Scalable Computing: Practice and Experience, vol. 2, no. 3, November, 1999, pp. 299-311, https://www.scpe.org/index.php/scpe/article/view/141

EXPERIENCES WITH DISTRIBUTED COMPUTATION OF TWIN PRIMES DISTRIBUTION

PATRICK H. FRY[†‡], JEFFREY NESHEIWAT[†§], AND BOLESLAW K. SZYMANSKI[†§]

Abstract. This paper outlines experiences with development of a distributed framework designed for parallel processing of an application using idle processor time on large numbers of heterogeneous computers. The system places no requirements on the performance or availability of the network interconnections. As such, it is most suitable to parallel applications with modest synchronization requirements and a high ratio of computation time to amount of data needed to define the problem domain subspace. We also provide details on our first application using this framework: a parallel computation which counts the distribution of twin primes, calculates Brun's Constant and the maximal distance between pairs of twin primes. Two primes are twins if they differ by two. We have gathered these results for an order of magnitude larger interval than previously computed. We present an overview of our implementation and provide preliminary results and future directions for the system.[1]

1. Introduction. There is a class of computations that requires vast computational resources and yet can be divided into independent subcomputations that do not require any inter-process communication. Examples of such problems are factoring large primes, breaking cryptographic codes, searches through large domains, enumeration of objects with predefined properties, computing statistical properties (averages, moments) of a class of objects in a large search space, etc. Often each subcomputation needs only a limited or constant volume of data to initiate the execution and reports a constant amount of data as the result. For the examples given above, the results would be binary values for searches or a few numbers for factoring, enumerations and moments. In such cases, even if the computation is divided into many pieces, the cost of communicating initial data and results is negligible compared to the cost of each subcomputation. Thanks to these properties, such computations can efficiently use idle processors linked through the Internet.

There have been many attempts to use idle processor time for useful computation [4, 11, 15, 16, 19, 22]. Using the Internet and idle processing time for distributed computation introduces difficulties not found in other parallel systems. The first problem is the Internet itself. Communication performance, in terms of end-to-end throughput, latency, and reliability, is relatively poor. The second problem is the availability of the computers. Each node could leave its "idle" state at any time, making the machine unavailable. This is in addition to the node becoming unreachable due to Internet network outages and transient failures. Hence, the framework must be capable of tolerating failed participating subcomputations. We are interested in large computations that even when parallelized, require months if not years to finish. Therefore, the framework must also be able to restart the computation in case of hardware failures to the entire system (eg. a power outage or server shutdown).

This paper outlines our experiences in applying a distributed heterogeneous environment to computing Brun's Constant and twin primes distribution over an number range two orders of magnitude larger than any previously traversed[17]. Although

[†]Department of Computer Science, Rensselaer Polytechnic Institute, Troy, NY USA 12180 - 3590({fryp, neshj, szymansk}@cs.rpi.edu)

[‡]The author acknowledges support from IBM Corp.

[§]The authors acknowledge support from NSF Grant 9527151

[1]An initial version of this paper was presented at Seventh International Symposium on High Performance Distributed Computing and published in its Proceedings [8].

we used the well known *farmer – worker* paradigm, this specific application to twin primes computation has led to several novel contributions. The relay agent allows nodes behind a firewall to contribute idle cycles. The *SCATTERS* tool is used to start computation on idle nodes. We have added new levels of reliability in the farmer to support the years necessary to complete this computation. We have also developed efficient ways to represent prime numbers and efficiently compute a sum of the large number of inverses with very high precision needed for evaluation of Brun's Constant. The average performance of our application is approximately 5 tera-instructions (5 trillion instructions) per week using over several hundreds of computers.

In the farmer – worker paradigm, the farmer is responsible for assigning work to be processed by its workers and collecting the results from the workers. In its most general form, the farmer – worker method makes no assumptions about the number of workers or the reliability of the connection between worker and farmer. Our design goals were to use as many processors as possible, and to enable a computation to proceed for many months. We focus on:

Reliability: We allow workers to drop off and join the computation at will. The farmer can fail and even checkpoint files can be lost without stopping progress of the computation.

Portability: We minimize memory, computation and system requirements of the worker and restrict the needed tools to basic standard (e.g., using TCP/IP for communication, gcc for the compiler and standardizing endian and data format for Unix and Linux, ensuring source code portability to NT and Windows95/98). Minimal host requirements open the problem up to a greater number of potential contributors.

Similar goals and means were used by the DESCHALL project which cracked the code in the RSA DES Challenge[22].

The majority of existing systems with similar goals impose significant system requirements. For example, *Legion* [11, 12] requires at least two daemons to run on each client and uses a specialized file system. If a client fails, the centralized configuration database must be manually updated if the client cannot be restarted. *Piranha* [4, 3] is built on a tuple-space based coordination language, *Linda* [9]. *Piranha* implements master-worker parallelism but assumes that the master process is persistent and requires Linda compiler and system on each participating machine. *Virtual BSP Computer* library [16] supports BSP computations over a network of non-dedicated workstations. This system migrates processes from workstations which become unavailable and supports scalability and tightly synchronized parallel computations. However, it requires an extended BSP library on each participating workstation and cannot survive a crash on the master process.

Our framework is best suited to parallel applications with modest synchronization requirements. The ratio of computation time to amount of data needed to define a portion of the problem domain space should be high. Some suitable applications would be search or enumeration algorithms. The amount of data returned to the farmer should be small as well. A "code-breaker," or decryption application is a good example[22]. The worker is given a region (interval of integers defined by just two numbers) in which to search for keys and returns a bit indicating "found" or "not found." Some simple performance statistics may also be returned.

We will also describe the first application of our farmer - worker system: computing twin primes distribution. This application was chosen both because it is a good representative of a class of applications which lends itself well to the farmer – worker

paradigm and its importance to number theory[13, 17]. We have already collected this information for an interval an order of magnitude larger than any previously computed and will have another order of magnitude on completion of the project.

A prime number is a positive integer that is evenly divisible by exactly two positive integers: itself and one. Two subsequent odd numbers that are both primes are called twin primes or twins for short. $(3,5)$, $(5,7)$, $(11,13)$, and $(41,43)$ are all twins. The Twin Primes Conjecture proposes that twin primes occur infinitely often[13]. Although it is still not known if the set of twins is infinite, Brun[2] proved that the sum of the reciprocals

$$B = (\frac{1}{3} + \frac{1}{5}) + (\frac{1}{5} + \frac{1}{7}) + (\frac{1}{11} + \frac{1}{13}) + (\frac{1}{17} + \frac{1}{19}) + \cdots$$

is convergent; unlike the divergent sum of the reciprocals of all individual primes. B is known as Brun's Constant [1]. This value has been estimated by summing the inverses of twin primes over ever increasing ranges [1, 7, 17] with the current upper limit of 10^{14}. For this application, our goal is to refine this estimate by enumerating twin primes two orders of magnitude further, up to 10^{16}. In addition to counting the number of twin primes and computing the sum of their inverses, we also find the maximum distance between twins and the location of these gaps. This data is of interest to number theoreticians[17].

