Prime Factorization Protocol*

Claire Badger
CSCE 689
Air Force Institute of Technology
claire.badger@afit.edu

Andrew Davis
CSCE 689
Air Force Institute of Technology
andrew.davis@afit.edu

Albert Taglieri
CSCE 689
Air Force Institute of Technology
albert.taglieri@afit.edu

Chris Voltz
CSCE 689
Air Force Institute of Technology
christopher.voltz@afit.edu

***Abstract*—Prime factorization is a highly challenging mathematical problem for large numbers. Solving this requires significant work that takes too much time to be considered useful on single-process systems. The difficulty of the problem quickly rises beyond the capability of these single-process systems, leading to computationally infeasible tasks. This difficulty of prime factorization for large integers reaches a level that is leveraged by security protocols to create reliable systems. Despite this hardness, there exists techniques that attempt to lessen the near impossibility of factoring these large values. The Prime Factorization Protocol is one such system. Here, prime factorization takes a distributed approach, leveraging the computational power of multiple machines in order to complete the task. This system is set apart from others through the ability to use multiple algorithms in order to factor more rapidly.**

I. Introduction

Prime factorization of large integers is a task that currently has no efficient solution to solve for. Additionally, the current best algorithms in this field are probabilistic so they are not guaranteed to always return the solution if one exists. That adds a balancing factor of how many times should the test be run before declaring the number to be prime or trying a different run of the algorithm. The Prime Factorization Protocol aims to leverage principles of distributed system to move towards a more efficient approach in this problem. Here, worker nodes work together using a number of different algorithms, to compute the prime factors of large integers. The system utilizes a hybrid, object-based, cluster architecture in order to efficiently solve prime factorization of large integers.

A user interface mirrors clients connecting to the system. Upon connection, users are provided a list of algorithms to employ against a large integer, as specified by them. The prime factors of this value are then computed and shown to the user. The PFP leverages C++ for computationally heavy operations and for communication between the user and nodes, as well as inter-nodal communication.

The following paper provides an overview of the PFP. *Background* outlines the process that takes place in the PFP, walking through user input on the user endpoints through processing on the node endpoints and back to results displayed to the user. *System Design and Methodology* discusses concepts, architecture, design considerations, and justifications of the system. Finally, *Results* covers an analysis of the system.

II. Background

This background will provide a brief run-through of the system. At the user endpoints, one parameter is requested: a large integer in which prime factors are to be found. The assumption is that this integer is a multiple of two large primes as figuring out the solution in that case has the most applicability. The user can input any values, which will be packaged by the system as work requests, to be worked on. This list is maintained at the user endpoint, with no more than one work request per user being sent to each node at a time. In other words, if there is 1 user and 4 nodes, then there can be 4 work orders out being worked on at that time.

Once a node endpoint receives a work request, it is stored in a queue. At each node endpoint, n work requests can be held, where n corresponds to the number of user endpoints connected to the node. Using the algorithm specified in the work request, the node will find the prime factors of the provided values, and returns the results to the user. Once computed, the node endpoint caches the results should the value be sent again. Additionally, the values are eventually propagated over the other node endpoints. Once the user receives the computed values, work requests as sent to other nodes is terminated.

There were several important distributed systems concepts that helped to achieve this task and drive the design decision enumerated above. One such concept is grid computing. Grid computing is a type of distributed system that leverages the use of multiple machines (that may or may not be similar in hardware and operating systems design) that all access the same shared resource on the system to accomplish a goal. The shared resources can include objects, processing power, memory, and data storage. It can use the power of parallelism to distribute work across different computers on the network which then update the shared resource based on the goal.

This is similar thinking to multi-threading, but instead of threads, the parallel work is being sent to machines which create a new process which might or might not create more than one thread. The advantage that grid computing has is that instead of the speed of the job being limited to the number of cores on one machine (in the case of multi-threading being used to achieve parallel computing), the job is only limited by the number of machines and the combined power of their cores, as well as how parallelizable the job is. In this way, grid computing can be a major advantage when it comes to costly jobs that can be parallelized.

