Augmenting the Remotely Operated Automated Mortar System with Message Passing

Zachary J. Ramirez, United States Military Academy
Raymond W. Blaine, United States Military Academy
Suzanne J. Matthews, United States Military Academy

Abstract. This paper looks at how the Message Passing Interface (MPI) can assist a prototype U.S. Army vehicle mounted mortar launcher system called the Automated Direct Indirect Mortar (ADIM). The ADIM's capabilities are augmented by the Remotely Automated Mortar System (ROAMS) by enabling fuzes to be set remotely. The performance of the initial ROAMS prototype, a threaded Python server using Raspberry Pis, is limited by Python's Global Interpreter Lock (GIL). In this paper, the prototype is redesigned using MPI and the C programming language to dramatically improve the efficiency of the system.

1. Introduction

The U.S. Army is developing a system called the Automated Direct Indirect Mortar (ADIM) system [1]. The ADIM is mounted to a high mobility multipurpose wheeled vehicle (HMMWV) and fires belt-fed 81mm mortar rounds. This system increases the capability of the conventional mortar by adding some key features. The most important features are the speed at which it can fire, stabilize, and re-fire, and the ability to conduct "shoot and scoot" missions. Shoot and scoot missions provide a key advantage, allowing the mortar operators to fire and leave the area before an enemy can acquire their location via radar and counter fire. This increased capability of lethality, provided by the rate of fire, survivability, and increased mobility, is essential to maintaining our technological superiority on the battlefield. However, the ADIM cannot be used to its full potential because of a limitation with current 81mm mortar rounds. Current mortar rounds must have their fuzes manually set prior to being loaded into the ADIM. This requires the system to be unloaded if the desired fuze setting is not available in the magazine, severely limiting the speed of operation.

In order for the ADIM system to reach its full and future potential, 81mm rounds must have several key qualities that current munitions lack. First, they must be "smart", accepting GPS locations allowing for flight alteration and precision fires. Second, the round's fuze setting must be able to be set and changed remotely. These capabilities do not currently exist. Additionally, the system must have a user-friendly interface, the rounds must be initially powered from the battery of a HMMWV, and they must retain power for the duration of flight.

An undergraduate capstone project at the United States Military Academy (USMA) called Remotely Operated Automated Mortar System (ROAMS) attempted to tackle these shortcomings during the 2013-2014 academic year. In the first iteration of this multi-year project, the focus was to optimize the fuze setting remotely.

The ROAMS system uses Raspberry Pis to simulate the hardware of the magazine server and individual smart-round mortars. The Raspberry Pi [2] is a popular, credit-card sized single-board computer (SBC) that retails for $35.00. The choice of Raspberry Pis enables the design team to cheaply prototype the hardware that would eventually be included in a custom integrated circuit (IC) for the smart rounds. Due to its ease of programmability, Python was selected as the language of choice to program the magazine server. The magazine server communicated with the smart-round Raspberry Pis using Python threads, simulating a classic "client-server" system.

The initial prototype worked well when interfaced with a test system consisting of four mortar rounds. However, the program's responsiveness and usability decreased as the number of fuze clients increased. This is especially problematic as the ADIM mortar chamber has a 20-round capacity. The responsiveness issue is particularly troubling because on the battlefield, every second counts. This work attempts to tackle this problem by redesigning the ROAMS system to support efficient remote fuze-setting.

This paper analyzes a redesign of the ROAMS magServer-client system using the Message Passing Interface (MPI) [3] and the C language. In order to test the scalability of the ROAMS system, a cluster of 21 Raspberry Pis was built to simulate the full ADIM system. We measure the performance of the MPI system and compare it to the client-server system implemented in Python. The MPI implementation results in a time reduction of up to 90 percent of the original Python prototype, suggesting that MPI is a promising technique to improve the speed of remote fuze setting.