This is a formidable computation which would take over 100 years on a single workstation. Parallelism and careful algorithm optimization are necessary to obtain results in more timely manner. The computation is highly parallel and there is no need for synchronization or communication between processes except for providing initial data and gathering results. A distributed environment in which idle processor time is utilized to run this computation is not only feasible but is also a highly efficient and economical means of obtaining the results.

This paper is divided in two parts. In the first part included in Section 2, we describe the architecture and design of each component in our distributed system. The second part, contained in Section 3, provides more details about the twin primes application. Specifically, we outline algorithms used for collecting our results efficiently and accurately. Section 4 shows our current results up to $4 \cdot 10^{15}$. Future work and conclusions are provided in Sections 5 and 6. Finally, Appendix A contains the source code of PWorker, the worker component of the twin primes application.

2. Distribution of Application. We assume that the problem to be executed is easy to parallelize by dividing the problem domain among processes. The framework provides a distributed environment which supports multiple platforms, has low overhead and is scalable. A server is needed to distribute the work to clients (parallel processes) and to collect the results. Client – client communication is not required.

A central server, or farmer, is responsible for assigning work to its workers and collecting results. Workers are free to join and leave the computation at will. A worker requests work from the farmer, processes the work, and returns the results if everything goes smoothly. The worker may also decide to quit the computation before reporting any results or be terminated if the computation takes too long (see Section 2.1.2).

Workers communicate with the farmer via the Internet. However, in some instances, there are firewalls between the worker and the farmer. To allow these workers to participate with minimum security risk, we developed a *relay* (see Section 2.3). The relay collects results from all workers behind the firewall and forwards the data

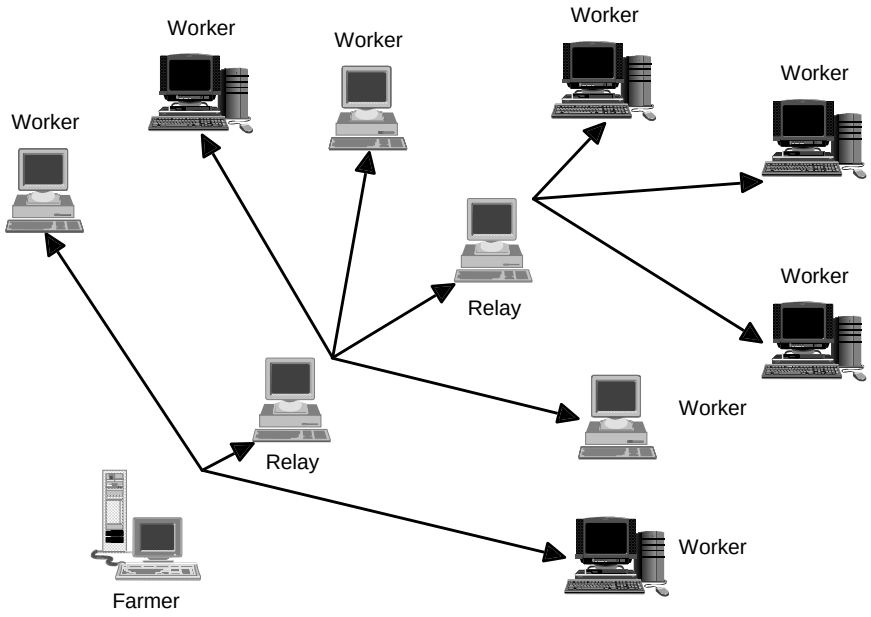


FIG. 2.1. *Communication paths between a farmer and its workers.*

to the farmer. Using the relay, a system administrator need only grant access for the machine running the relay instead of all participating workers. Relays provide *nested parallelism*: the clients of the farmer proceed in parallel, but some of them can be relays which then initiate parallel computation on the workers reporting to them (see Figure 2.1).

Work division is application dependent. For the twin primes distribution, the farmer divides the problem space into *intervals*. The transfer time of data needed to initiate the worker (discussed later) is a few seconds. The computation time of an interval should be minutes or longer to make the communication cost negligible. We have selected an interval size of 10 billion consecutive integers, which takes about 60 minutes on a fast PC or workstation. One million of these intervals need to be processed to reach our goal. The farmer stores the results from each interval, along with the worker's IP address, in a file.

The farmer and worker are written in C and use TCP sockets for communication. These were chosen for reliability and cross-platform support. We use a separate program which works off-line to combine the results from all completed intervals. A worker can open a TCP connection to the farmer directly or through a relay or series of relays. Figure 2.1 shows some possible communication paths. The relay is described in Sec. 2.3.

Message Format There are two main `struct`'s, `interval_type` and `message_type`, which are used to transfer data between farmer and worker. Figure 2.2(a) has the data format for each interval and Fig. 2.2(b) contains the message format. The message `struct` holds the data for up to `MAX_NUMBER_INTERVALS` (currently set to 16) intervals. `message_send(int sockfd, message_type *m)` and `message_recv(int sockfd, message_type *m)` are used to send and receive messages. `sockfd` is a TCP socket file descriptor and `m` is a pointer to the message.

```
/* Results for each interval */
typedef struct {
  long long       firsttwin;     /* First twin in interval          */
  long long       lasttwin;      /* Last twin in interval           */
  long            interval;      /* Interval Number                 */
  long long       maxdist;       /* Maximum distance between twins  */
  long long       maxdist_twin;  /* Starting number for maxdistance */
  long long       term0c, term0h, term0l, term1h, term1l, term2h,
                  term2l;        /* Brun Values                     */
  unsigned long   twincount;     /* Number of twins                 */
  long            time;          /* Interval processing time        */
} interval_type;
```

(a) *Interval* `struct`.

```
/* Message Type Structure */
typedef struct {
  long              mtype;                    /* Message Type               */
  char              identity[IDENTITY_SIZE]; /* IP address XXX.XXX.XXX.XXX  */
  long long         maxv;                     /* Maximum Value to compute   */
  long long         distance;                 /* Size of interval (1e10)    */
  long              maxfp;                    /* Number primes to precompute */
  long long         norm;                     /* Brun calculation constant  */
  long              client_id;                /* Farmer assigned ID number  */
  long              num_intervals;            /* Processed or to process    */
  interval_type     intervals[MAX_NUMBER_INTERVALS]; /* Interval data       */
  long              time;                     /* Total processing time      */
} message_type;
```

(b) *Message* `struct`.

