Emulating Volunteer Computing Scheduling Policies

David P. Anderson
University of California, Berkeley
davea@ssl.berkeley.edu

Abstract

Volunteer computing systems such as BOINC use several interacting scheduling policies, which must address multiple requirements across a large space of usage scenarios. In developing BOINC, we need to design and optimize these policies without direct access to the target nodes (volunteered PCs). To do this, we developed an emulation-based system that predicts the policies' behavior in specific scenarios. This system has been useful in the design and evaluation of policies, in software development, and in the resolution of problems occurring in the field.

1. Introduction

The consumer digital infrastructure (CDI) consists of mass-market devices (such as desktop, laptop and tablet computers, game consoles, and smart phones) and the communication networks that connect them. The CDI currently includes over 1 billion privately-owned PCs and 100 million GPUs capable of general-purpose computing, with a total computing capability of at least 20 ExaFLOPS, and on the order of 10 Exabytes of free disk space accessible via 1 Petabit/s of network bandwidth.

The CDI can be used for high-performance scientific computing by allowing computer owners to donate computing resources to research projects; this is called **volunteer computing**. BOINC [1] is the leading software platform for volunteer computing. The BOINC server software allows scientists to create **projects**, each of which operates a server that dispatches jobs. Using the BOINC client software, volunteers can **attach** their computers to one or more projects. Currently there are about 50 projects and 500,000 volunteered computers, providing over 5 PetaFLOPS of computing throughput.

Because many clients are behind NATs or firewalls, BOINC is "pull-based": all communication is initiated by the client. The client typically queues multiple jobs, perhaps from different projects. Hence scheduling in BOINC consists of three interacting components: **client job scheduling** (deciding which jobs to execute at a given point), **client job fetch** (deciding when and from where to request new jobs), and **job dispatch** (the server-side selection of jobs to send in response to a client request).

These scheduling policies are critical to BOINC. Poorly-performing or incorrectly implemented policies can reduce system throughput; equally importantly, they can frustrate and demotivate volunteers, possibly causing them to stop volunteering. Developing and evaluating policies, however, is made difficult by the unique properties of volunteer computing:

- The volunteered computers vary widely on many scheduling-related axes: hardware, availability, number and properties of attached projects, and so on. A combination of these factors is called a **scenario**. Scheduling policies should perform well across the entire population of scenarios.
- The volunteer computers are not directly accessible to BOINC software developers. We are not able to deploy new software on these computers, or log in to them.
- The scenario on a particular volunteer computer may be essentially impossible to reproduce on a BOINC development computer.

Historically, BOINC development has relied on a group of about 200 volunteer "alpha testers" who monitor the actions of their BOINC clients, communicate problems via email or message boards, and are willing to run experimental versions of the client. Alpha testers sample both central and extreme areas of the scenario distribution – some have multiple GPU boards, some attach to many projects, and so on. Thus, if alpha testers report no problems from a BOINC client with a given set of policies, we have some confidence that policies will perform well over most of the actual population of scenarios.

This approach, however, has significant limitations. For example, when an alpha tester reports a scheduling-related problem, it can be difficult to obtain information, such as trace message logs, needed to

understand and fix the problem. In addition, the alpha-test approach doesn't help us design scheduling policies for hypothetical situations in which, for example, the GPU/CPU speed disparity is greatly increased, or projects have much tighter latency requirements.

To address these issues, we developed a new way of studying BOINC scheduling policies. The system revolves around a **BOINC client emulator** (BCE) – a program that takes as input a description of a usage scenario, emulates (using the actual BOINC client code) the behavior of the client over some period of time, and calculates various performance metrics. An earlier version of BCE, which did not handle GPUs or multi-thread jobs, is described in [6].

We created a system in which volunteers can run BCE by pasting their BOINC client state files into a web form. Hence, when an alpha tester notices a bug or anomaly, they can, in many cases, reproduce it using BCE, and report it (together with their state files) to BOINC developers, who can then examine the problem under a debugger and fix it easily.

This paper describes the current BOINC scheduling policies, and several variants of each policy. We then describe BCE, and give some examples of its use to evaluate the policy variants.

