Local Scheduling for Volunteer Computing

David P. Anderson[1], John McLeod VII[2]

[1] U.C. Berkeley Space Sciences Laboratory
Berkeley, CA 94720
davea@ssl.berkeley.edu

[2] 652 Crescent Ridge Trail
Mableton, GA 30126
jm7@acm.org

Abstract

BOINC, a middleware system for volunteer computing, involves ***projects,*** *which distribute jobs, and* ***hosts,*** *which execute jobs. The local (host-level) scheduler addresses two issues: when to fetch new jobs from a project and, of the currently runnable jobs, which to execute. It seeks to simultaneously satisfy a number of constraints – such as maintaining given long-term ratios of work between projects, meeting deadlines for job reporting, and providing variety to the volunteer – using uncertain, dynamic information about resources and jobs. We describe these goals and factors, and discuss BOINC's local scheduling policies.*

1. Introduction

Volunteer computing is a form of distributed computing in which the general public volunteers processing and storage resources to computing projects. Early volunteer computing projects include the Great Internet Mersenne Prime Search [10] and Distributed.net [9]. Unlike other types of distributed computing, volunteer computing uses anonymous, untrusted resources.

BOINC (Berkeley Open Infrastructure for Network Computing) is a middleware system for volunteer computing [3]. It is being used for applications in physics, molecular biology, medicine, chemistry, astronomy, climate dynamics, mathematics, and the study of games. There are currently about 40 BOINC-based projects and about 400,000 volunteer computers performing an average of over 400 TeraFLOPS.

1-4244-0910-1/07/$20.00 ©2007 IEEE.

1.1) Projects and volunteers

BOINC projects are independent; each has its own database of jobs and volunteer accounts. Job durations vary widely between projects, ranging from a few minutes to several months (for example, Climateprediction.net [8]).

Volunteers participate by running BOINC client software on their computers (**hosts**). BOINC is open source and the client is available for most platforms, including Windows, Linux, and Mac OS X. Volunteers can **attach** each host to any set of projects, and can specify various **preferences** that constrain when and how their resources are used. For example, they can control the allocation of resources among projects.

There are advantages in attaching hosts to multiple projects. First, a given project may have periods when it has no work, so a host attached to several projects is less likely to become idle. Second, such a host may be able to do use its resource more fully, for example by downloading files for one project while computing for another.

1.2) The BOINC architecture

BOINC consists of client and server components (see Figure 1). The BOINC client runs projects' **applications**. The applications are linked with a runtime system whose functions include process control, checkpoint control, and graphics [5]. The client performs CPU scheduling (implemented on top of the local operating system's scheduler; at the OS level, BOINC runs applications at zero priority). It may preempt applications either by suspending them (and leaving them in memory) or by instructing them to quit.

All network communication in BOINC is initiated by the client. A client communicates with a project's task server [4] via HTTP. The request is an XML document that includes a description of the host

hardware and availability, a list of completed jobs, and a request for a certain amount (expressed in terms of CPU time) of additional work. The reply message includes a list of new jobs (each described by an XML element that lists the application, input and output files, including a set of **data servers** from which each file can be downloaded).

Some hosts have intermittent physical network connections (for example, portable computers or those with modem connections). Such computers may connect only every few days. During a period of network connection, BOINC attempts to download enough work to keep the computer busy until the next connection.

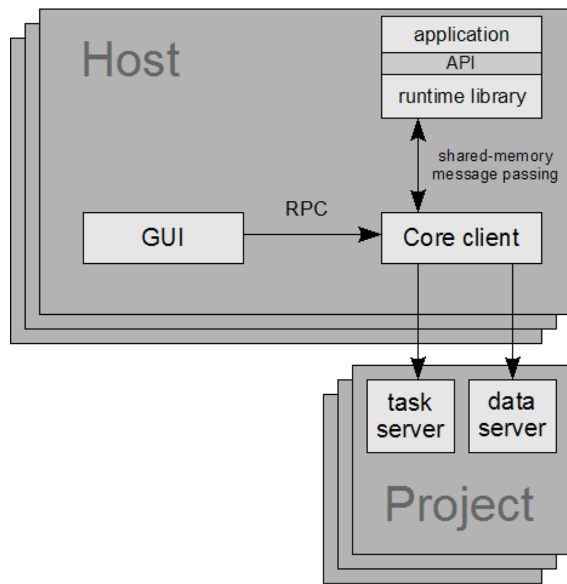

Figure 1: The BOINC architecture

1.3) Local scheduling issues

The BOINC client implements two related scheduling policies:

CPU scheduling: of the currently runnable jobs, which to run? Of the preempted jobs, which to keep in memory?

