Florida State University Libraries

Electronic Theses, Treatises and Dissertations The Graduate School

2015

Portable MPICH2 Clusters with Android Devices

Zachary Yannes

Follow this and additional works at the FSU Digital Library. For more information, please contact lib-ir@fsu.edu

FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

PORTABLE MPICH2 CLUSTERS WITH ANDROID DEVICES

By

ZACHARY YANNES

A Thesis submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Master of Science

2015

Copyright © 2015 Zachary Yannes. All Rights Reserved.

Zachary Yannes defended this thesis on April 24, 2015.
The members of the supervisory committee were:

Gary Tyson
Professor Directing Thesis

Zhi Wang
Committee Member

Xin Yuan
Committee Member

The Graduate School has verified and approved the above-named committee members, and certifies that the thesis has been approved in accordance with university requirements.

ACKNOWLEDGMENTS

I would like to thank my thesis advisor, Dr. Gary Tyson, for motivating me throughout my career at FSU and for all the help he has provided through the years. I would also like to thank Dr. Tyson and my thesis defence committee for their notes on this project. Finally, I would like to thank Dr. Liu and the entire staff of the CS department at FSU for the opportunities they have provided for me, including acceptance into the Master's program.

TABLE OF CONTENTS

List of Figures . . . v
List of Abbreviations . . . vi
Abstract . . . vii

1 Introduction **1**
- 1.1 Previous Research . . . 2
- 1.2 Android MPI Framework . . . 3

2 MPICH Overview **6**
- 2.1 Process Manager . . . 6
- 2.2 Process Management Interface . . . 6

3 Application Overview **8**
- 3.1 Configuration . . . 8
  - 3.1.1 Client Mode . . . 8
  - 3.1.2 Server Mode . . . 8

4 Evaluation **11**
- 4.1 Testbed Setup . . . 11
- 4.2 Overview . . . 11
- 4.3 IMB-EXT Results . . . 12
- 4.4 IMB-IO Results . . . 13

5 Discussion and Future Work **19**
- 5.1 Discussion and Future Work . . . 19

6 Conclusion **21**
- 6.1 Conclusion . . . 21

References . . . 22
Biographical Sketch . . . 23

LIST OF FIGURES

1.1 Example of running *mpiexec* with executable *helloworld* 1

1.2 Select smpd and mpiexec settings 4

3.1 Java Client listenting thread 9

3.2 Function to run executable in secure shell. Before running the executable, the HOSTNAME, HOME, and LD_LIBRARY_PATH environment variables and permissions are set 10

4.1 Results from the Accumulate IMB benchmark test 13

4.2 Results from the Window IMB benchmark test 14

4.3 Results from the unidirectional and bidirectional IMB benchmark tests 15

4.4 Results from the Open_Close IMB benchmark test 16

4.5 Results from the PRead IMB benchmark tests 17

4.6 Results from the PWrite IMB benchmark tests 18

LIST OF ABBREVIATIONS

AP - **A**ccess **P**oint
AOSP - **A**ndroid **O**pen **S**ource **P**roject
IMB - **I**ntel MPI **B**enchmark
MPI - **M**essage **P**assing **I**nterface
MPICH - **M**essage **P**assing **I**nterface **CH**ameleon
PM - **P**rocess **M**anager
PMI - **P**rocess **M**anagement **I**nterface
SMPD - **M**essage **P**assing **I**nterface

ABSTRACT

Recent surveys have shown that over 6.8 billion mobile phones are in use today where roughly 50% are running the Android Operating System [2]. This large population of devices can be utilized for message-passing interface (MPI) computing. There have been few examples of this in practice [1, 7] due to the complexity of installation and lack of functionality. However, I propose a new framework which permits client devices to dynamically join and leave an MPI-based network without drastically affecting its performance. It uses the MPICH2 library to execute processes and pass messages between the devices. Unlike the previous implementations, this framework uses Java (in the form of an Android application) to perform the initialization steps and execution of MPICH. As a result, the purpose of this paper is to demonstrate the feasibility of implementing the framework as an independent library.

CHAPTER 1

INTRODUCTION

Due to the prevalence of multi-processor devices, a rising field in computer science is multi-process implementation (MPI) programming. Unlike traditionally program execution, MPI programming allows the compiler to exploit the multiple processors in order to execute the program on multiple processes. The program is divided into separate tasks for the each processor. With frameworks like MPICH, the programmer can delegate processes not only to multiple processors but also to multiple devices.

MPICH is a portable, high-performance MPI standard [6]. It provides built in process managers such as Hydra and smpd. The process manager is executed on each device in order to initialize the server as well as perform authorization and connect to the server from the client devices. Once all the devices have started the process manager, *mpiexec* is called with the number of processes and the executable to run. For example, Figure 1.1 shows how to run the program *helloworld* with 32 processes.

```
1: mpiexec -n 32 ./helloworld
```

Figure 1.1: Example of running *mpiexec* with executable *helloworld*

Mpiexec passes the executable and parameters to the process manager which creates and executes the processes, performs message passing, and handles errors.

The Android system includes a built-in form of message passing through the Binder framework. The Binder framework provides the inter-process communication (IPC) mechanism for passing messages between processes in the form of a *Parcel*. Android handles the complexities of converting different objects into memory by having the user create an AIDL file. The user then implements and binds the AIDL interface from within a Java service. Next, the object class must be extended from the *Parcelable* class in order to instruct the Android operating system how to write and read the object to a Parcel. Although this method provides more functionality than extending from the Binder class or using a Messenger, it does not allow device-to-device message passing. For instance,

one application can create a *Rect* Parcelable, store the data, and pass it to a separate application on the same device. The reason is because the Binder framework sends the Parcelable data to the kernel for more efficient data transfer and access. Without a method for accessing another device's kernel memory, which would present difficult security issues, the Parcelable cannot be accessed by another device. However, a potential addition to this framework would be to implement an Android IPC driver which can communicate through WiFi.

The MPICH framework requires two libraries, libcrypto and libssl, which were cross compiled using an ARM toolchain created by the Android Native Development Kit (NDK). Once these libraries are available, the MPICH2 framework can be compiled for Android, again using the ARM toolchain to create the necessary executables: *smpd*, *mpiexec*, and *mpicc*. Since the Android operating system lacks the necessary libraries to execute MPICH2, the libmpi libcrypto libraries were cross compiled using the Android Native Development Kit (NDK). These libraries were then installed onto the */system* partition of each rooted Android device. To provide full MPI functionality, the framework must include initialization methods for both the server and client devices. The client setup includes connecting to the server, receiving the configuration files, and starting *smpd* and *mpiexec*. The server setup includes creating the WiFi access point, enumerating a list of client IP addresses, creating and distributing the configuration files, starting *smpd* and *mpiexec*, compiling the results, and displaying them on the server device.

