Advantages of Parallel Processing and the Effects of Communications Time

Wesley M. Eddy[1]
Ohio University
weddy@prime.cs.ohiou.edu

Mark Allman
NASA GRC / BBN Technologies
mallman@grc.nasa.gov

NASA Glenn Research Center Report Number CR-209455

Abstract

Many computing tasks involve heavy mathematical calculations, or analyzing large amounts of data. These operations can take a long time to complete using only one computer. Networks such as the Internet provide many computers with the ability to communicate with each other. Parallel or distributed computing takes advantage of these networked computers by arranging them to work together on a problem, thereby reducing the time needed to obtain the solution. The drawback to using a network of computers to solve a problem is the time wasted in communicating between the various hosts. The application of distributed computing techniques to a space environment or to use over a satellite network would therefore be limited by the amount of time needed to send data across the network, which would typically take much longer than on a terrestrial network. This experiment shows how much faster a large job can be performed by adding more computers to the task, what role communications time plays in the total execution time, and the impact a long-delay network has on a distributed computing system.

1. Introduction

Distributed computing provides a way to speed up the time it takes to perform a large task by dividing the task between multiple computers that can work simultaneously to complete the job in less time than a single machine requires. The goal of this project was to discover what relationship exists between the number of computers working on a task and the time they take to complete it, and also what effect the communications time between those computers has upon the overall time required to complete the task.

[1] This work was sponsored by the NASA Glenn Research Center.

The recent success of the SETI@Home project [SETI], where data gathered by a radio-telescope is processed by a large network of volunteers, shows that distributed computing can be very beneficial. Parallel processing has the potential to aid NASA in much of its work. For example, a data-gathering robot on the moon could make use of powerful Earth-based computers to analyze its surroundings, and then use the information produced by the analysis to decide where to go and what to focus its sensing devices on next. The same kind of situation could apply to a data-collecting satellite which might need to make decisions based upon the information it gathers

As a test problem we used the process of finding the N-th prime number. Because the primes become more rare as N becomes larger, it takes an increasingly longer period of time for a computer to find the N-th prime as N grows. As a demonstration, we created a program called *sprime* to calculate the N-th prime on a single computer. The results for various N are shown in Table 1 and Figure 1. As the graph illustrates, the problem is not linear as N increases, due to the thinning nature of primes. The time values given in Figure 1 represent the average over 30 runs for each value of N. Details on the workings of the *sprime* program can be found in Appendix A.

N	N-th Prime
250,000	3,497,861
500,000	7,368,787
750,000	11,381,621
1,000,000	15,485,863
1,500,000	23,879,579
2,000,000	32,452,843

