Measuring the Speedup of a Commercial Application on a Computer Grid

Tim Hoehn
SAFE Corporation
15565 Swiss Creek Lane
Cupertino, CA 95014 USA
1 (408) 741-5809
tim@SAFE-corp.biz

Robert Zeidman
SAFE Corporation
15565 Swiss Creek Lane
Cupertino, CA 95014 USA
1 (408) 741-5809
bob@SAFE-corp.biz

ABSTRACT

For applications that are highly processor intensive it can be advantageous to implement them on a distributed system or grid. The application is divided into sub-computations that are sent to computers that are networked together where these sub computations or "jobs" can run concurrently, resulting in a significant overall performance improvement. Because of the growing necessity to solve difficult calculable tasks, the improved performance of distributed computing systems, and the lower cost of computer hardware from which these systems can be constructed, distributed computing systems are becoming much more commonplace [12]. A distributed system that delivers linear speed-up means that a distributed system comprised of just four computers results in a time-savings of 75 percent[1]. This paper summarizes the "gridification" of our CodeSuite® application and describes in detail specific factors affecting the speedup as determined theoretically and experimentally, allowing us to further fine-tine the system. This kind of analysis and these factors can be used to fine-tune other similar gridified applications, particularly those that compare one set of files to another set of files on a file-by-file basis.

KEYWORDS

Distributed computing, grid computing, parallel computing, SWIG, computational complexity, performance computing.

1. INTRODUCTION

While the concept of employing distributed resources for executing a single task has been discussed for several decades, there are new tools available that make this a less daunting task. Developing a grid for a legacy application is still not a trivial undertaking, but modern tools have made it a more mainstream procedure [11]. In this article we discuss one example of taking a commercial application designed to run on a single PC and building a network grid to enable the distribution of processing load among the computers on the grid. We discuss the tradeoffs we made in techniques we use, what types of applications are good candidates for "gridification", and the quantitative performance gains we realized, based on theoretical and experimental data.

2. BACKGROUND

The stand-alone application we "gridified" is called CodeSuite®. This is a suite of tools for comparing two different sets of source code or binary code to help prove or disprove plagiarism in copyright infringement cases using a process known in the legal community as the "abstraction-filtration-comparison" test [9].

There are four primary components to CodeSuite. These components are CodeMatch®, BitMatch®, CodeCross™, and CodeDiff®. CodeMatch compares source code and determines correlation based upon several different algorithms. Since CodeMatch is often used to compare gigabytes of source code and its algorithms are very processor intensive—it can sometimes take weeks to finish its computations. BitMatch can compare binary files to binary files, or binary files to source code files, and is even more processor intensive because it works at the bit level. CodeDiff is used to determine the differences between sets of source files and is not quite as processor intensive as the other two. And finally CodeCross finds traces of non-functional source code that have been copied from one program to another. All four of these applications are used to assist in detecting software plagiarism by comparing two directories of possibly many thousands of source code files, then producing a report showing the matching patterns found based upon pre-specified parameters. These tools are used in court cases to help prove or disprove copyright infringement. Often in intellectual property litigation cases time is in short supply. For this reason we developed CodeGrid®, a network grid to enable resource sharing among a set of low cost, heterogeneous computers to improve the performance of CodeSuite.

2.0 CODEGRID

The main requirement for CodeGrid was that it be able to distribute and execute a large amount of processing to an automatically scalable number of nodes of a network and then format the results returned from each computer on the grid into a single report. It needed to have a load balancing mechanism and be fault-tolerance in case a node has problems during execution. The program had to be integrated with the existing CodeSuite software with a minimum of modifications [6].

There are many different ways to share resources between multiple computers over a network, and in our research we compared the respective advantages, disadvantages, and trade-offs of choosing one way over another. The main advantages of distributed computing are increased performance by sharing the load of resource-intensive applications, improved scalability, and fault tolerance. These advantages come at a cost however of added complexity, potential security problems, and increased manageability issues. In order to develop a successful distributed system, careful planning is necessary, as remote calls between computers on the grid can be 1000 times slower than local calls between processes running on a single computer. There has to be a balance between the added overhead of the network management and the size of the job itself.

