

GRID RESULT CHECKING

By

Peter C. Grant

Thesis

Submitted to the Faculty of the
Graduate School of Vanderbilt University
in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Computer Science

May, 2006

Nashville, Tennessee

Approved:

Date:

Julie A. Adams

Jeremy P. Spinrad

TABLE OF CONTENTS

	Page
LIST OF TABLES	iv
LIST OF FIGURES	v
Chapter	
I. INTRODUCTION	1
1. Problem Definition.....	2
2. Computational Model and Terminology.....	3
3. Voting	6
4. Spot Checking.....	9
5. Credibility-Based Fault Tolerance.....	12
6. Ringers	14
II. GRID RESULT CHECKING	19
1. Result Checking.....	19
2. Grid Result Checking.....	23
3. Combining Result Checking with Other Mechanisms.....	26
4. Incompetent Liars	39
III. IMPLEMENTATION.....	46
1. Timing.....	46
2. The Solver: The Revised Simplex Method.....	46
3. The Checker	47
4. Optimization	48
IV. DETERMINING A CHECKING RATIO	49
1. Motivation.....	49
2. Experiment Design.....	50
3. Hypothesis.....	52
V. RESULTS	55
1. Solver Results	55
2. Checker Results	58
3. Checking Ratio.....	61

VI. DISCUSSION AND CONCLUSIONS	63
1. Discussion.....	63
2. Conclusions.....	64
3. Future Work.....	66
REFERENCES	67

LIST OF TABLES

Table	Page
IV.1 Problem Size and Average Number of Iterations for Linear Programs.....	51
IV.2 Theoretical Checking Time for the Solution of a Linear Program.....	53
V.1 Timing Results for Solving Klee-Minty Problems	56
V.2 Interpolated Average Solution Times for Linear Programs.....	57
V.3 Timing Results for Checking Linear Program Solutions.....	59
V.4 Checking Time with Optimization Turned Down	60

LIST OF FIGURES

Figure	Page
IV.1 Checking Ratio Hypothesis for Linear Programming	54
V.1 Polynomial Fitting of the Solution Times of Klee-Minty Problems.....	56
V.2 Theoretical and Experimental Solution Times for Linear Programs	58
V.3 Theoretical and Experimental Checking Times for Linear Program Solutions....	59
V.4 Checking Time with Optimization Turned Down	60
V.5 Checking Ratio.....	61

CHAPTER I

INTRODUCTION

As the Internet has become faster and more ubiquitous, grid computing has grown in popularity and credibility. Researchers are trading their currently unused computing resources for the opportunity to use others' resources in the future. Businesses are solving problems more cheaply by harvesting unused computing cycles within their organizations. Multitudes of people are contributing to large-scale distributed computing projects such as SETI@Home and Folding@Home [Grid Computing Projects 2004]. Grid computing has great potential to solve very large and complex problems. However, solving these problems using resources that are not locally-owned introduces additional issues such as trust, security, privacy, and quality of service.

Naturally, any time a non-local computing resource is utilized in solving a problem, the result should be viewed with some skepticism. This thesis explores the various mechanisms that may be used to verify the correctness of computational results obtained from untrusted computing resources. First, Section 1 provides a more precise problem definition while Section 2 defines a simple computational model and the underlying assumptions of the model, providing the terminology used throughout this thesis. Finally, Sections 3-6 discuss each mechanism currently used and the accompanying concerns. Section 3 discusses *voting*, the simplest approach. Section 4 describes *spot checking*, a mechanism which verifies a portion of results objectively. Section 5 describes *credibility-based fault tolerance*, an approach that uses historical

information to calculate credibility. Section 6 describes a *ringer* approach that verifies results objectively and is very similar to spot-checking.

1. Problem Definition

When a problem is solved using non-local resources, the results may not be correct even if the computer program to solve the problem is flawless. The results could be incorrect for a number of reasons, one of which is that the program never ran on the non-local resource. Even if the operator of the resource is trusted and has no motivation to falsify results, this does not imply that the computing resource should be trusted, as it may be under an external influence. Computers today are subject to a wide variety of malware, such as viruses and trojan horses, that could adversely impact the results. Additionally, the results may have to travel across a network that contains malicious nodes. Therefore, no result can be deemed correct unless it can be verified using objective criteria. The problem explored in this thesis can be summarized as follows:

How can one ensure the correctness of a computational result obtained from a non-local resource?

Result checking is a means of verifying computational results without recomputing them. This thesis explores several of the traditional approaches used to ensure the correctness of results in grid computing projects and combines result checking with them to produce new variations. The thesis describes these new approaches and compares the efficiency of the new and traditional approaches. Result checking cannot be applied to every type of problem, so the achievement of a generalized approach is

impossible. However, where result checking can be applied, substantial gains in efficiency are expected.

2. Computational Model and Terminology

Computational Model

This thesis defines a computational grid to be any collection of computer resources connected by a network. The finer details, such as the nature of the resource and the network topology, are not addressed in this thesis because they are unlikely to impact the solutions to the problem.

The model of computation assumed in this thesis is a basic pull model [Buyya 2006] employed by many distributed computing projects. Each participant requests a task from the supervisor. After completing the task, the participant sends the result back to the supervisor. The participants are *untrusted*, so the results of a task may be incorrect. Therefore, the supervisor uses any available information to decide, for each result submitted, whether it corresponds to a correct or an incorrect answer for the task. When the supervisor obtains a correct answer for a task, the supervisor may decide that the task has been completed. At any time, the supervisor may *backtrack*, revisiting any decision made regarding answers and task completion. Additionally, the supervisor may choose not to assign a task to a participant. The supervisor may not, however, preempt a participant—all communication is initiated by the participants. The process continues as long as there are tasks to be completed.

Application Models

Application models are defined by the way in which tasks are scheduled and are dependent upon one another. This thesis considers the *independent tasks*, *bag of tasks*, and *workflow* application models [Venugopal et al. 2005]. A task in an independent task model does not depend on the results of another task, and the completion of a task is an end in itself. A bag of tasks application model is similar to an independent task model except that all of the tasks must be executed successfully. Workflows, on the other hand, are sequences of tasks where each task depends on the results of one or more predecessors. A task in the *process-oriented* application model requires finer-grained communication among participants [Venugopal et al. 2005]. Therefore, this thesis does not address process-oriented application models.

Incorrect Results

If a participant submits a result that the supervisor does not accept as the correct answer, the result is called a *lie* in this thesis, and the participant is called a *liar*, regardless of the decision criteria used by the supervisor. The supervisor may use any arbitrary criteria to judge the correctness of an answer, but these criteria should consist primarily of the correctness of results and other objective assessments. The participants should be made aware of these criteria because they form the contractual basis on which the participants and supervisors cooperate. Otherwise, the hidden elements of the decision criteria could erode the trust that the participants have in the supervisor and thereby discourage participation.

Types of Liars

Liars are classified into several groups depending on the motivation for lying or the means employed. This thesis formally introduces the concept of an *incompetent liar*. An incompetent liar is a participant who behaves truthfully but whose results are nevertheless incorrect. Such a liar could be incompetent because of software bugs that are only evident when performing a particular task. Additionally, the class of incompetent liars encompasses truthful participants whose hardware introduces errors or whose communications facilities lack integrity. Incompetent liars have been considered unintentional saboteurs in the past [Germain-Renaud and Playez 2003], but the solution approach elaborated in this thesis requires that this distinction be made.

Participants may lie to the supervisor to maximize some type of reward at a minimal cost to the participant [Du et al. 2004, Golle and Mironov 2001]. Lying in such a way is called rational cheating, or simply cheating. If the participant attempts to achieve this goal by minimizing the cost (i.e. by not performing tasks), then the participant is called a *lazy cheater* [Du and Goodrich 2005]. If, on the other hand, the participant attempts to maximize the reward (i.e. by keeping a prized result secret), then the participant is called a *hoarding cheater* [Du and Goodrich 2005].

Another, often ignored, kind of liar is possible in distributed systems: the *saboteur* [Sarmenta 2002]. Saboteurs intend to introduce falsified results and are not necessarily governed by the laws of rationality. Saboteurs simply may be motivated by a desire to cause mischief. However, saboteurs may intend to cause the project to fail by introducing incorrect results. Such saboteurs may indeed be governed by rationality, but their rationality may include factors external to the project.

3. Voting

Definition

Voting is perhaps the simplest mechanism to improve the security of grid computations. Therefore, voting serves to motivate the other, more complex approaches that exist primarily to address the shortcomings of voting. When a voting scheme is used, the supervisor assigns the same task to some number of participants [Sarmenta 2002]. As these participants submit their results back to the supervisor, the supervisor compares them to see if any subset of the responses forms a quorum of at least some predefined size m . The first m votes of the same result are accepted as the vote's outcome and are assumed to be the correct answer. Here it is important to note the distinction between a *result* and an *answer*. A result is a vote, while an answer is a vote's outcome.

A Simple Example

As an example, assume the supervisor assigns task T to four participants A , B , C , and D . Two of these participants A and B respond with one result R_1 , while C and D respond with a different result R_2 . If a 2-first voting scheme is used, the result accepted by the supervisor depends upon the order that the results are received by the supervisor. If the supervisor receives both votes for R_1 first, R_1 is accepted as the correct answer, and all other answers, including R_2 , are presumed to be incorrect.

Improvements

The supervisor could use the other, presumed-incorrect results to judge confidence in the answer. The supervisor has reason to be confident in the answer if all participants give the correct answer. If, on the other hand, there are two groups of participants providing different answers and separated only by a single participant, then the supervisor may suspect that one of the groups of participants is in *collusion* [Yurkewych et al. 2005], working together secretly to defraud the supervisor. The supervisor could use this information to blacklist the participants or otherwise thwart their attempts to undermine the project.

Additionally, the supervisor may use the other results to aid in debugging an implementation. For example, if all participants in the quorum report that they are operating on hardware from a particular vendor, and all of the participants outside the quorum report that they are using other vendors, then the supervisor could use this information to diagnose a fault in the implementation for that particular vendor. This kind of diagnosis would only be as accurate as the information provided by the participants, which could be arbitrarily incorrect.

Security of Voting

Voting addresses the issues associated with lazy and hoarding cheaters as well as saboteurs [Yurkewych et al. 2005, Sarmenta 2002]. If the supervisor assigns each task to at least m participants that are truthful and not colluding, then lazy cheaters, hoarding cheaters, and saboteurs can be caught using an m -first voting scheme. However, there is an interesting flaw in this assumption. An individual wishing to fool a voting scheme can

emulate a large number of participants, while each truthful individual is represented by a single participant—essentially, liars have more votes. This is a significant problem because the security of majority voting schemes is symmetric and heavily dependent upon the number of truthful participants. The maximum error in an m -first voting scheme follows, given that f is the fraction of colluding participants and m is the number of participants required for a majority [Sarmenta 2002]:

$$\text{Maximum error of voting} = \sum_{j=m}^{2m-1} \binom{2m-1}{j} f^j (1-f)^{2m-1-j}. \quad (\text{I.1})$$

There are, however, some inherent safeguards against lying [Yurkewych et al. 2005]. First, most individuals lack the resources to emulate a multitude of participants convincingly. The individual would require access to multiple computers with different internet connections to make the scheme undetectable. Second, the truthful participants are implicitly cooperating. Many liars would have to collude to compete effectively with the honest participants; otherwise, the liars would be working against each other.

The security of voting schemes can be improved by additional mechanisms which may help prevent collusion [Sarmenta 2002, Yurkewych et al. 2005]. The supervisor can decrease the possibility of successful sabotage by distributing task assignments temporally. For example, if the supervisor assigns a task to a participant, then the supervisor could wait for an answer from the first participant before assigning the same task to another participant. This way, the supervisor forces the participants to decide whether to lie before the participants know whether they can form a quorum and win the vote.