FIG. 2.2. *Message format.*

2.1. The Farmer. The farmer – worker paradigm provides a framework for providing the level of fault tolerance needed to respond to many transient failures on the worker side as well as for recovery from failures on the farmer side. Figure 2.3 illustrates how farmer and worker failures affect different regions of the problem's search space. An interval is the portion of the search space assigned to a given worker. Workers can be assigned multiple intervals at a time. When a worker has completed its work on an interval, or set of intervals, that data is sent to the farmer. The farmer performs a checkpoint operation every 200 messages. In the event of a failure, the farmer can recover by restoring its state from the checkpoint file. Intervals sent to the farmer that have not been checkpointed are lost and simply get reassigned. Most of the workers participate in the computation at idle-time. As a result, workers typically exhibit very frequent transient failures from the farmer's perspective. If the farmer does not get a response from a worker within 6 hours, the worker is timed out and its intervals are reassigned.

The farmer is responsible for assigning intervals to its workers and collecting results (as described in Section 2). The farmer monitors worker performance to maximize efficiency by dynamically changing the number of intervals assigned to each worker and reassigning intervals if a worker fails to return its results within a specified time period.

2.1.1. Identifying Workers. The farmer uses an ordered pair, (x, y), to identify each worker. x is the IP address of the worker's host. y is a unique identification

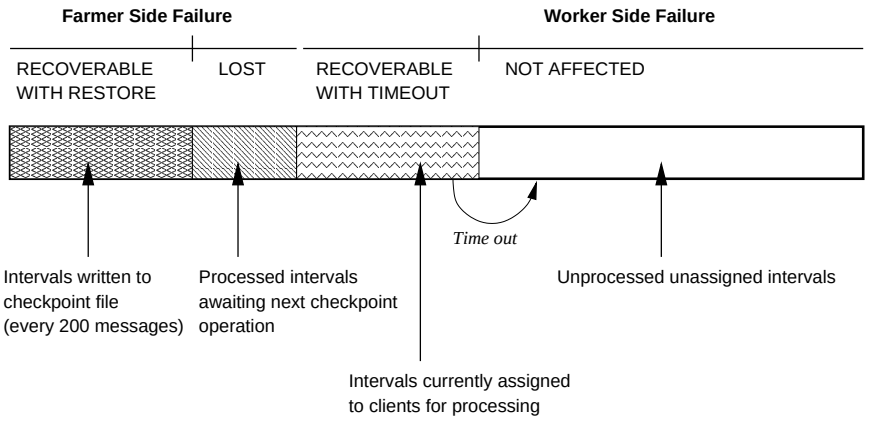


FIG. 2.3. *Segmentation of search space and overview of recoverability options.*

number assigned by the farmer. This enables multiple workers to run on the same host (e.g. multiprocessor architectures). This unique identifier is also useful in failure situations. If the farmer fails and a new one is started, the farmer can differentiate between workers which were processing for the previous farmer and its own workers.

There are at most one million workers needed to finish our computation. Assuming that for each successfully finishing worker, at most 10 fail, we can safely estimate that at most 10 million worker identifiers will be assigned. Each time the farmer is restarted, we add 10^7 to the initial identification number issued to the workers, thereby ensuring that two existing workers will never be assigned the same number.

2.1.2. Worker Timeouts. The farmer can lose contact with its workers for many reasons: network latency, failures on the workstation running the worker, or killed workers. The farmer handles all these problems using timeouts. If a worker fails to report in within a fixed time period, the farmer considers it dead and places the assigned intervals back in the ready queue along with all intervals waiting to be processed. If the worker reports in after this timeout period, it is not assigned any more work and is told to shut itself down. If the worker takes too long sending its finished work or receiving new work from the farmer, the worker shuts itself down. In this way we avoid receiving duplicate messages from the workers or assigning duplicate work.

2.1.3. Checkpoints. For every 200 messages received containing interval results, the farmer saves its state to a checkpoint file. This value of checkpoint frequency was selected as a compromise between the cost of checkpointing and the cost of farmer's failure. Each message contains an average of 3 intervals. Therefore a farmer crash would, on average, cause the loss of only 300 intervals. Each interval which has not yet been processed, either because it has not been assigned to a worker or is still currently being processed, is stored in the checkpoint file. Intervals which have already been processed are stored in another file which contains all completed work. If the farmer fails, it can be restarted in *recovery mode* by reloading its state from the checkpoint file. This enables processing to continue with only a few hours of lost work. Two checkpoint files are kept at all times. If the farmer crashes during the checkpoint process, the prior checkpoint file may be used to restart the farmer. The checkpoint files and processed intervals file are archived daily.

2.2. Workers. The workers perform twin primes computation. Upon startup, the worker connects to the farmer and receives a unique identification number and a limit specifying how far to enumerate twin primes. Using this limit, the worker reads $\sqrt{limit}$ primes from a binary file that contains all primes up to 10^8. The worker is then assigned several intervals of 10 billion over which it must compute the number of twins, Brun's Constant, and the maximum distance between twins.

The number of intervals assigned is dynamically set based on the response time of the worker. A window of 2.5 – 3.5 hours is defined as *acceptable*. Should the worker take more time to return its results, the number of assigned intervals is halved. Should the worker respond too soon, the number of intervals is increased by one . This ensures that the worker will communicate with the server roughly once every 3 hours.

2.2.1. Heterogeneity. Many problems arose when dealing with this mathematically intensive heterogeneous distributed application. Most portability issues revolve around the fact that the program relies on 64 bit integers, or `long long` integer types in C. The ANSI C standard does not presently support this data-type. As a result, many standard library functions had to be extended to support these longer integers while taking into account endian differences across architectures. For example, functions such as `ntohl()` and `htonl()` which convert `long` integers between network and host byte ordering had to be supplemented with `ntohll()` and `htonll()` routines which do the same for `long long` integers.

Other portability issues became apparent when moving to IRIX, AIX, and Windows 95/98/NT platforms. These had to do with signed versus unsigned `chars` being implicit. Furthermore, file I/O issues arose when reading the primes file with the Win32 client. Difficulties with the optimizer on AIX's native C compiler prompted the use of *gcc* for compiling the clients on all platforms. Currently, clients are available on a variety of platforms (Solaris, Sun OS, Linux, FreeBSD, IRIX, HP-UX, Win32, AIX).