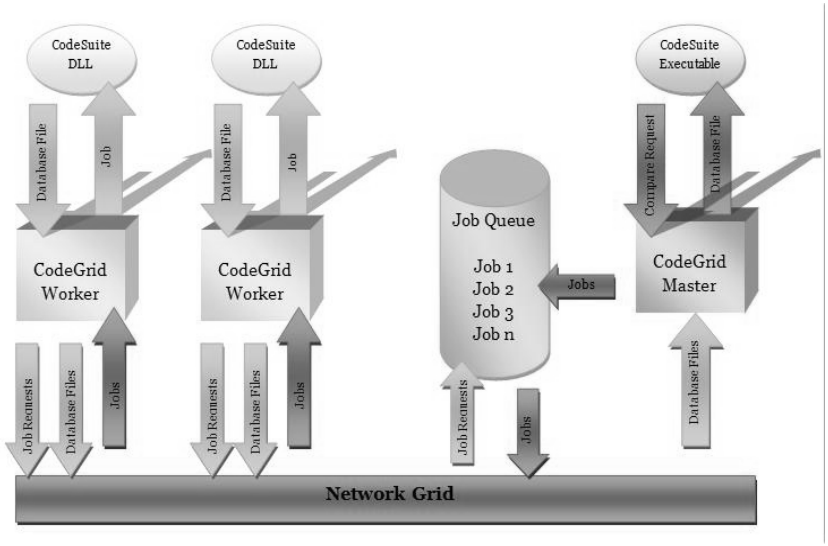


Figure 1. CodeGrid Overview

2.1 Research

Most tools available for the development of computing grids share similar architectures and communication strategies. Most of these approaches evolved from the same basic concepts. The three main decisions that had to be made in the development of CodeGrid were the programming language to use, the communications mechanism, and the work distribution mechanism. The first decision based upon our research was to use Java as our programming language. While this project could have been done using many different programming languages such as C++ or C#, we decided to use Java because it had the most to offer as far as available library routines, and would be the least restrictive as far as future portability and/or expansion to the Internet. Once we decided upon the programming language, the other decisions were heavily influenced by this decision.

2.1.1 Communications strategies and observations

CodeGrid has a "master" computer, which is the controller, and many "worker" computers that share their resources and perform computations. This is commonly known as the "master-worker paradigm." In our test grid we have four computers. Three of these computers have what I will call "remote" (from the master) workers, and one computer has a master and a "local" (to the master) worker.

We decided that Java's RMI (Remote Method Invocation) would be simpler than another, older but common communications mechanism - CORBA (Common Object Request Broker Architecture) and was the most advanced technology for client-server communications. Both CORBA and RMI are examples of distributed object computing middleware [3]. Our choice of using Java and RMI allowed us to use the "skeleton approach" and avoid delving into the underlying sockets layer programming [5]. We learned from our research that the learning curve for RMI was not as steep as CORBA. This decision played a huge part in the decision to use Java.

2.1.2 RMI callbacks

One of the areas of our program where performance was optimized was by limiting the number of RMI callbacks. Since RMI is based on object serialization, it is quite slow. To minimize the number of RMI calls we did things such as sending all the parameters each worker needs from the master at the same time using a single RMI call. Initially we had an RMI call for each parameter and the workers would request the parameters at the point they were needed. Changing this resulted in a substantial performance gain.

2.1.3 JNI calls

CodeSuite is written in C and its executable is a Windows dynamic link library (DLL). In order to use Java we needed a way for CodeGrid to communicate with the CodeSuite DLL. Java has just such a tool for native code translation called JNI (Java Native Interface). From our research we knew that JNI has a reputation for being difficult to use, but nonetheless a JNI wrapper would be necessary for the required function calls to the native CodeSuite DLL. We discovered a great freeware program called SWIG (Simplified Wrapper and Interface Generator)[14] that generates JNI wrappers for native C programs by using a simple interface file created from the native C header file. This file is run through SWIG and a wrapper is generated. This wrapper file automatically generated by SWIG is then compiled with the C program to create a bi-directional pathway for function calls between the C and Java programs. SWIG does have a learning curve of its own, but after using it and examining the code it generates our understanding of JNI and especially pointer handling between languages was greatly enhanced.