1.1 Previous Research

Due to time constraints, the scope of this project will be to use the MPICH process manager, *smpd*, on the Android devices. There have been a few brief explorations with MPI for Android [1, 7]. These implementations cross-compile MPICH2 for Android, executed on multiple rooted phones. In order to perform the cross-compilation, the authors created a cross-compiler using *buildroot*. I decided to use the *Cyanogenmod* ROM builder in anticipation of future development. The Cyanogenmod builder builds an Android ROM using a derivation of the Android Open Source Project (AOSP) source code, but provides methods for adding custom drivers and applications, as well as the cross-compiler. As a result, once a ROM is built, the output directory contains the required libraries for the cross-compilation: libcrypto and libssl.

By linking to the output directory during the configure phase, the MPICH2 library can be built using the traditional make/make install method. The experiments in [1, 7] demonstrated that the execution of MPICH on Android requires an extensive environment initialization which is performed manually. Furthermore, the number of processes spawned must be properly configured to match the number of available processors in the cluster. Finally, the security of the network must be improved on for the clients, in order to facilitate safe memory accesses and alleviate any doubts by the clients.

1.2 Android MPI Framework

In order to improve on the previous Android MPI designs, I have designed an Android application which performs the initialization of the environment automatically. First, the *smpd*, *mpiexec*, and the *MPI executable* (i.e. *helloworld*) files are written to internal memory on the device and permissions are changed to be executable. Second, the *$HOME* environment variable is set to the folder containing the written files which must have the same value on all devices when used by *smpd* and *mpiexec*. Third, the *hostname* of the device is changed from *localhost* by writing to */proc/sys/kernel/hostname* in order to differentiate between devices. Fourth, the application determines one device to act a server, which creates a portable Access Point (AP) network for the other devices to join. Fifth, the network clients are enumerated to create the *.smpd* configuration file, individual connections are established with the client, and the configuration file is distributed. The *.smpd* configuration file contains a passphrase required for execution, and a list of all dynamic client IP addresses. The final step is to execute *smpd* on the server and client devices, and then execute *mpiexec* with the MPI executable on the server. The application then waits for the execution to complete and displays the output of the program. In order to resolve the issue of a static client list, the application enumerates the network client list and redistributes the *.smpd* configuration file before every execution whenever new clients are added. This feature improves functionality with little overhead since the configuration file is only updated when a new client has been added. Therefore, pre-execution overhead is related to the number of new clients added between runs.

One important issue with the MPICH framework is the lack of security. *Smpd* provides minimal security by requiring a password upon execution. However, the password is stored in plaintext in the configuration file. Also, the communication between devices is over an insecure TCP connection.

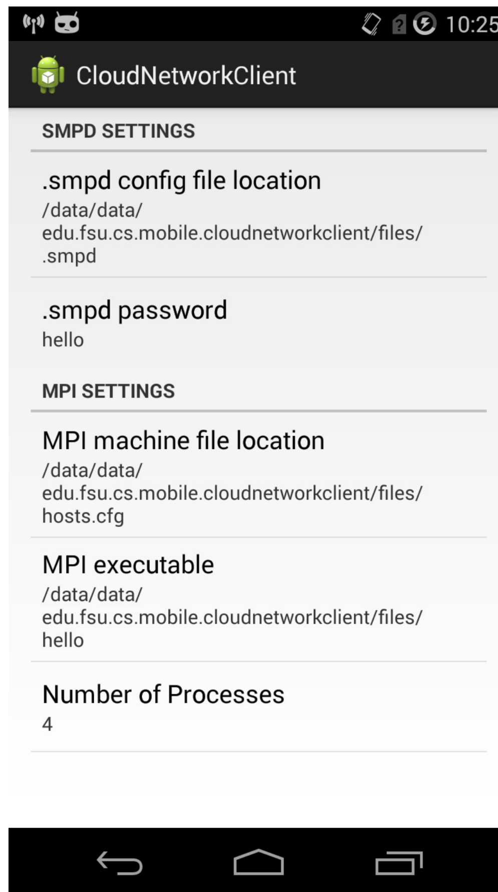

Figure 1.2: Select smpd and mpiexec settings

In order to safely distribute the configuration file without leaking the plaintext password, the file can be encrypted with a private key created upon starting the app. Then, the public key, which is retrieved by querying the server, is used to decrypt any received files. The security could be further increased by adding a routine in *smpd* to encrypt and decrypt messages sent between devices. However, these issues are contingencies for the main security measure, the WiFi network. Since the server creates a personal AP network, the only way for an attacker to retrieve packets would be by physically connecting to a device in the network or breaching the network wirelessly. As a safety measure for compromised networks, the server has the option to change network settings such as the password and the wireless encryption standard.

The remainder of this paper is organized as follows. First, in chapter 2 I provide an overview to the MPICH framework and describe the two main components: the Process Manager and Process

Management Interface. Second, in chapter 3 I provide an overview to the application implementing the Android MPICH framework. Third, in chapter 4 I analyse the performance of the framework by running the Intel MPI Benchmark and evaluate the feasibility based on the results. In chapter 5 I discuss key points in development of this framework and areas of future work. Finally, chapter 6 summarizes the information presented in this paper.

CHAPTER 2

MPICH OVERVIEW

In this section, the MPICH framework is broken down into the key components: the Process Manager and the Process Management Interface. The process manager performs the process tasks such as creation, memory passing, and synchronization. The Process Management Interface (PMI) provides a set of commands for interacting with the process manager between any device across the network.

2.1 Process Manager

After the environment is configured, the environment settings for each process are set by the process manager. Here, the process manager creates the initial state of each process in the pool by defining values such as process size, rank, app number, and id. First, the process manager creates and binds a socket between the parent (server) process and the spawned child process. Next, the starting state of the process is passed through the newly created socket. The remaining responsibility of the process manager is to send, decode, and respond to signals and commands. This includes maintaining a list of process IDs which must be updated as processes spawn and die.