2. Background

2.1 Attachment and resource share

A volunteer can **attach** each of their computers to any set of BOINC-based projects. Each attachment has a volunteer-specified **resource share** indicating the fraction of the computer's available resources that should be allocated to the project.

Each computer has processors of various **processor types** (currently CPU, NVIDIA GPU, and ATI GPU). There may be multiple instances of each type. Some computers have both NVIDIA and ATI GPUs.

Each project supplies clients with **jobs** that may use various combinations of the host's computing resources. The types of jobs supplied by a project may change over time, and there may be periods when a given project has no jobs available.

Resource share is intended apply to a host's aggregate processing resources, not to the processor types separately. For example, suppose a host has a 10 GFLOPS CPU and a 20 GFLOPS GPU. Suppose that the host is attached to projects A and B with equal resource shares, and that project A has both GPU and CPU applications, while project B has only GPU applications. In this case project A should be allocated 100% of the CPU and 25% of the GPU, while project B should be allocated 75% of the GPU (see Figure 1).

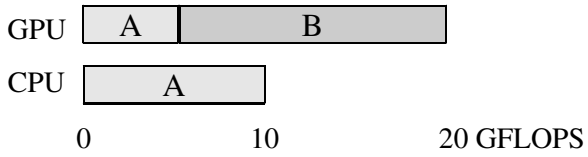

Figure 1: *A project's resource share applies to the host's combined processing resources. In this example, projects A and B, with equal resource shares, each get 15 GFLOPS.*

The notion of resource share is not precisely defined; for example, the time scale over which the ratio should apply is not specified; nor is the question of whether a project's share continue to accrue when it has no jobs available.

2.2 Hosts and preferences

The BOINC client probes and measures each host's hardware characteristics: the number and floating-point speed of its CPUs and GPUs, the size of its main memory and video RAM, and so on.

The behavior of the BOINC client on a given host is governed by a set of user-specified **preferences**. These indicate, for example, whether CPU and/or GPU computing should be done while the computer is in use, time-of-day limits on computing, limits on memory and disk usage, and so on.

On a given computer, BOINC is able to compute only when a) the computer is powered on and BOINC is running, and b) computing is allowed by the preferences. The BOINC client maintains basic availability data: the recent-average fraction of time when computing is allowed, when GPU computing is allowed, and when the computer is connected to the Internet.

2.3 Jobs

The properties of a BOINC job J include:

- The number of CPUs J will use (typically the number of CPU-intensive threads). This may be fractional.
- For each GPU type, the number of instances J will use. This may be fractional, meaning that J will use at most that fraction of the GPU's cores and memory. If J uses a GPU, we call it a **GPU job**; otherwise we call it a **CPU job**.
- The expected run time on the host.
- A **latency bound** that determines a deadline for completion of the job: the local deadline is the time when the job is dispatched to the host plus the latency bound. If the deadline is missed, the server will issue another instance of the job.

Almost all BOINC-based applications do regular checkpointing, so that they can exit and resume with minimal overhead. This study includes only such applications.

3. Scheduling policies in BOINC

Scheduling in BOINC involves interacting client and server policies. The client maintains a queue of jobs, and at any given point runs some of them on the host's hardware (more precisely, it runs them on the host's operating system, which may do its own scheduling). The client periodically issues **scheduler RPCs** to the job dispatchers of attached projects. Each RPC can report completed jobs and request new jobs.

3.1 Resource share accounting

The client policies require a way of deciding whether each project P has used too much or too little resource relative to its resource share SHARE(P). This determines priorities $PRIO_{sched}(P, T)$ and $PRIO_{fetch}(P)$ that are used for job scheduling for processor type T and job fetch respectively. We have used two approaches to this:

Local accounting: for each processor type T and each project P, the client maintains a "debt" D(P, T), which is incremented at a rate proportional to SHARE(P) and decremented as P uses resources of type T. $PRIO_{sched}(P, T)$ is D(P, T); $PRIO_{fetch}(P)$ is the sum of D(P, T), weighted by the peak FLOPS of T.

Global accounting: the client maintains REC(P), an exponentially-weighted recent average of the peak FLOPS (including all processor types) used by each project P. $PRIO_{sched}(P, T)$ and $PRIO_{fetch}(P)$ are both SHARE(P) – REC(P).