This is another area where substantial performance increases can be attained. This bridge between the Java Virtual Machine (VM) and the native DLL can be a source of congestion. Any time you create a new JNI bridge and pass data across that bridge through the JNI bottleneck there is a performance hit. Also transmitting values into and out of the VM is expensive. Primitive values are cheaper than object references because object references require some set-up so the Java garbage collector does not clean up objects that have references in native calls. A good optimizing JIT compiler will generally do a very good job of optimizing Java code and there is a significant performance penalty by making native calls due to the fact that

native method calls present a barrier to this optimization by the Java compiler. By combining as much work into a single JNI call as possible you can reduce the associated overhead. In our case we combined our native function calls into as few Java methods as we could in order to reduce the costs generated by them.

2.1.4 Network Speeds

It is important to ensure that the grid LAN is operating optimally to reduce the network transfer bottleneck. Our LAN is a 10/100 Ethernet and we used a benchmarking utility called NetPerf [7] to test the actual data rates we were getting. Upon testing we discovered we were getting between 83Mbits/sec and 90Mbits/sec, which is acceptable for a 10/100 network. While the file transfer isn't the dominant factor in our equation on large file sets, there is a significant amount of overhead associated with transferring all of the files for comparison and sending all the results back to the master machine.

2.1.5 Work distribution strategies

The basic unit of work is called a "job." Each job basically consists of a file from one of the two file sets being compared and its set of comparison parameters. We refer to these two sets or folders of files as the "input" directory and the "compare" directory. A job is basically an input file and a set of parameters. Each worker gets its own complete copy of the compare directory.

2.1.6 Push – Pull model tradeoffs

The two basic ways we could have distributed these jobs to the workers are known as the "push" model and the "pull" model [13]. In both models the master initiates the computation by dividing the work into jobs by some pre-defined criteria. With the push model, the master sends the jobs to the workers. With the pull model, the master puts the jobs in some shared container and waits for the tasks to be picked up and completed by the workers.

We chose the pull model. In this model all worker machines remotely fetch work from the master as needed rather than have the master divide up and distribute the work. Each worker continually requests work and sends back its results until all the work is completed as illustrated in Figure 1.The master then sorts and formats all the results into a database file from which various types of reports can be generated. We felt that this technique would be efficient since creating the database requires sorting the individual results from the workers. In this way we don't need to continually sort the results as they arrive at the master.

One of the advantages of using the pull model is that the grid automatically balances the load. This is because the set of work is shared, and each worker can pull work from the set at its own pace until there is no more work to be done. This method provides excellent load balancing regardless of worker speeds and network variations and provides good scalability.

While the push method would minimize network communication, we decided ahead of time that variations in job execution times would make this method more inefficient than the pull model in our case. In practice we see that there are indeed noticeable variations in how long it takes each worker to complete jobs. Often all of the workers finish and there is one worker still chugging away on an abnormally large job.

2.1.7 Local and remote files

CodeGrid distributes its files is by initially copying the compare directory from the local hard drive of the master to the local hard drive of each worker. While there is a lot of network overhead having to transfer one complete directory of files to every worker, one set of files must be accessed repeatedly by each worker, and this was the only practical option.

The files from the input directory reside in a shared directory on the local hard drive of the master and are accessed via the network by each worker. This works well because each worker needs to examine only a distinct portion of these files. This shared directory on the local hard drive of the master is mapped to a local directory name for each worker. These shared files are accessed strictly as needed by the workers when they get a job that calls for that file.

3. IMPLEMENTATION

3.1 Software and hardware

3.1.1 Software

For our Grid we used four identical PCs running Windows Vista Home Basic. The CodeGrid software was built with the Eclipse IDE and the Java SE Runtime Environment 6.

3.1.2 Hardware

Our Grid hardware consisted of four identical PCs with 1.8GHz AMD Sempron 3200+ processors, and 446MB of RAM connected with a Linksys FastEthernet 10/100 router.

4. PERFORMANCE

4.1 Theoretical Performance

Before we ran any tests we assumed that the best performance results we could achieve by building this grid would be a speed increase by a factor slightly less than the number of additional nodes. We guessed that the overhead of the grid would get smaller and smaller in relation to the amount of work simply by economies of scale. In our testing we attempt to prove this hypothesis. In order to simplify things and maintain consistency we used the same CodeSuite tool, CodeMatch, for all tests.