2.2 Process Management Interface

Each process can be interfaced through *PMI commands.* PMI commands convert the high-level structure of the PMI data structures to binary format which can be sent over sockets. This simplifies the distribution of the initial state information to each process to a simple set-get function methodology. The first command sent to any process is *MPI_Init or MPI_Init_thread*, which spawns the process, initializes communication, and distributes the environment settings. Then, the processes can perform a number of data transmission procedures such as unicast (MPI_get, MPI_set, MPI_Send, MPI_Recv), multicast (MPI_alltoall, MPI_allgather, MPI_allreduce), and broadcast (MPI_bcast). There are also non-blocking versions in which the command is prepended by an emphI, i.e. *MPI_Ibcast*. For finer granularity control of memory access, *MPI_Barrier* provides

a mutex or locking mechanism for shared memory. This prevents overwriting a variable with a deprecated value which is currently being accessed by another process. In other words, it maintains memory coherence between the devices without constantly having to synchronize. The final command called by a process is *MPI_Finalize*, which must be called by the same process which executed MPI_Init or MPI_Init_thread [6]. This command will complete the execution of the process by closing any remaining sockets and unlocking any local barriers.

CHAPTER 3

APPLICATION OVERVIEW

The motivation for building this project as an Android application is the necessity for portability and simplicity. The work on this application has led to a clean, fast framework which installs, initializes, and executes MPI programs over a network of Android devices.

3.1 Configuration

The first phase of the framework is mode selection. The application allows the user to choose either a client or server mode.

3.1.1 Client Mode

The concept of the client mode is to require as little user interaction as possible to initialize the MPICH cluster. This modular framework also helps to improves deployment and future development. Plus, in not requiring user interaction, the device can enter standby mode and conserve the battery. When the user enters client mode, the device is configured to connect to access points named *AndroidHPC*. Once the client is connected to the network, a thread starts in the background to listen for the server. Figure 3.1 shows part of the *ClientListenTask* class which is an extension of the AsyncTask. The ClientListenTask class is responsible for opening the ServerSocket and reading the configuration data sent from the server.

When the server writes to the socket, a secure shell is initiated to transfer the configuration files to the client device. This class is based on the *libsuperuser* library from [5]. The secure shell initializes with root permissions, which requires the device to be rooted. Finally, the device automatically executes *smpd* from the secure shell, which also runs in a background thread to be read by the host when needed.

3.1.2 Server Mode

Server mode has also been designed to reduce user interaction, but provides many options to customize the execution. When the user enters server mode, an access point is started from the host

```
 1:     /* Creates socket, reads configuration data */
 2:     @Override
 3:     protected String doInBackground(String... params) {
 4:     String line = "";
 5:         try {
 6:             line = openSocket(mAddr, mPort);
 7:         } catch (IOException e) {
 8:             Log.e("connectSocket()", e.getMessage());
 9:             return null;
10:         } catch (Exception e) {
11:             e.printStackTrace();
12:         }
13:         return line;
14:     }
15:
16:     /* Write configuration data to file */
17:     @Override
18:     protected void onPostExecute(String result) {
19:         super.onPostExecute(result);
20:
21:         if (result != null && !result.equals("")) {
22:             Log.i("OnPostExecute()", result);
23:             writeFile(mContext, ".smpd", result);
24:             mTaskListener.OnTaskCompleted(result);
25:         }
26:     }
27:
28:     private String openSocket(String server, int port) throws Exception {
29:
30:         ServerSocket serverSocket = new ServerSocket(port);
31:         String line = "";
32:         while (true) {
33:             /* Read line and return */
34:         }
35:     }
```

Figure 3.1: Java Client listenting thread

device (with the SSID “Android HPC”). The user can start a *mpiexec* process either by clicking the ”start” button, which runs the default MPI executable, or by long clicking the ”start” button and selecting an executable from the list. Before *mpiexec* can start, the server device collects a list of the IP addresses connected to the network, creates the configuration files, and then distributed them to the clients using secure shells. Figure 3.2 displays the function which initializes the secure shell with the correct environment variables and executable flags. The function can be used to start either *smpd* or *mpiexec*.

Like the ClientListenTask, *smpd* is executed on a background shell, followed by *mpiexec* with

```
 1: public String startExe(int code, String exe, String flags) {
 2:     String res = "";
 3:     String smpdfile = mDirectory+ "/" +mContext.getString(R.string.config_smpd);
 4:     String[] cmd;
 5:     if (code == SMPDCODE) {
 6:         cmd = new String[] {
 7:             String.format("echo %s > %s", mHost, HOSTNAMELOC),
 8:             String.format("export HOME=%s", mDirectory),
 9:             String.format("export LD_LIBRARY_PATH=%s", mLdPath),
10:             String.format("chmod 600 %s", smpdfile),
11:             mDirectory + "/" + exe + " " + flags };
12:     } else if (code == MPIEXECCODE) {
13:         cmd = new String[] {
14:             String.format("echo %s > %s", mHost, HOSTNAMELOC),
15:             String.format("export HOME=%s", mDirectory),
16:             String.format("export LD_LIBRARY_PATH=%s", mLdPath),
17:             String.format("chmod 600 %s", smpdfile),
18:             mDirectory + "/" + exe + " " + flags };
19:     }
20:     openRootShell(cmd, code, true);
21:     return res;
22: }
```

Figure 3.2: Function to run executable in secure shell. Before running the executable, the HOSTNAME, HOME, and LD_LIBRARY_PATH environment variables and permissions are set

the selected executable. Once the process completes, the returned data is parsed and displayed to the screen. Once the mode is selected, the user can also enter the *settings mode* to configure the execution of *smpd* and *mpiexec*, as shown in Figure 1.2. To make the settings persist between application runs, the selections are stored to *SharedPreferences*. Android SharedPreferences use an xml-styled format to store the settings to a local application folder which are loaded on application start up. The user can select both *smpd* and *mpiexec* settings. For *smpd*, the only necessary values are the config file and the password. For *mpiexec*, you can select the configuration file, the MPI executable, and the number of processes.

CHAPTER 4

EVALUATION

4.1 Testbed Setup

The testbed for this experiment is composed of two physical Android devices: a Samsung Galaxy Nexus with 1GB of memory and a Dual-core 1.2 GHz Cortex-A9 processor, and an LG D800 with 1.8GB of memory and a Quad-core 2.26 GHz Krait 400 processor [3]. The D800 was used as the server device, and the Nexus as the client. The devices were rooted prior to installation of the application in order to allow the application to execute root commands.

4.2 Overview

The goal of the experiment is to demonstrate whether the new framework can be implemented in a simple and efficient manner comparable to desktop MPICH clusters. The clusters were tested using the Intel MPI Benchmark (IMB) suite [4]. To compare the performance of the clusters, tests from the IMB-MPI2 test suite were used. Each test program performs i iterations of the command(s) between each device for varying message sizes: 0 to 2^{14}, with an increasing number of processes ($n = [2 : 16]$) over a maximum 5 second interval (except for Open_Close which has only one iteration). The number of iterations is approximated by from the interval window (5 seconds).