Alternatively, the supervisor could utilize a scheme whereby the tasks are uniquely obfuscated (through randomization or encryption) for each participant [Sarmenta 2002]. This would eliminate the participants' ability to collude because the lying participants would be unable to submit the same results without determining the obfuscation scheme. Meanwhile, the honest participants would submit the same results just by completing the task. The availability and security of this option depend upon the types of problems to be solved on the grid. Simple and obvious obfuscation schemes could offer extremely high security for some problems, while the security of other approaches may depend on the obscurity of the approach. If available, such a scheme would likely eliminate lazy cheaters and may help to deter saboteurs.

Additionally, the incorrect results, as described above, could be used to blacklist participants [Sarmenta 2002]. This is an intuitive approach, but it does require some care to implement properly. It would be wise to bootstrap the scheme by using trusted resources until there are a reasonable number of participants in the system with a long history of truthful behavior. Otherwise, the participants that have been presumed truthful could, in fact, be liars.

4. Spot Checking

Definition

Spot-checking is a scheme in which the supervisor checks a fraction of the results submitted by each participant. Both the supervisor and the participant must compute the results to *spotter tasks* [Sarmenta 2002]. The results of spotter tasks calculated by the

supervisor are compared to those submitted by the participant. If there is any difference between the two sets, the supervisor assumes that the participant is a liar and invalidates all results submitted by the participant. The supervisor may also blacklist the participant or silently ignore the participant's future results. Spot-checking specifically addresses lazy cheaters and saboteurs but ignores hoarding cheaters [Sarmenta 2002].

Security of Spot-Checking

The security of spot-checking lies in the participants' inability to distinguish a spotter task from a regular task because all spotter tasks are regular tasks. The set of spotter tasks may represent a fixed fraction of tasks assigned to that participant.

Alternatively the supervisor's decision to check a result may be modeled as a Bernoulli process or may be based on other assessments, such as credibility [Sarmenta 2002].

Some schemes may decide based on the time it takes for a participant to complete a task, whether to spot-check the task. The supervisor should not substitute such timing-based spot checks for regular checks but should instead supplement the regular checks.

Additionally, the supervisor should not complete tasks to verify results when the time taken by the participant is impossibly small because doing so could consume vast amounts of the supervisor's resources. Instead, the supervisor should consider the task incomplete and should consider the participant a lazy cheater.

Given that blacklisting can be enforced, f is the fraction of cheating participants, and k is the number of spot checks passed by each participant, the maximum error rate is bounded as follows [Sarmenta 2002]:

$$\text{Maximum error of spot checking} < \frac{f}{1-f} * \frac{1}{k * e}. \quad (\text{I.2})$$

Example Application

An interesting application of spot-checking uses a commitment-based sampling scheme to perform spot-checking when the size of a result is very large [Du et al. 2004]. This approach represents the result of a single task as a very large number of subtask results. The supervisor is only interested in certain of these results, so participants only submit a small portion of the results. The supervisor wishes to know whether the participant actually calculated all of the results.

The commitment-based sampling (CBS) scheme [Du et al. 2004] works as follows. 1) The participant builds a *Merkle Tree* containing the results. Merkle Trees are binary trees where the value at each leaf is a result, and the value of a non-leaf node is the hash value of the concatenation of its children's values. Therefore, to commit 2^n results, the participant need only submit the root of the Merkle Tree. 2) The supervisor then chooses a number of samples from the set of results and asks the participant to submit these results. The participant also submits the hash value of every vertex's sibling from the result (a leaf) to the root of the tree. 3) The supervisor then determines the hash value of the Merkle Tree containing this leaf and related hash values and compares this hash value with the previously-submitted root of the Merkle Tree. If all of these values match, then the participant is likely to be honest. The probability that cheating succeeds when the rate of cheating is $P(\text{cheat})$, the probability that the participant can guess a result is $P(\text{guess})$, and the number of samples per task is γ , is given by Equation I.3 from Du et al.

[2004]. Multiplying this equation by f , the fraction of lying participants, yields Equation I.4, the average maximum error of the CBS scheme.

$$P(\text{cheating succeeds}) = [1 - P(\text{lie}) + P(\text{cheat}) * P(\text{guess})]^{\gamma}. \quad (\text{I.3})$$

$$\text{Maximum error of CBS} = f * [1 - P(\text{lie}) + P(\text{cheat}) * P(\text{guess})]^{\gamma}. \quad (\text{I.4})$$

The security of commitment-based sampling lies in an assumption that the participants are rational. The scheme relies on the security of iterative hash functions, such as MD5 [Rivest 1992] and SHA-1 [Eastlake and Jones 2001], whose security has fallen under serious doubt because of recently-developed collision attacks [Wang et al. 2004, Lenstra 2005]. Therefore, the scheme is not secure against saboteurs because unlike rational cheaters, they could complete tasks and replace the results with results that provide the same hash.

5. Credibility-Based Fault Tolerance

Definition

Credibility-based fault tolerance is a very useful approach for securing grid computations because of its generality. The approach, described in [Sarmenta 2002], combines voting and spot-checking in a hybrid approach that automatically trades performance for correctness. The idea is relatively simple and clever. The supervisor determines each participant's credibility. When a participant submits a result, the

supervisor accepts it only if the probability of its correctness is above the threshold of acceptable error. Since the supervisor only accepts such results, the fraction of correct results should be above the threshold of acceptable error.

Security

The complexity of credibility-based fault tolerance lies in determining the credibility of a participant [Sarmenta 2002]. Each participant is assigned an initial credibility. When a participant submits a result, the supervisor determines the result's credibility based on the participant's credibility. If the credibility of the result is low, the supervisor takes actions to improve it. These actions are determined dynamically but can range from pure voting to pure spot-checking. The supervisor may attempt to improve a participant's credibility by spot-checking results from the participant, thereby improving the credibility of the current result.

The supervisor may request a vote from other high-credibility participants to improve a result's credibility. The supervisor ensures that the vote's outcome is likely to be correct by requiring the voting participants to have high credibility. Therefore, the supervisor may assume with high confidence that any voters who give incorrect results are liars. The supervisor may blacklist these participants as if they failed a spot check.

Problems

Sarmenta [2002] proposes a credibility-based scheme that uses voting to spot-check results. However, this approach is effective only if the participants are not allowed to adapt, and this is not a reasonable assumption to make. If participants can guess what

their credibility is, they can lie and collude only when their credibility is high. Since credibilities are monotonic in Sarmenta's scheme, participants know that their credibilities are high after submitting several correct results. Participants could start lying after achieving a high credibility. Therefore, the supervisor should continue normal spot-checking of high-credibility participants. High-credibility participants would be spot-checked automatically within Sarmenta's scheme if each participant's credibility decreases over time. The security of this particular credibility scheme is equivalent to voting when the participants adapt and the supervisor does not spot check high-credibility participants.

6. Ringers

Definition

Ringer schemes [Golle and Mironov 2001, Du and Goodrich 2005, Szajda et al. 2003] can be used in the same way that spot-checking schemes are used. A ringer task is a task that the supervisor assigns to a participant after the supervisor knows the results. Ringer tasks differ from spotter tasks in that the results of a ringer task are often known by the participant because of the way in which the problem is constructed. Ringer schemes are particularly effective when the ringer task requires significantly less time to create than to solve.

Simple Example

As an example, graph isomorphism [Du and Goodrich 2005] problems are easy to construct and relatively difficult to solve. A graph G is isomorphic to a graph H if any permutation of G 's vertices is equivalent to the graph H . The supervisor can easily construct a graph G and a graph H that is isomorphic to G . Likewise, the supervisor can easily construct two graphs that are not isomorphic. The supervisor knows whether the graphs are isomorphic by the construction of the problem, but the participant must complete the task. This type of ringer scheme must be specialized for each type of task to be solved, and it may be impossible to construct a ringer task in this manner for a particular problem type.

Hoarding Cheaters

Ringers can also be used when searching for high-value rare events [Golle and Mironov 2001]. Ringer schemes in this context are designed specifically to address the issue of hoarding cheaters. Each participant in Golle and Mironov's [2001] scheme is given a function to evaluate, an input range, and a screener function that determines which inputs map to outputs that are valuable. The task's results are comprised of those inputs within the given range that map to an output flagged by the screener as a high-value rare event. The screener or input may be modified such that the set of all such inputs found by the participants are a superset of the actual high-value rare events. The high-value events that are present in the results but are not the true high-value rare events are commonly called ringers. The supervisor reduces the perceived rarity of high-value events by introducing additional high-value events, thereby making the results more

difficult to guess. Since the participants cannot distinguish true high-value events from ringers, they must submit all high-value events. When the supervisor receives these results, the supervisor determines whether the participants submitted all of the ringers. Each ringer is analogous to a spot check, and the supervisor identifies a participant as a lazy or hoarding cheater if a ringer is missing.

Specializations

The basic ringer scheme has been specialized in a variety of ways and has been applied to several problems [Du and Goodrich 2005, Szajda et al. 2003]. Du and Goodrich [2005], for example, describe a scheme called chaff injection that is very much like the scheme described above. Input chaff injection refers to the obfuscation of high-value rare events by assigning tasks whose inputs are known to contain a set of rare events. Output chaff injection (also called criterion expansion) is a similar process, except that the extraneous rare events are created by the function or the screener. The resulting output in either case is a set of wheat (the actual rare events) and chaff (the extraneous rare events), and the participant cannot differentiate the wheat from the chaff. The approach taken by Du and Goodrich [2005] is different from that taken by Golle and Mironov [2001] because Du and Goodrich give the same expanded criterion to all participants, rendering collusion ineffective.

Security of Ringer Schemes

The security of a ringer scheme lies in the participants' inability to distinguish a ringer from a regular task or result [Golle and Mironov 2001]. If participants could

distinguish between regular tasks and ringers, then they could solve only the ringer tasks and lie about all other tasks. Therefore, the ringer tasks should be structured in the same manner as regular tasks and should require the same amount of time. Likewise, if participants could distinguish between regular results and ringer results in a search for high-value rare events, then the participants could report only the ringers and could hoard the high-value rare events.

The analysis of ringer schemes' security is identical to that of spot checking (Equation I.2) except that k represents the number of ringers returned or ringer tasks completed by each participant. The primary difference between spot checking and ringer schemes is when and the means by which the supervisor becomes aware of the answer to a ringer or spotter task. A supervisor in a spot checking scheme discovers the answer by recomputing tasks after participants have completed them. A supervisor in a ringer scheme knows the answer, by the construction of the problem, before assigning the task to the participant.

This chapter has defined the problem explored in this thesis and has described many viable existing solutions: voting, spot checking, credibility-based fault tolerance, and ringers. The chapter has discussed the specific problem domains and cheaters addressed by each solution. Additionally, the security of each approach has been qualified through discussion and quantified with theory where possible.

Chapter II discusses result checking and how it can be applied to the existing approaches described in this chapter to produce several new approaches. Additionally, Chapter II proves the efficiency of the existing and new approaches as well as the

conditions under which the new schemes are more efficient than the old. Chapter III discusses the implementation of the system used in the experiments. Chapter IV outlines an experiment to determine whether the approaches in Chapter II can ever be expected to be more efficient than those in Chapter I. Specifically, Chapter IV develops a way to determine the *checking ratio* of a particular implementation. Chapter V provides the results of this experiment. Chapter VI discusses the experimental results, draws conclusions from these results, and discusses possible directions of future work.

CHAPTER II

GRID RESULT CHECKING

This chapter provides an example-driven introduction to the theory of result checking, followed by a discussion about the use of result checking to ensure correctness in computational grids. Section 3 describes the efficiency metric used in this thesis, describes how new solution approaches can be formulated from existing schemes by applying result checking, and analyzes the efficiency of these new schemes compared to the efficiency of existing schemes. Finally, Section 4 discusses incompetent liars, the problems they pose, and how to address these issues.

1. Result Checking

Definition

Result checking has long been a part of complexity theory. The class of problems known as NP is defined by the existence of a checker for instances of these problems where the answer is *yes*. However, a problem that is in the class NP may not be a member of the class co-NP, which is defined by the existence of a checker for instances of these problems where the answer is *no*. Even when a problem is in NP and co-NP, a checker for the problem may not be a simple checker. Particularly, there is no known NP-complete problem which is a member of co-NP, as the existence of such a problem would imply that NP equals co-NP.