2.2.2. Automated Startup of Workers. Some of the challenges involved in running a computation on multiple workers are not matters of numerical or distributed computing, but system-administration issues which arise when trying to use computers which would otherwise be idle. In some cases, the owner of the machine is also the system administrator and can easily decide to start up a worker as a background process. In other cases, we want to take advantage of machines which are being used for other purposes and run workers when these machines are completely idle.

At Rensselaer, workers are run on a large number of public UNIX workstations by taking advantage of a separate project under development, called *SCATTERS* [5]. These public workstations are intended for students use from console only. When a student is logged in, it is expected they will have the full resources of that machine. While these machines were not purchased to solve large-scale numerical problems, they do have the potential to be a large source of cycles when they are idle.

SCATTERS consists of a collection of simple scripts which allow a systems-administrator to start up an arbitrary program (such as the twin primes worker) on a large number of workstations. The scripts are currently specific to Rensselaer's computing environment, but the ideas are simple enough to adapt to other environments.

There is one script which finds out which hosts are currently idle by parsing the output of standard UNIX commands such as `finger` and `rusers`. There is second script which takes a list of hosts and a single UNIX command as parameters. It uses a slightly modified version of `ssh` (as opposed to `telnet`) to log into the specified hosts

and execute the given command. A third script iteratively calls the second script. All of these scripts run on a single workstation, which is the *control station* for all of the SCATTERS processing.

Automated execution is started by running a Perl script which does two things. It starts a program (a worker, in this case), and every 10 seconds it checks to see if a user has logged into the workstation. Upon log in, the script will terminate the program and log off the system. Within seconds, all of the resources of the workstation are available to the console user. A system administrator enters the following command at the *control station* :

```
scatters_loop -count 200 -sleep 1800 -pubibm -pubsgi -pubsun
```

Workers are automatically started on as many as 245 workstations for the next 100 hours, when those machines are idle. The real limit to this `scatters_loop` script is how long the *control station* stays up before it needs to be rebooted (and that has generally been for weeks at a time). This is one of the largest single sources of cycles for the twin primes project.

2.3. The Relay. The relay enables workers to communicate with the farmer through a firewall. This relay receives worker messages, forwards them to the farmer, and sends the farmer's response back to the worker. This has proven useful for *Beowulf* class machines [21] in which only one node is accessible outside the cluster. The relay runs on the gateway node which has access outside the cluster. In a network with a firewall, the relay resides on the firewall.

We are currently prototyping a relay which encapsulates our farmer – worker messages in HTTP[6]. This allows our traffic to pass through HTTP Proxy Servers[6]. We have another relay here at RPI which extracts the messages from the HTTP stream and forwards them to/from the farmer. Using this method, the relay does not have to reside on the firewall. This is useful for locations where it would otherwise be difficult, for security or administrative reasons, to install our relay on the firewall. Instead this specialized relay resides on any computer on the same side of the firewall as the workers it's servicing.

3. Twin Primes and Brun's Constant. The program collects information about twin primes. Instead of storing each twin, we enumerate them. The $4 \cdot 10^{12}$ twins already found would require approximately 8 bytes of storage each, or 32000GB, if stored individually. Even with a more efficient storage method that would use one byte per pair (e.g. storing the difference between the twins), 4000GB would still be required.

Brun [2] proved the sum of inverses of all twins is convergent, in contrast with the divergent sum of inverses of individual primes. Using Hardy and Littlewood [14] an estimate of Brun's Constant can be computed from the sum of inverses of all twins up to a certain value, x. The accuracy of the estimate can be calculated from [1]. Prior to this work, the most accurate estimate of this constant was done by Nicely who reported the value $1.9021605778 \pm 2.1 \cdot 10^{-9}$ [17] based on $x = 10^{14}$. Increasing x to 10^{16} will increase the accuracy to $\pm 6.24 \cdot 10^{-10}$.

In addition to enumerating the twins and calculating the sum of their reciprocals, the maximum distance between twins is stored along with its location in each range of numbers analyzed. This information is of theoretical interest in number theory.

3.1. The Algorithm. A memory efficient method is required for storage of the 6 million primes used for the sieve and the range of numbers being analyzed. Section 3.1.1 describes these data storage techniques. Section 3.1.2 details the algorithm used

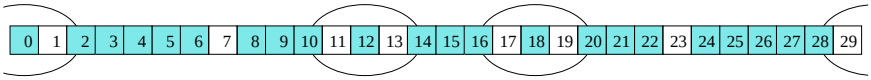


FIG. 3.1. *Potential primes and twins modulo 30. Multiples of 2, 3 and 5 are grayed out and potential twins are circled.*

to keep the calculation of Brun's Constant accurate.

3.1.1. Data Representation. The program uses the Sieve of Eratosthenes method to locate twin primes. All primes up to the square root of the limit must be known. For 10^{16}, all primes up to 10^8 are stored.

As the program has evolved, so has the method of storing these primes. Initially, an array of integers (4-bytes per prime) was used. There are just under 6 million primes smaller than 10^8, so 24MB was used. Such a memory requirement would have been a serious restriction on the number of machines which could participate in the computation. This memory requirement was reduced by using just one byte per prime by storing the difference between the current prime and its previous prime; thus using only 6MB. This is the method used by Nicely [17].

Currently, one bit for each *potential* prime up to 10^8 is used. Eliminating multiples of 2, 3, and 5 leaves only 8 potential primes for each 30 numbers. As shown in Figure 3.1, the potential primes are 1, 7, 11, 13, 17, 19, 23, and 29, modulo 30. Memory usage is reduced to $10^8/30 \approx 3.3$MB, halving the memory requirement from storing the distances.

There are 8 potential primes modulo 30, but only three pairs of these can be twins (circled in Figure 3.1). It is impossible for any number which is 7 or 23 modulo 30 to be a twin because the numbers distance 2 below and above are always multiples of 3 and 5. In prior implementations each odd number was treated as a potential twin member, 15 bits per 30 numbers. Now only 3 bits are used for every 30 numbers. And all multiples of 2, 3 and 5 are automatically removed.

With all multiples of 2, 3 and 5 eliminated, the program then eliminates all the multiples for all the remaining primes up to 10^8 (this process is referred to as shooting out). The smaller the prime, the more multiples it has to shoot out. We use a method which speeds elimination of multiples of initial k primes, $p_1, p_2, \ldots, p_k$. First we shoot out multiples of $p_4, p_5, \ldots, p_k$ for a range $g = p_1 * p_2 * \ldots * p_k$, called a *shooting gallery*. Then, we divide the prime twin enumeration interval into shooting galleries of size g. Note, that all multiples of the first k primes will occupy the same bit positions in each gallery (because a is divisible by p_i if and only if $a + m * g$ is divisible by p_i). Hence, by copying a bit pattern of the first shooting gallery to each subsequent one, we do not need to consider the first k primes when eliminating multiple numbers.