Work fetch: when to ask a project for more work, which project to ask, and how much work to ask for?

These policies have a large impact on the performance of BOINC-based projects. They must deal with changing and uncertain data; for example:

- Project-supplied job completion estimates may be off by orders of magnitude.
- Users may change preferences at any time. If resource shares are changed, a feasible workload can become infeasible.
- Projects may be off-line or have no work available for long periods (months or years).

The scheduler must do its best (relative to several often incompatible criteria) in response to changing situations. This paper describes the issues behind these problems in detail, discusses the policies currently used in BOINC, and proposes strategies for improving the policies.

2. The local scheduling problem

2.1) Inputs to local scheduling

To frame the local scheduling problem in more detail, we will describe its various inputs.

First, there are the host's hardware characteristics: number of processors, floating-point and integer benchmarks, RAM size, and so on. These are measured periodically by the BOINC client. The client also tracks various **usage characteristics**, such as the **active fraction** (the fraction of time BOINC is running and is allowed to do computation), and the statistics of its network connectivity.

Second, there are user preferences. These include:

- A **resource share** for each project. The fraction of a bottleneck resource R devoted to a project P should be roughly equal to P's resource share divided by the sum of resource shares of projects contending for R. Possible bottleneck resources include disk space, network bandwidth, and CPU time; only the latter is addressed here.
- Limits on processor usage: whether to compute while the computer is in use, the maximum number of CPUs to use, and the maximum fraction of CPU time to use (to allow users to reduce CPU heat).
- The maximum fraction of RAM to use while the computer is busy, and the maximum fraction to use while it's idle.
- *ConnectionInterval*: the typical time between periods of network activity. This lets users provide a “hint” about how often they connect, and it lets modem users tell BOINC how often they want it to automatically connect.
- *SchedulingInterval*: the “time slice” of the BOINC client CPU scheduler (the default is one hour).

Third, there are various settings and controls accessed via a graphical interface. For example, users can suspend and resume BOINC computation in its entirety, suspend and resume individual projects or jobs, attach and detach projects, and abort jobs.

Finally, each job has a number of project-specified parameters, including estimates of its number of floating-point and integer operations, an estimate of its maximum working-set size, a **deadline** by which it should be reported.

2.2) Job execution

Checkpointing is especially important in volunteer computing because the BOINC client, or the host, can be turned off at any time. BOINC's runtime system [5] lets the client tell applications when to checkpoint (based on user preferences for disk-access interval) and informs the client when a checkpoint has been done. Ideally, applications checkpoint promptly when BOINC asks them to, but some applications may have long periods when they can't checkpoint. Some applications never checkpoint.

BOINC allows applications to report their **fraction done** periodically during execution. This information is used to estimate completion time (see section 3.2) and is displayed to the user in the GUI and screen-saver graphics.

2.3) Redundancy and credit

Projects grant **credit** to users for the computational work they have contributed. Credit provides a basis for ranking users and teams, and is an extremely important incentive.

Volunteer computing participants are anonymous and untrusted, and the source code of the BOINC client is publicly available. Malicious participants can easily alter the client to report erroneous results, or to claim exaggerated credit for result.

BOINC supports **redundant computing** to increase the likelihood that only correct results are accepted and that credit is granted fairly. This works as follows. Two or more instances of each job are created and dispatched to clients. A completed result specifies an amount of **claimed credit** (typically based on CPU time and CPU benchmarks). If both results are returned before their deadline, and the results agree (according to project-specified criteria [18]) then the result is considered correct, and both users receive a **granted credit** equal to the minimum of the claimed credits. This reduces the payoff for claiming more credit than is deserved.