Run-time result checking is the process of determining whether a program's output is correct for a particular input. The process of checking certain problems takes asymptotically less time than solving the problems [Wasserman and Blum 1997]. Additionally, such checkers are often much simpler than the solutions and are therefore called simple checkers. Wasserman and Blum formalize the concept of a simple checker and in doing so require that the checker takes asymptotically less time than the solution algorithm [1997]. This requirement heuristically means that the checker cannot perform the same actions as the solution algorithm.

Checkers tend to have fewer bugs than the programs they check [Wasserman and Blum 1997]. First, the translation of an algorithm into an implementation is not always flawless. Bugs may be introduced at any stage of the process, especially where the algorithm is complex or where multiple developers are involved. Solutions that rely on operating system facilities, such as disk access and threading, may also be susceptible to race conditions. Second, solutions often employ numerically unstable algorithms and rely upon floating-point approximations to real or rational numbers. The underlying algorithms may have been proven correct for the set of infinite-precision numbers, but the introduction of fixed-bit approximations may introduce errors [Wasserman and Blum 1997]. Finally, checkers tend to be simpler than the programs they check, minimizing the need for multiple threads and multiple developers. Therefore, the likelihood of introducing a bug into the implementation is significantly reduced. Many checkers are indeed so simple that they can be verified formally and presumed to be entirely bug-free [Mehlhorn 2006].

Simple Example

As an example, comparison sorting algorithms can be complex and have asymptotic running times that are $\Omega(n \log n)$, where n is the size of the list. Verifying that the list has been sorted properly is simple and can be performed in $O(n)$ time [Wasserman and Blum 1997].

Complex Example

Solving linear programs is often a very complex process, the output of which is remarkably easy to check [Wright 1997]. Consider a linear program and its dual, of the following form:

Primal :

$$\begin{aligned} & \text{maximize} && c^T x \\ & \text{subject to} && Ax \leq b \\ & && x \geq 0. \end{aligned}$$

Dual :

$$\begin{aligned} & \text{minimize} && b^T y \\ & \text{subject to} && A^T y \geq c \\ & && y \geq 0. \end{aligned}$$

$$\begin{aligned} & \text{where } c, x \in \mathfrak{R}^n \\ & \quad b, y \in \mathfrak{R}^m \\ & \quad A \in \mathfrak{R}^{m \times n}. \end{aligned}$$

Any number of algorithms could be used to solve this problem: a variant of the simplex method or an interior-point algorithm, for example. If the chosen algorithm yields the solution vectors x and y , then checking the answer is trivial:

- verify that $x \geq 0$.
- verify that $y \geq 0$.
- verify that $Ax \leq b$.
- verify that $A^T y \geq c$.
- verify that $c^T x = b^T y$.

The vector x is the solution to the primal problem, the problem of interest, and y serves as a *certificate of optimality* for x [Wright 1997, Chvátal 1983]. If all of the above necessary conditions hold, then x is indeed the optimal solution to the primal linear program. While entire books have been dedicated to solving linear programs efficiently, the checker remains the same [Wright 1997, Chvátal 1983].

Additional complexity arises if the primal linear program can be unbounded or infeasible. Either of these cases requires the solution algorithm to provide a certificate that proves that the primal linear program is unbounded or infeasible. Such a certificate could be the result of solving the homogeneous self-dual embedding of the primal [Wright 1997]. Alternatively, if the linear program is known by its construction to be feasible and bounded, none of this additional complexity is necessary.

Generalizations

Problems with duals tend to be simple to check, with the notable exception of Monte Carlo algorithms [Mehlhorn 2006]. Examples include such problems as matchings and vertex covers, maximum flows and minimum cuts, and shortest paths and potential functions [Mehlhorn 2006]. Additional problems that have proven easy to check include graph characterization problems where a graph can be classified by the presence of a forbidden subgraph [Mehlhorn 2006]. Such graph characterization

problems include recognition algorithms for interval and comparability graphs [Kratsch et al. 2003].

Certifying Algorithms

Result checking has been refined, formalized and applied to many problems in graph theory [Kratsch et al. 2003, Mehlhorn 2006]. A *certifying algorithm* is an algorithm that provides a *certificate* (or *witness*) for each answer it produces [Kratsch et al. 2003]. This certificate provides evidence that the answer is correct. Therefore, a checker evaluates the answer and the certificate in the context of the original problem to determine the correctness of the answer without solving the original problem.

2. Grid Result Checking

Motivation

Result-checking has been addressed thoroughly in the context of debugging programs [Blum 1988, Blum and Kannan 1989, Wasserman and Blum 1997, Mehlhorn 2006]. However, the application of result-checking to grid computing has received less treatment [Germain-Renaud and Monnier-Ragaine 2005, Golle and Mironov 2001]. Golle and Mironov [2001] dismiss result-checking as an applicable approach for securing grid computations by merely stating that it is unknown how to check the results of every program. This is an undisputable fact, but it is certainly not a reason to ignore the approach, especially given that researchers are uncovering new methods of result-checking all the time [Kratsch et al. 2003]. The fact that many programs cannot be

checked has likely formed the motivation for developing many of the techniques currently employed to ensure security in grid applications.

Benefits

Result-checking via simple checkers confers a multitude of benefits. Simple checkers can be used to verify the correctness of a result without solving the problem itself. Moreover, this correctness can be checked more simply and efficiently than the solution can be found. Perhaps most importantly, this simplicity permits simple checkers to achieve complete confidence in a result, not just probabilistic confidence. This confidence is of particular importance in workflow applications, where future problem definitions depend upon current or past results. Any single incorrect result in a workflow could compromise future results and cause the entire computation to fail. Additionally, simple checkers, as a side effect in this scheme, may discover bugs in an implementation that other schemes would never find.

Caveats

The primary argument against using simple checkers for securing grid computations is the lack of generality [Germain-Renaud and Monnier-Ragainne 2005, Golle and Mironov 2001]: the approach can only be used when solving certain kinds of problems. However, result checking is not the only approach that lacks generality. Other approaches may be specifically targeted toward a particular class of problems, such as distributed search [Du and Goodrich 2005]. Additionally, some approaches rely on assumptions about the behavior of the participants [Sarmenta 2002] or the number of

problems to be solved [Du et al. 2004]. Another potential weakness of result-checking arises from its limited applicability: the approach mandates that any distributed task is checkable, which may counteract schemes to improve efficiency and *scalability* through information sharing or other means. Scalability is a measure of how well a solution performs as the size of a problem grows. The use of result-checking is therefore limited to *independent tasks*, *Bag-of-Tasks*, and *workflow* application models. Result-checking is ill-suited for use in tightly coupled applications, but these applications are typically run using tightly-coupled hardware instead of a grid.

Security of Result Checking

Depending on the problem, the solution algorithm, and the method of checking, an answer may have some degree of uncertainty associated with it [Wasserman and Blum 1997]. However, many checkers are able to verify an answer in such a way that the uncertainty of the answer is equal to the likelihood that there is a bug in the checker. Moreover, since checkers are by definition simple, the checkers should be easy to verify formally [Mehlhorn 2006]. Therefore, it is reasonable to expect that the security of a checker could be perfect.

Perhaps the most appealing attribute of grid result checking is that its security does not have the typical dependencies of other approaches. Instead, the security of grid result checking depends on the type of problem and the type of checker, meaning that the verification of results is likely to be more objective (see Section I.2). Additionally, the correctness of results does not depend entirely on the correctness of the solution's implementation, as it does with other approaches.

Additional types of result checking have been defined that have varying goals and levels of security. This thesis focuses on certifying algorithms as they are often very simple, and their security is often perfect, simplifying analysis. Other approaches, such as probabilistically checkable proofs [Aiello et al. 2000] and property testing [Goldreich 1997], verify results probabilistically. Absolute, objective correctness is crucial in workflow applications, where future tasks are formulated based on past results. Any single incorrect result in a workflow application model has the potential to undermine the entire solution.

3. Combining Result Checking with Other Mechanisms

Efficiency

Result checking can be combined with other mechanisms to achieve improvements in efficiency when certain conditions are met. Determining these conditions requires the development of an appropriate efficiency measure so that schemes may be compared. The efficiency of a scheme is calculated as the ratio of two times: the time required to complete n tasks on a trusted local resource; and the time required to complete the same task in the scheme. Equivalently, the efficiency of a scheme can be described as the amount of non-redundant work divided by the amount of work that will be performed. This thesis assumes that each participant has the same computational capacity and that each task requires $t(\text{solve})$ time units to solve. Likewise, results require $t(\text{check})$ time units to check. $P(\text{lie})$ is the Bernoulli probability that a participant lies

given that the participant is one of the lying participants. The fraction of lying participants is denoted f .

This efficiency measure overestimates actual efficiency because it does not account for communication costs or the redundancy required to guarantee quality of service [Kenyon and Cheliotis 2003]. Communication costs would be difficult to include without constraining the network topology and infrastructure. Furthermore, communication time should be negligible compared to the time spent completing a task; otherwise, the problem is likely not well-suited to solve on a grid. This efficiency measure is slightly biased in favor of schemes that include result checking because the measure does not account for the additional cost of transferring a certificate along with each result. However, this cost could be added to the checking time to remove this bias.

Additionally, the efficiency measure overestimates the actual efficiency of schemes that include result checking because it does not account for the time required to create certificates. Including the certificate creation time in the solution time would further complicate analysis and would make the results less intuitive. Therefore, the certificate creation time should be included in the checking time, resulting in an underestimate of the efficiency relative to other schemes.

Quality of service is not addressed in this thesis, but it deserves attention. Tasks with quality of service constraints require either stable participants or redundancy. None of the mechanisms presented in this thesis are able to enforce deadlines on participants' submission of results. Rational participants could be compelled to submit results quickly by offering a time-based reward. However, this would not prevent saboteurs from requesting a task and never completing it.

Voting

Voting with result checking is analogous to a peer review process. Peer review is a technique employed by many publishers to determine which literature is worthy of publishing. Many people submit work to the reviewers, and if the reviewers duplicated the work, it may take decades for anything to get published. Instead, the reviewers evaluate others' work to determine whether they meet certain criteria. If one piece of work does not meet the criteria, the reviewers evaluate the next piece of work.

The most intuitive approach to peer review in a computational grid would proceed as follows: the supervisor assigns a task to one participant. After receiving the result, the supervisor asks a group of participants to verify the results by giving an answer of either *correct* or *incorrect*.

This simple approach has a number of flaws but provides a good starting point for further discussion. True peer review processes use trusted reviewers, but this scheme employs reviewers that may be less trustworthy than submitters. Also, the reviewers know what the acceptable votes are, so they may vote without performing any work. Furthermore, if blacklisting is enforced, then the participants present are presumably truthful most of the time, since blatant liars would be caught and blacklisted. Therefore, reviewers have the motive, means, and opportunity to guess that every result is correct.

Alternatively, the supervisor may decrease the likelihood of lying by asking the participants for computational results instead of binary votes. For example, the two matrix-vector multiplications form the majority of the work to check the answer of a linear program. The supervisor could ask participants to vote by computing Ax and $A^T y$ without knowing which is Ax and without knowing b or c . The supervisor would then

have to perform two vector-vector multiplications and five vector-vector comparisons to transform the outcome of the vote into an assessment of the result's correctness. If the result is deemed incorrect, the original task of solving the linear program is assigned to a different participant.