3.2 Round-robin simulation

The BOINC client's baseline job scheduling policy is **weighted round-robin** (WRR); "weighted" means that projects are allocated time slices in proportion to their resource share. BOINC's client scheduling policies use **round-robin simulation** to predict the behavior of the system under WRR. The simulation is approximate: instead of modeling individual time-slices, it uses a continuous approximation. The outputs of the simulator are as follows (see figure 2):

- Whether each job will meet its deadline. Jobs that are projected to miss their deadline are called **deadline-endangered**.
- For each processor type T, the length of time SAT(T) that T is saturated (that is, all its instances are busy).
- For each processor type T, the number SHORTFALL(T) of idle instance-seconds within the maximum queue interval (see section 3.4).

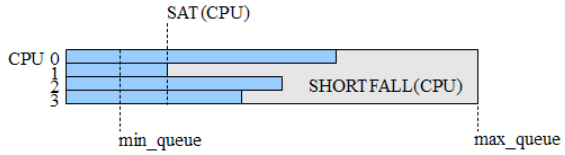

Figure 2: *The round-robin simulator predicts how long each processor instance will be busy given the current workload.*

3.3 Client job scheduling policies

Given a set of runnable jobs, this policy determines which jobs to run. The BOINC client does job scheduling by starting, suspending, and killing jobs.

The default policy is as follows. First, the scheduler does round-robin simulation as described above. The scheduler then builds an **ordered job list**. Jobs are selected in order of $PRIO_{sched}(P, T)$. Deadline-endangered jobs have precedence over others, and GPU jobs have precedence over CPU jobs. Running jobs that have not checkpointed yet have precedence over all others.

The scheduler then scans the ordered job list, running jobs and preempting others. Jobs are skipped if total memory usage would exceed the limit, or if GPUs

cannot be allocated. The scan stops when CPUs and GPUs are fully utilized.

In section 4, we compare the following variant policies:

JS-LOCAL: the baseline policy with local accounting.

JS-GLOBAL: the baseline policy with global accounting.

JS-WRR: a variant of JS-LOCAL with WRR only (deadlines are not used).

3.4 Client job fetch policy

This policy determines when to issue a scheduler RPC requesting jobs, which project to contact, and how much work to request. For each processor type T, a scheduler RPC request message includes:

instances(T): the client requests sufficient jobs to use this many instances of T, typically because that number of instances are currently idle

secs(T): the client requests sufficient jobs to use this many instance-seconds.

The job fetch policy has two queue-size parameters:

min_queue: the client attempts to maintain enough jobs to keep all processors busy for this period of time. Typically this reflects the duration of periods when the client is unable to fetch new jobs, for example because it is not connected to the Internet.

max_queue: if the client has enough jobs to keep a processor type T busy for this many seconds, it shouldn't get more jobs for T.

Currently these parameters are user preferences; in principle they could be derived from availability traces.

We compared the following job-fetch policies:

JF_ORIG: if, for a given processor type T, SHORTFALL(T) > 0, then let P be the project with jobs of type T for which $PRIO_{fetch}(P)$ is greatest. Request X*SHORTFALL(T) instance-seconds, where X is the fractional resource share of P among projects with jobs of type T.

JF_HYSTERESIS: if, for a processor type T, SAT(T) < min_secs, then let P be the project with jobs of type T for which $PRIO_{fetch}(P)$ is greatest. Request SHORTFALL(T) instance-seconds.

The main distinctions are that 1) JF_HYSTERESIS uses hysteresis, and 2) it asks a single project for the entire shortfall rather than dividing it among projects.

4. Evaluating scheduling policies

4.1 Scenarios

The BOINC client runs on about 500,000 computers . Each computer constitutes a "scenario" in which the scheduling policies operate. These scenario have many properties:

- The host's hardware characteristics (number and speed of processors, memory and disk sizes).
- The host availability: Hosts have widely differing availability patterns: some are available all the time, others are available periodically or randomly.
- The number of attached projects (possibly hundreds) and their resource shares.
- For each attached project, the lengths of its jobs (which may vary from minutes to months) and their slack time.
- For each attached project, the computer resources used by its jobs. This may include various combinations of CPUs and GPUs (multiple and/or fractional), and may change over time.
- The availability of projects: some projects are sporadically down for maintenance, or have no jobs.
- Errors (random or systematic) in *a priori* job runtime estimates.