If the results don't agree, or if one of the results is not reported by its deadline, the server generates an additional instance of the job, and sends it to a third host. This is repeated until a quorum of matching results is found or a limit on the number of instances is reached. Then, at some later point, the job's input and output files are deleted from the server, and eventually its database record is deleted.

A job instance that is computed correctly and reported by its deadline is almost certain to be granted credit. After the deadline passes, however, the chances of receiving credit decline, and beyond some point the computation is of no value to the participant (because it will not be granted credit) or to the project (because it will not contribute to a quorum).

2.4) Goals and scenarios

The scheduling policies have the following goals, in order of decreasing importance:

- Maximize the average rate at which the host is granted credit. To this end, the policies try to maximize CPU utilization and to avoid missed deadlines (and to balance these goals when they conflict).
- Enforce resource shares over the long term (this must be defined carefully: a project's share should not include periods when it is down or has no work).
- Maximize variety: if a host is attached to several projects with similar resource shares, the scheduler should avoid long periods (days or weeks) when the host works for a single project.

The performance of a scheduling policy is relative to a set of inputs: the hardware characteristics of a particular host, a set of projects and their jobs, user preferences, and so on. We call this a **scheduling scenario**. Scheduling scenarios can be arbitrarily complex: inputs can vary over time, projects may go up and down, job estimates may be wrong, and so on.

In this paper, we will restrict ourselves to very simple scenarios, of the form shown in Table 1.

ConnectionInterval	12 h		
MaxCPUs	1		
Projects	**P1**	**P2**	**P3**
Resource share	0.4	0.3	0.3
Job length	100 h	6 h	3 h
Job deadline	200 h	12 h	5 h

Table 1: An example scheduling scenario

Such scenarios assume that job length estimates are exact, each project's jobs are identical, the host is always running, and projects are always up.

2.7) Early scheduling policies

Early versions of BOINC used the following policies:

- CPU scheduling: round-robin between projects, weighted according to their resource share (we call this policy **weighted round robin**).
- Work fetch: keep enough work queued to last for *ConnectionInterval*, and divide this queue between projects based on resource share, with each project always having at least one job running or queued.

These policies work well in some scenarios. However, there are scenarios in which they fail disastrously. For example, in the scenario of Table 1, the jobs of each project will complete in 250, 20, and 10 hours respectively; every job is completed after its deadline, so (although the CPU is never idle) all the computing is wasted.

We also considered the use of earliest-deadline-first (EDF) scheduling [12]. This does only slightly better; in the scenario of Table 1, some of P3's jobs will be completed by their deadline, but all jobs of P1 and P2 will miss their deadlines.

3. Local scheduling policies in BOINC

We now describe the local scheduling policies used in the current version of the BOINC client. These policies avoid the problems described in the previous section, and have proven to work well in a wide range of real-world scheduling scenarios.

3.1) Terminology

A job J is **nearly runnable** if neither J nor its project is suspended, and J hasn't finished computing. A job J is **runnable** if, in addition, its input files have been downloaded. A project P is **runnable** if P has at least one runnable job; similarly for nearly runnable.

A project is **contactable** if the client is allowed to ask it for work (projects may be non-contactable for various reasons; for example, the client uses exponential back-off in response to RPC failures). A project is **potentially runnable** if it is contactable or nearly runnable. A project's **potentially runnable resource share** is its resource share relative to the set of potentially runnable projects.

The **wall CPU time** of a process is the amount of wall-clock time it has been running at the OS level. The CPU time may be significantly less, for example if the process does I/O or paging, or if CPU-intensive non-BOINC processes run at the same time.

MaxCPUs is the minimum of the user-specified CPU limit and the actual number of CPUs.

3.2) Job completion time estimation

Client scheduling requires estimates of the completion time of jobs, both unstarted and in progress. The more accurate these estimates are, the better the possible scheduling decisions.