The IMB-EXT tests selected are Unidir_Put/Unidir_Get, Bidir_Put/Bidir_Get, Accumulate, and Window. The Unidirectional Put/Get tests measures the unidirectional bandwidth for the execution of MPI_Put/MPI_Get by having one device send the message to the other device each iteration. The Bidirectional Put/Get tests perform the same test, except in each iteration both devices send a message. The Accumulate test measures the time to execute MPI_Accumulate performing a floating point MPI_Sum. The time is collected is the aggregated time, such that it is averaged over all the samples. The Window test measures the time to execute MPI_Win_create, MPI_Win_fence, MPI_Put, and MPI_Win_free.

The IMB-IO tests selected are Open_Close, PRead, and PWrite. The Open_Close test performs an MPI_File_open, MPI_File_write, and MPI_File_close to determine the time of one write task. The

PRead test contains three subtests: each process reads from one file concurrently with individual pointers (*indv*), each process reads from separate files concurrently (*priv*), and each process reads from one file concurrently with a shared pointer (*shared*). The PWrite test performs the same three subtests, but using MPI_File_write instead of MPI_File_read. For *indv*, the file is organized in disjoint contiguous blocks such that each process performs the operation for X/n bytes in the order it was accessed, where X is the message size and n is the number of processes. For *priv*, each process performs the operation for X/n bytes on a separate file, where X is the message size and n is the number of processes. Finally, for *shared*, the file is organized in disjoint contiguous blocks such that each process performs the operation for X/n bytes in a random order, where X is the message size and n is the number of processes.

4.3 IMB-EXT Results

The first two IMB-EXT tests, Accumulate and Window, both demonstrate the overhead of running common subroutines on the Android MPI cluster. Figure 4.1 shows the *MPI_Accumulate* task time for different message sizes. As the figure shows, the iteration time remains low for one or two processes, but increases linearly with the message size for 16 processes. Similarly, Figure 4.2 shows the time delay from running the Window functions over multiple processes.

The Unidirectional and Bidirectional Get and Put benchmarks only run using two processes because the commands only perform two-party communication. Figure 4.3 shows the throughput in (MBytes per second) for each subtest over different message sizes. This test is important to determine which message size to use to optimize the throughput for both getting and setting. For unidirectional communication, the throughput increases with the message size. Both commands provide maximum throughput at a message size of 2^{14}, MPI_Get with 55.24 MB/s and MPI_Put with 63.51 MB/s. For bidirectional communication, MPI_Get peaks at 49.95 MB/s with a maximum message size of 8192 bytes, while MPI_Put increases with the message size with a maximum throughput of 63.51 MB/s for 16384 bytes. However, the optimal message size would be around 1200 bytes to provide a throughput of roughly 39 MB/s.

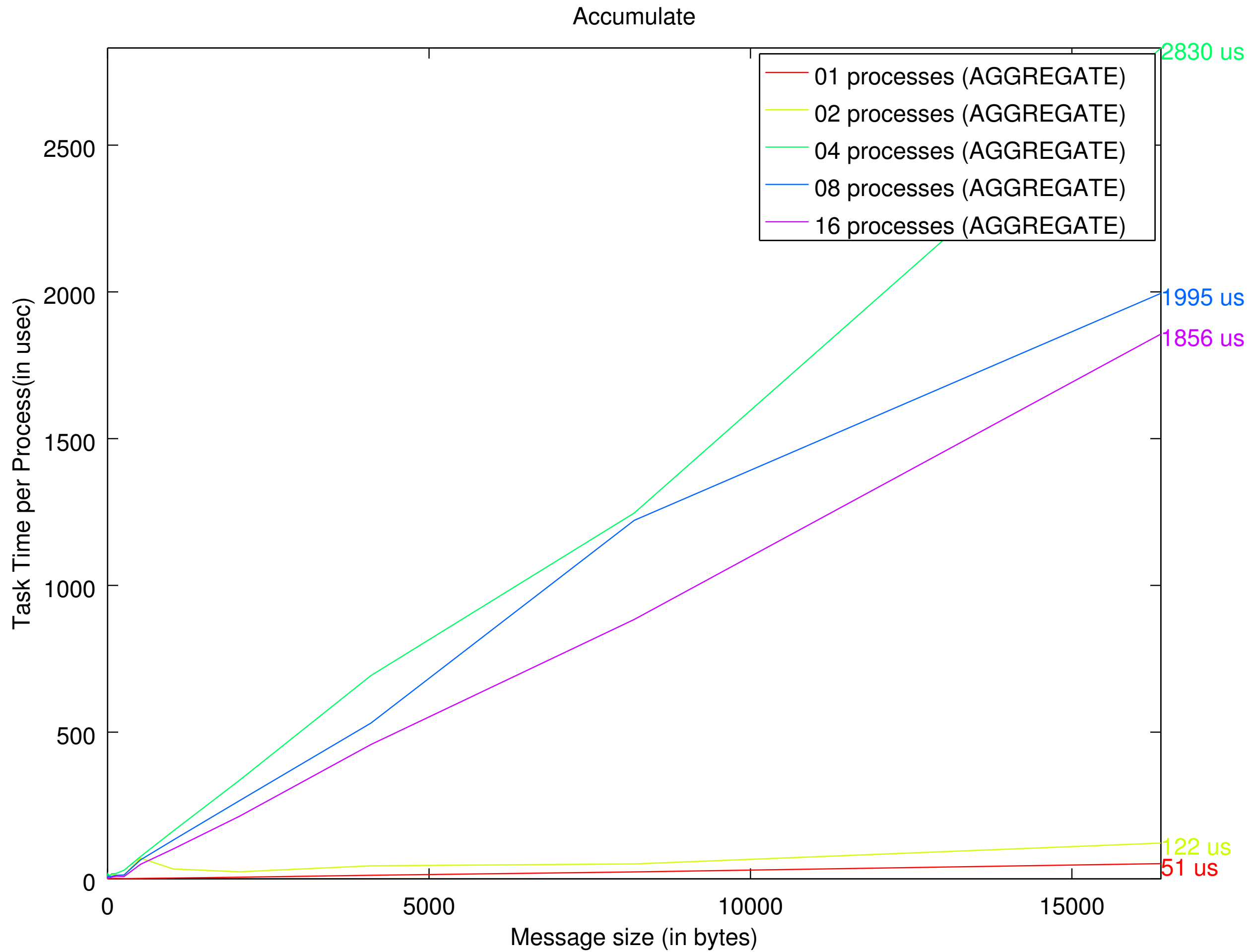

Figure 4.1: Results from the Accumulate IMB benchmark test

4.4 IMB-IO Results

The IMB-IO benchmark tests underline the runtime of MPI file I/O operations. The first test, Open_Close, opens the file with MPI_File_open, writes one byte using MPI_File_write, and finally closes the file with MPI_File_close using 1, 2, 4, 8, and 16 processes. Figure 4.4 displays the results from this test. As you can see from Figure 4.4, the average task time for $n = 2$ processes is approximately seven times that of $n = 1$ process. However, when the average runtimes are normalized to one process, for four or more processes the factor of runtime increasing is logarithmic, beginning with approximately 63 times the baseline task time. The reason for this performance degradation is from the overhead each process having to relocate the file pointer. As the number