The rest of the paper is organized as follows. Section II gives a detailed over view of the original ROAMS prototype, its key limitation, and motivations to transition to C and MPI. Section III describes the redesign of ROAMS to use MPI. Finally, preliminary results and conclusions are presented in Sections IV and V respectively.
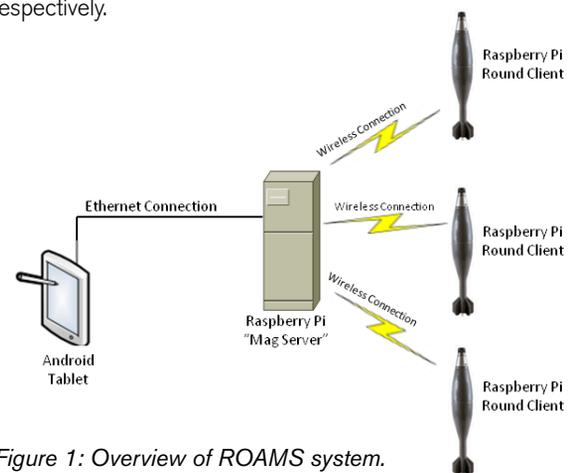

Figure 1: Overview of ROAMS system.

2. Overview of the ROAMS System

Figure 1 illustrates a simplified network layout of the ROAMS system prototype. A Raspberry Pi is used to simulate the microprocessor needed in each smart-round, and a central Raspberry Pi acts as the magazine server (or magServer). An Android tablet serves as an interface for soldiers to control the magServer and, by extension, the

mortar rounds themselves. The magServer as originally designed has four jobs:

- Establish connection with every mortar and store its state information.
- Establish connection with the tablet and provide an On-Demand list of mortars at its disposal.
- Accept and relay all commands from the user to the rounds.
- Provide setting verification on all rounds in its control.

The Android tablet accomplishes this by displaying the list of available mortar rounds for the user to select, and allows them to change the fuze setting and or GPS data. The magServer begins by setting up both a wired and wireless network interface. The server then connects to the user's Android interface device on the wired interface, and listens for mortar round connections on the wireless interface. When a mortar round is connected, the magServer adds it to the inventory and transmits to the tablet an updated list of mortar rounds. The server maintains a list of all rounds in its magazine, as well as their specific attributes (such as fuze setting and GPS coordinates).

The ROAMS remote fuze communication system was set up using a series of sockets following a classic server-client relationship. The magServer acts as the focal point between the client fuzes and the user interface. Whenever the server starts up, it runs a single-threaded Python script that accepts connections from the fuzes and the user interface. It then maintains a list of fuzes—with their relevant information—dynamically and sends update information out from the user interface. It also keeps the user interface updated on any change in fuze settings.

Python Limitations

A key limitation of the original ROAMS system was its use of Python to implement client-server threading. While a very popular language, Python is a very inefficient choice for multithreaded applications. This was highlighted in the late 2000's by David Beazley, who implicated Python's Global Interpreter Lock (GIL) as the source of its performance issues [4]. The GIL essentially forces Python programs to only run one thread at a time, even if a Python program is multi-threaded. This design decision exists to enforce memory safety in the Python interpreter.

Consequently, a program running two Python threads can run twice as slow as a Python program running a single thread. The Python community has resisted calls to remove the GIL, as doing so will reduce the safety of Python applications and reduce the speed of single threaded programs. All of these reasons suggest that Python is (for the immediate) a poor choice for creating a multi-threaded application.

Transition to C and MPI

These limitations forced the design team to explore other languages to better support multi-threading. The team settled on the C language, mainly due its native support for multi-threading, which is executed at the operating system level. While the onus for enforcing memory and thread-safety rests solely on the shoulders of the developer, C allows for more opportunities to enhance performance.

While C fully supports network socket programming over TCP/IP, the Message Passing Interface (MPI) library is used to enable the magServer to communicate with the individual clients. MPI is a standard in the high performance computing world, and is designed to enable efficient and scalable communication between multiple computers. The MPI library also has support for asynchronous communication and collective communication operations, which can drastically increase the rate at which messages are sent and received.

3. Methods

Figure 2 shows the custom 21-node Raspberry Pi B+ cluster built to simulate the full ADIM system. Each node in this cluster requires a USB wireless adaptor to both broadcast and receive wireless signals, similar to the intended implementation. Each node uses a 4GB microSD card to run the Linux operating system and store magServer and smart-round client program files. The cluster also requires a power supply to replicate the HMMWV battery for each node.

