Multilingual Support for A-JUMP

Usman A. Malik[1,2], Naveed Riaz[2], Sajjad Asghar[1], Mehnaz Hafeez[2] and Adeel ur Rehman[1,2]

[1]*National Center for Physics,Quaid-e-Azam University, Islamabad*
[2]*Shaheed Zulfikar Ali Bhutto Institute of Science and Technology, Islamabad*
usman@ncp.edu.pk, n.r.ansari@szabist-isb.edu.pk, sajjad@ncp.edu.pk, mhafeez04@yahoo.com,adeel@ncp.edu.pk

Abstract

The Architecture for Java Universal Message Passing (A-JUMP) is a message passing implementation based on MPJ specifications. It is written purely in Java. One of the design goals of A-JUMP is interoperability, where applications written in different programming languages must be able to communicate with each other. Moreover, programmers can write parallel applications using the programming language of their choice and their applications must also be portable across different platforms. The communication backbone of A-JUMP is High Performance Computing (HPC) bus. HPC bus is based on ActiveMQ – an implementation of Java Message Service (JMS). In this research work, we have augmented multilingual support for A-JUMP that includes language bindings for C/C++ and C#. Code distribution and execution on A-JUMP framework using C/C++ and C# applications is achieved. We have also provided APIs for the respective languages and tested the inter-language communication between C/C++, C# and Java applications. Performance results of code execution and communication of C/C++, and C# implementations are also presented.

Keywords: *HPC; Multilingual; A-JUMP; NMS; ActiveMQ-CPP*

1. Introduction

Several MPI like parallel programming models exist today that are written in different programming languages including C/C++, Java, Python and C#. These implementations are written using a single programming language and lack language interoperability support. Language interoperability ensures that application developers need not worry about learning and programming in a new language required by the parallel architecture they are targeting for [1]. The aim of this research deals with a similar endeavor using A-JUMP [2]. The existing implementation of A-JUMP is written only for Java and the research work presented has enhanced the A-JUMP architecture by enabling message passing between C++, C# and Java applications via A-JUMP, as well as providing client side APIs for the respective languages for application developers.

The remaining part of the paper is organized as follows: section-II covers the literature review; section-III and IV cover the multilingual support for A-JUMP using C# and C/C++ respectively; results are presented in section-V, and section-VI covers the conclusion and future directions.

2. Literature Review

The objective of the literature review is to depict some of the MPI implementations accomplished in C/C++, C# and Python programming languages and their possible analysis is also covered.

2.1. MPI Models in C/C++

LAM [3] is a programming environment and a development system for message passing multicomputer based on UNIX network. It has a modular micro-kernel. It provides a complete API for explicit message passing. The modular structure along with explicit node based communication makes it a suitable choice for building higher-level tools and parallel programming paradigms. It also provides a message buffering system.

MPICH [4] is a MPI implementation integrating portability with high performance. MPICH represents a sign of adaptability with respect to its environment, thereby, implying the portability characteristic at the cost of minimum efficiency loss. MPICH is implemented as a complete implementation of MPI. The communication styles offered include SCI, TCP, and shared memory. Several enhancements and performance improvements became part of the implementation in MPICH2.

Open MPI [5] is an open-source implementation of MPI. It supports homogenous and heterogeneous platforms. The major advantage of this implementation entails optimal communication performance and transparent handling of tasks. The communication protocols supported by Open MPI include shared memory, InfiniBand, Myrinet, TCP/IP, and Portals. The architecture and network properties are determined for each participating node and most efficient communication mode is chosen to achieve maximum performance.

MS MPI [6] is the Microsoft implementation of MPI that is based on MPICH2. MS MPI can be used to easily port the existing code based on MPICH2. It provides security based on Active Directory (AD) directory services and binary compatibility across different types of interconnectivity options. The cluster administrator can specify the resource allocation policies, thus preventing unauthorized jobs from accessing restricted nodes. It also provides fail-safe execution.