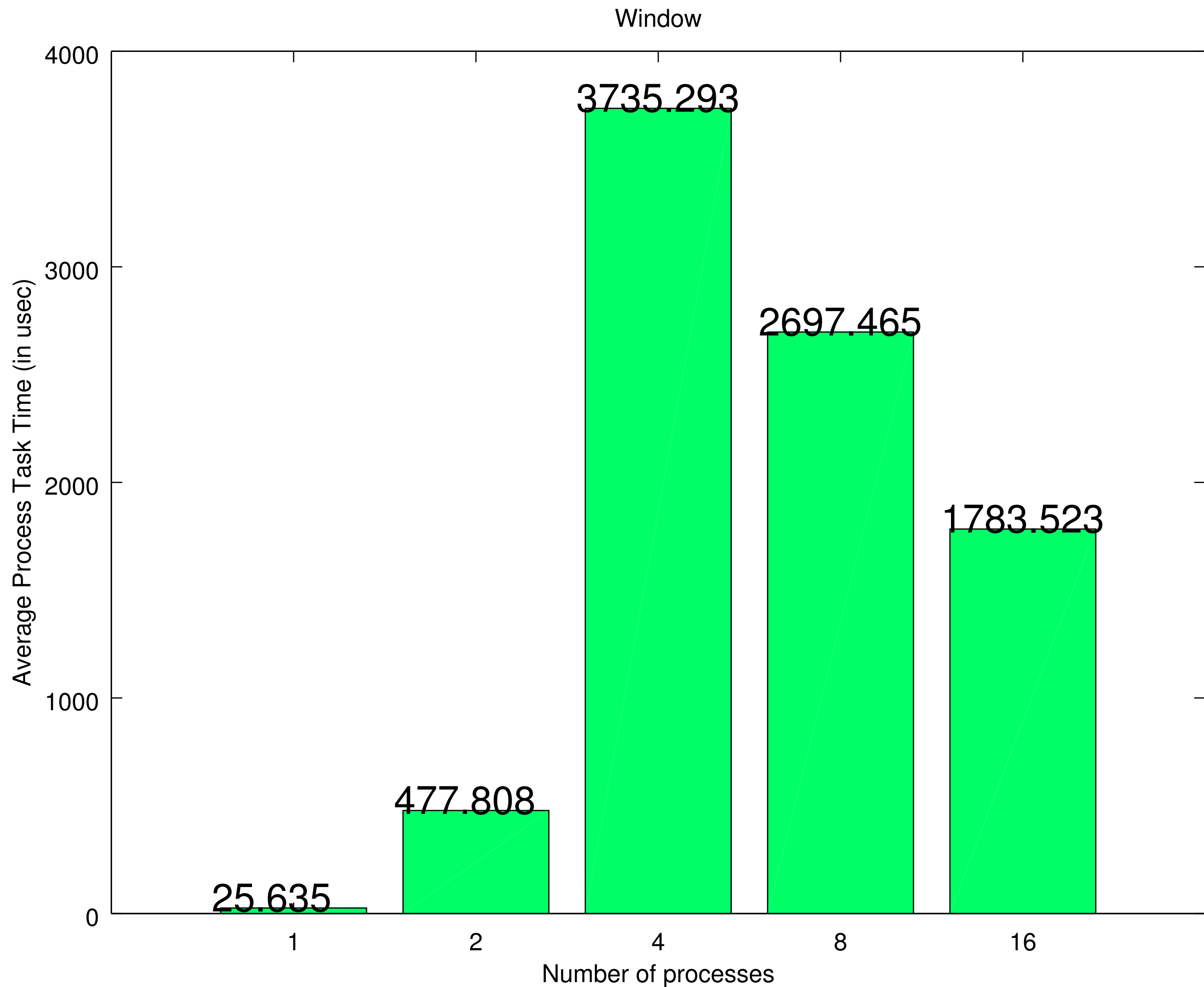

Figure 4.2: Results from the Window IMB benchmark test

of processes increases, the overhead propagates through each iteration, causing a greater data transfer lag. The remaining two tests demonstrate possible solutions to this issue by using different file pointer techniques.

The PRead and PWrite tests perform a variation of the Open_Close test but using three different techniques for file I/O operations, with either MPI_File_read or MPI_File_write operations, respectively. With the Open_Close test, each iteration performed all file I/O operations, but the PRead and PWrite tests perform the file open/close in the iteration depending on the subtest protocol. Figures 4.5 and 4.6 shows the throughput from both PRead and PWrite for the three

