResults;

(3) Update w.key and the queue order;

(4) Delete w from the queue if w is completed;

In Algorithm 1, after the first allocation (Step B(2)), a workunit and all of its Result tasks are stored in either QDG or QRF. Also, in Step B(1), the DG-server always assign tasks in QDG (or QRF) to a DG-worker (or an RF-worker). This strategy ensures that the Result tasks in the same workunit are assigned to the workers in the same system. We design this strategy to make tasks in a same group finish within the near time period, since the recorded execution time of a workunit group is determined by the member finishes last.

In Step B(2), the server moves tasks from Qinit to QDG or QRF according to the job consumption rate of the DG-workers and the RF-workers. Furthermore, in Step B(3), when selecting workunits from QDG (or QRF), the scheduling algorithm tends to select workunits with the higher progress ratio in order to complete workunits as earlier as possible.

In Step C, when the DG-server receives a completion report of a Result task, it updates the workunit progress status. The workunit is recorded as completed and is deleted from the queue if its Result tasks confirm a majority agreement on the execution results.

## 5 Experiments

In this work, we ran a series of simulated jobs on a Qube render management system in Formosa 3 and a Computer Game Desktop Grid (CGDG) as described above. Hereinafter, we will refer to the desktop grid system as the malleable job system and the render farm as the rigid job system. The experimental results are average of 100 runs that are conducted in two different experimental configurations. The first kind of short jobs use 1 core for 10 min, while the second kind long jobs use 1 core for 60 min. Note that in the render farm the computing resources may be unused because, when the system swaps, there will often be a delay, during which there will be unused resources.

Figure 4 demonstrates the effect of resource stealing on the render farm. In this figure, the normal resource utilization of the render farm is about 46 %. That is, in average, more than half resources are unused. Figure 4 (Left) shows that, when the malleable tasks are added to the render farm system, the system resources can be fully utilized. Since the malleable tasks are lightweight single-core tasks that ca be adapted to the fractured idle resources in the render farm, the system resources can be fully utilized by involving malleable tasks.


**Fig. 4.** (Left) System utilization of the high performance cluster: without malleable tasks (only rigid tasks) and with malleable tasks. (Right) Average number of task completion rate using the normal scheduler and the fragment-aware scheduler.

In Fig. 4 (Right), we conduct the experiment for completion rate of using the fragment-aware scheduler presented in the previous section. As mentioned earlier, the malleable jobs running on the stolen render farm resources can be aborted when rendering tasks arrive. That is, an malleable job may be aborted and the computation will be discarded. In this experiment, the completion rate can be improved by 30 % when using the fragment-aware scheduler.

## 6 Conclusion

In this paper, we propose a malleable job scheduling system for sharing resource between a high-performance render farm and a desktop grid. The proposed system maximizes the resource utilization by allocating unused resource fragments for the lightweight single-core malleable jobs. Our work demonstrates a successful integration of the two systems in which the desktop grid tasks gain significant resources without interfering with the original rendering tasks.

**Acknowledgments.** This study was supported by grant MOST-103-2221-E-029-027 from Ministry of Science and Technology, ROC.

## References

1. Kalé, L.V., Kumar, S., Jayant, D.: A malleable-job system for timeshared parallel machines. In: 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid, 2002, p. 230. IEEE (2002)
2. Anderson, D.P.: BOINC: a system for public-resource computing and storage. In: Proceedings of Fifth IEEE/ACM International Workshop on Grid Computing, 2004, pp. 4–10. IEEE (2004)
3. BOINC website. http://boinc.berkeley.edu/
4. Wu, I.-C., et al.: Job-level proof number search. IEEE Trans. Comput. Intell. AI Games **5**(1), 44–56 (2013)
5. Chen, Y.-W.: The study of the broker in a volunteer computing system for computer games. Master thesis, Department of Computer Science, NCTU (2011)
6. Han, S.-Y.: the study of the worker in a volunteer computing system for computer games. Master thesis, Department of Computer Science, NCTU (2011)
7. Zhou, S.: LSF: load sharing in large heterogeneous distributed systems. In: Workshop on Cluster Computing (1992)
8. Le Boudec, J.-Y.: Rate adaptation, congestion control and fairness: a tutorial. Web page, November 2005
9. Ali, G., et al.: Choosy: max-min fair sharing for datacenter jobs with constraints. In: Proceedings of the 8th ACM European Conference on Computer Systems. ACM (2013)
10. Qube Render Farm: (2015). http://www.pipelinefx.com