A COMPARISON OF SIMPLE TASK ALLOCATION WITH COMPLETE TASK CHECKING AND ROSENBERG TASK ALLOCATION WITH SPOT CHECKING

An Honors Thesis

Presented by

Derik DeLong

Submitted
May 2004

Guidance Committee Approval:

____
Professor Arnold Rosenberg, Computer Science

____
Professor Brian Levine, Computer Science

ABSTRACT

Title: **A Comparison Of Simple Task Allocation With Complete Task Checking And Rosenberg Task Allocation With Spot Checking**
Author: **Derik DeLong**
Research area: **Computer Science**
Guidance Committee Chair & Dept: **Arnold Rosenberg, Computer Science**
Guidance committee Member & Dept: **Brian Levine, Computer Science**

The emerging use of web-computation systems for large projects has made it a possible source for low cost computation power. The anonymous nature of the Internet has also made these projects a target for malicious users. Current protection techniques degrade performance heavily as well as suffer from susceptibility to collusion and invalid programming. Techniques to ensure correctness using internal solution verification facilities are examined. In particular, schemes in which all units are verified as well as one using spot-checking combined with accounting will be studied. The accounting is based upon concepts introduced by Arnold Rosenberg through the use of task allocation functions. Effects upon performance, particularly throughput will be examined. Additionally, the effectiveness of the two schemes to protect against incorrect units will be measured. The computational cost of using accounting will be measured and analyzed.

Honors Thesis (499T)

Introduction

Background

The popularity of the Internet that has made it an essential resource has yielded many new opportunities for taking advantage of the large amount of computing resources now in the hands of individuals. Inexpensive personal computers have large amounts of computation power that largely go untapped by the owning individuals. This is true for both private individuals as well as users in a corporate environment. These computers spend a large amount of time idle and even when in use, the primary uses (such as word processing and web "surfing") do not tax the computer to its full ability [1]. Indeed, the rate at which computer speed has grown has outpaced the actual requirements of those using the computers. This is particularly poignant in corporate environments whose users generally use the computers for little other than web browsing and word processing, neither of which are particularly demanding in terms of CPU use.

This has led to new paradigms of computing in which these resources get utilized at little or no cost to those using them remotely. One such service is the SETI@Home project [2]. Users sign up, install a client to retrieve data via the Internet, and process work units using that client. These clients essentially lease their computers to these projects for no cost. The main motivating factor is either competition with other users, or simply a belief in the value of the project's goals. However, the anonymous nature of the Internet combined with the lack of monetary motivation or true involvement with the project itself leads to some malicious users. Some of these clients will go to extreme lengths to either boost their own statistics or derail the project, even going as far as creating several usernames on message boards and talking to themselves [3].

Current Techniques

The current method of dealing with malicious users is the use of a voting scheme. The way in which it works is that the project allocates a single work unit some m times where they expect to get n copies of the solution back where n is some number of units less than m. The results are received and compared. The solution, which was returned as the majority answer, is then accepted as the correct solution. This approach is deficient in several ways. The first of which is the dramatically reduced performance that results. The performance is a mere fraction of theoretical maximum throughput possible [4]. In specific terms, throughput is 1/n of what is theoretically possible. Another deficiency is the lack of insurance that the voting will result in the correct result. Collusion or programming defects could escape this type of check. For the most part, SETI@Home ignores this problem because incorrect results for isolated units will not result in any extremely adverse effects. Other more accuracy dependent projects may find this risk unacceptable. Cases in which incorrect computation may lead to death must examine this fact more in depth.

Alternative Correctness Mechanisms

Projects may manually check submitted results to ensure correctness of data. The policy, which decides which units are checked, then becomes an important factor in how stringently data integrity is maintained. Complete checking (checking all units) will ensure perfect correctness of the data. However, this is computationally expensive to the host for the project and may negate any advantages to tapping into these distributed resources. Spot-checking (checking occasional units) is less computationally expensive, but introduces the possibility of false data in the system. Spot-checking may be threatening enough for users interested in lengthy membership in the project, but anonymity on the Internet and the ability to rejoin with a different alias weakens this concern. Malicious users may find the hassle of signing up again acceptable because they remain effective in their goal of entering false data into the system.