Table 1: The N-th prime grows faster as N becomes larger

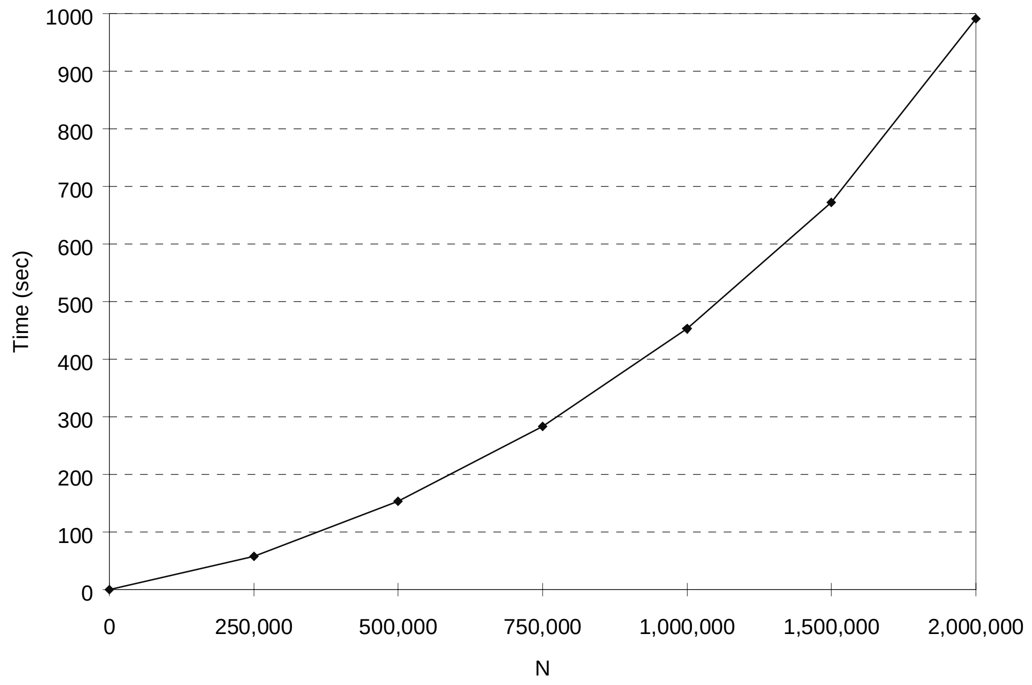

Figure 1: The time taken by one machine to calculate the N-th prime shows a rapid increase as N becomes larger.

2. Adding More Machines

To divide the problem amongst a number of computers we wrote two programs. The first program, *pprime* (parallel prime), is the client program that acts as the scheduler and splits the task into smaller parts which it assigns to other computers. The second program, *pprimed* (parallel prime daemon), is the server program that receives job requests from *pprime*, performs the required task and sends back the results.

The *pprimed* program that runs on each of the server computers receives a request from the client that consists of a range of numbers to check for primality and a maximum number of primes to find. The method used to determine whether or not a number is prime is the same as used in *sprime* and is described in Appendix A. When the daemon has finished checking the assigned range, or when the maximum number of primes has been reached, it stops checking and sends *pprime* a message echoing the range checked, how many primes were found, and the last prime found.

The *pprime* program can be configured to send requests to any number of computers running *pprimed*. For our tests, *pprime* ran on an Ultra-2 running SunOS 5.6, and *pprimed* ran on between one and ten nearly identical Pentium-Pro machines running NetBSD[2]. This was determined to be a good setup, because initial tests using the *sprime* program showed that the Sun machine was slower than any of the NetBSD machines. This could be typical of a real-life scenario in which one less-powerful computer could use its network to take advantage of faster computers that would otherwise be left idle. Figure 2 is a simplified diagram of our network. All network links are 10Mbps Ethernet.

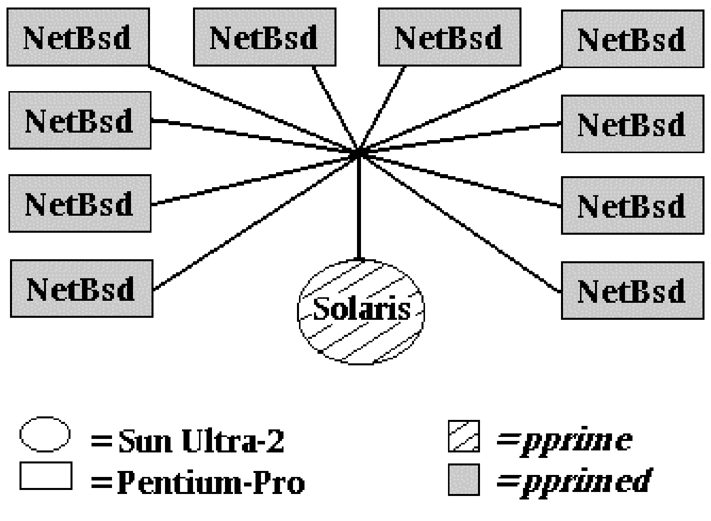

Figure 2: Our servers only communicated with the machine running pprime. All network links are 10Mbps Ethernet.

[2] The NetBSD machines used in our investigation were not identical. For instance, the machines ran various versions of the operating system (1.3.2, 1.3.3, and 1.4). In addition, the machines had differing amounts of RAM. The minor differences in the operating systems and RAM did not significantly impact the performance of the machines in tests with *sprime*, so for the sake of simplicity, they were all considered to be equally capable.

The Transmission Control Protocol (TCP) [Pos81] is used to provide reliable communications between the client and servers. The size of the messages exchanged between *pprime* and *pprimed* is always small enough to fit within one TCP segment and therefore TCP's congestion control algorithms [JK88] do not influence performance. Due to the low amount of other traffic and the close proximity of our computers to one another, the network conditions did not have a large effect upon the behavior of the system.