The computational complexity for the CodeMatch algorithm to find plagiarism within a set of N files is $O(N^2)$ since each file must be compared to each other file. Typically CodeMatch is used to search for plagiarism between two different programs, the first consisting of Ni files in the input directory and the second consisting of Nc files in the compare directory. Since each input file is compared to each compare file, the computational complexity is $O(Ni*Nc)$. This quadratic time complexity function grows rather rapidly as the number of files increases. CodeGrid allows a linear speedup of the computation, but the computation complexity remains inherently quadratic.

Figure 2 shows the theoretical relationships between the execution time and data size of CodeSuite and CodeGrid with various numbers of nodes.

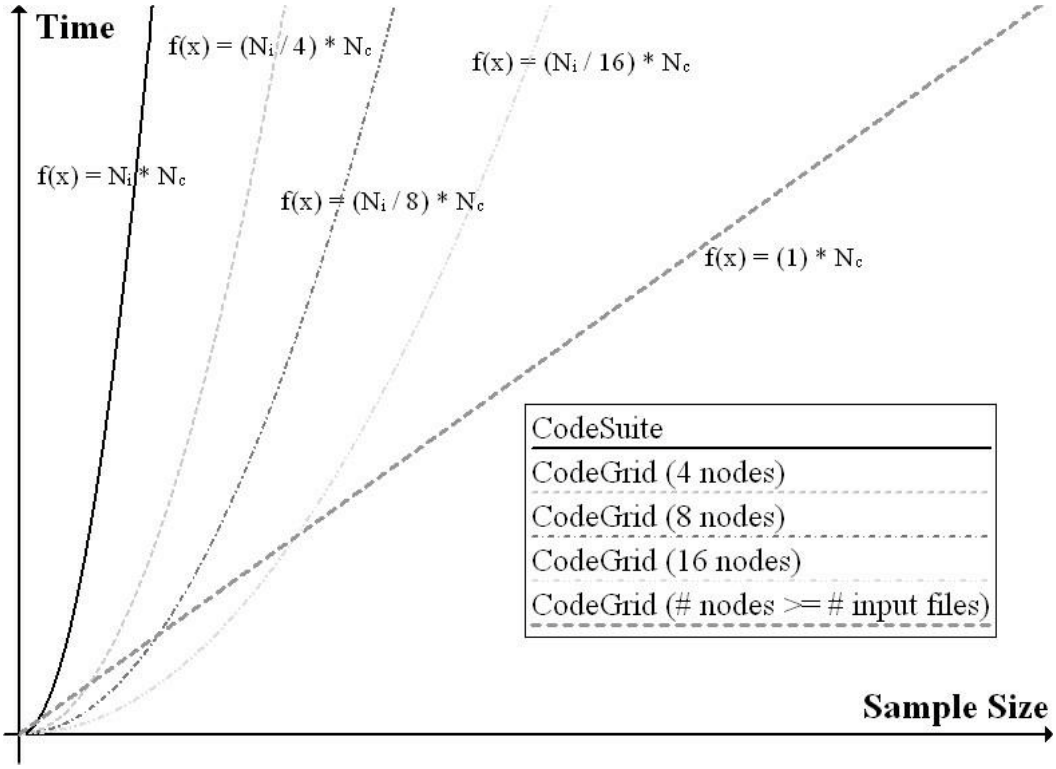


Figure 2. Theoretical Performance

An interesting thing to note is that when the number of nodes equals the number of input files *Ni*, then the performance is simply proportional to *Nc* as shown in Figure 2. Because CodeMatch cannot create jobs smaller than a single file, increasing the number of nodes beyond this has no effect on performance. Thus for any given comparison, for large numbers of nodes in the grid, performance approaches linear with respect to *Nc*, or O(*Nc*). In practice, this is difficult to achieve because we have worked on comparisons involving hundreds of thousands of files. We would need hundreds of thousands of nodes in order to reliably get a linear computation for any given comparison.

Below are the variables and equations we felt would be significant to the performance estimation of CodeGrid.

Variables

W = number of grid nodes.