Figure 2: Final Raspberry Pi cluster.

A custom power supply was built for the project that provides surge protection, voltage conversion, and eliminates the need for 21 separate power cords. The custom case design enables the entire system to be passively cooled. The magServer node also requires a special wireless adapter to host the wireless network. Cluster and implementation details are discussed in detail below.

Cluster Configuration Details

The master Raspberry Pi node acts as a wireless access point (WAP) and dynamic host configuration protocol (DHCP) server for the project using instructions procured from the Raspberry Pi HQ website [5]. The South Hampton Raspberry Pi cluster tutorial [6] was a starting point to set up MPI on our cluster.

This application uses a custom DHCP server to assign IP addresses to each node in the cluster, requiring some additional configurations not outlined in the South Hampton tutorial. For example, the SSH configuration file was modified to disable reverse DNS lookup. Next, a Python script was added to send each worker's IP address to the magServer when the system initially boots up. This enables the magServer to automatically know at start-up the number of available worker nodes (active rounds) and their respective IP addresses.

ROAMS MPI Implementation

In the context of ROAMS, the magServer can be thought of a "master" node that passes messages to a series of "worker" nodes (smart rounds) in the ADIM magazine. Upon start up, the magServer has a list of the available "active" rounds in the magazine. Each message sent from the magServer to a particular smart round contains a set of commands to set its fuze. Each worker, upon receiving its message and setting its fuze, sends back a confirmation message.

For the scope of this paper, the design uses point-to-point communicators MPI_Send and MPI_Recv to implement the communication model. The MPI_Send function enables the magServer to send a message to a worker node. The MPI_Recv function allows a worker node to receive a message from the magServer. Thus, a pair of MPI_Send/MPI_Recv communicators is necessary each time a message is sent from the magServer to the worker nodes, or vice versa.
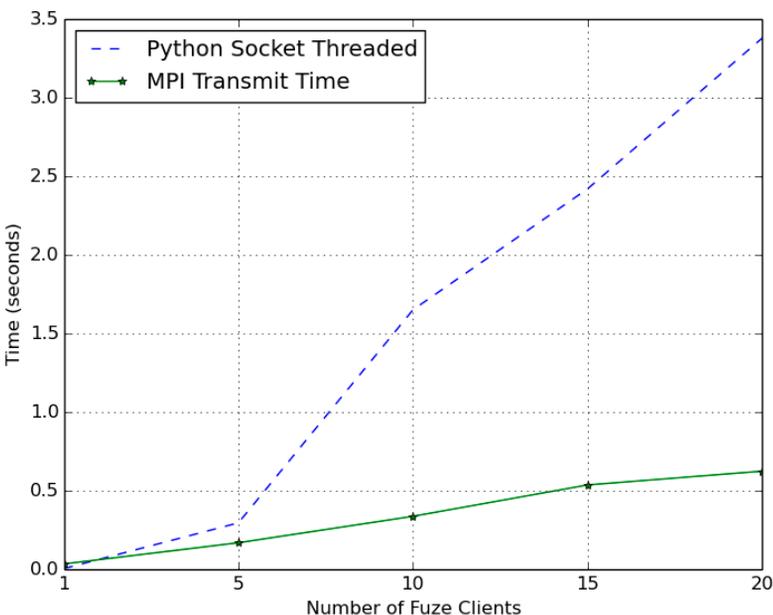

Figure 3: Time spent sending orders to new fuzes.

4. Results

The scalability of MPI compared to the Python client-server program is benchmarked by measuring two stages of execution: the time taken to send a message indicating a change in one or more clients' status (Figure 3), and the time taken to receive acknowledgement from the fuze clients that the change was made and implemented (Figure 4).

These experiments do not consider the time needed to communicate to the user interface, as scaling efficiency issues are not applicable in this context. The experiments also don't reflect the amount of time needed to acquire fuze clients during operation. This is due to the current system's inability to properly simulate when a mortar is fired. The conclusion section includes a discussion detailing what a proper future simulation of the process will look like, and some hypotheses on running time.