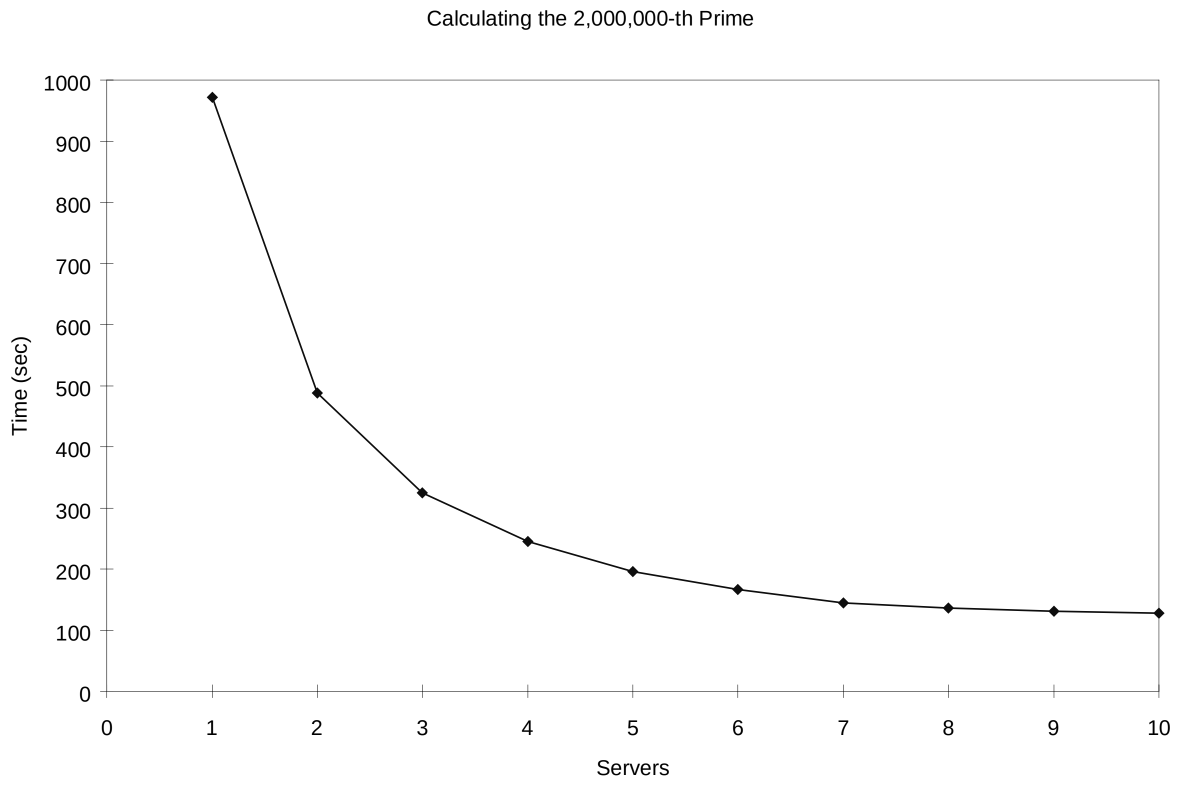

Figure 3: Adding more servers reduced the calculation time

Figure 3 plots the average length of time that our system took to calculate the two-millionth prime over thirty trials as a function of the number of servers. As expected, utilizing more servers caused the job to be completed much faster. The figure also shows that the gains in performance time decrease with every machine that is added. It seems that the curve can be approximated by saying that the time in which a parallel processing system can complete a task equals the time required for one server to complete the task, divided by the total number of servers being used. Or symbolically: $T_x = T_1 / x$, where x is the number of servers, and T_x is the time required for x servers to do the job. This equation is for a system operating under what are considered ideal circumstances, where all the servers are equally capable machines, it takes zero time to send information between machines, and the problem can be divided equally. Figure 4 depicts the percent difference in calculation time between the results achieved in our experiment and what the ideal equation predicts. Figure 5 gauges the benefits we experienced by adding more computers by presenting the decrease in calculation time from the single-server calculation time as a function of the number of servers.

