High-Performance Task Distribution for Volunteer Computing

David P. Anderson
Eric Korpela
Rom Walton

Space Sciences Laboratory
University of California, Berkeley
{davea, korpela, rwalton}@ssl.berkeley.edu

Abstract

Volunteer computing projects use a ***task server*** *to manage work. Clients periodically communicate with the server to report completed tasks and get new tasks. The rate at which the server can dispatch tasks may limit the computing power available to the project. This paper discusses the design of the task server in BOINC, a middleware system for volunteer computing. We present measurements of the CPU time and disk I/O used by a BOINC server, and show that a server consisting of a single inexpensive computer can distribute on the order of 8.8 million tasks per day. With two additional computers this increases to 23.6 million tasks per day.*

1. Introduction

Volunteer computing is a paradigm in which large numbers of computers, volunteered by members of the general public, provide computing and storage resources. Early volunteer computing projects include the Great Internet Mersenne Prime Search [10], SETI@home [1], Distributed.net [6] and Folding@home [12]. Volunteer computing is being used in high-energy physics, molecular biology, medicine, astrophysics, climate study, and other areas.

BOINC (Berkeley Open Infrastructure for Network Computing) is a middleware system for volunteer computing [2]. BOINC is being used by a number of projects, including SETI@home, Climateprediction.net [5], LHC@home [13], Predictor@home [16], and Einstein@Home [7]. Volunteers participate by running a BOINC client program on their computers. They can "attach" each computer to any set of projects, and can control the resource fraction devoted to each project.

BOINC-based projects are autonomous. Each project operates a server consisting of several components:

- **Web interfaces** for account and team management, message boards, and other features.
- A **task server** that creates tasks, dispatches them to clients, and processes returned tasks.
- A **data server** that downloads input files and executables, and that uploads output files.

These components share various data stored on disk, including relational databases and upload/download files (see Figure 1).

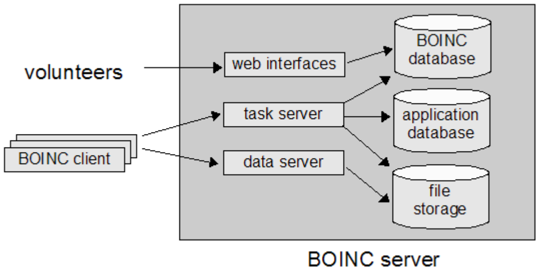

Figure 1: A BOINC server consists of several components, sharing several forms of storage.

This paper addresses the design and performance of the BOINC task server. The other components can also impose significant server load, but are outside the scope of this paper.

Each client periodically communicates with the task server to report completed work and to get new work. In addition, the server performs a number of background functions, such as retrying and garbage-collecting tasks. The load on a task server depends on the number of volunteer hosts and their rates of communication. The number of volunteer hosts in current projects ranges from tens to hundreds of thousands, and in the future may reach tens or hundreds of millions. If servers become overloaded, requests fail and hosts become idle. Thus, server performance can limit the computing capacity available to a volunteer computing project.

While developing and deploying BOINC we solved a variety of server performance problems. This paper describes the design of the BOINC server software, and presents measurements of the CPU time and disk bandwidth used by its various components. From these measurements we conclude that, using a single computer costing about $4,000, a BOINC project can dispatch about 8.8 million tasks per day. If each client is issued one task per day and each task uses 12 CPU hours on a 1 GFLOPS computer, the project can support 8.8 million clients and obtain 4.4 PetaFLOPS of computing power. With two additional server computers, a project can dispatch about 23.6 million tasks per day.

2. The BOINC computing model

Grid computing [9] involves resource sharing between organizations that are mutually accountable. In contrast, participants in a volunteer computing project are not accountable to the project (indeed, their identity is unknown), and the volunteered hosts are unreliable and insecure.

Thus, when a task is sent to a host, several types of errors are possible. Incorrect output may result from a hardware malfunction (especially in hosts that are "overclocked"), an incorrect modification to the application, or a intentional malicious attack by the volunteer. The application may crash. There may be no response to the project, e.g. because the host dies or stops running BOINC. An unrecoverable error may occur while downloading or uploading files. The result may be correct but reported too late to be of use.