Ideally, we would like BOINC's scheduling policies to perform well for all possible combinations of these properties. In practice, we want the policies to perform well for as much of population of actual scenarios as possible.

4.2 Figures of merit

We evaluate scheduling policies according to several figures of merit:

Idle fraction: the fraction of processing capacity (as measured by peak FLOPS of all processor types) that was idle during the emulation period.

Wasted fraction: the fraction of processing capacity (as measured by peak FLOPS) that was used for jobs that did not complete by their deadline.

Resource share violation: the RMS over projects P of the difference between P's share of processing resources and the amount it actually received.

Monotony: a measure of the extent to which the system ran jobs of a single project for long periods (such behavior is undesirable for many volunteers).

RPCs per job: the average number of RPCs per job. The lower this is, the less load is placed on project servers.

Each quantity is scaled to lie in [0, 1], where 0 is good and 1 is bad.

The performance metrics conflict; in general we cannot minimize them simultaneously. The overall evaluation of a policy is a subjectively-weighted combination of the metrics.

4.3 The BOINC client emulator

The **BOINC client emulator** (BCE) takes as input a scenario description and a set of flags selecting the job scheduling, job fetch, and server deadline-check policies. It simulates the behavior of the client in that scenario for a given time period, and reports the resulting values for the figures of merit listed above. It also generates a "time-line" visualization of processor usage, and a message log detailing the scheduling decisions.

BCE uses a mix of emulation and simulation. The implementation of job scheduling, job fetch, and preference enforcement uses the same source code as the BOINC client. In terms of scheduling, BCE reproduces the exact behavior of the client; hence it "emulates" the client.

Other components of the system are simulated: a) job execution is simulated, and run times are normally distributed; b) host availability is modeled as a random process in which available and unavailable periods have exponentially distributed lengths; c) BOINC schedulers are simulated with a simplified model.

We developed a controller script that does multiple BCE runs and generates graphs summarizing the figures of merit. For example, it can compare scheduling policies across one or more scenarios, or do a parameter sweep over a scenario parameter.

In addition, we developed a web interface to BCE (http://boinc.berkeley.edu/sim_form.php), with the goal of allowing alpha testers to submit scenarios to BOINC developers. When alpha testers see a scheduling-related problem, they can upload their BOINC client state files through this interface, and verify that they see the same problem in BCE. The input files are saved on the server. They then report the problem to BOINC developers, who can then investigate the problem in a controlled, reproducible environment, without further involvement of the tester.

5. Emulation results

We used the BCE to compare the scheduling policies described in section 3. We used the following scenarios:

Scenario 1: CPU only, two projects.
Scenario 2: 4 CPUs and 1 GPU. GPU is 10X faster than one CPU. Two projects, one with CPU jobs, one with both.
Scenario 3: CPU only. Two projects, one with very long low-slack jobs.
Scenario 4: CPU and GPU. Twenty projects with varying job types.

Unless otherwise specified, the simulation period is 10 days.

5.1 Job scheduling: use of EDF

To study the effect of using EDF scheduling for deadline-endangered jobs, we compared JS-WRR to JS-GLOBAL. In scenario 1, we varied the slack time of project 1. The job runtime is 1000s, and we varied the latency bound from 1000s to 2000s. The results are shown in figure 3.

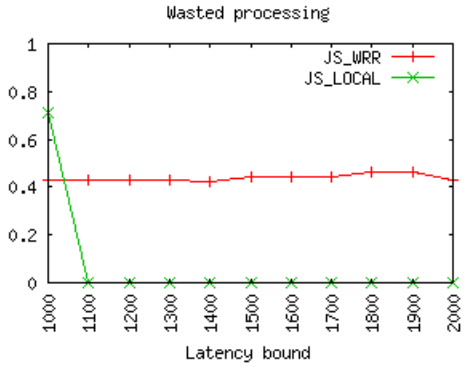

Figure 3: *A job-scheduling policy that incorporates deadlines wastes less processing time.*

With zero slack, neither policy can meet the deadlines for project 1, so half the processing is wasted. For larger slack times, JS-LOCAL has significantly lower wasted time because it gives priority to deadline-endangered jobs.