The efficiency of m -first voting is given by Equation II.1, where m is the required quorum size, which is always an integer greater than or equal to two. Therefore, the maximum efficiency of m -first voting is 50%. This coarse granularity is one of the largest problems with voting: it cannot be fine-tuned because the number of votes is an integer. The appeal of m -first voting, however, lies in where this inefficiency exists: in the participants. Voting requires only coordination and vote-counting by the supervisor, making the approach extremely scalable. Additionally, voting is very cost-effective because participants' resources far outnumber those of the supervisor and are typically cheaper to use.

$$\begin{aligned} \text{Efficiency of } m\text{-first voting} &= \frac{n * t(\text{solve})}{m * n * t(\text{solve})} \\ &= \frac{1}{m}. \end{aligned} \tag{II.1}$$

The security of r -first peer review is equal to that of m -first voting when $r = m - 1$, and the efficiency of r -first peer review is given by Equation II.2, where r is the number of reviewers and α , the *checking ratio*, is the ratio of checking time to solving time for a task. The efficiency measure must be augmented to capture the inefficiency that occurs in the peer review scheme when a result is incorrect. Therefore, Equation II.2 includes f , the fraction of lying and colluding participants. Like the security analysis of voting in

Chapter I, this analysis assumes that the lying participants always lie. Therefore $P(\text{lie})$, the probability that participants lie, is not included in Equation II.2 because it is always one.

$$\begin{aligned}
 \text{Efficiency of } r\text{-first peer review} &= \frac{n * t(\text{solve})}{n * t(\text{solve}) + \frac{n}{1-f} * r * t(\text{check})} \\
 &= \frac{1}{1 + \frac{1}{1-f} * r * \frac{t(\text{check})}{t(\text{solve})}} \\
 \text{Introduce } \alpha &= \frac{t(\text{check})}{t(\text{solve})} \\
 \text{Efficiency of } r\text{-first peer review} &= \frac{1}{1 + \frac{\alpha * r}{1-f}} \\
 &= \frac{1}{\frac{1-f + \alpha * r}{1-f}} \\
 &= \frac{1-f}{1-f + \alpha * r}.
 \end{aligned} \tag{II.2}$$

Theorem 1 describes the conditions under which the peer review scheme is more efficient than voting. Practically, the minimum α required for peer review to perform better than voting is determined when f is one-half. Neither voting nor peer review should be used when f is as high as one-half because answers are as likely to be wrong as they are to be correct. Therefore, peer review can be expected to be more efficient than voting when checking requires less than one-half of the time solving requires.

Theorem 1. r -first peer review is more efficient than m -first peer voting when $r = m - 1$ if and only if $\alpha < (1 - f)$.

Proof. Set $\frac{1-f}{1-f+\alpha*r} > \frac{1}{m}$ and solve for α :

$$\frac{1-f+\alpha*(m-1)}{1-f} < m$$

$$\frac{1-f+\alpha*(m-1)}{1-f} < \frac{1-f}{1-f} * m$$

$$1-f+\alpha*(m-1) < (1-f)*m$$

$$\alpha*(m-1) < (1-f)*m - (1-f)$$

$$\alpha < \frac{(1-f)*m - (1-f)}{m-1}$$

$$\alpha < \frac{m-1}{m-1} * (1-f)$$

$$\alpha < (1-f). \quad \text{QED.}$$

Theorem 1 makes sense intuitively. r reviewers in a peer review scheme must check an average of $1/(1-f)$ results for every task completed in a voting scheme because reviewers must check the lies submitted by participants. Both schemes must complete the task at least once, and this completion counts as a vote.

Spot Checking

Result checking can be applied to a spot checking approach very easily: the supervisor simply checks a fraction of results instead of recomputing them. If certifying algorithms are used, then the security of the two approaches is equivalent. However, the application of certifying algorithms yields improvements in efficiency if checking the result requires less time than recomputing the result. Furthermore, improvements in the efficiency of the scheme can be traded easily for improvements in security because both

are determined by $P(\text{check})$, the probability that each result is spot checked. When certifying algorithms are used to spot check results, the approach will be called *spot certification*.

The efficiency of spot checking is given by Equation II.3. Intuitively, this equation states that the efficiency of spot checking is determined by the fraction of correct results received relative to the probability that results are checked. The efficiency of spot certification is given by Equation II.4, where again α is the ratio of checking time to solving time for a task.

$$\begin{aligned}
 \text{Efficiency of spot checking} &= \frac{n * t(\text{solve})}{n * t(\text{solve}) + \frac{n}{1 - P(\text{lie}) * f} * P(\text{check}) * t(\text{solve})} \\
 &= \frac{1}{1 + \frac{P(\text{check})}{1 - P(\text{lie}) * f}} \\
 &= \frac{1}{1 - P(\text{lie}) * f + P(\text{check})} \\
 &= \frac{1 - P(\text{lie}) * f}{1 - P(\text{lie}) * f + P(\text{check})}.
 \end{aligned} \tag{II.3}$$

$$\begin{aligned}
\text{Efficiency of spot certification} &= \frac{n * t(\text{solve})}{n * t(\text{solve}) + \frac{n}{1 - P(\text{lie}) * f} * P(\text{check}) * t(\text{check})} \\
&= \frac{1}{1 + \frac{1}{1 - P(\text{lie}) * f} * P(\text{check}) * \frac{t(\text{check})}{t(\text{solve})}} \\
\text{Introduce } \alpha &= \frac{t(\text{check})}{t(\text{solve})} \\
\text{Efficiency of spot certification} &= \frac{1}{1 + \frac{\alpha * P(\text{check})}{1 - P(\text{lie}) * f}} \\
&= \frac{1}{1 - P(\text{lie}) * f + \alpha * P(\text{check})} \\
&= \frac{1 - P(\text{lie}) * f}{1 - P(\text{lie}) * f + \alpha * P(\text{check})}.
\end{aligned} \tag{II.4}$$

Theorem 2 describes the conditions under which spot certification is more efficient than spot checking. This condition is also very intuitive—if checking a result takes longer than completing the task, it makes more sense to complete the task. Spot certification is particularly interesting if α is very small. Consider the following example. Spot checking is used with a $P(\text{check})$ of one-tenth, and spot certification is used with a $P(\text{check})$ and an α of one-tenth. The security of the two schemes is the same, but the efficiency of spot checking is at most 91% while the efficiency of spot certification is at most 99%. This efficiency gain could be traded for additional security by increasing $P(\text{check})$. If $P(\text{check})$ is set to one, then the efficiency of the schemes is the same, but the security is drastically different—the supervisor in the spot checking scheme only checks ten percent of the results, but the supervisor in the spot certification scheme verifies every result. This means that a workflow application could be guaranteed to complete

successfully using spot certification but that such guarantees could not be made with spot checking.

Theorem 2. Spot certification is more efficient than spot checking if and only if $\alpha < 1$.

Proof. Set $\frac{1 - P(\text{lie}) * f}{1 - P(\text{lie}) * f + \alpha * P(\text{check})} > \frac{1 - P(\text{lie}) * f}{1 - P(\text{lie}) * f + P(\text{check})}$ and solve for α :

$$1 - P(\text{lie}) * f + \alpha * P(\text{check}) < 1 - P(\text{lie}) * f + P(\text{check})$$

$$\alpha * P(\text{check}) < P(\text{check})$$

$$\alpha < 1. \quad \text{QED.}$$

Credibility-Based Fault Tolerance

Credibility-based fault tolerance is a scheme that combines voting and spot checking to achieve a finely tuned balance between security and efficiency. However, the complexity of the scheme deters attempts to analyze it by any method other than simulation. Furthermore, Sarmenta [2002] assumes that participants are not adaptive. It has yet to be proven whether credibility-based fault tolerance is more secure or efficient than pure spot checking or voting schemes under realistic circumstances.

However, if credibility-based fault tolerance can be applied effectively without restricting the behavior of participants, the scheme can benefit from result checking because credibility-based fault tolerance relies on voting and spot checking. Spot checking could be replaced by spot certification, and the improvements in efficiency imply that results may be checked more often, leading to faster credibility increases. Additionally, the supervisor could use peer review instead of voting when the condition specified by Theorem 1 is met. Aggregate credibility information could be used to

approximate f , the fraction of lying participants, and the checking ratio α can be derived theoretically or approximated through experimentation.

Ringers

Ringers can be incorporated into a peer review scheme because the supervisor is asking the reviewers to solve decision problems. This is analogous to asking participants to determine whether an output is valuable by applying a screener function. The supervisor may decrease the likelihood of lying by asking participants to check ringer results in addition to the actual results. As with any ringer scheme, the ringers would have to be indistinguishable from actual results and would have to be constructed in such a way that the supervisor knows the answer. Ringers could be used with a peer review scheme when the verification criteria are very complex and cannot be easily separated from a decision problem. The same level of security could be obtained by using spot checking in this manner.

The efficiency of a basic ringer approach is given by Equation II.5. $t(\text{create})$ represents the time required to create a ringer, $P(\text{check})$ is the probability that a task is a ringer, and the creation ratio β is the ratio of creation time to solving time for a ringer task.

$$\begin{aligned}
\text{Efficiency of ringers} &= \frac{n * t(\text{solve})}{\frac{n * t(\text{solve})}{1 - P(\text{check})} + \frac{n * P(\text{check}) * t(\text{create})}{(1 - P(\text{lie}) * f) * (1 - P(\text{check}))}} \\
&= \frac{1}{\frac{1}{1 - P(\text{check})} + \frac{P(\text{check})}{(1 - P(\text{lie}) * f) * (1 - P(\text{check}))} * \frac{t(\text{create})}{t(\text{solve})}} \\
\text{Introduce } \beta &= \frac{t(\text{create})}{t(\text{solve})} \\
&= \frac{1}{\frac{1}{1 - P(\text{check})} + \frac{\beta * P(\text{check})}{(1 - P(\text{lie}) * f) * (1 - P(\text{check}))}} \\
&= \frac{1}{\frac{1 - P(\text{lie}) * f + \beta * P(\text{check})}{(1 - P(\text{lie}) * f) * (1 - P(\text{check}))}} \\
&= \frac{(1 - P(\text{lie}) * f) * (1 - P(\text{check}))}{1 - P(\text{lie}) * f + \beta * P(\text{check})}.
\end{aligned} \tag{II.5}$$

Theorem 3 describes the conditions under which using ringers is more efficient than spot checking. This theorem shows that spot checking may be more efficient than using ringers even when the creation of ringer tasks is negligible.

Theorem 3. Ringer schemes are more efficient than spot checking if and only if $\beta < P(\text{lie}) * f - P(\text{check})$.

Proof. Set $\frac{(1-P(\text{lie}) * f) * (1-P(\text{check}))}{1-P(\text{lie}) * f + \beta * P(\text{check})} > \frac{1-P(\text{lie}) * f}{1-P(\text{lie}) * f + P(\text{check})}$ and solve for β :

$$\frac{1-P(\text{lie}) * f + \beta * P(\text{check})}{(1-P(\text{check}))} < 1-P(\text{lie}) * f + P(\text{check})$$

$$1-P(\text{lie}) * f + \beta * P(\text{check}) < (1-P(\text{lie}) * f + P(\text{check})) * (1-P(\text{check}))$$

$$\beta * P(\text{check}) < (1-P(\text{lie}) * f + P(\text{check})) * (1-P(\text{check})) - (1-P(\text{lie}) * f)$$

$$\beta < \frac{(1-P(\text{lie}) * f + P(\text{check})) * (1-P(\text{check})) - (1-P(\text{lie}) * f)}{P(\text{check})}$$

$$\beta < \frac{(1-P(\text{lie}) * f) * (-P(\text{check})) + P(\text{check}) * (1-P(\text{check}))}{P(\text{check})}$$

$$\beta < P(\text{lie}) * f - 1 + 1 - P(\text{check})$$

$$\beta < P(\text{lie}) * f - P(\text{check}). \quad \text{QED.}$$

The maximum error of a peer review scheme augmented with ringers is given by Equation II.6, and its efficiency is given by Equation II.7, where k is the number of ringers correctly detected by each participant. The same level of security may be obtained by using spot certification to verify that the outcome of a peer review process is correct. Equation II.6 was derived much like II.3, by multiplying Equation I.2 by the probability that a participant lies when submitting a result that is later reviewed.