Currently we use galleries of size

$$30 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 = 223092870$$

which occupies about 2.8MB of memory. This is the largest gallery which can be indexed by four-byte integers. This allows us not to shoot out multiples of 7, 11, 13, 17, 19 and 23 which are eliminated by copying a bit pattern of the first gallery. Every new gallery is created by bit copying from this bit pattern.

Consider $7^2 = 49$. This number would be shot out in the bit pattern (eliminating a potential 17, 19 twin pair). Table 3.1 shows some multiples which will automatically

$223092870 + 49$	$=$	223092919
$(2 \cdot 223092870) + 49$	$=$	446185789
$(3 \cdot 223092870) + 49$	$=$	669278659
	$\vdots$	

TABLE 3.1
Multiples of seven are eliminated by copying the first gallery.

be shot out of future galleries.

The described method saves processing time. While only the first nine primes are eliminated out of the millions of primes which have to be examined, these nine are the ones which would have required the most processing had they been manually shot out of each gallery. Eliminating more primes would require eight byte integers for indexing in the galleries and as a result would increase processing time.

To estimate the savings attributable to the described method, we notice that the work that shooting out multiples of a prime causes is proportional to this prime's inverse which is proportional to the number of prime's multiples in a large range. Hence, the work of eliminated initial primes is $\sum_{i=4}^{9} \frac{1}{p_i} = 0.4652\ldots$, where p_i denotes i-th prime (counting 2 as the first prime). We do bit pattern copying instead, where each step copies 32 bits representing 320 numbers, so this work is $\frac{1}{320} = 0.003$. We verified that $\sum_{i=10}^{100,000} \frac{1}{p_i} = 1.41\ldots$. $p_{5761455} = 99,999,989$ is the last prime $\leq 10^8$, so the last prime that our algorithm will process. Hence, by approximating primes from $100,000$ to 5761455 by an arithmetic series starting with $p_{100,001}$ and with the step $\ln p_{100,001}$ we obtain $\sum_{i=10}^{5761455} \frac{1}{p_i} \approx 1.64$ as the estimate of the work needed to stamp out the remaining primes.

Hence, without eliminating initial 9 primes, the computation time would be proportional to 2.11 and with elimination this factor drops to 1.64 indicating about 25% gain. Estimating from the source code and the time it takes to compute a single range of 10 billion numbers, the program spends about 10 instructions or 20 cycles to shoot out a multiple, so the total computation will require 300 tera-cycles (trillion cycles – we are using floating point operations sparingly, so we cannot measure our progress in Teraflops). So far, over 100 tera-cycles have been used. Cycles which would otherwise have been idle.

3.1.2. Accuracy in Computation of Brun's Constant. The main difficulty in this part of the computation is to maintain the numerical precision of the result. Potentially, each floating point operation contributes an error to the result, so after each inverse operation, we know the result with the relative error $\pm 2^{-55}$ and each addition adds an error of the same magnitude. We want to add inverses of primes up to 10^{16} and there are about 10^{12} of them, so the error would potentially add up to leave only four digits of precision out of the initial 16 decimal digits of precision. Of course, with high probability, the error will be smaller because some errors will cancel each other out. Still this is not a satisfactory solution.

Fortunately, we can use already known values of Brun's Constant for the initial primes up to $b = 10^{14}$. Hence, we need to compute only an addition to Brun's Constant representing the sum of inverses of twins bigger than b. This addition is of magnitude of at most 10^{-3}, thus adding three digits to the number of significant digits of the result (Brun's Constant is about 2).

To further increase precision, we can use long division operations to produce a

32 digit result. The procedure for such long division uses two floating point divisions to create 16 digits of the result each and four long integer multiplications to compute a precise remainder from the first division. Hence, it is a rather costly operation. However, the resulting precision of the final result (Brun's Constant addition) is 34 digits if the multi-word addition is used.

To avoid direct summation, to increase the number of significant digits and to decrease the number of long divisions, we can use approximations in which some of the summations are replaced by a single operation of multiplication and addition of some small corrective value. Only few digits of the corrective value will impact the result, so loss of precision in the correction is irrelevant.

Let $[s, s+r]$ where $r < s/2$ denote the range in which we compute a partial sum of Brun's Constant, n denotes the number of twins in this range and $s+t_1, s+t_2, \ldots, s+t_n$ are the subsequent twins. Of course $0 \le t_1 \le t_2 \ldots \le r$. Then, for some $0 < c < r$ the partial sum which we want to compute is

$$P(s,r) = \sum_{i=1}^{n} \frac{1}{s+t_i} = \frac{1}{s+c} \sum_{i=1}^{n} \frac{1}{1+(t_i-c)/(s+c)}$$

However,

$$\frac{1}{1+x} = 1 - x + x^2 \ldots + (-x)^k \ldots$$

and

$$|\frac{1}{1+x} - \sum_{i=0}^{k} (-x)^k| = |\frac{x^{k+1}}{1-x}| < 2|x^{k+1}|$$

for $-1/2 < x < 1/2$ which is satisfied for $x = (t_i - c)/(s+c)$ because $c < r$ implies $x > -r/(s+r) > -r/s > -1/2$ and $c > 0$ yields $x < r/s < 1/2$.