Fault tolerance is a major concern on any distributed system as well. In order to maintain a dependable system, it has to be available, reliable, safe, and maintainable. Because there are multiple systems that run the worker code available to use, the whole entire system should be available as long as there is at least one worker node that is running. Reliability can be maintained by the proper communication channels being set up between the servers. Although the servers are hardware that is given to us and there can't be much done in the care of crashes on the server, checks in place that can manage malfunctions of nodes can improve the reliability of the systems.

At the very least, it is important that fault-silent errors can be converted to fault-stop errors. This could be implemented through "heartbeat checks" sent from one node to each of the other nodes regularly, which is essentially just a simple message that requires a response back. If a response is not given within a certain time frame, then it can be assumed that the node that is not responding has failed. So in this case, once the nodes find that one node has stopped functioning and communicate that with the user, so that it can resend the information that was sent to the faulty node so that information is not lost and not put into the overall calculation. This retains the reliability while also ensuring the system is maintainable.

III. System Design and Methodology

PFP's architecture leverages many different key architectural design principles in order to create a system that is robust, reliable, efficient, and secure. The system designed and outlined in this paper consists two main components: a client-side that simulates the user interface, and a server-side that manages incoming requests and performs prime factorization, hereafter referred to as users and nodes, respectively.

In this section, the overall architecture of the PFP will be outlined, covering user and node components, their relation to one another, and the relation among inter-nodal communication. Work requests will be covered including their purpose, communication method, and format. Following is a brief section outlining the replication of data and it's consistency. Fault tolerance and programming language implementation complete this section.

A. Architecture

The architecture of the PFP is comprised of a hybrid, object-based approach in order to efficiently segregate work between each component. The objects match to the components outlined above, each one with a completely disjoint purpose than the other. The method calls between each object take place over a network, passing data to be worked with. This data is passed directly from the users to nodes. Communication is split up between a client side (user endpoints) and a server side (node endpoints), as can be seen in figure 4. The nodes operate in a peer-to-peer connection to maintain relevant information on the network topology, thus the hybrid architecture. The following sub-sections will elaborate on each component.

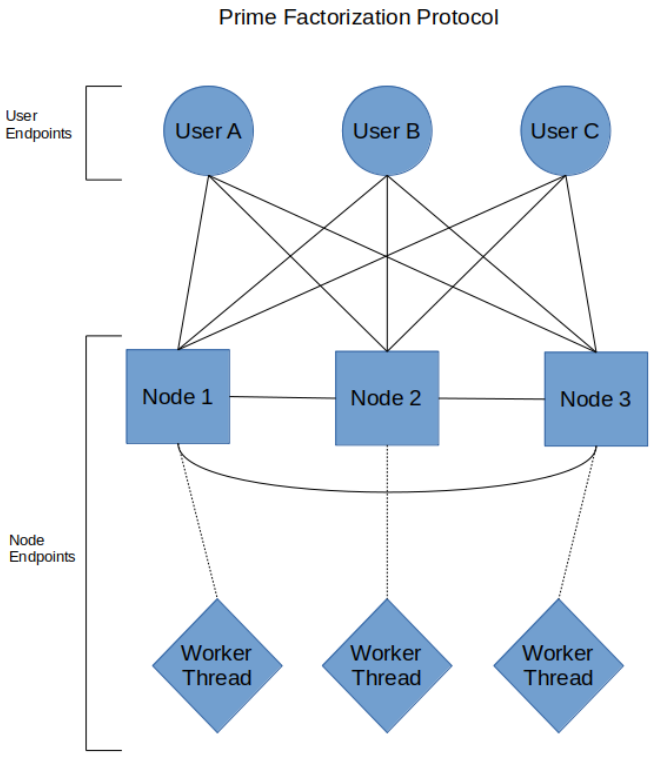

Fig. 1. The PFP architecture. Solid lines represent connectivity, dash lines represent part of those connected to

1) User Component: In this system, the user component has two main purposes: I/O operations, and pseudo-process management of the nodes. The former simply gets integer values to factor from the interface along with the algorithm to do so with. This information is packaged into a PFP-specific work request, outlined later. Once received, this data moves to the second purpose of the user component; communication

with the nodes.