2.2. MPI Models in Python

MYMPI [7] is a Python based implementation of MPI standard covering a subset of MPI specifications. It can be used with a standard Python interpreter. It mostly follows the C and Fortran MPI syntax and semantics, thus providing better integration of Python programs with these languages. MYMPI provides better control for writing parallel applications.

pyMPI [8] is a near complete implementation of MPI specifications. The customized interpreter itself runs as a parallel application. This requires different programming style on different machines. The custom Python interpreter calls the MPI_INIT when it starts taking away the control from the programmers.

MPI for Python [9] is an open source, object-oriented Python package providing MPI bindings built on top of MPI-1 and MPI-2 specifications. It is capable of communicating any Python object exposing a memory buffer along with general Python objects.

2.3. MPI Models in C#

The initial release of MPI.Net [10] consisted of two libraries. The first consists of C# bindings, which is a low-level interface following C++ bindings of MPI-2 specifications; and the second is MPI.Net, a high level interface influenced by Object Oriented MPI (OOMPI) based on C++ library. It uses shadow communicators for serialized objects.

3. Multilingual Bindings for A-JUMP using C#

The following language features kept in mind while adding C# support to existing A-JUMP were language popularity and library support; platform independence; and interoperability.

C# is one of the most popular programming language and has an extensive library support behind it. Moreover, it has various improvements over other programming languages including Java, and it is constantly evolving. It is platform independent and does exist on multiple platforms. Mono [13] is a cross platform implementation of C# and Common Language Runtime (CLR) which is binary compatible with Microsoft. Another important factor for adding multilingual bindings is interoperability. The design goal is exchanging messages between different programming languages without the use of any native code. If we can demonstrate message interoperability (message exchange) between applications written in C# and other programming languages it would also be possible to do the same using any other programming language supported by the Microsoft .Net framework [14].

The layered architecture of Multilingual Bindings for A-JUMP is presented in Fig. 1 shown below. The newly added components Message Adapter (MA), Application Manager (AM), and Compatibility Manager (CM) are distinguished in the diagram.

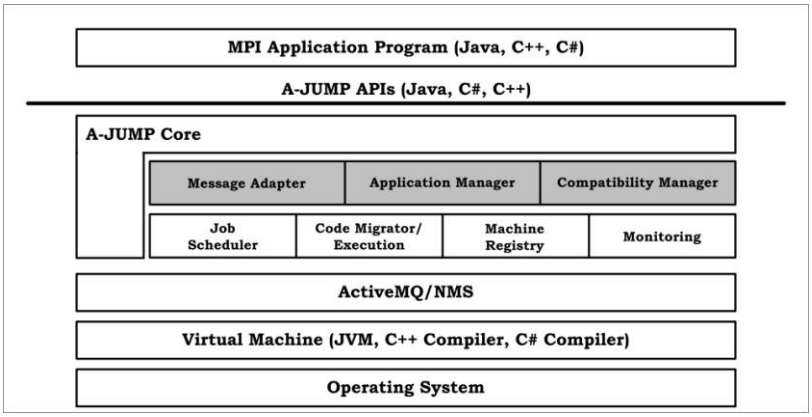

Figure 1. Layered Architecture of Multilingual Bindings for A-JUMP

3.1. Implementation Details

For adding C# language bindings for A-JUMP we have used Apache NMS (.Net Messaging Service) which provides the .Net Message APIs. The NMS API allows building .Net applications using C# and any other languages supported by .Net framework. With NMS APIs, one can connect to multiple message brokers using a JMS

style API. The existing A-JUMP framework is enhanced by adding components (for supporting multiple language bindings), and C# APIs. The component level architecture of Multilingual Bindings for A-JUMP is presented in Fig. 2 shown below.