5.2 Job scheduling: local/global policies

To study the effect of local and global resource-share accounting, we compared JS-LOCAL and JS-GLOBAL in scenario 2. The results are shown in figure 4.

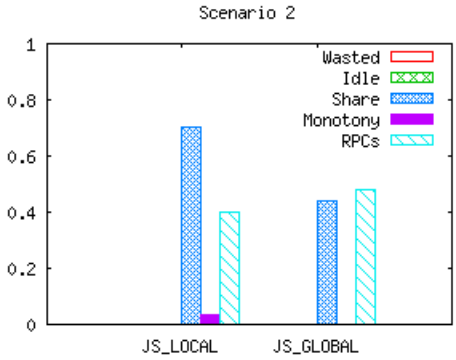

Figure 4: *A resource-share accounting policy that spans processor types reduces resource share violation*

JS_LOCAL divides CPU time evenly between the 2 projects, while JS_GLOBAL devotes all the CPU time to project 1; the latter policy respects resource share as much as possible while still maximizing throughput.

5.3 Job fetch: hysteresis

To study the value of hysteresis, we compared JF-ORIG and JF-HYSTERESIS in scenario 4. The results are shown in figure 5.

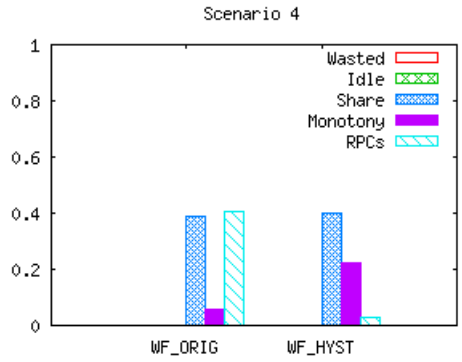

Figure 5: *A job-fetch policy with hysteresis reduces the number of scheduler RPCs.*

Because a typical scheduler RPC fetches multiple jobs, WF_HYST reduces the number of scheduler RPCs in this case. Monotony increases because each RPC fetches multiple jobs, and as a result the client may have jobs from only one project for some periods.

5.4 Long low-slack jobs

We studied a particular scenario (scenario 3) in which project 1 has long (million-second) low-slack jobs. These jobs are immediately deadline-endangered, forcing the client to run them to the exclusion of other jobs. Project 2 has normal jobs.

We studied JS-REC in this scenario, and varied the half-life parameter A used in averaging REC. The results are shown in figure 6.

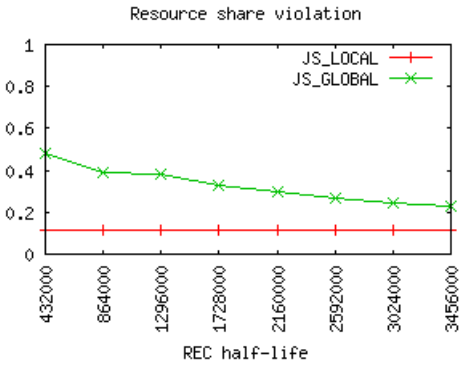

Figure 6: *In a scenario with long low-slack jobs, credit estimate half-life affects resource share violation.*

When A is small, the system has a short memory: after a project 1 job completes, the system quickly forgets that project 1 has exceeded its resource share, and as a result share violation is high. Increasing A to several times the long job size reduces this effect.

6. Discussion and conclusion

6.1 Related work

This work is an extension of the work of Kondo *et al.* [6], in which an emulator was used to study BOINC client scheduling policies. The earlier system did not model GPU or multithread jobs; the policies studied in the current work were developed to handle such jobs.

Mutka [7] did a simulation-based study of policies for scheduling jobs with deadlines on a Condor-based desktop grid; Rood and Lewis [8] did a similar study that sought to minimize average makespan. This context differs from ours because, for example, BOINC jobs do not have an overall deadline, BOINC has no provision for migrating jobs between hosts, and BOINC handles coprocessors.

Estrada *et al.* [4] used the emulation approach to study server-side scheduling policies in BOINC. Their system, EmBOINC, used a simulator (driven by either traces or by an analytic model) of a dynamic population of volunteer hosts, and used emulation of the BOINC server. It complements the current work.