The user maintains a list of active nodes that can be connected to and requests work to be done. This list can be updated throughout the process in order to connect to more nodes and thus speed up the prime factorization process. This update is done by receiving information from currently connected nodes, each of which maintains a topology of the network and informs of new node connection. In addition to currently connected nodes, the user endpoint maintains a queue of requested work and a list of currently executing tasks, as seen in figure 2.

Once a work request is completed by the node, the user then sends out the next work request in the queue. Since some algorithms require multiple runs to find the prime factorization of a number, the currently executing tasks list maintains which node is completing which task as any given point. Therefore, depending on the algorithm, once a user receives a completed work order, it may need to send additional orders on the received value.

As any node replies with a completed task, the user sends a *STOP* command to other nodes that are also working on the problem. This *STOP* informs the nodes receiving it that a solution has been found and further work on that task order can cease. Once prime values have been found by the nodes, the user is informed and task execution can either continue, per the user's request, or end.

It is important to maintain transparency to the end user. To facilitate this, output is suppressed on the user end so that the user doesn't have to look at where they are connecting.

2) Node Component: Nodes outline the worker functionality of the system. These sit at some geographically separated location from the user, waiting for work requests for come in. Here, three main purposes exist: communication between the user, communication between other nodes, and prime factorization. Communication is covered in the *Communication* section.

Prime factorization by nodes utilizes C++. When work requests come in, the node maintains a queue of currently connected users along with their request, as seen in figure 2. Each node will have at most one work request from each user (where this pipeline is maintained at the user endpoint). Once a task is complete, the node sends the results back to the user and begins the next task in the queue. Should there not be any tasks, the node remains idle with the exception of the heartbeat and network updates, covered in *Communication*.

3) Hybrid Communication: As outlined in the previous two sections, the design of the system combines both client-server principles as well as peer-to-peer. The former is done through the communication of users and nodes. Users need not worry about any other user on the network, and only require a solid communication line between the node(s) they are connecting to. They will, at any given time, connect to as many nodes as requested by the system so as to increase computational power, though this aspect will not be seen by the physical user. The peer-to-peer aspect plays a role only in inter-node communication on the server level as will be outlined further into this paper.

B. Architectural Justification

The PFP is broken up into two distinct components using an object-based approach due to the obvious modularity present in the system. That is, a user component and node component best represent the ability to efficiently communicate and get work done.

The nature of an object oriented architecture gives an advantage in this system with regards to the algorithms. If a new algorithm is added, there is much less to change in PFP and it can be added without affecting the code for the current algorithms. Likewise, if a more efficient implementation for one of our algorithms needed to be added, then we could do so without editing the other algorithms currently in use.

Each user endpoint maintains their queue of work requests so as not to overload the already computationally heavy work being done on the node endpoints. This also allows a persistent connection between users and nodes to ensure results are filtered to the correct endpoints.

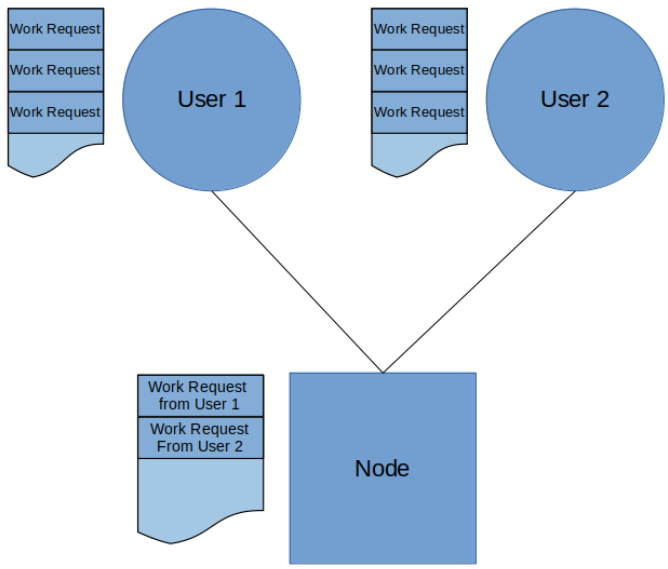

Fig. 2. The PFP architecture. Solid lines represent connectivity, dash lines represent part of those connected to