We can estimate the remaining CPU time of an unstarted job as the estimated number of floating-point operations (supplied by the project) divided by the floating-point benchmark of a CPU (measured by the BOINC client). However, this can give consistently wrong estimates because either a) the application is memory-intensive, while BOINC's benchmarks are not; or b) the application may have been compiled with more or less optimization than the BOINC client.

BOINC corrects for these factors with a per-project **duration correction factor**, an estimate of the ratio of actual CPU time to originally estimated CPU time. As will be seen, it is better to err on the high side, so this is calculated in an asymmetric way; increases are reflected immediately, but decreases are exponentially smoothed.

Secondly, a job's completion time depends on its wall CPU time, which may be much greater than its CPU time (for example, if the application does lots of I/O or paging, or if the host typically has other CPU-intensive processes running). BOINC corrects for this with a **CPU efficiency factor**, computed as a smoothed average of CPU time to wall CPU time.

Finally, many BOINC applications report their fraction done periodically during execution, and (if the application developer has implemented it carefully) this provides a reliable estimate of completion time. The reliability of an estimate based on fraction done presumably increases as the job progresses. Hence, for jobs in progress BOINC uses the estimate

$$FA + (1-F)B$$

where F is the fraction done, A is the estimate based on elapsed CPU time and fraction done, and B is the estimate based on benchmarks and floating-point count.

3.3) Debt

BOINC's local scheduling policies are based on the notion of **debt**. The debt to a project is the amount of wall CPU time owed to it, relative to other projects. BOINC uses two types of debt; each is varied over a set S of projects. In each case, the debt is recalculated periodically as follows:

WallCPUTime(P) = wall CPU time used by project P during most recent period
A = sum of *WallCPUTime(P)* for P in S
R = sum of *ResourceShare(P)* for P in S

For each project P in S:

*W = A*ResourceShare(P) / R*
Debt(P) += W – WallCPUTime(P)

Debt(P) is then normalized so that its mean is zero.

Short-term debt is used by the CPU scheduler. It is varied over the set of runnable projects, and is bounded so that the maximum short-term debt is no greater than 86,400 (i.e. one day).

Long-term debt is used by the work-fetch policy. It is varied over the set of potentially runnable projects.

3.4) Simulating weighted round-robin scheduling

The CPU scheduling and work fetch policies use a simulation of weighted round-robin scheduling applied to the set of nearly runnable jobs. The simulation takes into account parameters such as active fraction and CPU efficiency. It is a continuous approximation, and doesn't reflect the discrete nature of scheduling or of host inactivity. It produces the following outputs for each job J and project P:

- *DeadlineMissed(J)*: whether J misses its deadline.
- *DeadlinesMissed(P)*: the number of jobs J of P for which *DeadlineMissed(J)*.
- *TotalShortfall*: the amount of additional work (measured in CPU time) needed to keep all CPUs busy for the next *ConnectionInterval* seconds.
- *Shortfall(P)*: the additional work (measured in CPU time) for project P needed to keep it from running short of work in the next *ConnectionInterval* seconds.

In the example shown in Figure 2, projects A and B have resource shares 2 and 1 respectively. A has jobs A1 and A2, and B has job B1. The computer has two CPUs. From time 0 to 4 all three jobs run with equal weighting. At time 4 job A2 finishes. From time 4 to 8, project A gets only a 0.5 share because it has only one job. At time 8, job A1 finishes.

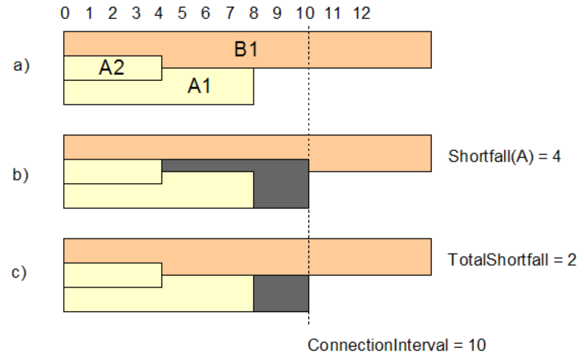

Figure 2: Round-robin simulation example

Shortfall(A) (the area of the dark region in Figure 2b) is 4, *Shortfall(B)* is 0, and *TotalShortfall* (the area of the dark region in Figure 2c) is 2.