6.2 Future work

This work can be extended in a number of directions:

- Characterize the actual population of scenarios, and develop a system, perhaps based on Monte-Carlo sampling, to study policies over the entire population.
- Study other policy alternatives. Over the last few years, scores of policy changes have been proposed, primarily on the BOINC alpha-testing and development email lists. Many of these merit study.
- Study multiprocessor scheduling policies other than earliest-deadline-first (EDF). EDF is optimal for uniprocessors but not multiprocessors. Although optimal scheduling for multiprocessors is NP-complete, there are heuristics that perform better than EDF in many cases [3].
- Increase system throughput by enforcing resource share across a volunteer's hosts, rather than for each host separately. For example, if a particular host is well-suited to a particular project, it could run only that project, and the difference could be made up on other hosts. Bertin *et al.* [2] have proposed a system that, in effect, optimally enforces resource share across multiple volunteers.
- Take application memory usage into account. The presence of memory-intensive applications may have a significant impact; for example, it may be possible to use only a single processor instance at times.
- Model file transfers. Jobs are assumed to be runnable immediately after dispatch. For data-intensive applications (those with large input or output files) this is not a realistic assumption. It would be important to model an additional scheduling policy: the order in which files are uploaded and downloaded.
- Simulate scheduler behavior more realistically, for example by emulating it as has been done in EmBOINC [4].
- Model project unavailability and the sporadic availability of particular types of jobs (for example, GPU jobs).
- Model applications that checkpoint infrequently or never.
- Model inaccurate job runtime estimates.

6.2 Conclusion

We created an emulation-based system for predicting the behavior and performance of the BOINC client in a wide range of usage scenarios. This system has been extremely valuable, both for studying scheduling policies and for debugging the system. In particular, we concluded that in some scenarios:

- EDF scheduling reduces wasted processing.

- Global resource-share accounting reduces share violation.
- Job-fetch hysteresis reduces the number of scheduler RPCs per job.
- In scenarios with long jobs, a longer averaging half-life reduces resource share violation.

We believe that the emulation approach would be useful in the development of any distributed system where usage scenarios cannot easily be duplicated on development machines. The initial effort in factoring the software to allow emulation, and in developing the emulator, is offset by more efficient development and debugging, and by a resulting system that performs better and is more robust.

References

[1] David P. Anderson. BOINC: A System for Public-Resource Computing and Storage. 5th IEEE/ACM International Workshop on Grid Computing, November 8, 2004, Pittsburgh, USA, pp. 1-7.

[2] Remi Bertin, Arnaud Legrand, Corinne Touati. Toward a Fully Decentralized Algorithm for Multiple Bag-of-tasks Application Scheduling on Grids. IEEE/ACM International Conference on Grid Computing (Grid), Tsukuba, Japan, 2008.

[3] Dell'Amico, Silvano Martello. Optimal Scheduling of Tasks on Identical Parallel Processors. ORSA Journal on Computing, Vol. 7, No. 2. (1 January 1995), pp. 191-200.

[4] Trilce Estrada, Michela Taufer, David Anderson. Performance Prediction and Analysis of BOINC Projects: An Empirical Study with EmBOINC. Journal of Grid Computing 7(4) Dec. 2009, p. 537-554.

[5] Bahman Javadi, Derrick Kondo, Jean-Marc Vincent, and David P. Anderson. Discovering Statistical Models of Availability in Large Distributed Systems: An Empirical Study of SETI@home. To appear, Transactions on Parallel and Distributed Systems.

[6] D. Kondo, D.P. Anderson, and J. McLeod VII. Performance Evaluation of Scheduling Policies for Volunteer Computing. 3rd IEEE International Conference on e-Science and Grid Computing, Bangalore, India, 10-13 December 2007.

[7] M. W. Mutka, Considering Deadline Constraints When Allocating the Shared Capacity of Private Workstations, International Journal of Computer Simulation, vol. 4, no. 1, pp. 41-63 1994.

[8] Brent Rood and Michael J. Lewis. Scheduling on the Grid via Multi-State Resource Availability Prediction, 9th Grid Computing Conference, 2008.