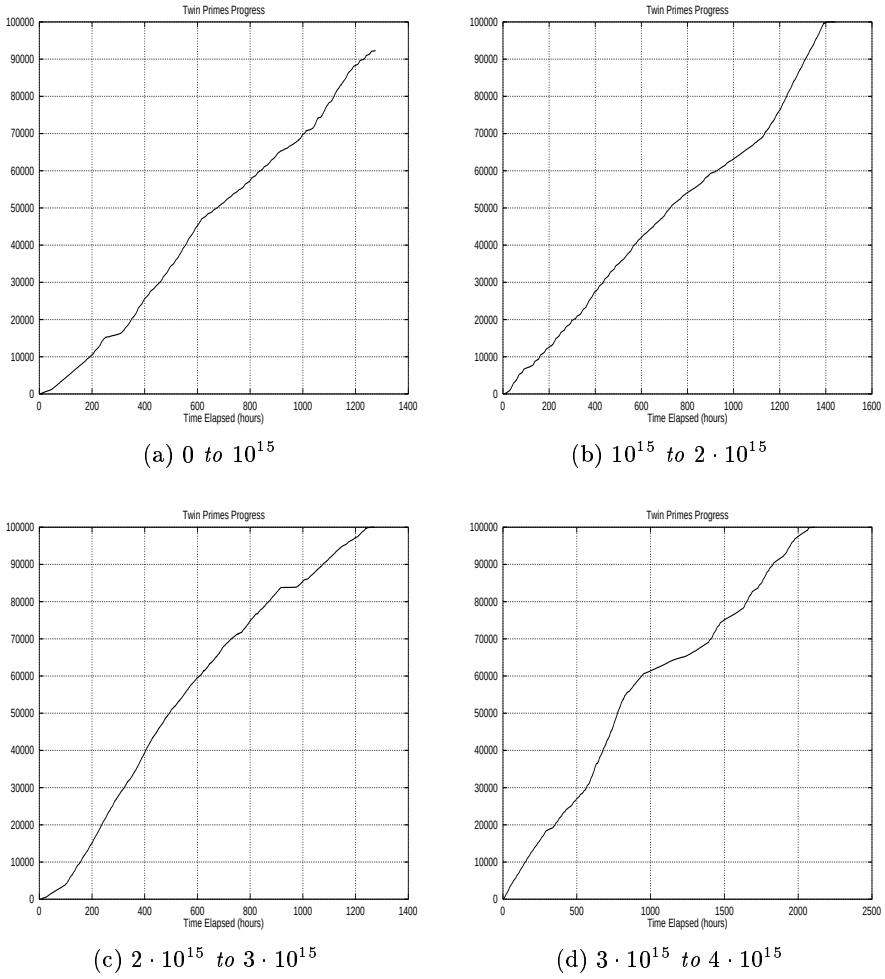
We will keep partial results as integers (the floating point results of some operations will be multiplied by the proper power of 10 and converted into a long long integer). This representation adds digits of precision to the result if the computed partial sums are added using multi-word arithmetic. The exact number of additional digits will depend on the absolute value of the result, which on the other hand is a function of b.

$$P(s,r,c) \approx \frac{1}{s+c} \left(n - \frac{1}{s+c} \left(\sum_{i=0}^{n} (t_i - c) - \frac{\sum_{i=1}^{n} (t_i-c)^2}{s+c} \right) \right)$$

The first term's numerator (i.e. n) can be computed exactly and using long division described above we can get 32 digits of precision of the result.

The second term will be minimized if we select an integer $c = \sum_{i=1}^{n} t_i/n$. The sum $\sum_{i=1}^{n} t_i/n$ can be computed exactly for $r < 10^9$ in long long integer. With this choice of c, the second term is less than n/s^2, so it can be represented as an integer. The third term is smaller than $n \cdot r^2/s^3$ and for $r < 10^8$ the summation can be computed exactly as an integer.

Finally, we ignore all the higher terms, which contributes to the numerical error of the result. The magnitude of this error can be estimated as at most twice the sum of fourth terms which in turn is less than $2 \cdot n \cdot r^3/s^4$.



(a) 0 *to* 10^{15} (b) 10^{15} *to* $2 \cdot 10^{15}$

(c) $2 \cdot 10^{15}$ *to* $3 \cdot 10^{15}$ (d) $3 \cdot 10^{15}$ *to* $4 \cdot 10^{15}$

FIG. 4.1. *Progress of twin primes computation (time in hours vs. number of intervals, 1 interval* $= 10^{10}$*).*

Using Nicely's result[17], we can set $b = 10^{14}$ and the final precision of the sum of inverses will be at least 30 decimal digits for $r = 3003$. For convenience, we select the size that is the divisor of the gallery size. Using the alternative method of performing long division for each pair of twins, we could get 34 decimal digits of precision, but with 3000 times more long divisions executed which would dominate the computation time of the whole algorithm.

4. Results. The largest range for which all twin primes were found has been growing together with the capabilities of computers [1, 17] and currently is 10^{14} [17]. As of the writing of this paper, spare cycles on over 325 computers have been used to compute twin primes up to $4 \cdot 10^{15}$, Thus far, 4,205,621,313,841 twins have been found. Additionally, the maximum distance between two twins, up to $4 \cdot 10^{15}$, is 28842 and occurs at (2797282815481499, 2797282815481501). The largest twin found by our program to date is (3999999999999191, 3999999999999193). These results were obtained over the course of 9 months and continue to be updated.

Figure 4.1(a) shows the progress made from start to 10^{15}. The slope flattens out between 200 and 400 hours into the computation. This is the result of a server crash which had to be restarted in *recovery mode*. The slow ramp up is the result of workers being started up manually. The slope decreases again 600 hours into the computation when 16 off-site *Beowulf* nodes were suspended for benchmarking purposes.

Figure 4.1(b) shows the progress from 10^{15} to $2.0 \cdot 10^{15}$. During this time, the server has not crashed at all. At approximately 100 hours into the computation, one of the relay agents went down and is evident by the change in slope. As the computation progressed, the slope slowly decreased followed by a sharp increase after 1100 hours. This trend is due to the academic calendar at Rensselaer Polytechnic Institute. The majority of workers run on public workstations when they are idle. As the semester progressed, the workstations were in use most of the time and rarely idle. The sharp increase in slope coincides with the end of classes hence the idle workstations were tasked with computing twin primes.

Figure 4.1(c) shows progress made from $2.0 \cdot 10^{15}$ to $3.0 \cdot 10^{15}$. During this interval there was one server crash approximately 900 hours into the run which lasted 48 hours. Despite this crash, the most progress was made thanks to additional cycle contributions.

Figure 4.1(d) shows the progress at time of printing of this paper, from $3.0 \cdot 10^{15}$ to $4 \cdot 10^{15}$. Note that this interval will take longer than the previous one. This is attributed to lowered contribution of academic computing resources as the school year begins at many universities.

To date, the farmer has crashed on two occasions after $1.0 \cdot 10^{15}$. Both crashes were attributed to external factors such as power failure. During this time, one relay crash occurred which only affected workers at one site. The relay and its associated workers were restarted with no impact on the farmer. On average, a server crash has occurred every 100 days. Total server uptime is 208 days. After a server crash, the server is simply restarted in *recovery mode* and continues assigning intervals from where the last checkpoint file was created. Initial computation from 0 to $1.0 \cdot 10^{15}$ were confirmed using Nicely's results up to $1.0 \cdot 10^{15}$ [18].

5. Future Work. Extensions to this work involve furthering the existing twin primes computation and also applying this fault tolerant distributed computing framework to similar problems. To maximize utilization of idle workstations, our attention has been focused on the Windows 95/98/NT platform. This operating system is unique in that workers cannot be started and stopped remotely in the same way as under UNIX. We are looking into two solutions to this problem. The first involves initiating the worker when the screensaver engages and suspending when the screensaver disengages. The other solution will require developing a specific service that listens on a port for worker start and stop requests that come in remotely, as is currently the case for workers running on other platforms. The advantage of the latter solution is that it can be applied to a large class of mathematical, enumeration, and search problems.