2.1) Persistent redundant computing

Because the above problems occur with non-negligible frequency, volunteer computing requires mechanisms for **validation** (to ensure that outputs are correct) and **retry** (to ensure that tasks eventually get done). BOINC provides a mechanism called **persistent redundant computing** that accomplishes both goals. This mechanism involves performing each task independently on two or more computers, comparing the outputs, looking for a "quorum" of equivalent outputs, and generating new instances as needed to reach a quorum.

In BOINC terminology, a **job** is a computational task, specified by a set of input files and an application program. Each job J has several scheduling-related parameters:

- DelayBound(J): a time interval that determines the deadline for instances of J.
- NInstances(J): the number of instances of J to be created initially.
- MinQuorum(J): the minimum size of a quorum.
- Estimates of the amount of computing, disk space, and memory required by J.
- Upper bounds on the number of erroneous, correct, and total instances. These are used to detect jobs that consistently crash the application, that return inconsistent results, or that cause their results to not be reported.

A **job instance** (or just "instance") refers to a job and specifies a set of output files. An instance is **dispatched** to at most one host. An instance is **reported** when it listed in a scheduler request message. If enough instances of a job have been reported and are equivalent, they are marked as **valid** and one of them is selected as the job's **canonical instance**.

BOINC implements persistent redundant computing as follows:

1) When a job J is created, NInstances(J) instances for J are created and marked as unsent.
2) When a client requests work, the task server selects one or more unsent instances and dispatches them to the host. Two instances of the same job are never sent to the same participant, making it unlikely that a maliciously incorrect result will be accepted as valid. The instance's deadline is set to the current time plus DelayBound(J).
3) If an instance's deadline passes before it is reported, the server marks it as "timed out" and creates a new instance of J. It also checks whether the limit on the number of error or total instance of J has been reached, and if so marks J as having a permanent error.
4) When an instance I is reported, and its job already has a canonical instance I*, the server invokes an application-specific function that compares I and I*, and marks I as valid if they are equivalent. If there is no canonical instance yet, and the number of successful instances is at least MinQuorum(J), the server invokes an application-specific function which, if it finds a quorum of equivalent instances, selects one of them as

the canonical instance I*, and marks the instances as valid if they are equivalent to I*. Volunteers are granted **credit** for valid instances.

2.2) Scheduling policy options

BOINC's task server can be configured for any of several scheduling policies. In all policies, an instance it sent to a host only if the host has sufficient memory and disk and is likely to complete the instance by its deadline. Two instances of the same job are never sent to the same participant. The policy options are as follows:

- Bag-of-tasks: no restrictions beyond the above.
- Homogeneous redundancy: once an instance of a given job J has been sent, further instances of J are sent only to numerically equivalent hosts [18].
- Locality scheduling: a host H is preferentially sent instances that use data files currently resident on H.

The measurements in this paper use the bag-of-tasks policy. The other policies impose higher server load.

3. BOINC task server architecture

3.1) Task server components

BOINC implements a task server using a number of separate programs, which share a common MySQL database (see Figure 2).