Twz = worker initiation time (initial RMI call and parameter transfer). This is the time it takes a worker to prepare to start.

Twc = transfer time for the compare directory from the master to a worker.

Twca = average transfer time for the compare directory from the master to a remote worker.

Twi = average transfer time for the input files to a remote worker.

Twr = total RMI call execution time per worker (for each job request, database append request, etc).

Twx = CodeSuite execution time per worker. This could be execution time for CodeMatch, CodeDiff, BitMatch, etc.

Two = grid-related overhead time per remote worker.

Twd = transfer time for a remote worker's database files returned to the master.

Jw = jobs completed per worker W.

Fw = fraction of work done by worker W.

Twtot = overall time by worker W.

Ni = amount of code in input folder.

Nc = amount of code in compare folder.

Txc = cumulative time to transfer the compare directory to all remote workers.

Tm = time for all remote worker RMI calls.

Ta = re-assembly time for all databases on the master. This is how long it takes the master to sort and merge all of its individual worker database files into the final complete database file.

Ttot = calculated total CodeGrid execution time

J = number of jobs total.

Rw = data transfer rate per worker

Fo = fraction of the complete run that is grid overhead. This is the percentage of the overall time that isn't attributed to CodeSuite itself.

Tg = measured total CodeGrid execution time

Equations

The average transfer time for the compare directory from the master to a worker, Twca, equals the average transfer time for the compare directory from the master to a worker in a grid of W nodes. This is the sum of all the transfer times divided by the number of remote workers, keeping in mind that Twc for node 1 is 0 since it resides on the same machine as the master.

$$Twca = (\sum Twc) / (W–1)$$

Equation 1. Average compare file transfer time

The time to transfer the compare directory to all remote workers, Txc, equals the cumulative transfer time for the compare directory from the master to all workers in a grid of W nodes.

$$Txc = \sum Twc$$

Equation 2. Total compare file transfer time

The average data transfer rate per worker is calculated from the size of the compare directory divided by the time to transfer the compare directory to each of the three remote workers.

$$Rw = Nc / Twca$$

Equation 3. Average data transfer rate per worker

The transfer time for the input files to a remote worker, Twi, equals the amount of code in the input folder divided by the average transfer rate per worker times the number of workers minus one divided by the number of workers.

$$Twi = (Ni / Rw) * ((W-1)/W)$$

Equation 4. Average input file transfer time

The grid-related overhead time per remote worker, Two, equals the sum of a workers initiation times plus the average transfer time for the input files and compare files plus the total RMI call execution time per worker plus the return time for all database files from a remote worker.

$$Two = Twz + Twi + Twc + Twr + Twd$$

Equation 5. Overhead time per remote worker

The fraction of work done by worker W, Fw, equals the jobs completed per worker W divided by the total number of jobs.

$$Fw = Jw / J$$

Equation 6. Percentage of work done by worker

The total working time Twtot per worker equals the fraction of work completed by the worker times the total time for all work to be completed.

$$Twtot = Fw * Tg$$

Equation 7. Total working time per worker

The CodeSuite execution time per worker, Twx, equals the overall time by worker W minus the overhead for remote worker W.

$$Twx = (Twtot – Two)$$

Equation 8. CodeSuite execution time per worker

The fraction of the complete run that is grid overhead, Fo, equals the overall time by all workers divided by the total time for all workers.

$$Fo = (\sum Two) / (\sum Twtot)$$

Equation 9. Fraction of grid overhead

Total calculated CodeGrid execution time Ttot equals the total overhead per worker plus the total CodeSuite time per worker.

$$Ttot = (\sum Two) + (\sum Twx)$$

Equation 10. Total grid execution time

4.2 Performance Measurements

In our tests some values were measured while other values were calculated from the measured values. We obtain these measured values through the use of output statements and by using a tool called RMI Spy [2], which reports all incoming and outgoing RMI calls along with the amount of data passed and the time each call took. The results from our tests are shown in Table 1. Note that measured values are in clear cells while calculated values are in shaded cells.