$$\text{Maximum error of ringer peer review} < \frac{f}{1-f} * \frac{\sum_{j=m}^{2m-1} \binom{2m-1}{j} f^j (1-f)^{2m-1-j}}{k * e}. \quad \text{(II.6)}$$

$$\begin{aligned}
\text{Efficiency of ringer peer review} &= \frac{n * t(\text{solve})}{n * t(\text{solve}) + \frac{n * m * t(\text{check}) + n * t(\text{create})}{(1 - P(\text{lie}) * f) * (1 - P(\text{check}))}} \\
&= \frac{1}{1 + \frac{m * \frac{t(\text{check})}{t(\text{solve})} + \frac{t(\text{create})}{t(\text{solve})}}{(1 - P(\text{lie}) * f) * (1 - P(\text{check}))}} \\
\text{Introduce } \alpha &= \frac{t(\text{check})}{t(\text{solve})} \\
\text{Introduce } \beta &= \frac{t(\text{create})}{t(\text{solve})} \\
&= \frac{1}{1 + \frac{m * \alpha + \beta}{(1 - P(\text{lie}) * f) * (1 - P(\text{check}))}} \\
&= \frac{(1 - P(\text{lie}) * f) * (1 - P(\text{check}))}{(1 - P(\text{lie}) * f) * (1 - P(\text{check})) + m * \alpha + \beta}
\end{aligned} \tag{II.7}$$

4. Incompetent Liars

Incorrect Results

An incompetent liar is a participant who behaves truthfully but whose results are nevertheless incorrect. There are many reasons why a supervisor could receive an incorrect result from a participant. The accuracy of a result seen by the supervisor generally depends on the participant's software, hardware, computing platform, and honesty, combined with the communication infrastructure between the participant and the supervisor. Therefore, an incorrect result implies that there is a software bug, there is a platform or hardware problem, the participant is lying, or some communication was corrupted in transit:

$$\begin{aligned} (\text{Incorrect Result}) \Rightarrow & (\text{Bug in Software}) \vee (\text{Hardware/Platform Problem}) \vee \\ & (\text{Lying Participant}) \vee (\text{Corrupted Communication}). \end{aligned} \quad \text{(II.8)}$$

Communication is typically secure because grid computing platforms can use some form of reliable protocol with encryption and checksums for communication between the participant and the supervisor. Hardware and platform problems often cause the participant to crash completely, but some hardware issues do introduce errors into computations [Kahney 2001]. While hardware or communication problems may cause a truthful participant to submit an incorrect result, these problems form an integral part of the participant's behavior. Furthermore, each participant's hardware and communication facilities may be different and are likely to be independent, and the supervisor cannot control these factors. Therefore, participants with faulty hardware should be treated the

same as liars because the results are equally valueless due to factors outside the control of the supervisor.

This leaves only two likely and relevant reasons why a result could be incorrect: either there is a software bug, or the participant is lying. If the supervisor distributes software that contains bugs, these bugs are present system-wide: no matter which participant completes a task, the result will be incorrect. The ability to detect software bugs is a side effect of approaches that employ result checking, but the issue *must* be addressed. If an approach employs result checking but does not attempt to detect software bugs, the supervisor will presume that every participant is lying.

Only one of the approaches described in Chapter I can be modified to detect software bugs: ringers. When a basic ringer approach is used, the supervisor knows a task's answer before assigning the task to a participant. The supervisor knows whether the participant submits an incorrect result, but the means of verifying the result are separate from the completion of the task. Therefore, the supervisor could detect software bugs by recomputing some fraction of the incorrect results submitted.

Likewise, when input chaff injection is used, the supervisor injects inputs which should correspond to particular outputs. A software bug could prevent all of the participants from detecting these ringers and submitting them to the supervisor. One could avoid having to detect software bugs by testing inputs prior to injection to verify that the software properly detects the ringers. However, this may not always work properly because some combination of inputs could still trigger a bug. Therefore, the supervisor would have to recompute the task to determine definitively whether there is a bug.

Software bugs appear to be correct answers in all of the other schemes described in Chapter I. The supervisor in a voting scheme accepts the result as correct because all of the participants have the same software and task, so the same bug appears in all of the results. Likewise, the supervisor in a spot checking scheme may accept incorrect results because the results are only as correct as the software. Software bugs cannot be detected using these schemes because results are not verified objectively—instead, the proper execution of the implementation is considered proof of a result’s correctness. Therefore, the maximum error of each scheme described in Chapter I is actually $(1 - f) * P(\text{bug})$ higher, where f is the fraction of lying participants and $P(\text{bug})$ is the probability that a software bug causes the implementation to yield an incorrect result.

Peer review and spot certification can detect bugs in software, but some efficiency may be sacrificed. The simplest way to detect bugs is to spot check all of the tasks that are completed incorrectly. More complex schemes might utilize certain conditions or parameters in determining when it would be wise to spot check tasks. Timing information could be used, as described in Chapter I, to rule out obvious lazy cheaters. Also, multiple participants giving the same incorrect result in a peer review scheme may suggest the presence of a bug.

Although credibility-based fault tolerance is not thoroughly analyzed in this thesis, the approach is discussed because it could be one of the best ways to determine when to spot check tasks for software bugs. The credibility of a participant is exactly the type of additional information the supervisor needs to distinguish between lies and software bugs. The supervisor could do this by spot checking incorrect results from high-credibility participants and ignoring incorrect results from low-credibility participants.

Blacklisting

When blacklisting can be enforced in schemes that include result checking, it must be applied more carefully because of the presence of detectable software bugs. When a task's result fails verification, the participant who submitted the result may be *suspended* pending a spot check for the task in question. A suspended participant is no longer allowed to request tasks from the supervisor. The supervisor may choose to assign the task to another participant or may recompute the task later, with idle resources. If there are many participants in the system, the supervisor may choose to delay the spot check until multiple participants have completed the task incorrectly. If the supervisor or any participant completes the task successfully, the suspended participants are blacklisted. If the supervisor fails to complete the task successfully, the supervisor *exonerates* all of the participants who have been suspended because of their failure to complete that particular task. An exonerated participant may continue receiving tasks from the supervisor.

Efficiency

The efficiency calculations for peer review (Equation II.2) and spot certification (Equation II.4) do not account for the additional work required to detect software bugs because it can be proven that no efficiency is lost in searching for software bugs unless a software bug is detected.

Theorem 4. It is possible to search for software bugs using peer review and spot certification in such a way that efficiency is lost only when a bug is found.

Proof.

If no participant is ever suspended, then

No efficiency is lost searching for bugs because no results are deemed incorrect.

If a participant is suspended for submitting an incorrect result, then

If the task is completed successfully by the supervisor or a participant, then

Suspended participants lied about the result, so no efficiency is lost.

If the task cannot be completed successfully by another participant, then

The participant is suspended.

If the supervisor cannot complete the task successfully, then

The supervisor has detected a software bug, and

If suspended participants lied about results, then

No efficiency is lost.

If suspended participants were truthful, then

Efficiency is lost. QED.

It should be noted that the proof of Theorem 4 makes the assumption that the supervisor does not assign tasks redundantly—at any time, each task may be assigned to only one participant or the supervisor. This assumption is reasonable because any redundant task assignment would obviously cause a loss of efficiency, even if the supervisor does not search for software bugs.

Additionally, the proof of Theorem 4 assumes that the only way that a participant can determine a correct answer is by using the same software that all of the other participants are using. Practically, this assumption means that all truthful participants

observe the same software bugs for each task. This assumption could be violated by bugs resulting from the randomization of tasks that may be used to prevent collusion.

Furthermore, the notion of software bugs in this thesis does not include operating system facilities or racing conditions. Thus, Theorem 4 applies specifically to software bugs, such as an incorrect algorithm or incorrect translation of an algorithm into an implementation, that are present in every participant's software.

While efficiency may not be lost when searching for software bugs, the scalability of the approach may suffer. As the number of tasks increases, so too will the number of tasks that must be recomputed by the supervisor due to lying participants. Saboteurs could easily degrade the scalability by submitting incorrect results, perhaps causing the supervisor to complete every task. Therefore, care should be taken when choosing which tasks to recompute.

This chapter has discussed result checking and its application to computational grids. Several new approaches have been created by combining result checking with traditional approaches. This chapter has analyzed the efficiency of the new and the traditional approaches and has proven theorems that stipulate when the use of result checking yields greater efficiency. This chapter also analyzed the security of the new approaches. The efficiency and security analysis both lead to the same conclusion: *result checking has great potential for ensuring correctness in a computational grid, provided that 1) result checkers may be designed for the problems solved on the grid and 2) the checking ratio is low.*

The remaining chapters in this thesis are devoted to the design, creation, and execution of an experiment to determine an upper bound on the average-case checking ratio for linear programming. Linear programming was chosen because it is easily verifiable and widely applicable. Moreover, linear programs may appear as tasks in workflow applications such as branch-and-bound [Eckstein et al. 2001]. This experiment will offer some insight into the level of efficiency that might be expected when grid result checking is used.

CHAPTER III

IMPLEMENTATION

1. Timing

Implementations were timed on a 1680.864MHz Intel Pentium 4 system with 256MB RAM running the Linux 2.4.27 kernel. The timing was performed using *gettimeofday*, which obtains the elapsed time since the Epoch with microsecond resolution. This method of timing is not completely accurate because it does not account for context switches and scheduling time. However, this additional overhead can be expected to exist in any computer system. Attempts were made to minimize this overhead by suspending the execution of all nonessential processes during test execution and by giving test processes the highest priority available.

2. The Solver: The Revised Simplex Method

The implementation of the revised simplex method used in this thesis is a modified version of Vanderbei's implementation of the two-phase revised simplex method with an eta factorization of the basis [Vanderbei 2001]. The simplex method was chosen because it is easily understood and because it performs well in practice. Vanderbei's implementation was chosen because the underlying algorithm is simple and is analyzed thoroughly by Chvátal [1983]. Vanderbei's implementation was modified to allow for timing.

Timing was performed by inserting calls to *gettimeofday* before and after calls to the *solver* subroutine. Furthermore, the *solver* subroutine was modified so that it no longer prints anything to the screen because this functionality involves unnecessary delays. No input or output operations are performed in the timed sections of the code.

3. The Checker

The implementation of the checker consists of two parts. The first part constructs a simple linear program of a specified size and constructs its solution. The second part consists of the actual checking, where the solution is checked against the linear program's constraints. Only the second part of the checker is timed, and this timed code consists of the following operations, as described in Chapter II:

- verify that $x \geq 0$.
 - verify that $y \geq 0$.
 - verify that $Ax \leq b$.
 - verify that $A^T y \geq c$.
 - verify that $c^T x = b^T y$.
- where $c, x \in \Re^n$
 $b, y \in \Re^m$
 $A \in \Re^{m \times n}$.

The two operations $Ax - b$ and $A^T y - c$ are performed within the same set of loops to improve the performance of the code. The linear programs and their solutions are constructed in the following form:

$$A = \begin{pmatrix} 2 & \dots & 2 \\ \vdots & \ddots & \vdots \\ 2 & \dots & 2 \end{pmatrix}$$

$$b_i = \frac{n * (n-1)}{2} * i, \quad \forall i = 1, \dots, m$$

$$x_j = j - 1, \quad \forall j = 1, \dots, n$$

$$c_j = \sum_{k=1}^n k * (k-1), \quad \forall j = 1, \dots, n$$

$$y_i = i - 1, \quad \forall i = 1, \dots, m$$

4. Optimization

Additional performance gains might be expected in both the solver and the checker if optimized linear algebra libraries are used. However, the solution provided by Vanderbei does not make use of such libraries, so such libraries have not been used within the checker. Both the solver and the checker were compiled with gcc 4.0.2 using optimization flags to improve performance. These optimizations may impact the accuracy of general computations, but the solutions obtained were verified accurate when the following optimization flags were used: `-march=pentium4 -O2 -fomit-frame-pointer`.

CHAPTER IV

DETERMINING A CHECKING RATIO

This chapter describes the design of an experiment to determine a worst average-case checking ratio for linear programming. Many of such checking ratios could exist for linear programming because the checking ratio can be affected by changes in several components, including but not limited to the following: algorithm, compiler, optimizations, implementation, hardware, and operating system. This chapter explains how the checking ratio obtained through experimentation will be a worst average-case checking ratio for the particular system described in Chapter III.