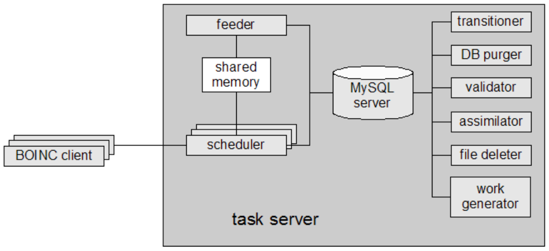

Figure 2: The components of a BOINC task server

- The **work generator** creates new jobs and their input files. For example, the SETI@home work generator reads digital tapes containing data from a radio telescope, divides this data into files, and creates jobs in the BOINC database. The work generator sleeps if the number of unsent instances exceeds a threshold, limiting the amount of disk storage needed for input files.
- The **scheduler** handles requests from BOINC clients. Each request includes a description of the host, a list of completed instances, and a request for additional work, expressed in terms of the time the work should take to complete. The reply includes a list of instances and their corresponding jobs. Handling a request involves a number of database operations: reading and updating records for the user account and team, the host, and the various jobs and instances. The scheduler is implemented as a Fast CGI program run from an Apache web server [3], and many instances can run concurrently.
- The **feeder** streamlines the scheduler's database access. It maintains a shared-memory segment containing 1) static database tables such as applications, platforms, and application versions, and 2) a fixed-size cache of unsent instance/job pairs. The scheduler finds instances that can be sent to a particular client by scanning this memory segment. A semaphore synchronizes access to the shared-memory segment. To minimize contention for this semaphore, the scheduler marks a cache entry as "busy" (and releases the semaphore) while it reads the instance from the database to verify that it is still unsent.
- The **transitioner** examines jobs for which a state change has occurred (e.g., a completed instance has been reported). Depending on the situation, it may generates new instances, flag the job as having a permanent error, or trigger validation or assimilation of the job.
- The **validator** compares the instances of a job and selects a canonical instance representing the correct output. It determines the credit granted to users and hosts that return the correct output, and updates those database records.
- The **assimilator** handles job that are "completed": i.e., that have a canonical instance or for which a permanent error has occurred. Handling a successfully completed job might involve writing outputs to an application database or archiving the output files.
- The **file deleter** deletes input and output files that are no longer needed.
- The **database purger** removes jobs and instance database entries that are no longer needed, first writing them to XML log files. This bounds the size of these tables, so that they act as a working set rather than an archive. This allows database management operations (such as backups and schema changes) to be done quickly.

The programs communicate through the BOINC database. For example, when the work generator creates a job it sets a flag in the job's database record indicating that the transitioner should examine it. Most of the programs repeatedly scan the database, enumerating records that have the relevant flag set, handling these records, and clearing the flags in the database. Database indices on the flag fields make these enumerations efficient. When an enumeration returns nothing, the program sleeps for a short period.

Thus, a BOINC task server consists of many processes, mostly asynchronous with respect to client requests, that communicate through a database. This approach has the disadvantage of imposing a high load on the database server. One can imagine an alternative design in which almost all functions are done by the scheduler, synchronously with client requests. This would have lower database overhead. However, the current design has several important advantages:

- It is resilient with respect to failures. For example, only the assimilator uses the application database, and if it unavailable only the assimilator is blocked. The other components continue to execute, and the BOINC database (i.e., the job records tagged as ready to assimilate) acts as a queue for the assimilator when it runs again.
- It is resilient with respect to performance. If back-end components (e.g. the validator or assimilator) perform poorly and fall behind, the client-visible components (the feeder and scheduler) are unaffected.
- The various components can easily be distributed and/or replicated (see below).

3.3) Distribution of components

The programs making up a BOINC task server may run on different computers. In particular, the BOINC database may run on a separate computer (MySQL allows remote access). Many of the programs require access to shared files (configuration files, log files, upload/download data files) so generally the server computers are on the same LAN and use a network file system such as NFS.

The server programs may also be replicated, either on a multiprocessor host or on different hosts. Interference between replicas is avoided by having each replica work on a different subset of database items. The space of database identifiers is partitioned: if there are n replicas, replica i handles only items (e.g., jobs) for which (ID mod n) = i.

4. Performance measurements

We made performance measurements of the various server components. All measurements were made on a Dell 3850 PowerEdge server, with 2 GB of RAM and 2 Intel Xeon processors running at 2.4 Ghz. The storage is 3 160 GB SCSI disks configured as Raid 0, with a peak throughput of about 100 Mbytes/sec. The server runs the Linux 2.4.21-20.ELsmp kernel. It runs the 4.0.22 version of MySQL, with the “max” configuration settings. The BOINC database tables use InnoDB.

We used a synthetic workload consisting of 50,000 jobs, each with NInstances(J) = 2 and MinQuorum(J) = 2. Thus there were 100,000 instances. We measured each phase of the server’s functions in handling these jobs and instances. In each phase, we ran the necessary programs and the MySQL server on a single host, and measured MySQL CPU time and disk traffic, and application CPU time. We averaged these measurements over several runs.

This workload is simpler than the workload in a real BOINC system. For example, the synthetic workload involves a single user account and host. The user does not belong to a team, so there is no lookup or update of team records. The synthetic workload does not model instance timeout and errors. The resulting database fits entirely in RAM, so little disk reading is done.

4.1) Work creation

We ran a program that creates 50,000 jobs, then ran the transitioner, which creates two instances per job.