Table 1. Test results

	Node 1	Node 2	Node 3	Node 4	Totals/Averages
Tg (sec)	-	-	-	-	3,000
Ttot (sec)	-	-	-	-	3,037
Ni (bytes)	-	-	-	-	4,602,866
Nc (bytes)	-	-	-	-	2,675,445
J					504
Jw	76	137	158	133	504
Twc (sec)	0	20	29	36	85
Twd (sec)	0	22	15	11	48
Ta (sec)	-	-	-	-	20
Twz (sec)	0	10	15	16	41
Twr (sec)	0	4	4	4	12
Tm (sec)	-	-	-	-	12
Rw (bytes/sec)	-	-	-	-	94,427
Twca (sec)	-	-	-	-	28.33
Two (sec)	0	56	63	67	223
Txc (sec)	-	-	-	-	85
Twi (sec)	-	-	-	-	37
Fw (%)	15%	27%	31%	26%	100%
Twtot (sec)	452	815	940	792	3,000
Twx (sec)	452	759	877	725	2,814
Fo (%)	-	-	-	-	7.42%

We discovered that during the transfer of compare files to workers, the transfer time to each subsequent worker gets longer as the workers get the files because workers begin requesting jobs and returning database files once they get their compare files. This increases the network traffic. By measuring the time it takes for the compare directory to transfer (we have console messages at each worker showing the beginning and end of transfer) we determined that it takes 20 seconds to transfer to the first remote worker, another 29 seconds to transfer to the second, and another 36 seconds to transfer to the third. This is a total of 85 seconds as measured to completely transfer to all 3 remote workers. Since we calculated Rw equals about 94,427 bytes per second we deduced that it should take about 37 seconds to transfer all of the files in the input folder to the three remote workers.

To estimate the total RMI call execution time (Twr) for the remote workers we used the values given by RMI Spy. For each job executed by remote workers there are RMI calls for incoming calls for the methods; getJob(), returnReceipt(), and append(), and outgoing calls for the method getHostName(). According to the data captured by RMI Spy the average combined time for all four nodes is about 20ms. In our test case we have 504 jobs so 504 X 20ms = 10,080ms or a little over 10 seconds total.

It is because of the additional machine and network overhead of the master that the local worker takes longer to process jobs and therefore completes fewer jobs than its remote counterparts. We noticed that in our 4 node test with 504 jobs, worker 1 (local to the master) completed 76 jobs total, worker 2 completed 137 jobs, worker 3 completed 158 and worker 4 completed 133.

Some other measurements were taken by using output statements, for example the Ta (database re-assembly time) variable was measured this way.

4.3 Summary of Actual performance gains

As we expected, the performance of CodeGrid is better than that of CodeMatch, even though for a small value of files it is possible for CodeMatch to have faster running time than CodeGrid. One of the most surprising things about the results of our testing was the minimal impact of the grid overhead on performance.

In a different test, we look closer at the point where CodeGrid performance surpasses CodeMatch. As we can see in Figure 3, when comparing code with less than 250Kbytes, CodeGrid showed no performance increase over CodeMatch due to overhead, but once the data size gets larger than 250Kbytes we can start to see performance improvements. While there is some variation, once we get beyond 250Kbytes CodeGrid begins to outperform CodeMatch consistently and our four-node grid actually becomes more than four times faster at one point.

We conducted four tests running CodeMatch on the same data sets but varying numbers of nodes. These tests show the deviations from the ideal values and are shown in Table 2. The first directory had 4.5 Mbytes of code and the second had 2.6 Mbytes. The tests were comprised of running standalone CodeMatch and running CodeMatch on CodeGrid with 2, 3, and 4 nodes. We expected that since our test took 190 minutes on stand-alone CodeMatch, if each machine can do an equal amount of work in a grid situation, then 2 machines on a grid should ideally take about 95 minutes (190 / 2) and 3 machines should ideally take about 63.3 minutes, etc. This is neglecting any overhead from the grid implementation and network tasks.

Table 2. Results by the number of nodes.

Nodes	Ideal execution time (minutes)	Actual execution time (minutes)	% Deviation
1	190	190	0%
2	95	93	-2.1%
3	63.3	64	1.0%
4	47.5	50	5.0%

Most of our tests were conducted with randomly but similar sized directories. Since CodeGrid copies one complete directory (the compare directory) to each worker prior to processing, and each worker only gets a share of the input directory, we wanted to see how much effect different directory sizes have. We were curious whether the same set of files would generate different results if the compare directory was small with a large input directory or vice-versa. To test this we ran three tests.