1. Motivation

The efficiency of schemes that use certification is heavily dependent upon the ratio of checking time to solving time, called the *checking ratio*. Therefore, it is important to determine this ratio theoretically and to examine whether the theoretical ratio holds under practical circumstances. Linear programming has been selected as an example problem for this analysis because of its generality. Furthermore, linear programs may be used to solve integer linear programs in a workflow application model [Eckstein et al. 2001]. Integer linear programs are NP-complete, so if these problems can be solved on a grid, any problem in NP can be solved on a grid. Thus it suffices to determine the theoretical and empirical checking ratio for linear programming, which will indicate whether it is feasible to solve integer linear programs using a workflow

application model on a grid. The results from the experiment will likewise yield numerical results for the expected efficiency of each scheme introduced in Chapter II.

The checking ratio derived should represent the average case as much as possible, over-estimating when necessary. The analysis of algorithms usually focuses on worst-case asymptotic running time, which presents two problems when used for determining the checking ratio: 1) Good developers exploit the fundamental characteristics algorithms to achieve a running time that is better than the worst case, so the theoretical checking ratio is unrealistic. As an example, consider bubble sort, a simple algorithm that requires $\theta(n^2)$ time in the worst case to sort a list of size n . If the number of inversions in the list is restricted, bubble sort will likely perform better than most general-purpose sorting algorithms. 2) Asymptotic running times omit constants and minor terms. These terms play a crucial role in determining the checking ratio. As an example, consider a checker that requires n time and two solution algorithms: one that requires n time and one that requires $10n$ time. Asymptotic analysis would treat these algorithms the same, but the associated checking ratios differ by an order of magnitude. Thus, the checking ratio should be derived by counting or approximating the number of significant operations an algorithm performs in the average case.

2. Experiment Design

This section details an experiment that yields a worst average-case checking ratio for the system described in Chapter III. The experiment proceeds as follows:

- 1) Table 4.1 in Chvátal contains the average number of iterations required by the simplex method for particular problem sizes [Chvátal 1983]. These averages were obtained using Monte Carlo methods. The relevant portion of this table is reproduced in Table IV.1.

Table IV.1—Problem Size and Number of Iterations [Chvátal 1983]

Problem Size (n = m)	Number of Iterations
10	9.40
20	25.2
30	44.4
40	67.6
50	95.2

- 2) Klee-Minty problems of size one through seven are constructed. Klee-Minty problems are a class of pathological linear programs for which the simplex method requires $2^n - 1$ iterations [Chvátal 1983]. Klee-Minty problems were selected because they require a number of iterations that is exponential in the size of the input. This experiment solves Klee-Minty problems of size one through seven 10,000 times each to determine the average solution time. These times are fit with a second-degree polynomial using MATLAB, and solution times are interpolated for each number of iterations in Table IV.1. Solving the Klee-Minty problems requires the maximum number of iterations relative to the problem size, so conversely the problem size is minimized relative to the number of iterations. Thus, interpolation against these data for a given number of iterations yields the minimum solution time for each of the

iterations. The iterations in Table IV.1 are averages for the corresponding problem size, so the solution time obtained through interpolation serves as a lower bound on the average solution time of a dense linear program. However, solution time is in the denominator of the checking ratio, so the lower bound on the solution time yields an upper bound on the checking ratio—a worst average case.

- 3) Dense linear programs and their corresponding solutions of size 10, 20, 30, 40, and 50 are constructed and checked 1,000 times each.
- 4) Data from Steps 2 and 3 are combined to determine the checking ratio for each of the five problem sizes in Table IV.1.

3. Hypothesis

The worst-case asymptotic running time of the revised simplex method is exponential in the size of the input. However, in practice this method performs quite well, typically requiring less than $3m/2$ iterations. Each iteration requires time bounded by $32m + 10n$ floating point operations. Therefore, the average-case running time of the revised simplex method is expected to be roughly equivalent to $48m^2 + 15mn$ floating point operations [Chvátal 1983]. A complete discussion of the revised simplex method can be found in Chapter 7 of Chvátal [1983].

Checking the solution of a linear program is much simpler than solving one. Table IV.2 contains the steps required for checking the solution, along with the required running time of each step. The time of each step is determined by counting the floating

point arithmetic operations in the implementation of the checker. Summing the times from Table IV.1, the total time required to check a solution is roughly equivalent to $2mn + 2m + 2n + 1$ floating point operations.

Table IV.2—Theoretical Checking Time.

Operation	Time
verify that $x \geq 0$	0
verify that $y \geq 0$	0
verify that $Ax \leq b$	$m * n + m$
verify that $A^T y \geq c$	$m * n + n$
verify that $c^T x = b^T y$	$n + m + 1$

The theoretical checking ratio is obtained by dividing the number of operations that the checker requires by the number of operations that the revised simplex method requires.

Hypothesis. Linear programming has a checking ratio as described by Equation IV.1, so the checking ratio will converge to approximately 0.033, as illustrated by Figure IV.1.

$$\text{Linear programming theoretical checking ratio} = \frac{2mn + 2m + 2n + 1}{48m^2 + 15mn}. \quad \text{(IV.1)}$$

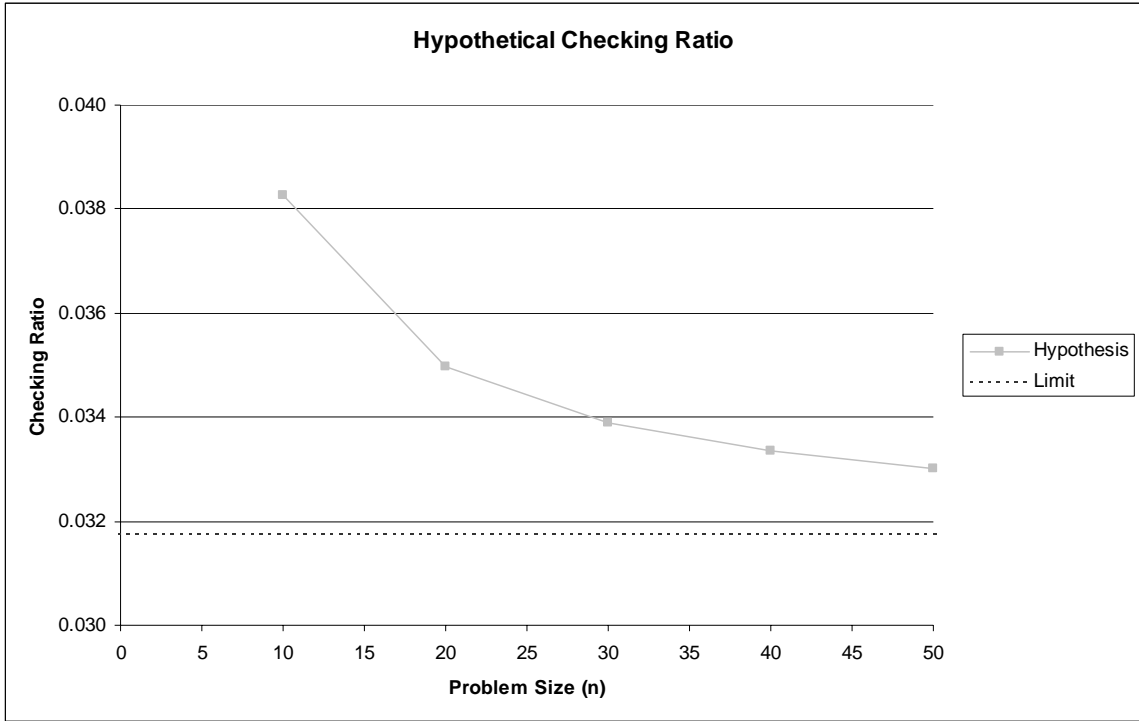


Figure IV.1—The Hypothetical Checking Ratio for Linear Programming.

This chapter has developed a hypothetical checking ratio for linear programming as well as an experiment to verify the hypothesis. The remaining chapter presents the results of this experiment and discusses the implications of those results.

CHAPTER V

RESULTS

This chapter presents the results of the experiment designed in Chapter IV to determine a checking ratio for linear programming. The implications of these results are then discussed. Primarily of interest is the effect of compiler optimization on the checker and how this optimization affects the checking ratio. Additionally, the efficiency calculations derived in Chapter II are revisited now that an empirical checking ratio is known. Numerical results are considered for the efficiency of spot certification and peer review.

1. Solver Results

Table V.1 contains timing results obtained from solving the seven Klee-Minty problems. These solution times were fit with a second-degree polynomial using MATLAB. This polynomial takes the form $1.7e-8x^2 + 1.4e-5x + 8.9e-5$, where x is the number of iterations, and values of the polynomial are in seconds. Figure V.1 shows that a second-degree polynomial fits very well, and the norm of the residuals is $1.9e-5$.

Table V.1—Timing Results for Klee-Minty Problems.

Problem Size (n = m)	Iterations	Time to Solve (sec)	STDev (Time to Solve)
1	1	0.000104	2.63E-06
2	3	0.00013	3.47E-06
3	7	0.000191	3.35E-06
4	15	0.0003	3.89E-06
5	31	0.000562	1.21E-05
6	63	0.001045	3.85E-05
7	127	0.002172	0.000238

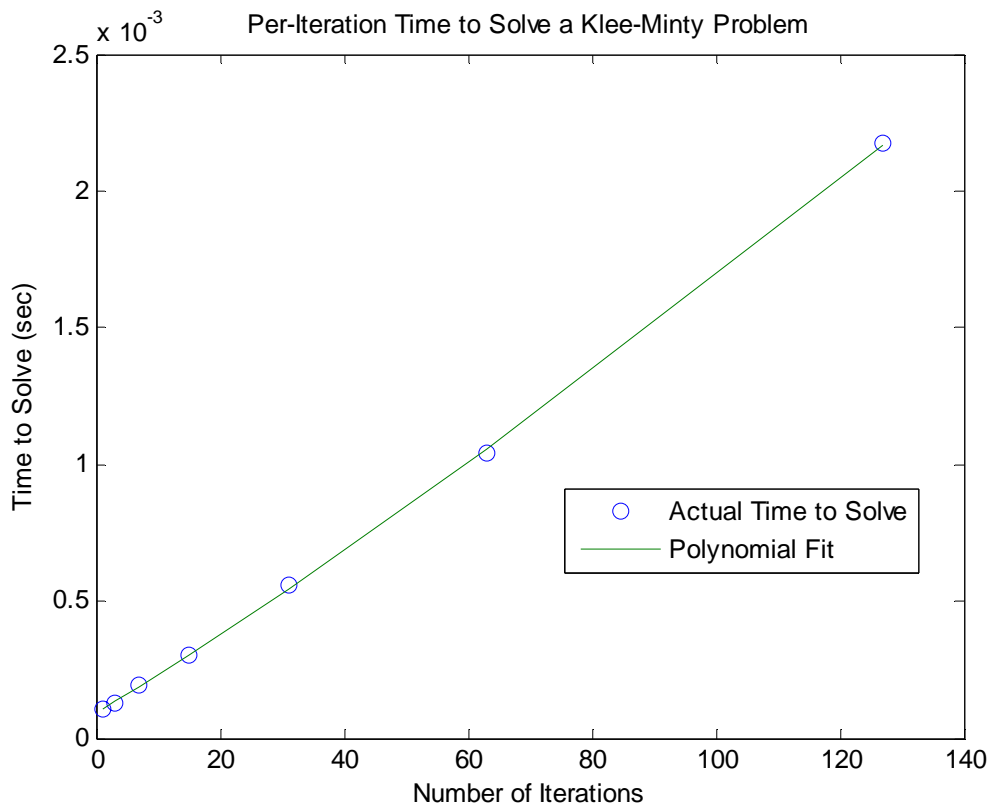


Figure V.1—Polynomial Fit of the Solution Time of Klee-Minty Problems.