Elapsed time	44 seconds
MySQL CPU time	15 seconds
Work generator CPU time	1.2 seconds
Transitioner CPU time	30 seconds
Disk traffic	76 MB write, 0.3 MB read

4.2) Work dispatch

We ran a driver program that generates a sequence of scheduler requests, piping them into a scheduler, with a feeder running concurrently. Each request gets one new instance, and reports the successful completion of the previous instance. Thus there were 100,000 scheduler requests.

Elapsed time	522 seconds
MySQL CPU time	130 seconds
Driver CPU time	37 seconds
Feeder CPU time	12 seconds
Scheduler CPU time	320 seconds
Disk traffic	50 MB write, 7 MB read

In an operational server, the scheduler runs under Apache using Fast CGI, a mechanism that allows a single process to handle many requests. Our performance measurements differ from this only in the absence of the overhead of network connection establishment and the forwarding of data from Apache to the scheduler process. This does not use a significant fraction of total CPU, does not occur in the MySQL server.

4.3) Validation

We ran the transitioner (which flags jobs as needing validation) and a "dummy validator" that marks all instances as valid and marks all jobs as ready to assimilate.

Elapsed time	255 seconds
MySQL CPU time	118 seconds
Transitioner CPU time	1.12 seconds
Validator CPU time	162.53 seconds
Disk traffic	317 MB write, 1.2 MB read

4.4) Assimilation

We ran a "dummy assimilator" that marks jobs as assimilated.

Elapsed time	107.32 seconds
MySQL CPU time	12 seconds
Assimilator CPU time	97.33 seconds
Disk traffic	50 MB write, 0.9 MB read

4.5) File deletion

We ran the transitioner (which marks assimilated jobs as ready for file deletion) and then ran the file deleter.

Elapsed time	49.73 seconds
MySQL CPU time	45 seconds
Transitioner CPU time	3.26 seconds
File deleter CPU time	16.38 seconds
Disk traffic	264 MB write, 1.5 MB read

4.6) Database purge

We ran the database purger, which purges jobs and their instances.

Elapsed time	92.2 seconds
MySQL CPU time	45 seconds
Database purge CPU time	69.27 seconds
Disk traffic	338 MB write, 2 MB read

5) The limits of task server performance

The performance measurements from the previous section give an upper bound on the rate at which a BOINC project can process instances. We calculate the instance-dispatch rate at which a system component (CPU or disk) will become "saturated". We define this as a state where device utilization is above 50%.

5.1) Single server computer

Summing CPU times and disk bandwidths over all phases, we see that handling 100,000 instances uses 980 CPU seconds and 1105 MB of disk I/O. On our reference machine (with 2 CPUs) the CPU is saturated at a rate of 8.8 million instances per day. At this rate, disk utilization is about 2.5%, so the CPU is the bottleneck resource.

If each client is issued one instance per day and each instance uses 12 CPU hours on a 1 GFLOPS computer, the project can support 8.8 million clients and obtain 4.4 PetaFLOPS of computing power

5.2) Multiple server computers

Suppose the project can devote several computers (identical to our reference computer) to server functions. All server components except the MySQL server can be replicated arbitrarily, so the MySQL server is the system bottleneck. Summing the MySQL CPU load over all phases, we see that handling 100,000 instances uses 365 CPU seconds. The MySQL server's CPU will be saturated at a rate of 23.6 million instances per day. Disk utilization is about 6.5% at this rate.

At 23.6 million instances per day, the CPU utilization of other functions is as follows:

Transitioner	0.048
Work generator	0.0016
Scheduler + feeder	0.45
Validator	0.22
File deleter	0.02
DB purger	0.09

Thus these functions can be handled by two other reference computers without CPU saturation.

5.3) The effects of database size

Our synthetic workload involves a database that fits entirely in RAM. To study the effects of larger databases, we measured SETI@home's task server. SETI@home's BOINC database occupies 36 GB of memory, and its database server has 8 GB of RAM. The CPU load on the database server is about 4%. The average disk rate is 0.7 MB/sec read and 0.01 MB/sec write. Thus, the MySQL server does 17 MB of disk I/O per CPU second. With the synthetic workload, the MySQL server did 3.02 MB of disk I/O per CPU second.

Thus, for SETI@home, a non-RAM-resident database increases disk I/O by a factor of roughly 6. If we apply this factor to the cases in sections 5.1 and 5.2, we see that disk utilization increases to about 15% and 40% respectively, so that CPU is still the system bottleneck.