A newer version of the twin primes worker is currently being developed and tested that provides the following additional functionality:

- **Real-time feedback.** The worker will output characters to standard output representing the number of galleries processed and the elapsed time since communicating with the farmer.
- **Delivery of architecture and host information.** In addition to sending twin prime data, workers will update the server with information about the

architecture and operating system running the worker.

These changes were motivated by overwhelming demand from the scores of users contributing cycles by running workers on their computers. These enhancements will also provide valuable information for debugging purposes. One of the issue not addressed yet, is the security. The users are willing to participate in this computation only if they trust that the worker is not mischievous and will not corrupt their machines resources. To built this trust, we provide an access to the source code that is relatively straightforward. For more complex applications, it will be important to implement workers in Java as an applet that can only execute in the safe JVM environment.

The mechanism that we have employed for enumerating twin primes is generic in that the farmer assigns portions of a problem to many workers while maintaining a high degree of fault tolerance and ensuring recoverability. The problem specific computations are done by the worker. This mechanism is similar to that employed by the DESCHALL project[22]. Both mechanisms are designed to allow an arbitrarily large number of clients to join the computation while making no assumptions as to reliability. The major difference is that DESCHALL was searching for a specific key to crack the code whereas we are enumerating a large number of twin primes and using these values in subsequent calculations. Our workers return a significant amount of data to the farmer for analysis. Given the nature of these two problems, i.e., search vs. enumeration it is interesting how both projects, although developed independently, converged on a similar mechanism for parallelizing their respective algorithms. Hence, our future plans include making the farmer and its accompanying communication protocol generic so workers can be developed to solve a range of computationally intensive problems in number theory, encryption and other areas.

We are also considering extending our protocol to support parallel computations which require synchronization steps. The distributed environment we are using is more volatile (workers are frequently joining and leaving the computation pool) than those traditionally used for this type of parallel computation. We propose to use replication of the computation to minimize the effect of individual workers being dropped out of the worker pool.

Consider the following example. In the first stage of the synchronous step n workers will initiate computation and by a time-out, $p \leq 1.0$ fractions of them will finish. The remaining unfinished $n(1-p)$ intervals are then computed by survivors with about $\frac{1}{1-p}-1$ workers assigned to each interval. As a result, with high probability $1-(1-p)^{\frac{1}{1-p}-1}$ the step will finish or the system will initiate the third stage with even higher replication level, and so, a higher probability of finishing. For example, in our computation $n = 200$, $p = 1/5$, so in 99.84% cases we will be able to finish in two stages, yielding a respectable efficiency of 50% compared to non-synchronous case. A maximum independent set problem is an example of a synchronous parallel computation [10] with a low communication requirement if proper data distribution is used [16]. We plan to use our system with a protocol extended for synchronous computation to this problem.

6. Conclusions. We have described our experiences with the farmer – worker approach to computing twin primes distribution. Although this is a well known distributed computation paradigm, our experience applying this framework to twin primes has led to several novel contributions: the relay which can be used to facilitate participation of nodes behind a firewall, the SCATTERS scripting tool used to start computation on idle nodes, new methods of reliability in the farmer to support the lengthy execution time, and efficient methods for representing prime numbers in mem-

POS	INTERVALS	DOMAIN / GROUP
1	165163	rpi.edu (247 hosts)
2	65901	cs.rpi.edu (40 hosts)
3	46957	momentum.cs.rpi.edu
4	33842	jpl.nasa.gov (33 hosts)
5	26292	nycap.rr.com (5 hosts)
6	9944	stu.rpi.edu (10 hosts)

TABLE 6.1
Top 6 contributors.

ory and computing Brun's constant. In addition to application of existing techniques and these contributions, we have already surpassed the current record for largest interval searched by an order of magnitude. Our goal is to compute all twins another order of magnitude to 10^{16}. At the current rate with approximately 325 workers, this goal should be reached in a year.

6.1. Call for Cycles. With your help, we can reach our goal sooner. Workers are available for the following platforms:

- AIX4
- HP-UX
- Linux (elf) and FreeBSD
- IRIX and Cray Origin 2000
- SunOS 4.1.3 and Solaris 2.5 and 2.6
- Windows 95/98/NT

All of the information needed to contribute spare CPU cycles to this effort is available at: *http://www.cs.rpi.edu/research/twinp*

To date, the greatest contributors are ranked in Table 6.1. The `rpi.edu` machines include the public computing labs at Rensselaer. `cs.rpi.edu` include the Computer Science departmental workstations and `momentum.cs.rpi.edu` is a 12 processor Cray Origin 2000. `jpl.nasa.gov` are workers run at NASA / JPL's High Performance Computing Systems and Application Group. The remaining entries represent students' personal computers, their contribution was significant during the school year and has dropped off as the semester drew to a close. INTERVALS represent a range of 10 billion.

6.2. Acknowledgments. Special thanks go to Dr. Thomas Nicely whose collaboration has proved invaluable to this project. The authors also wish to thank the following people for their support and assistance: Garance A. Drosehn (RPI-CIS) for automating startup of workers on public workstations using SCATTERS[5]. Nathan Schimke (RPI-CS) for automating startup on CS department workstations. Thomas Cwik (JPL) for access to HPCC resources, Josh Wilmes (RPI-CS) for assistance in porting to AIX, Robert Dugan (RPI-CS) for assistance with porting to Win95/98/NT.

Thank go to everyone who has supported our efforts by donating spare CPU cycles or by providing useful feedback, in particular: David Armstrong (RPI), James Bonanno (RPI), Frank Brodsky (APO), Jonathan Chen (RPI), Will Coleda (UT), Erik Collin (EIS), Barnaby Court (RPI), Steve Czetty (RPI-JPL), Dan Daglas (NYU), Brett Hogden (RGE), Mark Hulber (RPI-CS), Scott Jackson (Brown), Thomas Jennings (RPI), Simon Karpen (RPI), Tim Klink (UT), Mat Maessen (UT), Kristopher Nasadowski (APO), Matthew Schnee (Digital), Yuan Shi (Temple), Chris Stevens

(RPI), Kim Stevens (RPI-CS), Dave Thompson (RPI).

REFERENCES

[1] R. P. Brent, *Irregularities in the distribution of primes and twin primes*, Math. Comp., 29 (1975), pp. 43–56.