Table V.2 contains the interpolated solution times for each number of iterations in Table IV.4. These values are also displayed in Figure V.2, along with the theoretical solution time from Chapter IV. The theoretical solution time has been scaled and shifted so that the first point in the curve matches the first experimental data point. Please note that this was performed only so that the two could be compared visually—the theoretical values are upper bounds on the number of arithmetic operations, and the experimental values are expressed in seconds. Even so, the two series are very similar. The deviations that do exist could have had any number of causes, including but not limited to real-world scheduling, memory size and latency, processor architecture, and optimizations. It would be nearly impossible to determine the role that each of these causes plays in causing deviations from the theoretical values.

Table V.2—Problem Size, Number of Iterations, and Solution Time.

Problem Size ($n = m$)	Number of Iterations	Time to Solve (sec)
10	9.40	2.2465E-04
20	25.2	4.5885E-04
30	44.4	7.5484E-04
40	67.6	1.1292E-03
50	95.2	1.5983E-03

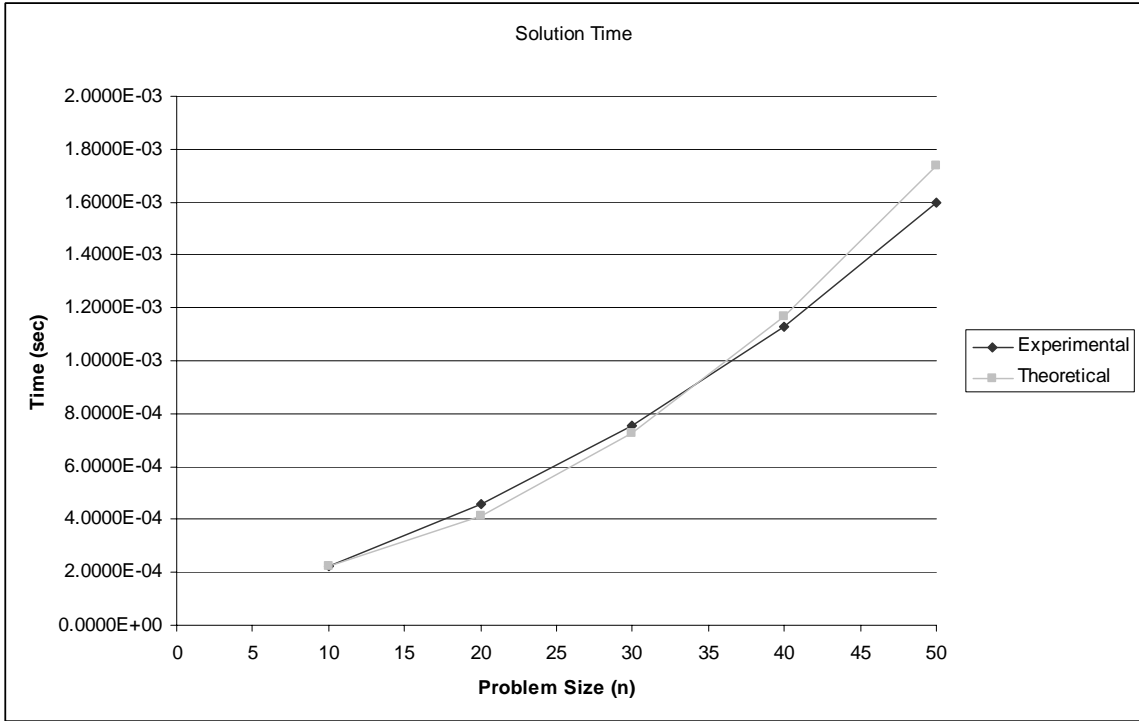


Figure V.2—Theoretical and Experimental Solution Times.

2. Checker Results

Table V.3 contains timing results obtained from checking solutions of size 10, 20, 30, 40, and 50 1,000 times each. Figure V.3 illustrates these results, along with the corresponding scaled and shifted theoretical values. It is obvious that something has changed the very nature of these results—the difference between the theoretical and experimental values is staggering. The primary suspect is the optimization of the checker. Therefore, the same set of experiments was run with optimization turned down to -O, and the results are presented in Table V.4 and Figure V.4. It is easy to see from Figures V.3 and V.4 that changing the optimization of the checker had a drastic impact on the running time. Optimization changes were not tested on the solver because the goal of the experiment is to obtain an upper bound on the checking ratio, and removing

optimizations from the solver would counteract this goal. Extrapolating the results from the checker, it seems likely that removing optimization from both the checker and the solver would bring the experimental checking ratio closer to the theoretical value.

Table V.3—Experimental Checking Time.

Problem Size (n = m)	Time to Check (sec)	STDev (Time to Check)
10	7.75E-06	4.84E-07
20	1.08E-05	7.96E-07
30	2.77E-05	1.29E-06
40	3.12E-05	1.88E-06
50	4.47E-05	1.98E-06

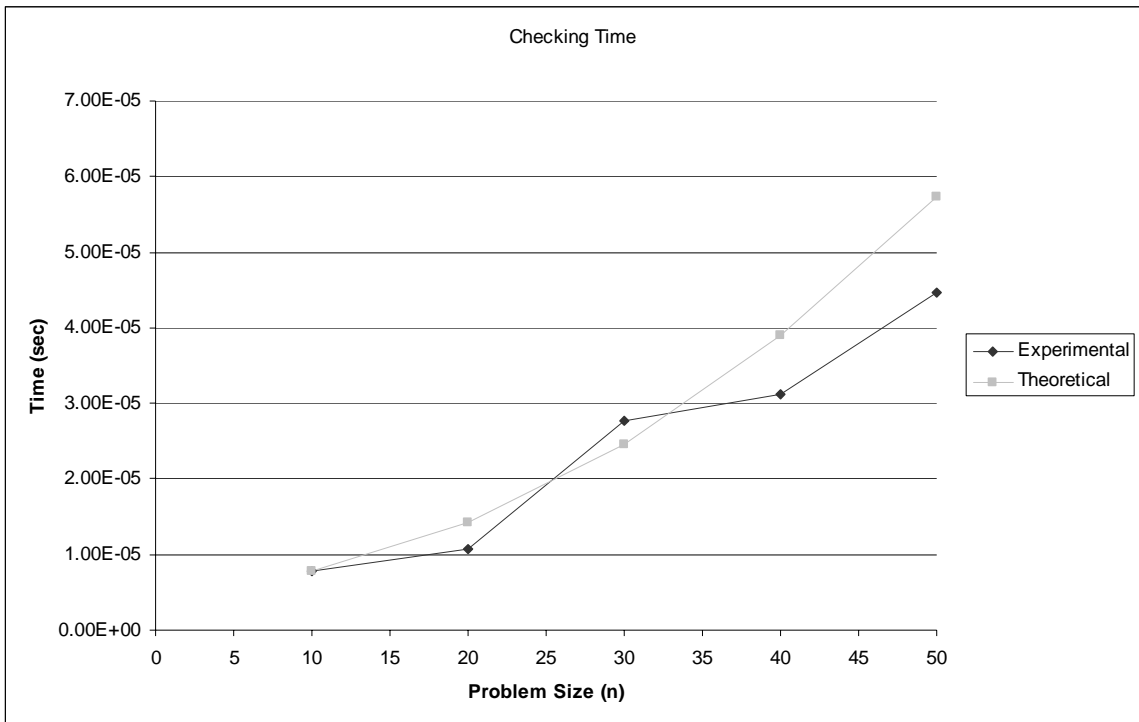


Figure V.3—Theoretical and Experimental Checking Times.

Table V.4—Checking Time with Optimization Turned Down.

Problem Size (n = m)	Time to Check (sec)	STDev (Time to Check)
10	9.04E-06	7.62E-07
20	1.37E-05	8.54E-07
30	2.66E-05	1.38E-06
40	4.15E-05	2.03E-06
50	5.98E-05	2.55E-06

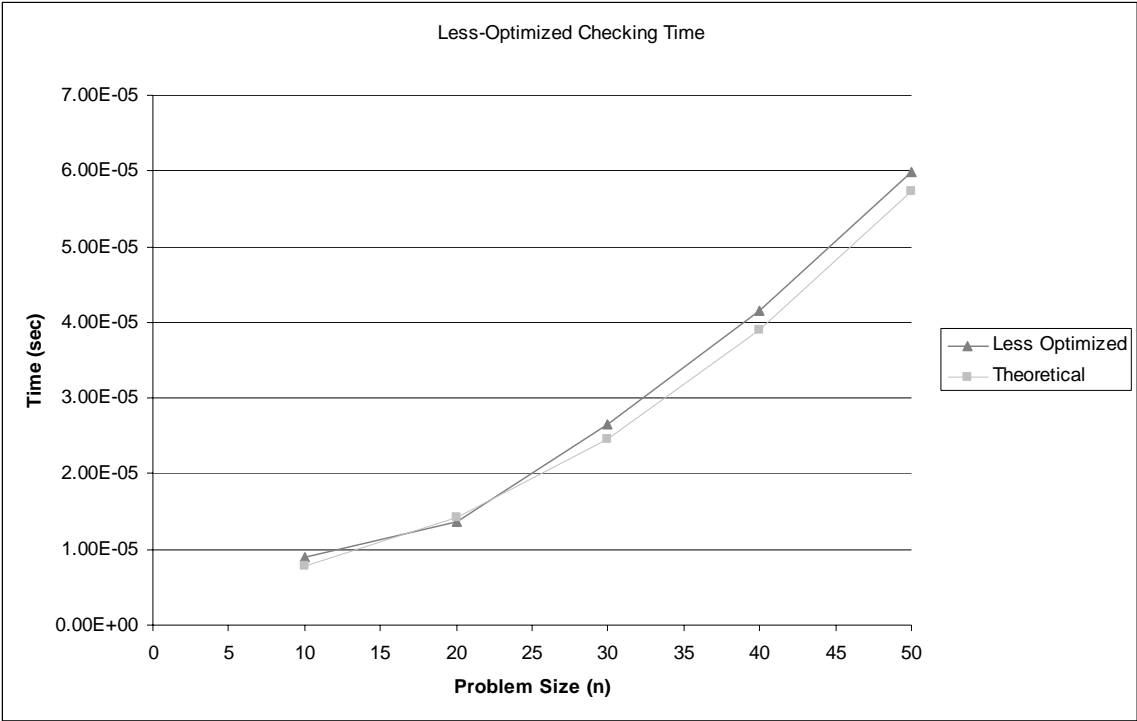


Figure V.4—Checking Time with Optimization Turned Down.

3. Checking Ratio

The checking ratio is determined by dividing the checking time by the solving time. Figure V.5 shows two different experimental checking ratios and the theoretical checking ratio. It may be worth noting that in this case, the theoretical values do not require shifting or scaling—these values are derived directly from Equation IV.1. The first experimental checking ratio occurs when both the checker and the solver are optimized, while the second checking ratio was computed using a fully-optimized solver and a less-optimized checker. With the exception of one data point, the optimized checker yields a checking ratio below the theoretical value. Likewise with one exception, the less-optimized checker yields a checking ratio above the checking ratio.