5.4) Network bandwidth

Network bandwidth may also be a system bottleneck. BOINC scheduler request and reply messages average about 10 KB. The average network bandwidth needed to dispatch 8.8 million instances per day would therefore be about 8.2 Mbits/sec (both incoming and outgoing). At 23.6 million instances per day the network bandwidth is about 21.9 Mbits/sec.

Most volunteer computing participants use home computers, so this traffic goes over the commodity Internet. Some research institutions have connections to the commodity Internet that are expensive and/or slow, and for which the above data rates would be infeasible. However, it is possible to circumvent this problem by using a separate dedicated connection (currently $1000-$2000 per month for a 100 Mbps connection).

This network traffic is exclusive of file upload and download, which may be a performance issue. The BOINC architecture allows data servers to be located anywhere; they are simply web servers, and do not access the BOINC database. Current BOINC-based projects that use large files (Einstein@Home [7] and Climateprediction.net [5]) use replicated and distributed data servers, located at partner institutions. The upload/download traffic is spread across the commodity Internet connections of those institutions.

5.5) Exponential backoff of client requests

The BOINC task server performs best if the request arrival rate is stable. If the server is down for an extended behavior (e.g. several hours or days), it can potentially be overwhelmed by client requests when it comes back up. This can drive various parts of the task server (e.g. the database server) into modes that cause the task server as a whole to perform much worse than normal. To avoid this problem, the BOINC client uses random exponential backoff [15, 17] when server requests fail. As a result, the request arrival rate remains stable even after long server outages.

6. Optimizing database queries

Database performance dominates the performance of a BOINC task server. The BOINC server programs were originally developed using a database interface layer providing basic operations (insert, delete, update, select one record, enumerate a sequence of records). Each operation reads or writes all fields of the table.

This interface is simple for the programmer but in some cases performs poorly. For example, the transitioner processes jobs J for which transition_time(J) < now, and for each such job examines all of its instances. Originally the instances were fetched with a separate query for each job. We replaced this with a query that returns jobs and their instances in a single stream (see Figure 3). The optimized query is:

```
SELECT *
FROM job
LEFT JOIN instance
ON job.id = instance.jobid
WHERE job.transition_time < now
LIMIT 1000
```

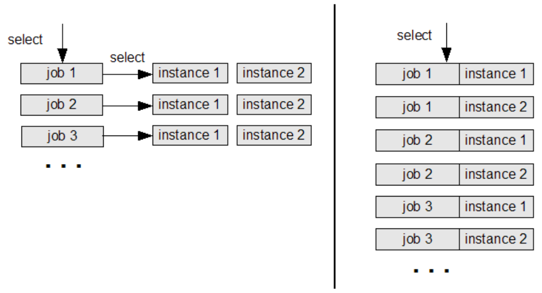

Figure 3: Old and new enumerations of jobs and their instances.

The transitioner scans the stream returned by this query. It identifies complete groups of instances by noting when the job ID changes. (If the query returns fewer than 1000 rows, the last group is complete.)

This query uses more bandwidth from the database server because each job is sent multiple times. However, the number of queries (and hence round-trips to the database server) is reduced by a factor of several thousand, providing much better system performance. A similar technique is used in the validator and feeder.

The BOINC server database code does not use transactions, because they are not supported in some versions of MySQL and they often reduce performance. This decision led to some bugs involving conflicting updates. We fixed these problems using a combination of techniques:

- Most fields are modified only by one program.
- Updates modify only the fields that have actually changed. This also improves performance, because large fields (like "blobs" containing XML text) are updated only when necessary.
- When possible, updates are relative (using increment or max()) rather than absolute.

7. Related work

The design of the BOINC task server is based largely on the experience of SETI@home [1]. When SETI@home was released in May 1999, it quickly acquired about 400,000 participants. Its task server (based on two Sun workstations) was soon overwhelmed. The server was modified to use a feeder/shared-memory scheme. Result validation was add to SETI@home as an afterthought, but not integrated with the credit system. This led to various "credit-cheating" attacks, which motivated the BOINC design.

Folding@home [12] uses a two-level task scheduler. Clients contact a main server (which has a project-wide database) and are forwarded to one of several experiment-specific servers, each of which has a database of jobs for that experiment. This design is perhaps even more scalable than BOINC, at the expense of greater database management overhead.