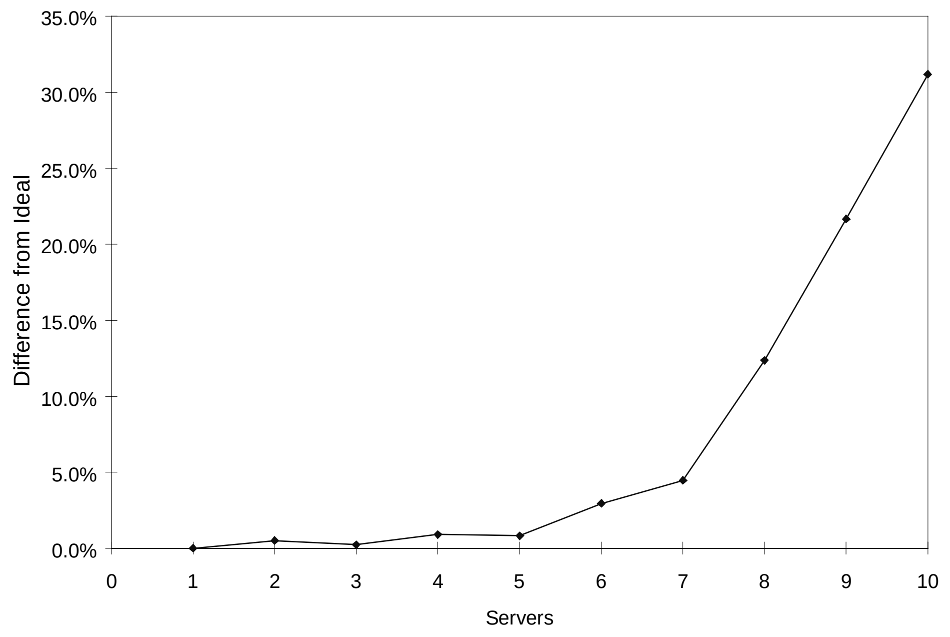

Figure 4: As more machines are added our system performs progressively worse than an ideal system.

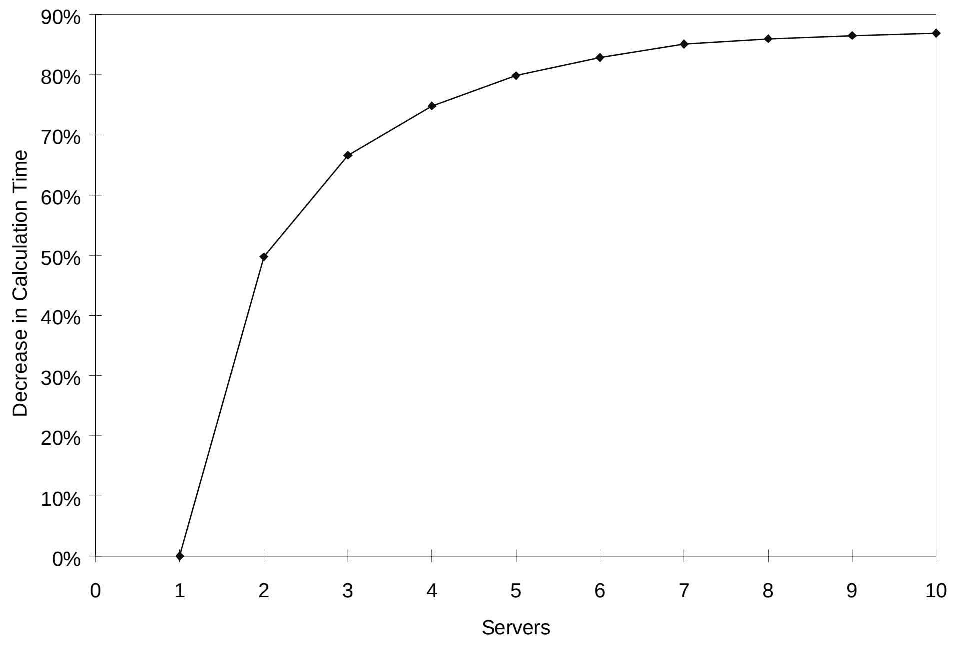

Figure 5: The performance improvement obtained by adding more machines goes down with each machine added.

Figure 4 illustrates that using up to five servers, our system very close to ideal. But, as more machines were added, the results begin to differ substantially the ideal case. Figure 5 also attests to a sharp decrease in calculation time at first, followed by a prolonged leveling off. From these charts it is clear that the contributions of additional machines become less effective as the number of servers is increased. This is due to the client program's inefficiency in dealing with the rising complexity of coordinating the work of multiple servers. We may be able to use a different scheduling algorithm to remedy this problem, however that task was left as future work.