For each execution stage, the running time of the threaded Python implementation is compared against the MPI version. To illustrate scalability, the run time is measured as the number of

supported fuze clients is increased from one to twenty, in increments of five. We measure the percentage of run-time reduction by use the equation $(1-M/P)\times100$ where M and P are the execution times of the MPI and Python implementations, respectively.

Sending a Message to Fuze Clients

Figure 3 shows the average time it takes each implementation to send fuze data to all the clients. In this particular execution stage, the Python implementation performs moderately well, with execution time ranging from 0.00267 seconds on a single fuze to 3.37702 seconds on twenty fuzes. While the MPI implementation also experiences a modest increase in running time, it takes 0.03241 on a single fuze and 0.62245 seconds on twenty, requiring less than a second to compute regardless of the number of fuzes. This represents an 81.56 percent reduction in time for transmitting messages to the full twenty rounds.

Receiving Confirmation from Fuze Clients

Figure 4 depicts the average time it takes the Python version to receive confirmation from all the fuze clients compared to the MPI implantation. When dealing with five fuzes, it takes 1.529 seconds for the Python implementation to receive confirmation. However, as the number of fuzes increases to fifteen, the Python threaded version takes on average 5.677 seconds. At twenty clients, it takes the Python implementation 8.337 seconds on average. In contrast, the MPI implementation takes 0.05375 seconds on average to receive confirmation from a single fuze, 0.74295 seconds for fifteen fuzes, and 0.83812 seconds for twenty. This corresponds to reduction in running time of 89.95 percent.
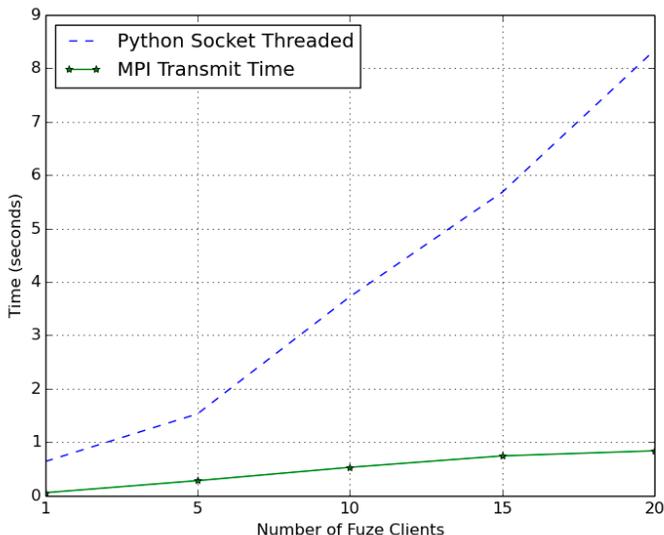
5. Conclusion

Figure 4: Time required to receive confirmation from fuzes.

The experimental results clearly show the benefit of using the MPI implementation for remotely setting fuze clients on ROAMS. Using MPI allows ROAMS to reduce the time necessary to acquire fuze information by up to 89.95 percent, corresponding to a speed up of 10.54. In all execution stages, it takes MPI less than a second to perform the desired task, regardless of the number of fuzes. In contrast, the Python implementation can take up to ten seconds.

While the difference may seem marginal on the surface, every

second counts on the battlefield. The current ADIM system has a fire rate of 30 rounds per minute, or a round every two seconds. Therefore, the reductions from 3.3 and 8.3 seconds to less than a second correspond to possibly two to four more rounds directed at the enemy. When a soldier is in contact with the enemy in a firefight, two to four mortar rounds could be the difference between achieving the objective and failing to suppress the enemy. In order for the ADIM to be useful, a soldier needs to know as soon as possible that the changes were received and his equipment is ready for use so he can continue to react to the ever changing battlefield.