[2] V. Brun, *La série 1/5 + 1/7 + 1/11 + 1/13 + 1/17 + 1/19 + 1/29 + 1/31 + 1/41 + 1/43 + 1/59 + 1/61 + ... où les dénominateurs sont 'nombres premieres jumeaux' est convergente ou finie*, Bull. Sci. Math., 43 (1919), pp. 124–128.

[3] N. Carriero, E. Freeman, D. Gelernter, and D. Kaminsky, *Adaptive parallelism and Piranha*, Computer, 28 (1995), pp. 40–49.

[4] N. Carriero, D. Gelernter, D. Kaminsky, and J. Westbrook, *Adaptive parallelism with Piranha*, Tech. Report 954, Yale University, 1993.

[5] G. A. Drosehn, *SCATTERS - a Simple Cool Admin Tool To Everywhere Run Something.* http://www.rpi.edu/~drosehn/Projects/SCATTERS.html.

[6] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee, *Hypertext Transfer Protocol – HTTP/1.1*, Network Working Group, IETF, Jan. 1997. RFC 2068.

[7] C. E. Fröberg, *On the sum of inverses of primes and twin primes*, Nordisk Tidskr. Informationsbehandling (BIT), 1 (1961), pp. 15–20.

[8] P. H. Fry, J. Nesheiwat, and B. K. Szymanski, *Computing twin primes and brun's constant: A distributed approach*, in Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing, Chicago, IL, July 1998, IEEE Computer Society, pp. 42–49.

[9] D. Gelernter, M. Jourdenais, and D. Kaminsky, *Piranha scheduling: Strategies and their implementation*, Tech. Report 983, Yale University, 1993.

[10] M. K. Goldberg and D. L. Hollinger, *Database learning: a method for empirical algorithm design*, in Proc. Workshop on Algorithm Engineering, 1997.

[11] A. S. Grimshaw, A. Nguyen-Tuong, and W. A. Wulf, *Campus-wide computing: Early results using legion at the University of Virginia*, Tech. Report CS-95-19, University of Virginia, Mar. 1995.

[12] A. S. Grimshaw, W. A. Wulf, J. C. French, A. C. Weaver, and P. F. R. Jr., *A synopsis of the Legion project*, Tech. Report CS-94-20, University of Virginia, June 1994.

[13] R. K. Guy, *Unsolved Problems in Number Theory*, Springer–Verlag, 2 ed., 1994.

[14] G. H. Hardy and J. E. Littlewood, *Some problems of 'Partitio Numerorum' III: On the expression of a number as a sum of primes*, Acta Math, 44 (1922), pp. 1–70.

[15] G. Judd, M. Clement, and Q. Shell, *The DOGMA approach to high-utilization supercomputing*, in Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing, Chicago, IL, July 1998, IEEE Computer Society, pp. 64–70.

[16] M. Nibhanupudi and B. K. Szymanski, *Runtime support for virtual BSP computer*, in Proc. Workshops at 12th Intern. Parallel Processing Symposium, Springer Verlag, 1998.

[17] T. Nicely, *Enumeration to 1e14 of the twin primes and Brun's constant*, Virginia Journal of Science, 46 (1996), pp. 195–204.

[18] T. R. Nicely, *Private communication.* See also http://www.lynchburg.edu/public/academic/math/nicely/gaps/gaps.htm.

[19] J. Pruyne and M. Livny, *Interfacing Condor and PVM to harness the cycles of workstation clusters*, Future Generation Computer Systems, 12 (1996), pp. 67–85.

[20] M. J. Quinn, *Parallel Computing: Theory and Practice*, McGraw-Hill, Inc., 1994.

[21] T. Sterling, D. Becker, J. Salmon, D. Katz, P. Angelino, D. Ridge, and J. Lindheim, *How to build a Beowulf: A tutorial*, Oct. 1997. Presented by the Center for Advanced Computing Research, California Institute of Technology.

[22] R. Verser, *The $10,000 DES challenge.* http://www.frii.com/~rcv/deschall.htm.

[23] A. Y. Zomaya, ed., *Parallel & Distributed Computing Handbook*, McGraw-Hill, Inc., 1996, ch. 5.

Appendix A. Source for PWorker `main()`.

```
int main(int argc, char *argv[]) {

  int sockfd, i;
  message_type message;                    /* Message for farmer comm.   */
  int start_time, end_time;                /* Timers                     */
  int done = 0;

  init_message(&message);                  /* Initialize message         */

  /* Open Connection to Farmer*/
  if ((sockfd =tcp_client_connect(argv[1], atoi(argv[2]))) < 0)
    error("Error establishing client socket");

  /* Send Request for work to farmer*/
  message.mtype = RTW_REQUEST;             /* Ready to work              */
  message = hton_message(message);         /* Host to network conversion */
  if (message_send(sockfd, message) < 0)   /* Send request               */
    error("Error sending request");

  /* Receive Response */
  if ( message_recv(sockfd, &message) < 0) /* Receive response           */
    error("Error receiving work order");
  message = ntoh_message(message);         /* Network to host conversion */
  tcp_close(sockfd);

  /* Worker Initialization */
  start_time = time(NULL);                 /* Start timer                */
  setup(&message);                         /* Initialize worker          */
  end_time = time(NULL);                   /* Stop timer                 */

  /* Send First Gal Request */
  message.mtype = GAL_REQUEST;             /* Request some work          */
  message.time = end_time - start_time;    /* Send time to set up worker */

  /* Send Request */
  message = hton_message(message);
  if ((sockfd =tcp_client_connect(argv[1], atoi(argv[2]))) < 0)
    error("Error establishing client socket");
  if (message_send(sockfd, message) <0)
    error("Error sending request");

  while(!done) {                           /* Main computation loop      */
    /* Receive set of Galleries to process */
    if ( message_recv(sockfd, &message) < 0) /* Receive set of intervals   */
      error("Error receiving work order");
    tcp_close(sockfd);
    message = ntoh_message(message);

    /* Process Galleries */
    if (message.num_intervals <= 0) {
      /* No more galleries left on farmer to process, shutdown worker      */
      done = 1;
      continue;
```

```
    }

    start_time = time(NULL);                /* Start timer                */
    run(&message);                          /* Process work in message    */
    end_time = time(NULL);                  /* Stop timer                 */

    /* Request next set of galleries */
    message.mtype = NXT_REQUEST;
    message.time = end_time - start_time;
    /* Send Request */
    message = hton_message(message);
    if ((sockfd =tcp_client_connect(argv[1], atoi(argv[2]))) < 0)
      error("Error establishing client socket");
    if (message_send(sockfd, message) <0)    /* Send results and           *
                                              * request more work          */
      error("Error sending request");
  }

  tcp_close(sockfd);

  exit(0);
  return 0;
}
```