Test1 - 3 Mbytes input directory, 1 Mbyte compare directory

Test2 - 2 Mbytes input directory, 2 Mbytes compare directory

Test3 - 1 Mbytes input directory, 3 Mbytes compare directory

As it turns out the directory transfer size doesn't seem to be a significant factor -- far less than we thought it would be. As seen in Table 3, the variation between various size directories in CodeGrid is about the same percentage-wise as the difference between stand-alone CodeMatch runs.

Table 3. Results by Compare Directory Size.

Input directory / Compare directory	CodeMatch	CodeGrid
2 Megs / 2 Megs	58 min 53 sec	16 min 40 sec
1 Meg / 3 Megs	51 min 53 sec	14 min 57 sec
3 Megs / 1 Meg	46 min 11 sec	13 min 19 sec

5. FUTURE WORK

One limitation of our analysis is that it was difficult to completely control all of the factors independently. For example, since data is transferred to three nodes while processing is going on, the measured transfer times include some processing time. In a future study, we would like to put breakpoints into the CodeGrid code to stop processing while transfers are taking place and do similar kinds of work to more precisely isolate each of the factors that contribute to the operation of CodeGrid.

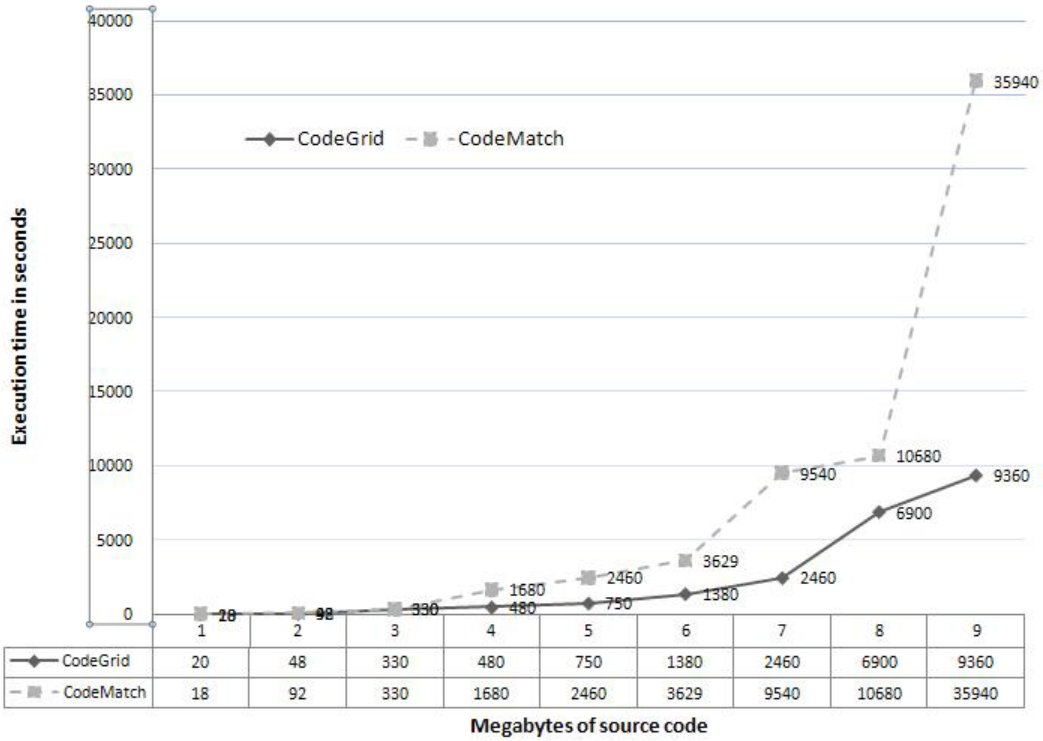
Since the real unit of work on CodeGrid is the "job" we want to run tests based upon number of jobs instead of number of bytes of code. Since jobs can be of any size we will need to do some work to determine the average job size. Some other interesting data could be generated by testing based upon lines of code, number of files, etc. Also testing with combinations of the above may produce some interesting results.

We would also like to do more testing with a faster network and with additional nodes.