C. Communication

The PFP employs a hybrid communication method, using peer-to-peer connection as well as client-server connection through the use of TCP connections. The former is used in inter-nodal communication, allowing nodes to maintain an accurate topological mapping of the network in its current

ALG	ARGC	ARG0_LEN	ARG0_DATA
0	1	32	1A52F ... 78B4

Fig. 3. In this example, the Prime Sieve algorithm is used, there is one data value being sent of length 32 bytes, with the value 0xB478 ... F521A.

state. There are two ways this is done: a heartbeat method and a data-sharing method. Every minute, each node sends a heartbeat informing all connected nodes that they are still up and running. Every fifteen minutes, each node sends details of themselves to all connected nodes that includes their status and newly found data as outlined in the *Replication* section.

Work requests, as specified by user endpoints, are sent using PFP specific formatting, as outlined in figure 3. Here, *ALG* specifies the algorithm to use when factoring. Algorithm specific flags are found in table I, *Algorithm Flags*.

TABLE I
ALGORITHM FLAGS

Value	*Algorithm*
0	Pollard's Rho
2	ECM
3	Quadratic Sieve

ARGC outlines the number of values to compute, where each value is given as a length, followed by the value itself. *ARG0_LEN* specifies the length of the value. Finally, *ARG0_DATA* is the value to factor, formatted in little-endian.

The client-server approach is used between the user endpoints (clients) and the node endpoints (server). Users need not worry about connections with other users, nor inter-connectivity between nodes. They simply connect to n nodes in order to request work be done. User endpoints do maintain a list of all reachable nodes in order to distribute work as widely as possible. This list is updated upon connection from a node, which in turn is maintained through the heartbeat method employed above.

D. Communication Justification

TCP connection is used on all communication channels due to the large amount of data being sent and the importance of entire numbers making it to their destination. Should a partial number arrive anywhere at anytime, the validity of the computed result will be incorrect, thus displaying wrong data back to the user. Receiving partial data here corresponds to receiving just parts of the large integers being factored, leading to completely wrong numbers being given to the both the user and the workers should partial information be transmitted.

Nodes communicate with one another in order to create a robust system that smoothly and effectively hands potentially downed nodes. This concept is further evaluated in *Fault Tolerance*.

The PFP data packages were designed in such a way that allows future development while limiting redesign of communication. That is, should future expansion be made, an RPC can easily be employed using the specifications outlined above.

E. Replication and Consistency

Replication occurs between node endpoints upon information dissemination. This data corresponds to requested numbers to factor and their prime factorization upon completion. By replicating this data, there can be simple lookup tables prior to computing prime factors of such large numbers, saving resources and time. Should a value not be found in the table, it is computed and shared between other node endpoints in the system.

Consistency is maintained through an eventual model. That is, should a request for work from any user endpoint not come in to any node endpoint, nodes will eventually converge to consistent states. This is due to the inter-nodal communication outlined above in which data information is shared every fifteen minutes.

F. Fault Tolerance

Fault tolerance is a major concern on any distributed system. It is vital to create a system that is available, reliable, safe, and maintainable. Each of these is maintained in the PFP due to the inter-connectivity of nodes on the server side.

The system is made fault tolerant through design practices described in prior sections. Namely, node endpoint failures or stops are tracked and communicated to user endpoints. Should a node fail at any point, the heartbeat protocol catches this and informs other connected nodes and connected users. Upon realization that a node has failed, user endpoints will no longer send work requests to this node. Should the downed node come back online, the user eventually receives this information from other nodes they are connected to and can again send work requests to the node.

Redundancy in work requests being sent to multiple nodes allows for continued work as well, despite a faulty node endpoint. Since the user sends similar requests to multiple nodes, they are guaranteed a result unless the entire system crashes. The implementation here is simply an example utilizing only three node endpoints. In full deployment, there will be many more, meaning a system crash is indicative of problems beyond our control.