To address this concern, accounting can be introduced. Lists of units completed by users can be maintained and when a particular user's spot-checked unit is found to be false, their past units may be checked. This forces further examination of users that have been found to submit bad results, increasing the likelihood of discovering bad results when checking. Generally, accounting in this way had been ignored for space and computational reasons. Keeping simple lists of completed work units can take large amounts of space. Retrieving this information is linear and the space used is excessive and would for any significantly large project exceed the size of fast memory.

To achieve accounting but keep the information more manageable, task allocation functions may be used as defined by Arnold Rosenberg [5]. This accounting is maintained not with pure lists of work unit identifiers, but using strides and steps. This greatly compresses the lists being maintained. Additionally, it makes lookup logarithmic instead of linear, which is another gain.

The adoption of a spot-checking policy with accounting will reduce falsely accepted units, but still leaves a possible point of exploitation for malicious users. If malicious users are willing to sign up repeatedly, they could submit only a few units before leaving and signing up again. By minimizing the length of the stay, it's possible to avoid having a unit checked. Therefore, it becomes necessary to check at least one unit when a client times out to ensure that the each user has a unit checked at some point no matter their behavior.

The combination of these different techniques is more effective than simple spot-checking, but it is not perfect. A tradeoff of accuracy for decreased computational use is made.

Problem Types

For transformational computations in which complete checking of a solution requires a repetition of the computation, the check-all method is impractical. It would require the same amount of computational resources to check as one could muster from the volunteer pool. In fact, it's the equivalent of doing the computation with one's own computational resources. In this situation, spot-checking with accounting is still a viable solution. Comparison of the two task allocation systems becomes uninteresting and therefore, that class of problems is ignored.

Solutions to NP-Complete problems can be solved in a fraction of the time that it takes to solve them. Checking all completed units becomes a viable possibility, because using outside resources multiplies the amount of work accomplished over an interval of time. Therefore, this paper will concentrate upon comparing a check all unit policy with a spot-checking policy that includes the use of Professor Rosenberg's lightweight accounting scheme. Additionally, the style of problem being solved will be NP-Complete.

Operational Logistics

In order to demonstrate the importance of these policies, the logistics of a web computation system must be explained. Users connect to submit finished work units as well as receive a new work unit. If a user does not reconnect within a pre-determined amount of time, they are considered “timed-out” and the unit that they were processing is added back to the pool of available work units.

The check-all unit policy, which will be referred to as the simple task allocator for the duration of this paper, operates by keeping very simple records for the connecting users. The number of incorrect units and whether that user has timed out is contained within that record. Exceeding a pre-determined number of failed units (which will also be referred to as strikes) invalidates that user as does timing out. A connecting user’s entries are consulted to determine validity as a user. If the user is valid, then the submitted unit is drawn accepted into the system to processing and a new unit is returned. The units are allocated from a simple FIFO queue to connecting users with no specific reasoning other than being the next unit in the queue. Every unit is added to the queue of units to get checked and because of this, accounting of which worker did a particular unit is unimportant.

In the spot-checking policy, units are checked randomly at a rate specified at run-time. More complicated user records are also maintained. In addition to the fields contained in the records that the simple task allocator use, it also keeps records of which work units that user has worked on. The task allocation algorithm drawn from Rosenberg’s paper defines these records. This sets up a series of range queries that can be used to look up the old records. The function chosen decides which unit is allocated to a user when he connects to submit a completed unit. If a randomly checked unit is found to be false, the completing user’s completed units are all checked to make sure that they are correct. Additionally, if a user times out, one of his completed units is checked to make sure that at least one unit of his is verified for correctness.

Implementation

Programming Language

Java was chosen as the implementation language for cross platform and extensibility reasons. The source code is open and available at [6]. The cross-platform aspect of Java was important because while the source code was developed on Mac OS X, it was run on a Linux computer. Additionally, the underlying system can be used on any of a number of platforms for further experiments, which may study factors neglected for this study. The extensibility allowed by Java makes the use of different task allocation schemes trivial. Indeed, the framework built for this study can be used to study other allocation schemes and different problem types (such as transformations for which checking time equals work time).

Threading

While the parallel aspect of a web-computation would seem to suggest the use of threads would be beneficial for the simulator, the use of threads leads to non-deterministic computation. Thread scheduling in the JVM is rather dynamic which makes enforcing a particular ordering of events hard. It also introduces synchronization issues, which may cause incorrect behavior. Additionally, real time is used to simulate time passage in a thread based system. In place of an hour, a second may be used. However, processing time has significant effects upon event scheduling. Additionally, the simulation takes a large amount of time even when the amount of time used to represent an hour is small.