6. CONCLUSION

From our testing of our grid we came up with a couple of surprising results. We expected the RMI and JNI bridges to be more significant factors, and we expected the compare directory versus input directory sizes to be a more significant factor due to the bottleneck of transferring the complete compare directory over the network to multiple machines. The overhead of all the network transfers and remote method calls are actually much less significant than we had originally assumed it would be.

Another surprising thing is the percentage of work that the master machine does is much less than expected. We had assumed that since the master and worker on the master are local to each other and the data needed by the local worker doesn't have to be transferred across the network, this worker would do well more than its share of the work. In fact the opposite proved true. Since the master on this machine is catering to three other remote workers as well as the local worker, the local worker in fact does significantly less work than the remote workers.



	1	2	3	4	5	6	7	8	9
CodeGrid	20	48	330	480	750	1380	2460	6900	9360
CodeMatch	18	92	330	1680	2460	3629	9540	10680	35940

Figure 3. Performance Close-Up

We intend to use this study as a guideline for which factors to improve in order to increase the performance of CodeGrid. We hope others can look at what we've done and use it as a model for determining which factors of their systems actually affect performance and which do not.

REFERENCES

[1] Abbas, Ahmar, *Grid Computing: A Practical Guide to Technology and Applications*, Charles River Media, 2003.

[2] Beryozkin, Genady, RMI Spy Plug-in for Eclipse version 2.0. (2002-08). Retrieved Nov 13, 2008, from http://www.genady.net/rmi/v20/

[3] Buschmann, Frank, Henney, Kevlin and Schmidt, Douglas C. *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing, Volume 4*, John Wiley and Sons, 2007.

[4] Goodrich, Michael T. and Tamassia , Roberto, *Data Structures and Algorithms in Java, Second Edition*, John Wiley and Sons, 2001.

[5] Grandinetti, Lucio, *Grid Computing: The New Frontier of High Performance Computing*, Elsevier Science and Technology Books, Inc., 2005.

[6] Hoehn , Timothy and Zeidman, Bob, *Grid-Enabling Resource-Intensive Applications*, DDJ Oct. 10, 2007

[7] Jones, Rick, NetPerf Homepage. (n.d.). Retrieved May 15, 2008, from http://netperf.org

[8] McConnell, Jeffrey J., *Analysis of Algorithms: An Active Learning Approach*, Jones and Bartlett Publishers, 2001.

[9] Richards, John, *The "Abstraction, Filtration, Comparison" Test*, Ladas & Parry Intellectual Property Law, http://www.ladas.com/Patents/Computer/SoftwareAndCopyright/Softwa06.html, July 2002,

[10] Rosen, Kenneth H., *Discrete Mathematics and its Applications*, 5th Edition, McGraw-Hill Book Company, Inc., 2003.

[11] Schopf, Jennifer M. and Nitzberg, Bill, *Scientific Programming, special issue on Grid Computing*, Vol. 10, No. 2, pg. 103-111, August 2002.

[12] Semchyshyn, Yuriy, and Fedasyuk, Dmytro, *Analysis of Computational Complexity and Time Losses of the Distributed Computing Systems*, CADSM'2007, February 20–24, 2007, Polyana, UKRAINE.

[13] Silberschatz, Galvin, and Gagne, *Operating System Concepts with Java*, 7th edition, John Wiley & Sons Publishing Inc, 2007.

[14] Welcome to SWIG. (April 23, 2008) Retrieved May 15, 2008, from http://www.swig.org

AUTHORS

Tim Hoehn is a Research Engineer at SAFE Corporation. He has written technical papers on grid computing and has a pending patent on Internet search. Tim graduated from The University of Washington with a bachelor's degree in software engineering.

Robert Zeidman is a Senior Member of the IEEE and president of SAFE Corporation, the leading provider of tools for comparing and measuring software intellectual property. Among his publications are technical papers on hardware and software design methods as well as three textbooks -- *Designing with FPGAs and CPLDs*, *Verilog Designer's Library*, and *Introduction to Verilog*. He has taught courses at engineering conferences throughout the world. Bob holds five patents and earned a master's degree in electrical engineering at Stanford University and bachelor's degrees in physics and electrical engineering at Cornell University.