3. Effects of a Communications Delay

With the small network of computers in our lab, which were all within a few feet of each other and connected by 10 Mbps Ethernet, the communication time between computers was negligible. In a real world implementation however, the geographic distances between the client and servers might be much greater than that of our lab's network. Therefore, the time it would take to communicate could be significantly larger. To simulate a longer communications time, the *pprimed* program which runs on each server machine was designed with a variable delay option. The delay could be set to any time period, which the program would wait for between finishing its calculations and sending its results back to the client, thus simulating a longer network delay. Our delay represents the twice the time needed for a trip over the network, since our program only waits once, but two network trips are required per request (one to get the request to the server and then another to return the results back to the client).

The influence of communication time in the total calculation time is dependent upon the number of times the computers must communicate. If the number of messages sent over the network can be kept to a minimum then the network delay will create less of an increase in the total calculation time. In situations where the size of the job is known, it can simply be divided evenly between the number of machines available and there would only need to be one request sent to each machine and one response received back from each, thus minimizing the effect of communication time to one round-trip over the network.

In our case however, *pprime* usually has to send multiple requests to each server. Since the density of the prime numbers decreases, the program cannot predict how big the total job will be, so it sequentially assigns ranges of numbers to all the servers until N primes are found. The size of the chunk of numbers each machine is told to check is simply equal to the number of primes left to find (N minus the number of primes found). Since this would result in a large number of small chunk sizes as the number of primes found approaches N, there is also a built-in minimum chunk-size of 25,000. A result of this algorithm is that as machines are added, less total requests must be sent since each salvo of requests searches a larger total area. We expect that other parallel processing systems where the size of the job is not known beforehand can be expected to show similar behavior. Table 2 lists the average number of requests that *pprime* will send as a function of the number of servers, when calculating the two-millionth prime. As expected the number of requests drops off as servers are added to the system.

Servers	Requests
1	90.0
2	86.0
3	80.9
4	75.2
5	68.2
6	58.9
7	44.7
8	38.3
9	37.1
10	30.8

Table 2: Average number of requests sent (90 trials)

Even traveling at the speed of light, communications between the Earth, satellites, and the moon experience a significant propagation delay because of the large distances that must be crossed. As useful delays for our tests, we chose 0.25 seconds, 0.5 seconds and 2.6 seconds. NASA's ACTS satellite can cover a large area of the Earth, making it a good candidate for use in creating a satellite based distributed computing system. Since it takes roughly 250 milliseconds to get a signal to up to the ACTS satellite and then back down to Earth, 0.25 seconds is used to simulate a situation in which a geosynchronous data-collecting satellite uses Earth-based computers to process its data. The 0.5 second delay approximates two messages traveling over the ACTS satellite link which would be the case if the satellite was used to connect ground based computers. Another scenario we envisioned was having our weaker client computer on the moon. In this case it would use more powerful Earth-based computers to do work as its servers. A delay of 2.6 seconds was chosen to replicate this situation because it takes about 1.3 seconds to get a signal up to the moon or back down. These were two areas we envisioned as possible uses of distributed computing for NASA. Figure 6 shows the average time it took to find the two-millionth prime when all of our servers were set with the specified delays over thirty trials.