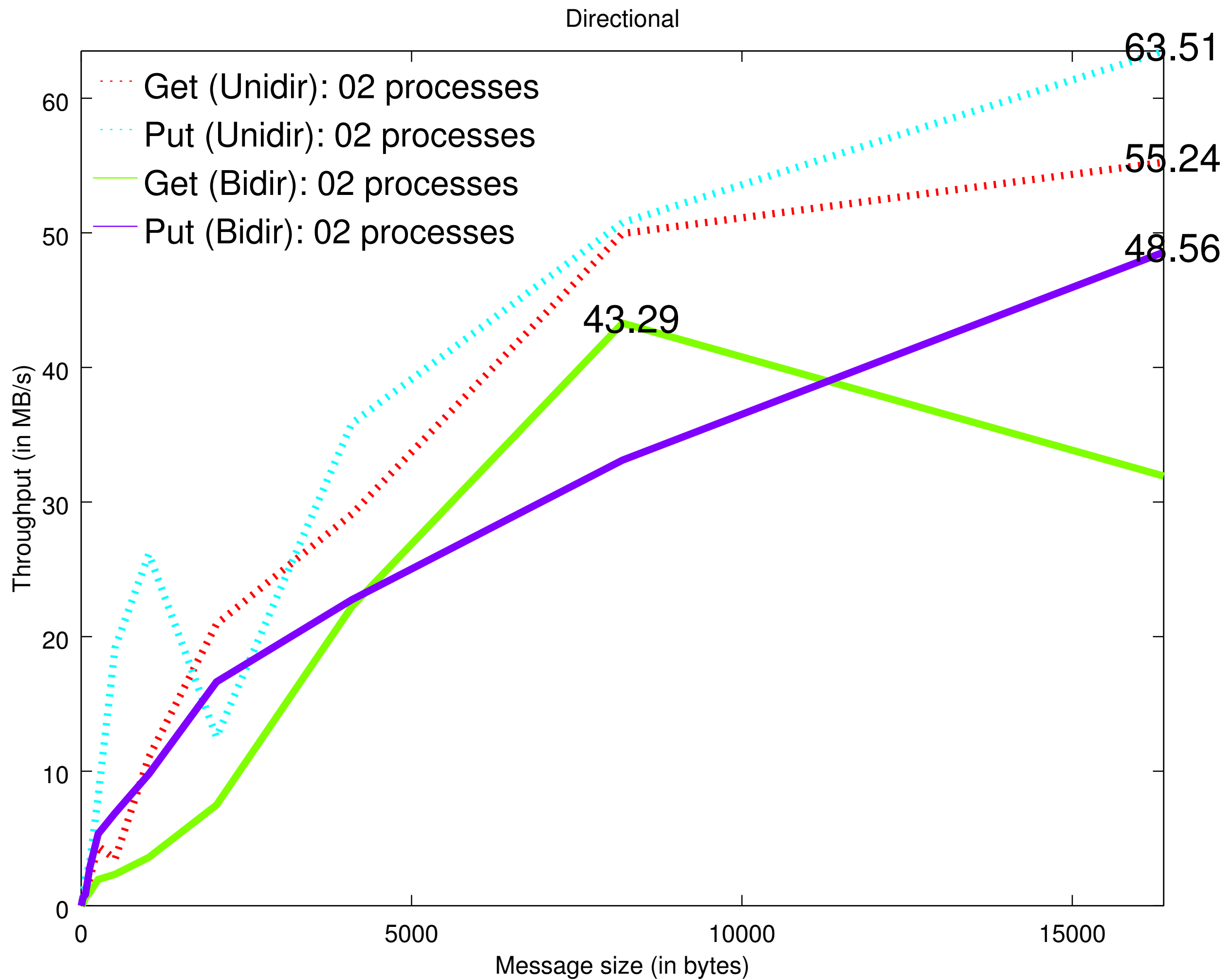

Figure 4.3: Results from the unidirectional and bidirectional IMB benchmark tests

subtests.

Running PRead with individual file pointers, each iteration must perform a file open and close, causing a substantial decrease in throughput for $n > 1$ processes, roughly 13%. However, for PRead with private file pointers, each iteration does not require a synchronization step between the processes since they each have their own file pointer. Therefore, we can achieve up to 168% increase in throughput for 4 processes with a message size of 128, with a maximum throughput of 4405.78 MB/s for a message size of 16384. Finally, for PRead with shared pointers, we experience the greatest performance increase of 243% with 4 processes and a message size of 4 bytes, and a maximum throughput of 764.18 MB/s with a message size of 16384. This likely results from the use