3.5) CPU scheduling policy

BOINC's CPU scheduler consists of two parts: a **job selection policy** that computes a list of jobs that should ideally run (the **run list**), and an **enforcement policy** that attempts to run these jobs, while possibly postponing the preemption of jobs that haven't checkpointed recently (to avoid wasted CPU time).

The job selection policy uses earliest-deadline-first (EDF) for projects with jobs that are in danger of missing their deadline, and weighted round-robin among other projects if additional CPUs exist. This allows the client to meet deadlines that would otherwise be missed, while providing variety and honoring resource shares over the long term. The policy is:

1. For each project P, set *AnticipatedDebt(P)* to *ShortTermDebt(P)*.
2. Let P be the project with the earliest-deadline runnable job among projects with *DeadlinesMissed(P)* > 0. Let J be P's earliest-deadline runnable job not on the run list.
3. If such a J exists, add J to the run list, decrement *AnticipatedDebt(P)* by the expected payoff (*SchedulingInterval/MaxCPUs*), and decrement *DeadlinesMissed(P)*.
4. If there are more CPUs, and projects with *DeadlinesMissed(P)* > 0, go to 1.
5. If the size of the run list is *MaxCPUs*, stop.
6. If a job J is currently running, and has been running for less than *SchedulingInterval*, add J to the run list and go to 5 (this ensure that jobs receive their full time slice no matter how often the job selection function runs).

7. Find the project P for which *AnticipatedDebt(P)* is greatest, select one of P's runnable jobs (picking one that is already running, if possible, else the one received first from the project) and add that job to the run list.
8. Decrement *AnticipatedDebt(P)* by the expected payoff and go to 5.

The job selection function runs when a job is completed, when the end of the user-specified scheduling period is reached, when new jobs become runnable, or when the user performs a UI interaction (e.g. suspending or resuming a project or job).

The enforcement policy is implemented by a function that is called by the job selection function at its conclusion, and whenever a job checkpoints. Let X be the set of jobs in the run list that are not currently running, and let Y be the set of running jobs that are not in the run list. The enforcement policy is as follows:

1. If *DeadlineMissed(J)* for some J in X, then preempt a job in Y, and run J (preempt the job with the least wall CPU time since checkpoint). Repeat as needed.
2. If there is a job J in Y that has checkpointed since the last call to this function, preempt J and run a job in X.

3.6) Work-fetch policy

Returning to the scenario shown in Table 1, we see that if the host always has at least one job for each project, it will miss deadlines, regardless of the CPU scheduling policy. We now describe a work-fetch policy that fixes this problem.

A project P is **overworked** if

$$LongTermDebt(P) < -SchedulingInterval$$

This condition occurs if P's jobs consistently have tight deadlines, and are therefore run in EDF mode by the CPU scheduler.

The work fetch policy function runs periodically, and whenever an event occurs that could create a need for more jobs. The function either a) decides to not fetch work, or b) selects a project P to ask for work, and decides how many seconds of work to request. The policy is as follows:

1) Find the project P for which

$$LongTermDebt(P) - Shortfall(P)$$

is greatest. Consider projects that are overworked or that have projected deadline misses only if there is an idle CPU.

2) If there is such a project P, set its work request to

$$max(Shortfall(P), TotalShortfall/PRRS)$$

where PRRS is P's potentially runnable resource share.

In the scenario of Table 1, the system initially fetches jobs for all three projects. After a brief period of round-robin CPU scheduling, the system switches to EDF mode and run only projects 2 and 3. As this continues, the long-term debts of projects 2 and 3 decrease until they become overworked, at which point no work is fetched from them. The system reaches a steady state in which project 1 uses one CPU most of the time, and either project 2 or 3 has no work queued.