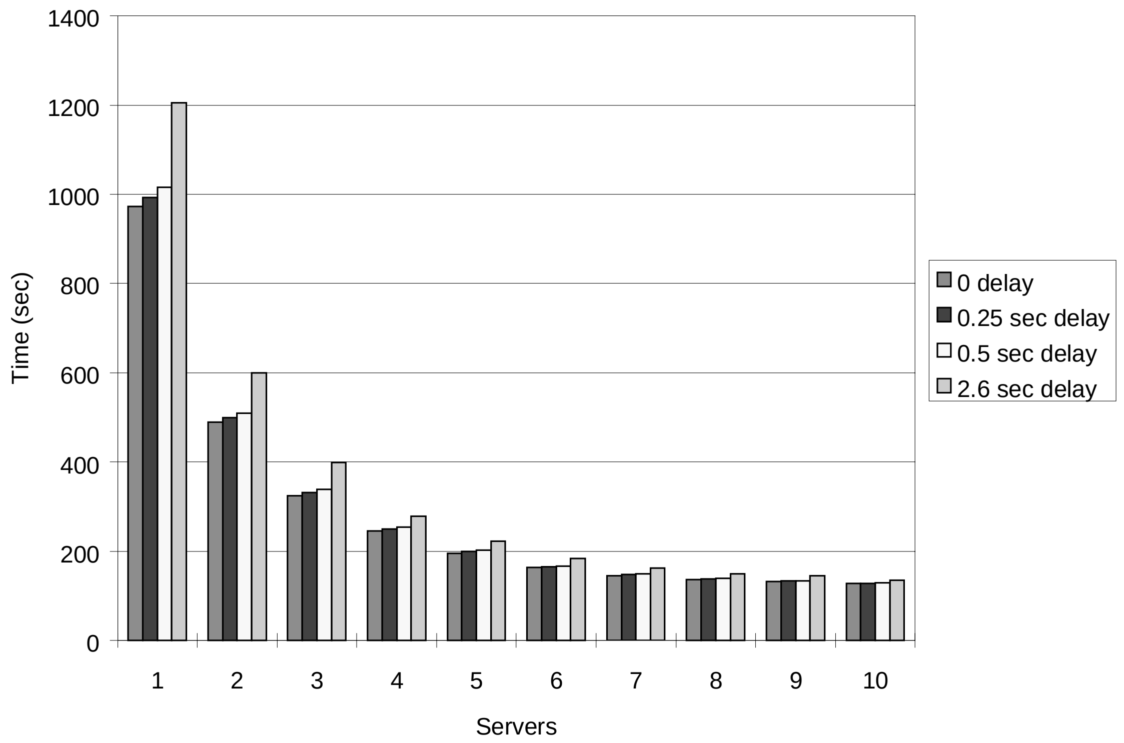

Figure 6: Time to find Two-Millionth Prime when a delay is introduced

From Figure 6, we can see that as more servers are introduced, the extra run-time added by the network delay goes down considerably. This is a direct effect of the assignment algorithm we used in *pprime*, which sends less requests as it works with more machines. Using Table 2, we can predict roughly how many requests will need to be sent to a particular number of servers and dividing the average number of total requests by the number of servers, we can get the average number of requests sent to each individual server. The total effect of the network delay then, is approximately equal to the average number of requests sent to an individual server multiplied by the network delay. Since some machines end up processing more requests than others, a more accurate figure for the total calculation time can be obtained by adding the zero-delay calculation time to the product of the maximum number of requests sent to any one server and the network delay.

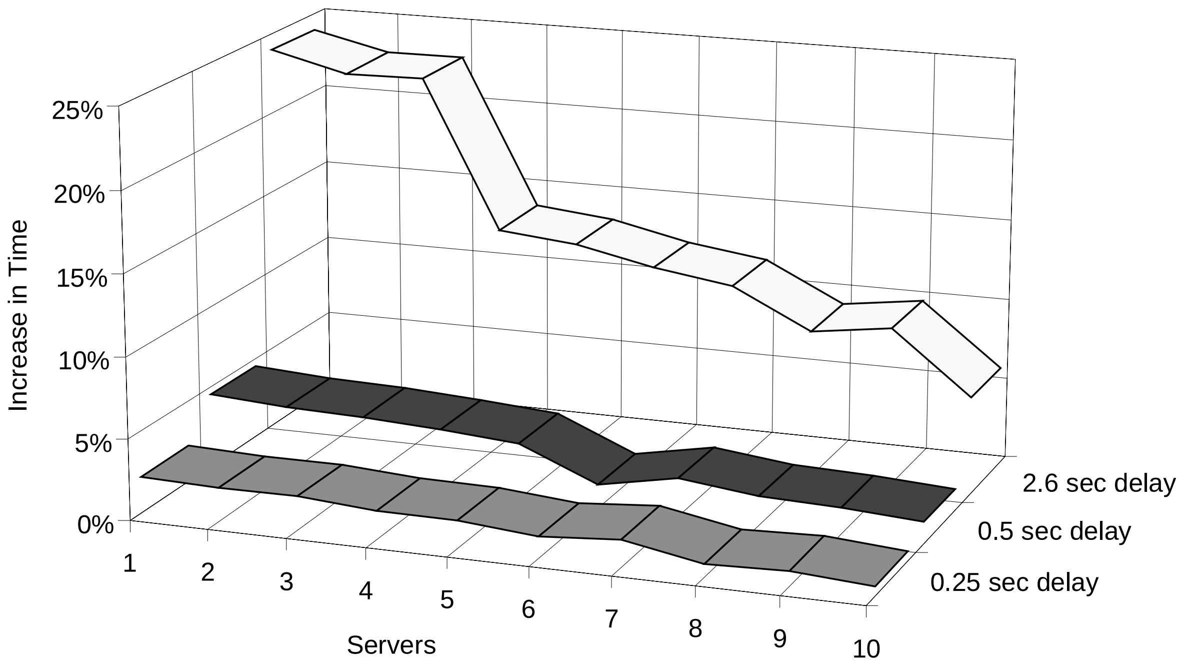

Figure 7: Increase in calculation time caused by delay as a percentage of the zero-delay calculation time**.**