Xtremweb [8], an experimental middleware system for volunteer computing, has a task server that uses MySQL. The Xtremweb web site reports that this server can dispatch 1 million tasks per week (0.14 million per day).

Commercial systems for volunteer and "desktop grid" computing (such as United Devices [19] and Entropia [4]) have roughly the same server functions as BOINC, and use relational databases to store task and participant data. Measurements of their server performance are not available.

There is a large body of work on scheduling for Grid systems [9]. These schedulers have functions that differ from BOINC's; they deal with complex workflows rather than single tasks, and they do not deal with redundancy and credit. Published results describe the performance of the schedules, not the schedulers. Govindaraju et al. [11] studied the performance of XML generation and parsing in SOAP, an interface layer used in the current generation of Grid systems. The overhead of this layer (roughly 1 CPU second per 100,000 floating-point numbers) would create a severe bottleneck in a large volunteer computing project. BOINC does its own XML generation and parsing.

Other researchers have proposed distributed schedulers; an extreme example is Liljeqvist and Bengtsson [14], who describe a Grid scheduler implemented in network routers. Such systems are often hard to deploy and debug in practice.

8. Conclusion

We have shown that a BOINC task server, running on inexpensive hardware, can potentially dispatch tens of millions of tasks a day. The database server (and in particular its CPU) is typically the system bottleneck.

The network bandwidth needed for task serving is typically only a few Mbps, and because of BOINC's multi-project design, a project's task server need not be highly available. Hence, at least for task serving, large-scale volunteer computing can be done using the hardware resources typically available to small research projects. There is no need for expensive servers and hosting facilities.

Several aspects of the BOINC server design contributed to achieving this level of performance:

- The use of a shared-memory work cache, replenished by a separate process, to reduce database traffic.
- Optimization of database queries, and in particular using joins to reduce the number of queries.
- The avoidance of high-overhead data representation layers.

This work was supported by the National Science Foundation under grants SCI-0221529 and SCI-0438443. We thank Jeff Cobb, Matt Lebofsky, and Bob Bankay for their help in identifying and diagnosing performance problems in SETI@home's BOINC server, which motivated much of the work described here.

9. References

[1] D.P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, D. Werthimer. "SETI@home: An Experiment in Public-Resource Computing". *Communications of the ACM*, 45(11), November 2002, 56-61.

[2] D.P. Anderson. "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.

[3] M.R. Brown. "FastCGI: A High-Performance Gateway Interface", *Fifth International World Wide Web Conference*, 6 May 1996, Paris, France.

[4] A. Chien, 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.

[5] Climateprediction.net, http://climateprediction.net/

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

[7] Einstein@Home, http://einstein.phys.uwm.edu/

[8] G. Fedak, C. Germain, V. Néri and F. Cappello. "XtremWeb : A Generic Global Computing System", CCGRID2001 Workshop on Global Computing on Personal Devices, May 2001.

[9] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann: San Francisco (CA), 1999.

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

[11] M. Govindaraju, A. Slominski, K. Chiu, P. Liu, R. Engelen, M. Lewis. "Toward Characterizing the Performance of SOAP Toolkits". 5th IEEE/ACM International Workshop on Grid Computing, pp. 365-372, Nov. 8, 2004, Pittsburgh, PA.

[12] S.M. Larson, C.D. Snow, M. Shirts and V.S. Pande. "Folding@Home and Genome@Home: Using distributed computing to tackle previously intractible problems in computational biology". *Computational Genomics*, Horizon Press, 2002.

[13] LHC@home, http://athome.web.cern.ch/athome/

[14] B. Liljeqvist and L. Bengtsson. "Grid Computing Distribution Using Network Processors". 14th IASTED Conference on Parallel and Distributed Computing Systems, 2002, Cambridge, MA.

[15] R.M. Metcalfe and D.R. Boggs. "Ethernet: Distributed packet switching for local computer networks", *Commun. ACM* 19 (7), 395-404 (1976).

[16] Predictor@home, http://predictor.scripps.edu/

[17] N.-O. Song, B.-J. Kwak and L. Miller. "On the Stability of Exponential Backoff", *J. Res. Natl. Inst. Stand. Technol.* 108, 289-297 (2003).

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

[19] United Devices, http://www.ud.com