3.7) Memory-aware scheduling

Although they run at the lowest OS priority, BOINC applications can impact user-visible performance because of their memory usage. We modified the CPU scheduling policy to reflect this. Limiting memory usage reduces the CPU time available to BOINC, and on some systems BOINC would do no work at all. One goal of our design is to provide user-adjustable controls over this trade-off. A second goal is to maximize the CPU efficiency of BOINC applications; that is, to ensure that they don't thrash. On a multiprocessor, it may sometimes be more efficient (in terms of total CPU time per wall time) to run fewer jobs than the number of CPUs. A third goal is to support applications that can trade off memory usage for speed. Such applications should be made aware of the current memory constraints, so that they can adapt accordingly.

BOINC periodically measures the working set size of all running BOINC applications. To accommodate spikes in memory usage, BOINC smooths the working set size.

The job selection function computes the available RAM, based on preferences. In building the run list, it keeps track of RAM used so far, and skips any task that would cause this to exceed available RAM.

The enforcement policy computes the available RAM, based on preferences. While running tasks, it keeps track of RAM used so far and skips any task that would cause the limit to be exceeded. It preempts tasks that haven't checkpointed if they would cause the limit to be exceeded.

In addition, a function runs every 30 seconds or so and computes the working sets of all running jobs. If the total is too large, it triggers CPU scheduler enforcement (see above). If a job's working set is too large for it to ever run, it is aborted.

These policies may cause some jobs to not run for long periods. For example, suppose that

- A 2-CPU machine has 1 GB RAM,
- There's a small-RAM job X with a close deadline
- There's a 1 GB job Y
- There are several small-RAM jobs.

In this case, Y won't run until X has finished, even if it is more deserving (in terms of debt) than the other small jobs. However, Y won't starve indefinitely. Eventually it will run into deadline trouble, at which point the BOINC CPU scheduler will run it ahead of the other jobs.

4. Related work

A number of platforms for volunteer computing have been developed [2, 6, 7, 13, 14, 15]. These platforms all differ from BOINC in crucial respects (they are restricted to a single project, or have no job deadlines, or allow only one queued job at a a time) and so do not address the scheduling issues described here.

BOINC's weighted round-robin scheduling is analogous to fair-shared scheduling in time-sharing systems [11]. Earliest-deadline-first scheduling and its optimality properties were described by Liu and Layland [12]. There is a substantial amount of work in completion-time estimation and its use for scheduling [17].

Memory-aware scheduling was introduced and explored by Agrawala and Bryant [1]. In our work, however, memory requirements are not known in advance.

5. Conclusions and future work

We have described the issues involved in local scheduling for volunteer computing, and have presented policies that have proven to work well in the real world. Key aspects of these policies are: 1) the notion of debt, in its two forms; 2) the use of deadline scheduling (but only when necessary); and 3) careful attention to job completion estimation.

Currently, client and server scheduling are not well integrated. The client asks for N seconds of work, and the server sends it jobs that can run in available memory, and that, if started immediately, would finish by their deadline. However, since the server has no information about work queued or in progress on the client, it can send jobs that will cause deadlines to be missed. To remedy this, we plan to have the client send information about queued and in-progress work, including completion time estimates. The server will use this information to do a deadline-scheduling based simulation to decide what jobs, if any, can safely be sent.

Network connection interval is currently a user preference. Many users are unaware of this preference or don't set it correctly. We plan to eliminate it by having the BOINC client record statistics about periods when the host is powered off or not connected, and base scheduling decisions on these statistics.

The local scheduling policies currently reflect memory constraints only after the fact. For example, if a host has 2 CPUs but only enough RAM to run one job at a time, BOINC will fetch work on the assumption that both CPUs will be used, and deadlines will be missed. This can be fixed – for example, by modifying the round-robin simulator to reflect memory-aware scheduling.

We plan to develop a simulation-based framework in which we can evaluate, compare and study scheduling policies. Currently we rely on “thought experiments” and empirical evidence – we make a change to the scheduler, run it in-house and with alpha testers, then release it to the 400,000+ BOINC participants. It is difficult to know if a change has had the intended effect, and if a change causes a major problem, it can waste lots of computing power. A simulation-based testbed would avoid these problems.