G. Programming Language Implementation

With prime factorization being a mathematically heavy problem, especially in the sense of large numbers, C++ has

been employed in the PFP. Other languages (such as Python) were tested for connectivity of endpoints, but ultimately C++ was defaulted to for system-wide use. This is attributed to overall system performance and speed in maintaining the singular language.

In order to handle large numbers, the library gmp was selected. While, C++ data types have size constraints, gmp can allow PFP to deal with numbers as large as necessary without having to change code. However, gmp allows PFP to scale almost indefinitely, providing it has algorithms that are effective at the larger numbers.

IV. Results

Number Factored	Time taken (s)
2491	2.166615618
143081	2.161948273
36574511	2.159060191
810413431	2.205360794
6265039103	2.159058025
10866186077	2.208508722
47986845481	2.198133004
123462282263	2.235958375
252231959429	2.155657956
1038811397557	2.165508165
64301794795831	2.207783587
6426212544756457	2.201140113
642827783896348493	2.204525649
813179197227970681	2.167924242
81317159771617207733	2.215621350

Fig. 4. Table of Results from running PFP

The above table summarizes the results from running PFP on numbers that are the result of two primes being multiplied together. The last two were products of two 10-digit primes. The system can scale to handle larger numbers. However, the quadratic sieve code that we used, which is the best algorithm currently included in PFP for that size number, was from another source and PFP had difficulty cracking numbers of that size. If we were to continue development, we would look for another implementation to use or base our implementation off of. However, our system scales fairly well to deal with this challenge and continued work could yield even better results.

The results could vary from trial to trial as PFP uses random inputs to the algorithms for starting values so sometimes better values are used earlier than in other runs.

V. Conclusion

Prime factorization of small numbers is a difficult task. Once these numbers become sufficiently large, the difficulty in computing their prime factors become a task with no known efficient solution. Single-process approaches prove useless in this endeavor. To combat this, the Prime Factorization Protocol aims to scope the problem from a single-process to multiple processes.

The protocol, as outlined and defined in this paper, utilizes multiple principles in designing distributed systems to ensure a number of key points are maintained. A hybrid architecture is implemented to ensure transparency in user interaction as well as fault tolerance in node connectivity. The PFP maintains inter-connectivity through heartbeat messages sent between node endpoints, leading to a fault tolerant system that essentially creates itself.

Communication is handled through the sending of PFP specific work requests. These allow for any connected node to complete any request, should the parameters fit. This flexibility on both the user end and the node end allows for future expansion including new algorithms to be employed, more users and nodes to run, and outdated code to continue usefulness over time.

Finally, future work includes expanding the system to be comprised of more worker nodes so as to allow for more user connections. A larger system provides more nodes to perform work, and therefore more data to be collected and processed for users. Additionally, adding different algorithms for prime factorization to further compare results and speed of algorithms against one another. Since the algorithms that are best at handling the large numbers typically have situations that they excel at solving, adding more options provides faster coverage. Additional future work involves actually implementing the heartbeat packet that goes through the nodes so that they know the network and can adapt to worker nodes dropping in and out and share knowledge as necessary. The caching concept is not currently implemented in our system, but that would be a useful addition. More security and error checking could be helpful however, since it is a prime factoring system, we deemed these as less important due to their already being other comparable services available. Employing these provides a more robust system that can be expanded beyond just prime factorization.

Appendix A

PFP was developed and deployed on Linux. Multiple servers were used in developing, testing, and deployment of the system. Therefore, multiple IP addresses are managed.

Compilation and Execution

Source code can be found at:

- https://github.com/adtag4/Prime.git

Provided is an install script and a Makefile. First, run the install script to install *libgmp3-dev* and *gmp-doc*. Once done, run *make*.

The paper, in AFIT Thesis format, fits page requirements. If this format is requested, please let one of the authors know at the aformentioned emails.

APPENDIX B

The authors and team behind PFP have put in over 350 combined man hours to date. If there are problems connecting to nodes please let the team know as they are operated on a server and thus the team would need to restart them if they go down.

You can reference the RFC in documentation for technical details for PFP.