These results also indicate the superiority of using MPI to achieve system scalability. MPI is a long-standing standard for a reason. The experimental results clearly show that it is faster than standard Python sockets for broadcasting and receiving messages from many nodes. We encourage other developers designing server-client systems to explore MPI as a potential library to improve performance. Future work will explore other MPI operations, such as collective and asynchronous communication constructs, in a further effort to improve performance in the ROAMs system.

Notably, the current experimentation does not include the amount of time needed to maintain the list of active fuze clients. As each round is fired from the chamber, it becomes "inactive." When a new mortar is inserted into the chamber, the round becomes "active." In both cases, the magServer needs to know about the change of status in individual rounds to maintain an accurate list of the mortars available at any given time.

Message passing can assist in keeping the magServer updated as follows. Every time a mortar is fired, it sends a message to the magServer indicating that it is no longer active. The magServer, upon receiving the message, will need to remove the mortar's IP address from the list of "available" IPs. When a new mortar is added to the chamber, it sends a message to the magServer notifying that the round is active. Upon receiving the message, the magServer adds the new IP address to the list of "available" IPs. Regardless of whether a new round is "acquired" or "disabled/fired", the cost is a single send/receive operation plus the time needed to update the list.

The preliminary results suggest that the time needed to send/receive a single message using MPI takes between 0.03 and 0.05 seconds, a trivial amount. Since ADIM's capacity is twenty rounds, it is hypothesized that the time needed to update the list is negligible. A thorough simulation of mortars firing and being reloaded is needed to fully test this hypothesis. We plan to make this the focus of our future work.

Acknowledgments/Disclaimer

ABOUT THE AUTHORS

Zachary Ramirez is a 2nd Lieutenant in the U.S. Army, Transportation Corps. He graduated with his B.S. in computer science from the United States Military Academy in 2014. He completed the work on this paper as part of an independent study supervised by Dr. Matthews and MAJ Blaine. He is currently stationed in the 916th Sustainment Brigade at Ft. Irwin, CA. He was recently selected to become part of the new Cyber Corps and will make the transition in August 2016.

E-mail: zachary.j.ramirez3.mil@mail.mil

Raymond Blaine was commissioned a Signal Officer and recently became a Cyber Officer. His assignments include a variety of duty positions at Fort Bragg, N. C. He also has served two tours in OIF and one tour in OEF, as a Platoon Leader, Aide-de-Camp to the Chief of Staff MNC-I, and as S6 for 2-508 PIR respectively. He is an Assistant Professor at USMA.

E-mail: raymond.w.blaine.mil@mail.mil

Suzanne J. Matthews is an assistant professor of computer science at the United States Military Academy, West Point. She received her Ph.D. in computer science from Texas A&M University, and her M.S. and B.S. in computer science from Rensselaer Polytechnic Institute. Her honors include a Texas A&M University Dissertation Fellowship, a Rensselaer Master Teaching Fellowship, and memberships in the Upsilon Pi Epsilon and Phi Kappi Phi honor societies.

E-mail: suzanne.matthews@usma.edu

REFERENCES

1. Kowal, Eric and Lopez, Ed. Revolutionary Mortar System to Boost Speed, Accuracy, Enhance soldier Safety [Online]. Available: <http://www.army.mil/article/147037/Revolutionary_mortar_system_to_boost_speed__accuracy__enhance_Soldier_safety/, 2015>.
2. Raspberry Pi Model B+ Data Sheet [Online]. Available: <https://www.adafruit.com/datasheets/pi-specs.pdf >, 2014.
3. Gropp, William, Ewing Lusk, and Rajeev Thakur. Using MPI-2: Advanced features of the message-passing interface. MIT press, 1999.
4. Beazley, David. Understanding the Python GIL [Online]. Available: <http://www.dabeaz.com/python/UnderstandingGIL.pdf. 2010>.
5. How-To: Turn a Raspberry Pi into a WiFi Router [Online]. Available: <http://raspberrypihq.com/how-to-turn-a-raspberry-pi-into-a-wifi-router/>
6. Cox, Simon . Steps to Make Raspberry Pi Supercomputer [Online]. Available: <http://www.southampton.ac.uk/~sjc/raspberrypi/pi_supercomputer_southampton_web.pdf, 2013>