A single threaded model is advantageous. In order to simulate parallelism, a sorted set (priority queue) was used to store events. Events are treated as objects. Each is stamped with a simulated time and sorted according to those times. This allows events to specify when other events occur without being affected by yet other events in the system. Examples of events include client connections, timeout testing of units, submitted work unit check completion, and client joins. The program then simplifies down to a single thread that services the event with the lowest stamped time. This forces the entire simulation to be deterministic by forcing a particular order for all events. Additionally, computation time of any single event has no effect on other events because they are unaware of real time and act solely in simulated time. Finally, it speeds the simulation up because the next event that's due to occur is processed immediately instead of any actual time passing. Each event is triggered when removed from the queue, effecting its changes and adding a new event to the queue if defined to do so. After each of these events, status information is polled from the statistics manager and written out to a file. In a thread-based system, the log writes had to be periodic, causing less granularity in the log as well as a lack of determinism.

Task Allocation Agnosticism

Both the check-all policy and spot-check with accounting policy use the same framework. Both implement an interface called the TaskAllocator interface, which allows that part of the system to be modular while the rest of the system remains static. This helps to avoid any possibility of engineering another part of the system to favor a particular scheme. Other portions of the simulation are completely unaware of which class is actually in place behind that interface. This has the additional benefit of allowing yet other task allocation techniques without modification of the underlying system.

Timeouts

The time out mechanism in the simulation does not use individual timers to track work units. Instead, it checks the units on a period of slightly longer than the maximum expected completion time. However, in order to avoid excessive work unit examination, the units are kept in a queue that is sorted based upon allocation time. Therefore, units are checked in that enumeration only as long as the units have actually timed out.

Status Information Management

The statistic mechanism for the simulation relies upon outside feedback from other parts of the system. Instead of requiring the individual parts to maintain information necessary for the computation of these numbers, they simply register events with the statistics mechanism, which maintains the number independently of the rest of the system. This is a simple method and fast method of maintaining these important numbers, but care must be taken when constructing a new task allocator to ensure that it correctly communicates with the statistics manager. Failure to do could result in incorrect information. This is mentioned as an aside as the task allocators used in this study correctly register events.

One measure that is recorded into the logs is not deterministic due to a failing in Java. There is no way to determine actual CPU time for any set of operations, and therefore, CPU utilization of task allocators is approximated using real time. Because one point of interest for this paper is the amount of work required by both the simple and Rosenberg task allocators, this information must be tracked. This work is actually performed unlike other parts of the system, which are simply simulated. Ideally, pure CPU time would be the way to measure this work because activity in the host computer would have no effect on this metric and therefore make it deterministic. However, this is not possible and therefore depending on background operating system activity, these values can vary. This was minimized and therefore can be treated as noise, with no significant effect.

Note: Diagrams of the classes involved in the implementation are attached. These diagrams show how the different parts interact.

Experiments

Simulator Parameters

There are sixteen variables that can be varied for the simulator. Several are options simply for the simulation itself and do not affect the behavior of the entities within the simulation (the random number seed and end time). The variables studied were the task allocator used (Rosenberg and simple), malicious user rate (10%, 0%, 25%, and 0.1%), unit check rate (2%, 10%, 1%, and 0.1%), client quit rate (25%, 50%, 0%, and 10%), check to work time ratio (4.3471e-4, 0.01, and 0.001), and new client spawn rate in simulated hours (0.1, 0.01, and 0.5).

The statically set variables for the experiments were the mistake rate (0.1%), the base work time for a unit in simulated hours (8), maximum additional time for unit processing (4), the number of checking servers (1), maximum users (10,000), initial number of users (1000), number of strikes (1), number of work units (1,000,000), random seed (0), and length of simulation in simulated hours (1000).

Check to Work Ratio

The 4.3471e-4 check-ratio was a ratio found using separate experiment. The time to find the solution and checking the solution for a series of 100 different subset-sum problems was collected and the ratio was then calculated. The details of the subset-sum problem can be found in [7]. The source code used for this experiment is included. The set being worked on is 300 elements large, with elements ranging between 1 and 1000, adding up to 90,000. This allowed for a realistic ratio be used as the basis when other variables are being varied.