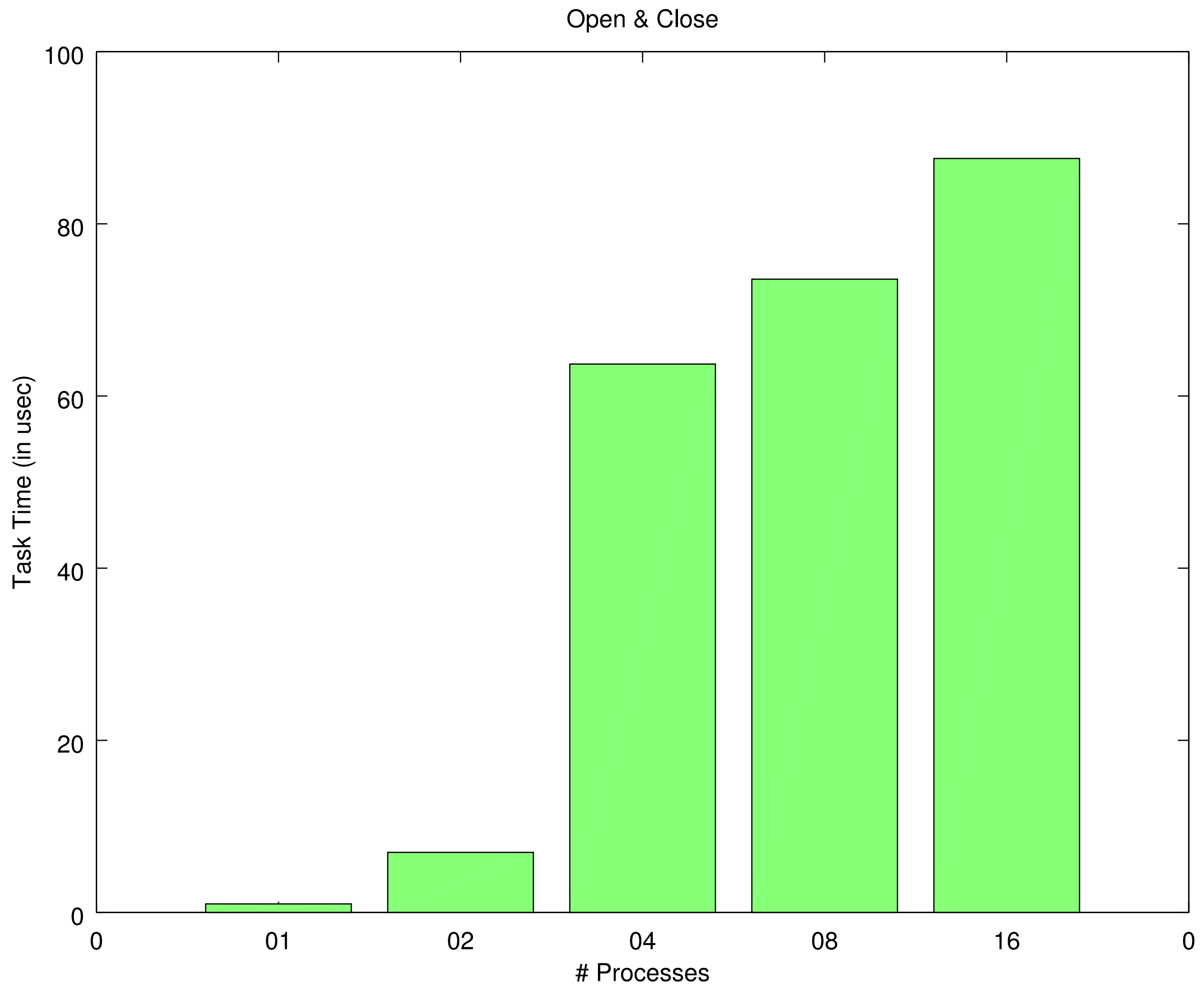

Figure 4.4: Results from the Open_Close IMB benchmark test

of a shared file pointer, which must be synchronized between processes for each iteration. Overall, the greatest result shown by Figures 4.5 and 4.6 is that the use of private file pointers consistently provides the highest throughput. MPI_File_read and with private file pointers initializes each process with a block of the file and points the private file pointer to the start of that block. Therefore, each process can iterate through the file independently without performing synchronization at the end of each message, but synchronizing after all messages in each block have completed.

As expected, PWrite followed the pattern of PRead in four processes producing the highest throughput. However, the PWrite benchmark showed that individual and shared file pointers

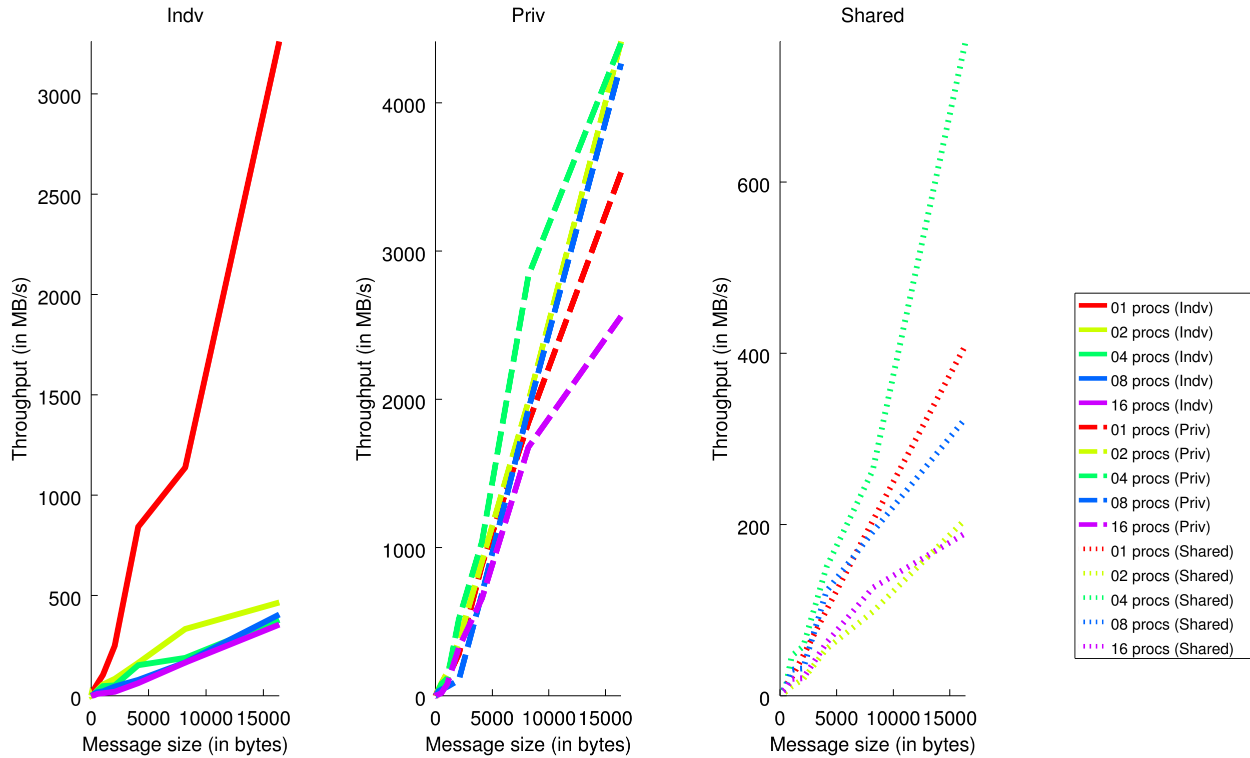

Figure 4.5: Results from the PRead IMB benchmark tests

produced substantially higher throughput than private file pointers. As Figure 4.6 shows, for four processes and a message size of 16384, shared and individual file pointers produce 578.62 and 534.01 MB/s respectively. On the other hand, private file pointers only produce a maximum of 25.34 MB/s. This results from the synchronization required in writing versus reading from a file. When MPI_File_read is executed, the file pointer changes its address, but the content remains the same. However, MPI_File_write changes both the address of the file pointer, and the contents of the file. Therefore, shared file pointers must resynchronize after every iteration, while individual file pointers only have to update their individual blocks in the file and synchronize after all iterations.

From the file I/O IMB tests, the results show that MPI File I/O operations produce the highest throughput when using four processes. Furthermore, MPI_File_read produces the highest throughput using private file pointers and a message size set at 128 bytes, while MPI_File_write produces the highest throughput using shared file pointers and a message size set at 16384 bytes.

While the experiments did not demonstrate performance at larger scales (due to the lack of test