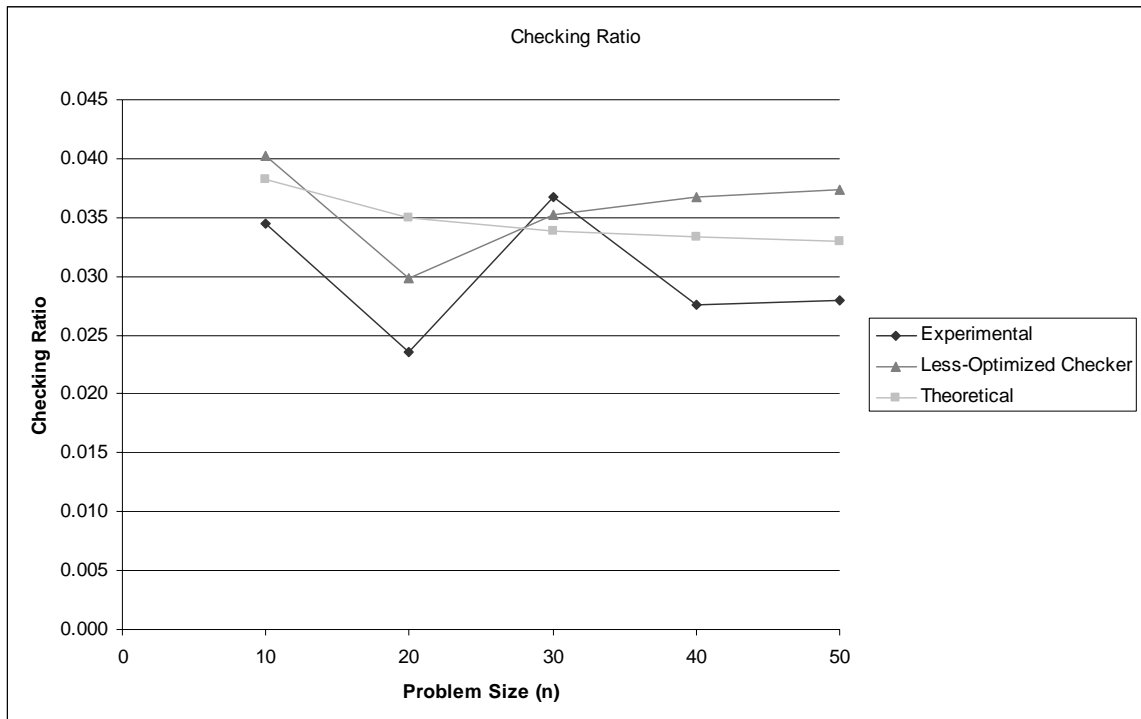


Figure V.5—Checking Ratio.

Overall, the use of the optimized checker resulted in an average 16.6% deviation from the theoretical values, while the less-optimized checker resulted in an average 9.45% deviation from theoretical values. The average checking ratios were 2.94% for the optimized checker, 3.59% with less optimization, and 3.47% for the theoretical value. All of these assessments suggest that optimization drastically improves the checker's performance and thus the checking ratio.

The experimental and less-optimized checking ratios, for a problem of size 30, are greater than the theoretical checking ratio. The erratic behavior of the optimized checker (Figure V.3) for problems of size 30 likely can be attributed to irregularity in the optimization. However, since both the experimental and less optimized checking ratios are greater than the theoretical checking ratio, the most likely cause is underestimation of the solution time.

CHAPTER VI

DISCUSSION AND CONCLUSIONS

1. Discussion

The empirical checking ratio was lower than predicted, and it was shown through additional experimentation that effective compiler optimization of the checker was the factor that caused the crucial difference. This was a very predictable result: *simple checkers are in fact so simple that they can be optimized a great deal, not only by the programmer but by the compiler as well.* Despite underestimating the time required to solve linear programs, the optimization of the checker drove the checking ratio below the anticipated value.

The actual value of the checking ratio is compelling—2.94%. Provided that every result is correct, the efficiency would be a staggering 97.1%. Thus, spot certification could feasibly be used to check every result submitted. This perfect level of security and very high efficiency is an obvious improvement over spot checking and suggests that workflow applications could be completed successfully on a grid if the supervisor can dedicate one computer for every 34 dedicated participant computers. Therefore, it may be feasible to use a small cluster to check results from thousands of participants.

Integer linear programming is one such workflow application that could be run successfully on a grid using spot certification. It is possible to solve an integer linear program by solving many relaxed subproblems, each of which is a linear program. Moreover, since integer linear programming is NP-complete, any problem in NP may be

reduced to an integer linear program in polynomial time. Therefore, any problem in NP may be solved on a grid using spot certification.

The checking ratios observed suggest that peer review is more efficient than voting for solving linear programs when the two schemes have the same level of security. When half of the participants are liars, peer review with one reviewer is 94.4% efficient, and when none of the participants are liars, the scheme is 97.1% efficient. Voting with two participants offers the same level of security but is only 50% efficient. More than 97.1% of the participants would have to be liars for peer review to perform worse than voting.

The experiment performed in this thesis shows that checking ratios achieved in practice can be as low as predicted by counting the significant operations of the checker and solver. Moreover, the checking ratio observed in this thesis is lower than the hypothesized value because compiler optimizations significantly reduced the running time of the checker. Optimization can be expected to impact the running time of the checker more than it affects that of the solver because the checker is typically much simpler than the solver.

2. Conclusions

This thesis discusses various mechanisms for ensuring that computational results from a grid are correct. These mechanisms include voting, spot checking, ringers, and credibility-based fault tolerance. Three new approaches are introduced that achieve the same goal by using result checking. These approaches are called peer review, spot

certification, and ringer peer review. Each approach, new and old, is evaluated in terms of security and computational efficiency.

Necessary and sufficient conditions for each new approach are derived that dictate when the approaches are better than their conventional counterparts. After deriving these conditions, it became apparent that little was known about the checking ratio that plays such a crucial role in determining the efficiency of spot certification and peer review. Therefore, this thesis presents an experiment to determine the checking ratio for linear programming.

The results from this experiment show that spot certification and peer review can be drastically more efficient than spot checking and voting, the approaches from which they are derived. The average checking ratio for linear programming was determined to be 0.0294. This means that checking the solution of a linear program requires 2.94% of the time required to discover the solution using the revised simplex method. Therefore it would be reasonable to expect spot certification to be at least 90% efficient even with perfect security. Achieving perfect security with spot checking requires the supervisor to compute every result, thereby rendering the grid useless.

The results from the linear programming experiment also suggest that simple checkers benefit a great deal from compiler optimizations, which in turn means that the checking ratio benefits a great deal from such automatic optimizations. The likely cause of this is the simplicity of checking a result. Result checkers may be so simple that each of the variables required in the checking process can be mapped to a register. Additionally, the simplicity of result checkers may enable software developers to hand-optimize code more easily, producing even further improvements in efficiency.

When utilized in grid computing, result checking evaluates the correctness of results objectively. Other approaches evaluate results subjectively, depending on assumptions about participants' behavior or on the correctness of a potentially complex implementation. Furthermore, result checking offers software developers a means to discover bugs without extensive testing or formal verification. Although it is not a general approach, where result checking can be applied it offers potentially drastic efficiency improvements and an added layer of security.

3. Future Work

While spot certification and peer review can be extremely efficient, they cannot be applied to every problem. Therefore, future work will include the design and analysis of new certifying algorithms so that more problems can be solved on grids using these new approaches. Also, it is unclear how spot certification and peer review would fare when quality of service is necessary, so some future work in this direction may be warranted.

The future will also bring the execution of additional experiments to determine the checking ratios of various certifying algorithms. The primary problem encountered in performing these experiments is the difficulty of achieving a true average-case analysis. Average case analyses are usually performed by assuming that inputs are uniformly distributed, and often this is not a reasonable expectation. Therefore, it may be best to perform these experiments with a particular form of a particular problem in mind. Regardless of how the experiments are performed, these experiments could play a vital role in the evolution of more efficient grid security mechanisms.

REFERENCES

- Aiello, W., Bhatt, S. N., Ostrovsky, R. and Rajagopalan, S. Fast Verification of Any Remote Procedure Call: Short Witness-Indistinguishable One-Round Proofs for NP. *International Colloquium on Automata, Languages and Programming*, 2000, pp. 463-474.
- Blum, M. Designing Programs to Check Their Work. Technical Report 88-09, International Computer Science Institute, 1988.
- Blum, M. and Kannan, S. Designing Programs that Check their Work. In *21st ACM Symposium on the Theory of Computing*, 1989, pp. 86-97.
- Buyya, R. Grid Resource Management and Application Scheduling. The GridBus Project. <<http://www.gridbus.org>> Accessed 10 March 2006.
- Chvátal, V. *Linear Programming*. New York: W. H. Freeman and Company, 1983.
- Du, W. and Goodrich, M. T. Searching for High-Value Rare Events with Uncheatable Grid Computing. In *Proceedings of the Applied Cryptography and Network Security (ACNS) Conference*, 2005, pp. 122-137.
- Du, W., Jia, J., Mangal, M., and Murugesan, M. Uncheatable Grid Computing. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, 2004, pp. 4-11.
- Eastlake, D. and Jones, P. US Secure Hash Algorithm 1 (SHA1). Request for Comments 3174, Network Working Group <<http://www.ietf.org/rfc/rfc3174.txt>> September 2001. Accessed 1 March 2006.
- Eckstein, J., Hart, W. E., and Phillips, C. A. PICO: An Object-Oriented Framework for Parallel Branch and Bound. In *Inherently Parallel Algorithms in Feasibility and Optimization and their Applications*, (eds.) D. Butnariu, Y. Censor, and S. Reich. Amsterdam: Elsevier Science Publishers, 2001, pp. 219-265.
- Germain-Renaud, C. and Monnier-Ragain, D. Grid Result Checking. In *Proceedings of the Second Conference on Computing Frontiers*, 2005. ACM Press, 2005, pp. 87-96.
- Goldreich, O. Combinatorial Property Testing (a survey). *Electronic Colloquium on Computational Complexity (ECCC)* 4(56), 1997.

- Golle, P. and Mironov, I. Uncheatable Distributed Computations. In *Topics in Cryptology (CT-RSA 2001) – The Cryptographer’s Track at RSA Conference 2001*, (ed.) David Naccache, 2001, pp. 425-440.
- Grid Computing Projects. <<http://www.grid.org>> 2004. Accessed 10 March 2006.
- Kahney, L. Cheaters Bow to Peer Pressure. *Wired News*, 15 February 2001.
- Kenyon, C. and Cheliotis, G. Creating Services with Hard Guarantees from Cycle-Harvesting Systems. *International Symposium on Cluster Computing and the Grid (CCGRID 2003)*, 2003, pp. 224-231.
- Kratsch, D., McConnell, R. M., Mehlhorn, K., and Spinrad, J. Certifying Algorithms for Recognizing Interval Graphs and Permutation Graphs. *ACM-SIAM Symposium on Discrete Algorithms (SODA 2003)*, 2003, pp. 158-167.
- Lenstra, A. K. Further Progress in Hashing Cryptanalysis.
<<http://cm.bell-labs.com/who/akl>> 26 February 2005. Accessed 8 March 2006.
- Mehlhorn, K. Certifying Algorithms: An Attempt of a Theory. Presentation available at <<http://cs.ioc.ee/yik/schools/win2005/mehlhorn/TallinnTalkTheory.pdf>> Accessed 7 February 2006.
- Rivest, R. The MD5 Message-Digest Algorithm. Request for Comments 1321, Network Working Group <<http://www.ietf.org/rfc/rfc1321.txt>> April 1992. Accessed 1 March 2006.
- Sarmenta, L. F. G. Sabotage-Tolerance Mechanisms for Volunteer Computing Systems. *Future Generation Computer Systems*, 18(4):561-572, 2002.
- Szajda, D., Lawson, B., and Owen, J. Hardening Functions for Large-Scale Distributed Computations. *IEEE Symposium on Security and Privacy*, 2003, pp. 216-224.
- Vanderbei, R. *Linear Programming: Foundations and Extensions*, Second Edition. Norwell: Kluwer, 2001, pp. 136-141. <<http://www.princeton.edu/~rvdb/LPbook>> Accessed 17 March 2006.
- Venugopal, S., Buyya, R., and Ramamohanarao, K. A Taxonomy of Data Grids for Distributed Data Sharing, Management and Processing, Technical Report, GRIDS-TR-2005-3, Grid Computing and Distributed Systems Laboratory, University of Melbourne, Australia, April 21, 2005.
- Wang, X., Feng, D., Lai, X., and Yu, H. Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD. Cryptology ePrint Archive: Report 2004/199. 17 August 2004.

Wasserman, H. and Blum, M. Software Reliability via Run-Time Result-Checking. *Journal of the ACM*, 44(6), November 1997, pp. 826-849.

Wright, S. J. *Primal-Dual Interior Point Methods*. Philadelphia: Society for Industrial and Applied Mathematics, 1997.

Yurkewych, M, Levine, B. N., and Rosenberg, A. On the Cost-Ineffectiveness of Redundancy in Commercial P2P Computing. In *Proceedings of the ACM Conference on Computers & Communications Security*, November 2005, pp. 280-288.