Client Quite Rate

The default quit-rate was based upon statistics provided by SETI@Home about the behavior of their own users [8]. The general behavior for their users is a logarithmic drop off. Half of their users complete a single unit before quitting. However, this is not typical of all web computation systems, especially systems in corporate setting or in situations in which completing work provides some kind of reward. Also, SETI@Home attracts the most attention of any web computation project, subjecting it to the most users with no real interest in the actual project. Therefore, a smaller quit-rate of 25% was chosen as the default.

Malicious Users

Malicious user rates are generally unknown because the safeguard against such efforts is voting currently which doesn't actually track the number of users submitting incorrect units. Ten percent was chosen as the default to introduce some, but not major accounting complication.

Unit Check Rate

The unit check-rate was also chosen without any specific background reasoning. One out of fifty was chosen as the default because it struck a nice balance between doing little work and minimizing the chances of false positives.

Client Spawn Rate

The client spawn rate had several values with no special significance. 0.1 was chosen as the default because it was the compromise between the other two values.

Results

Graphs

Graphs of the various mutations within the experiment are attached. The individual runs are labeled as follows:

Variable	**Number**
Check to Work Ratio	1
Spawn Time	2
Quit Rate	3
Malicious Rate	4
Check Rate	5

The settings for each experiment are summarized in the table below:

Variable/Case	**1 (default)**	**2**	**3**	**4**
1	4.3471e-4	0.01	0.001	-
2	0.1	0.01	0.5	-
3	0.25	0.5	0.0	0.1
4	0.1	0.0	0.25	0.4
5	0.02	0.1	0.01	0.001

Note: Default values for variables were used for any variables not being varied.

Eight types of graphs are generated for each set of variables:

Label	**Description**
acctime	The amount of real time consumed by accounting processing.
checked	The number of work units checked.
checkqueue	The number of work units waiting in the check queue.
completeratetime	The number of units completed per simulated hour.
completions	The total number of completed units including rejects.
falseposnum	The number of falsely accepted units.
falserate	The percentage of false positives to completions including rejections.
falseratenorej	The percentage of false positives to completions excluding rejections.

Accounting Time

The amount of accounting time never exceeded 35 seconds (see Accounting Time 3.3) and generally didn't go above 5 seconds. Considering that these simulations ran for

1000 simulated hours, both task allocation methods do not constitute a major factor in the use of the CPU in the server. In fact, with an upper bound of 35 seconds, the accounting time is only 9.7223e-04% of the server's CPU use. However, the Rosenberg based task allocator uses between 2 and 3 times the amount of CPU time used by the simple allocator.

Check Queue Size

Check queue size differed widely between the two schemes, as one might intuitively guess. In situations in which the single server was sufficient to process checking the units coming in for the simple task allocator, the check queue size stayed very low and stable. However, the Rosenberg-based task allocator had a wildly changing queue size, which would vary between moderately large and empty. The sudden growth of the queue corresponds with the discovery of a false unit from a user and checking of the old units, which causes the sudden submission of many units for checking (see Check Queue Size 1.1). The Rosenberg scheme shows its strength in cases where checking all the units exceeds the capacity of the server. This happens when the check to work ratio is high (see Check Queue Size 1.2) or users infrequently quit (see Check Queue Size 3.3).

False Positive Rate (No Rejections)

Spot-checking, even with Rosenberg task allocation, will incorrectly accept a certain number of incorrect units. Dependent upon several factors, the ratio of falsely accepted work units to total work units accepted may change. The variable with the most significant effect is the malicious user rate (see False Positive Rate No Rejections 4.1-4.4). In particular, comparing experiment 4.1 to 4.4, we can see a very large difference.

Simulation 4.1 converges to 2.3753294% while simulation 4.4 converges to 12.53475%. That's 5.27 times larger.

This rate is also affected by the check rate (see False Positive Rate No Rejections 5.1-5.4). The higher the spot-check rate, the lower the false acceptance rate will be. In particular, comparison of simulation 5.2 and 5.3 (which exhibit spot check rates of 10% and 1% respectively) shows that using 10% instead of 1% yields half the amount of false positives. 10% yields a 1.326% false positive rate while 1% yields a 2.637% false positive rate.