Figure 7 illustrates that increasing the number of machines reduces the negative effects of the network delay. This is, once again, a result of issuing less requests per server being as the number of servers is increased. Figure 7 also indicates that adding servers is more effective at reducing calculation time in longer delay networks than networks with a short delay. This is because in longer delay systems the communications time is a bigger portion of the total calculation time so a reduction in the number of network round-trips has more effect.

4. Conclusions and Future Work

This experiment has shown that it is possible to use several computers in parallel to solve problems that take long periods of time to complete on a single machine, and that by using more computers the total calculation time can be drastically reduced. The results of this experiment also show that for tasks which involve multiple requests over a long-delay network, adding more machines to the parallel processing system can also help to reduce the negative effects of the network's delay. These are important results for NASA because they show that cheaper less-powerful computers placed on the moon, or linked by a satellite connection can effectively use parallel processing techniques with more powerful machines elsewhere on the network.

Future work in this area involves analyzing the effects of even longer delays, such as a Mars-based computer using servers on Earth. It is expected that at a longer delay the size of the task must be proportionally larger for the system to be effective. For tasks involving transfers of large amounts of data, networks with different data-transfer rates should be considered. In a real-world situation, there might also be varying network delays to each server, which should also be analyzed. Changes in these network conditions could result in the need for a different scheduling algorithm. In addition, tests should be performed on a more dynamic network, since competing traffic could have some influence on the system.

Acknowledgments

The authors would like to thank David Juedes from Ohio University for giving us some helpful information about primality testing. Dan Glover at NASA-GRC also provided feedback which was useful in the writing of this paper.

References

[Bres89] David M. Bressoud, *Factorization and Primality Testing*, Springer Vertag, 1989.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, *Introduction to Algorithms*, The MIT Press, 1990, pages 838-844.

[JK88] Van Jacobson and Michael J. Karels. Congestion Avoidance and Control. In ACM SIGCOMM, 1988.

[Pos81] Jon Postel. Transmission Control Protocol, September 1981. RFC 793.

[SETI] The Search for Extraterrestrial Intelligence, http://www.setiathome.ssl.berkeley.edu, 1999.

Appendix A. Checking for Primality

One of the main portions of the programs used in this experiment is the algorithm that checks a particular number for primality. Although there are faster methods for performing this test [Bres89] such as the APR test, the Miller-Rabin test [CLR90], the

Cyclotomy test, and several Monte Carlo methods, the method used in the *sprime* and *pprimed* programs is probably the most obvious and the easiest to understand. Most of the other methods are also intended to be used upon numbers that have hundreds or even thousands of digits, while we were only testing numbers that could be contained in four bytes of memory. Our goal also was not to find the fastest or most efficient way to find primes. Rather, finding primes was simply a test problem used to make generalizations about the behavior of a parallel processing system in a long-delay environment.

A number is considered prime if it is an integer greater than one, and its only factors are itself and one. Any number which is not prime is called a composite, with the exception of the number one, which is considered to be a unit. So, to check a number X for primality, all that must be done is to make sure that none of the integers less X will divide it evenly. One way to speed this task up is to first check to see if two is a factor of X, and then only make sure that odd numbers are not factors. This is possible because if X is even, then dividing by two will immediately show that X is a composite, and if X is odd, then it cannot have any even numbers as factors. Actually, as long as the square root of the number is not a whole number, only the integers less than the square root of X must be checked, since factors come in pairs, and all the pairs whose first term is greater than X's square root will be mirror images of pairs whose first term was less than the square root of X.

When looking for primes, our programs, *sprime* and *pprimed*, skip over even numbers because we know that there are no even primes greater than two. To check an odd number X for primality then, our programs use the C language's modulus operator (%) to ensure that there is no remainder when X is divided by any of the odd integers less than one plus the square root. As soon as a number is found that divides X evenly, X is taken to be a composite and the next odd integer, X + 2, is checked for primality. If no numbers can be found to divide X evenly, then X is considered prime.