The simulator should allow specifying scenarios in detail: job completion time distributions, checkpointing patterns, server availability, and so on. More generally, it should allow the specification of a distribution of scenarios (perhaps based on observed hosts and projects), so that one can study the average performance of scheduling policies, as well as their performance in particular cases. It should also accommodate trace-based simulations [16]. Creating such a simulator will be a formidable effort, but will yield a powerful tool for improving the efficiency of volunteer computing.

References

[1] Agrawala, A. K. and Bryant, R. M. “Models of memory scheduling”. SIGOPS Oper. Syst. Rev. 9, 5 (Nov. 1975), 217-222.

[2] Alexandrov, A.D., M. Ibel, K.E. Schauser, K.E. Scheiman. "SuperWeb: Research issues in Java-Based Global Computing". In Proceedings of the Workshop on Java for High performance Scientific and Engineering Computing Simulation and Modelling. Syracuse University, New York, 1996.

[3] Anderson, D.P. "BOINC: A System for Public-Resource Computing and Storage". 5th IEEE/ACM International Workshop on Grid Computing, pp. 365-372, Nov. 8 2004, Pittsburgh, PA.

[4] Anderson, D.P., E. Korpela, and R. Walton. "High-Performance Task Distribution for Volunteer Computing". 1st IEEE International Conference on e-Science and Grid Computing, Melbourne, Dec. 2005, pp. 196-203.

[5] Anderson, D.P., C. Christensen, and B. Allen. "Designing a Runtime System for Volunteer Computing", to appear in Supercomputing 06.

[6] Baratloo, A., M. Karaul, Z. Kedem, and P. Wyckoff. "Charlotte: Metacomputing on the Web". In In Proceedings of the 9th Conference on Parallel and Distributed Computing Systems, 1996.

[7] Chien, A., B. Calder, S. Elbert, and K. Bhatia. "Entropia: architecture and performance of an enterprise desktop grid system", J. Parallel Distrib. Comput. 63 (2003) 597-610.

[8] Christensen, C., T. Aina, D. Stainforth, "The Challenge of Volunteer Computing With Lengthy Climate Modelling Simulations", Proceedings of the 1st IEEE Conference on e-Science and Grid Computing, Melbourne, Australia, 5-8 Dec 2005.

[9] Distributed.net, http://distributed.net

[10] GIMPS, http://www.mersenne.org/prime.htm

[11] Henry, G.J. "The Fair Share Scheduler". Bell Syst. Tech. J. 63, 8, Part 2 (Oct. 1984), 1845-1857.

[12] Liu, C.L. and J.W. Layland. "Scheduling algorithms for multiprogramming in a hard-real-time environment". Journal of the ACM 20,1 (1973), 46—61.

[13] Neary, M.O., B.O. Christiansen, P. Capello , K.E. Schauser, "Javelin: Parallel Computing on the Internet", Future Generation Computer Systems 15 (1999) pp 659-674.

[14] Nisan, N., S. London, O. Regev, N. Camiel, "Globally distributed computation over the Internet---the POPCORN project", Proceedings of the International Conference on Distributed Computing Systems (ICDCS'98), 1998.

[15] Sarmenta, L.F.G. and S. Hirano. "Bayanihan: Building and Studying Web-Based Volunteer Computing Systems Using Java". Future Generation Computer Systems, 15(5/6), 1999.

[16] Sherman, S. W. and Browne, J. C. "Trace driven modeling: Review and overview". In Proceedings of the 1st Symposium on Simulation of Computer Systems (Gaithersburg, Maryland, United States, June 19 - 20, 1973). H. J. Highland, Ed.

[17] Spooner, D.P., S.A. Jarvis, J. Cao, S. Saini and G.R. Nudd, "Local grid scheduling techniques using performance prediction", IEE Proceedings Computers and Digital Techniques, 150(2), pp. 87-96, 2003.

[18] Taufer, M., D. Anderson, P. Cicotti, C.L. Brooks III. "Homogeneous Redundancy: a Technique to Ensure Integrity of Molecular Simulation Results Using Public Computing". Heterogeneous Computing Workshop, International Parallel and Distributed Processing Symposium 2005, Denver, CO, April 4-8, 2005.