Complete Rate

The complete rate (or average completion time for a unit) shows very similar information for unit checking policies for all but two cases. These two styles diverge in simulation 1.2 and 3.3 (see Complete Rate 1.2 and 3.3). Simulation 1.2 shows a difference when the check to work ratio is large. Simulation 3.3 shows that malicious rate has a rather negative effect upon throughput. When the check rate is too high, the throughputs diverge but both converge to their own respective rates. This also results in a static ratio between the two. For simulation 3.3, the difference continues to grow with time.

Completions

The graphs generated by the number of completed units in the simulations reveal differences in only two cases between the two algorithms (see Completions 1.2 and 3.3). In simulation 1.2, the two complete units at a linear rate, but there is a multiplier difference between the two with the Rosenberg task allocator exhibiting three times the completions. In simulation 3.3, the simple task allocator completes tasks in a linear manner, but the Rosenberg task allocator does so quadratically.

Conclusions

Task Allocator Computational Cost

Neither the simple task allocator, nor the Rosenberg based task allocator uses significant computer resources. The linear difference between the two styles is also unimportant because neither grows at a rate great enough to become significant over time. Both grow at a linear rate.

Spot-Checking Effects

The check queue size, complete rate, and completions suggest the same thing. When the cost of checking incoming units exceeds the capacity of the checking server(s), the check queue begins to grow unboundedly. The rate that units are submitted exceeds the rate at which they can be checked, causing a backlog. The capacity of the system will outpace the rate at which the host can verify the results. The cost of checking becomes the limiting factor.

False Positive Rate

While the Rosenberg task allocation scheme attempts to minimize false positives, it is not 100% effective like the simple task allocator. Even worse, as the malicious rate increases, the percentage of completed units that are false increases. The results are many times more inaccurate when the percentage of users that are malicious is increased.

Increasing the check rate also affects this rate and increasing the number of units checked helps further reduce the number of false positives. However, this is not a linear relationship and doubling the check rate does not divide the false acceptances in half.

Final Remarks

The additional computational cost of adopting the Rosenberg task allocator over the simple task allocator is insignificant due to the small total cost of either. Neither would consume many resources. The resources are so few that even though the Rosenberg algorithm consumes 2-3 times as much, it's still extremely small, and therefore should not be taken into account when choosing between the two.

While the Rosenberg task allocation system is significantly more efficient than the simple task allocator, it's only a viable solution if one is willing to accept a certain amount of incorrect data. Further, one must be aware of the number of malicious users that they are likely to attract. If their estimate is significantly less than the true number, the real ratio of incorrect data to completed data will be higher than expected, which may invalidate the project or have other adverse effects.

In situations where the risk of false positives is acceptable, it then becomes necessary to determine whether the rate which clients submit units could exceed the rate at which the checking server(s) can process all those units. If so, then using the Rosenberg task allocator becomes a good way to maintain maximum throughput at a small inaccuracy cost. This improved throughput can be many times the throughput of the simple version. Over time, this leads to a very large performance disparity. Additionally, the spot check rate can be varied to maximize accuracy while avoiding a backlog and reduced performance.

References

[1] C. Weth, U. Kraus, J. Freuer, M. Ruder, R. Dannecker, P. Schneider, M. Konold, H. Ruder (2000): Xpulsar@home – School help Scientists. *1st International Symposium on Cluster Computing and the Grid.*

[2] E. Korpela, D. Werthimer, D. Anderson, J. Cobb, M. Lebofsky (2000): SETI@home: massively distributed computing for SETI. In *Computing in Science and Engineering* (P.F. Dubois, Ed.) IEEE Computer Soc. Press, Los Alamitos, CA.

[3] Kahney, Leander. "Cheaters Bow to Peer Pressure." Wired Magazine Online. 15 Feb. 2001 <http://www.wired.com/news/technology/0,1282,41838,00.html>.

[4] D. Kondo, H. Casanova, E. Wing, F. Berman (2002): Models and scheduling guidelines for global computing applications. *Intl. Parallel and Distr. Processing Symp. (IPDPS'02).*

[5] Rosenberg, Arnold L. (2003): Accountable Web-Computing. *IEEE. Trans. Parallel and Distr. Systs. 14*, 97–106.

[6] http://godlikenerd.com/webcomp/

[7] K. Mehlhorn (1984): *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness.* EATCS Monographs on Theoretical Computer Science, Springer-Verlag, Berlin.

[8] User Statistics. <http://setiathome.berkeley.edu/userresults.html>.