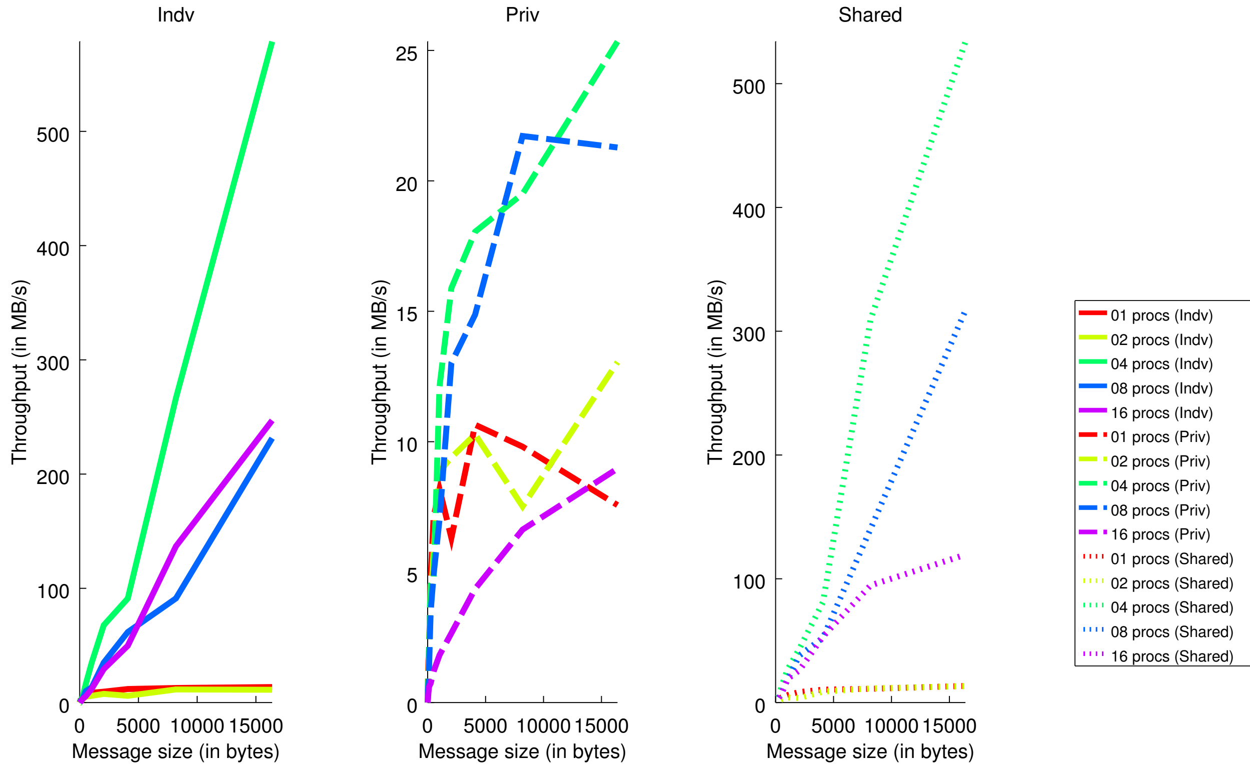

Figure 4.6: Results from the PWrite IMB benchmark tests

devices), it did highlight which operations result in minimal overhead and which operations result in substantial overhead. Analysis of the program execution traces revealed that the major source of the overhead is memory management. As the number of processes increases for operations with short time iterations, the overhead of synchronizing between each interval aggregates over time. In other words, the framework provides better throughput for large message sizes, resulting in fewer synchronization steps, and therefore less overhead aggregation. The minimum message size for which the throughput outperforms that of one device was measured to be 8192 bytes.

CHAPTER 5

DISCUSSION AND FUTURE WORK

5.1 Discussion and Future Work

The decision of using *smpd* as the process manager came out of necessity when at the start of my research, smpd was the only compatible PM provided with MPICH2. Since then, MPICH3 has been released with a much more encompassing library, as well as a new PM known as Hydra. This update provides several new components such as starting nameservers, selecting different interfaces and launchers, and binding processes to threads.

I have since compiled the MPICH3 library for Android using the Hydra PM. The Hydra PM uses SSH to communicate between the devices which provides additional security. In order to communicate using SSH, each device must create an RSA or DSA public and private key. The application can be modified to collect the public keys of each client device, create a *authorized_keys* file containing each public key, and distribute it to each device instead of the *smpd* configuration file. The Android RM can be developed, like Hydra, to accept various interfaces. Then, after developing the necessary drivers, the RM could connect devices via Bluetooth, NFC, or possibly even media devices.

One point that should be brought up with this project is that it is hoped to be developed into an independent component or module, rather than a library. One reason for this is because of the network card performance on mobile devices. Unlike business-grade routers, the access points created by mobile devices can hold a very limited quota, especially when referring to it as an HPC. Therefore, the solution is to allow multiple networks to overlap, provided the security agrees. Much like wide area networks, it would allow devices to run as either hosts or clients without affecting the other networks negatively, but rather merging the networks. One way to accomplish this is by having one device operate as a bridge for the two networks.

One issue discovered during testing is that *mpiexec* performs multiple searches at the start of execution to find library and include files. This results in a significant initialization overhead. A possible performance increase could result from recompiling the MPICH2 library with hardcoded

paths to library and include files. Furthermore, the application can be made more portable by compiling static libraries for *libmpi* and *libcrypto* which can be installed on the device dynamically upon application installation.

CHAPTER 6

CONCLUSION

6.1 Conclusion

The purpose of this paper is to introduce a new framework for autonomous initialization and execution of MPICH clusters on Android devices, and to evaluate the feasibility of such a framework. First, I introduced the MPICH and AIDL message passing frameworks and briefly compared their deployment methods. Second, I extensively described the execution procedures of MPICH. Next, I listed the configuration process and the Intel MPI Benchmark tests used to profile the Android cluster. Then I evaluated the results of the IMB tests and highlighted the configurations providing the best performance. Finally, I discussed the application of this framework and improvements to be made in the future.

The Android MPICH framework deployed on two devices provided interesting results when profiled using the IMB MPI and file I/O test cases. Primarily, the cluster is best used for programs with large message sizes and 4 processes, especially when performing file I/O operations. Furthermore, due to the overhead from synchronizing at each iteration, the cluster performs greater from using private file pointers for MPI_File_read and shared file pointers for MPI_File_write.

From this data, we can conclude that the proposed framework is feasible for creating portable MPI clusters on Android devices. The framework was applied to a cluster of two devices, and provided a significant performance increase for large message sizes. This is due to the overhead aggregation from the synchronization step between iterations.

REFERENCES

[1] Felix Busching, Sebastian Schildt, and Lars Wolf. Droidcluster: Towards smartphone cluster computing–the streets are paved with potential computer clusters. In *Distributed Computing Systems Workshops (ICDCSW), 2012 32nd International Conference on*, pages 114–117. IEEE, 2012.

[2] Andy Favell. Global mobile statistics 2014 part a: Mobile subscribers; handset market share; mobile operators, May 2014.

[3] gsmarena. Gsm arena, March 2015.

[4] Gergana Slavova (Intel). Intel mpi benchmarks 4.0, October 2013.

[5] Jorrit Chainfire Jongma. libsuperuser, January 2015.

[6] MPICH. Mpich documentation, March 2015.

[7] Iulian VÎRTEJANU and Costică NIŢU. Programming distributed applications for mobile platforms using mpi.

BIOGRAPHICAL SKETCH

Zachary Yannes is currently in his second year of Graduate school in the Department of Computer Science at Florida State University. In December 2012, he graduated from the Florida State with his Bachelors in Computer Science, and then began the Masters program the following semester. Zachary has been working at the FSU Mobile Lab for four years where his field of research focuses on mobile development and architecture.