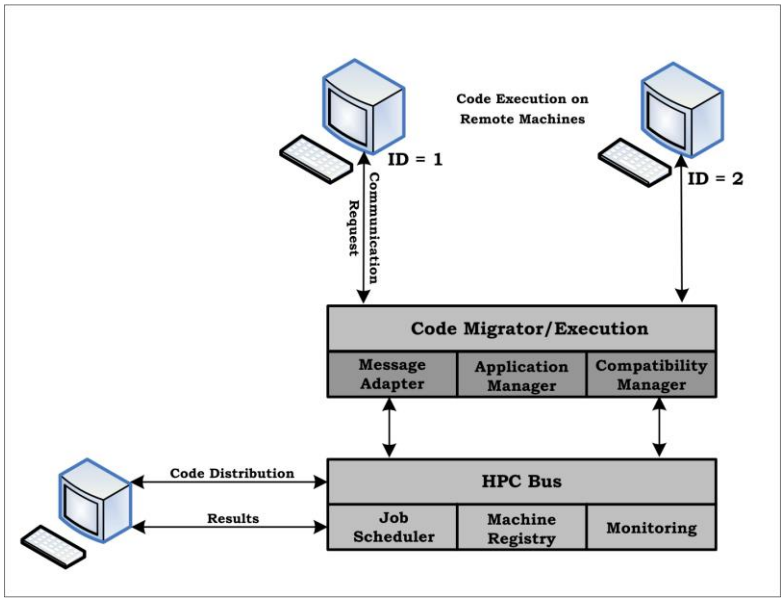

Figure 2. Component Level Diagram of Multilingual Bindings for A-JUMP

The details of the newly added components are discussed as follows;

3.1.1. Application Manager: The Application Manager (AM) passes the source application (the one sending the messages) information to the Message Adapter (MA). The Application Manager ensures that correct information with respect to source application type (i.e., the language in which the application is written) and the host operating system type is passed to the Message Adapter. The messages are transferred in the same way as in original A-JUMP. The MA is running along with Code Execution / Migrator / Adapter on every node that is registered in the A-JUMP architecture. The MA then passes this information to Compatibility Manager (CM) for further processing.

3.1.2. Message Adapter: The Message Adapter (MA) receives all messages being sent and received in the A-JUMP framework. It identifies the type of message received, the source application type, the application message is destined to, destination application type, and the APIs that have been used to send the message. The information received by MA is passed on to the Compatibility Manager (CM). Message Adapter is the message exchange hub and receives all messages being sent and received within the framework. This functionality of the Message Adapter can also be used for message persistence for adding fault tolerance in the A-JUMP framework at the message level.

3.1.3. Compatibility Manager: The Compatibility Manager (CM) makes messages compatible amongst different type of applications running in the A-JUMP framework. The CM performs the actual conversion on the basis of the information received from the Message Adapter. In case the source and destination applications are of same type (i.e., they are written in the same language), it simply forwards the message unchanged to the destination application. Otherwise it converts the incoming message, makes it compatible with the destination application type and then forwards it to the destination

application. The message is converted based on the application identifier (i.e. type of the application), which is also sent along with the message. A reference table is maintained for data type conversions amongst applications written in different programming languages. So far the conversion is only possible for basic data types i.e. integers and characters.

3.1.4. C# APIs: The A-JUMP communication APIs have been enhanced by adding C# APIs that facilitate the programmers in writing parallel code using C#. These APIs are close to MPI specifications for Java (mpiJava 1.2) [11]. The communication APIs offer support for asynchronous communication for message passing and code distribution, which is a distinct feature of A-JUMP among other MPI implementations.

The workflow is same as in original A-JUMP (Java), however all the messages are now sent to the Message Adapter running along with Code Migrator/Execution. The sender application sends the information to Message Adapter. The code distribution mechanism is highlighted in Fig. 3 shown below. The figure shows how the implementation for C# is integrated with current A-JUMP framework. HPC bus has been used for job submission and result collection. It is apparent from the diagram that the new implementation has integrated very well without much change the existing layout. When a user wants to submit a job, he submits it to the HPC bus of A-JUMP which then sends it to Code Adapter for execution. The Code Adapter also assigns ranks to the nodes. The MPI communication calls are provided as APIs for inter process and intra-process communications. C# APIs have also been developed as part of the enhanced framework.

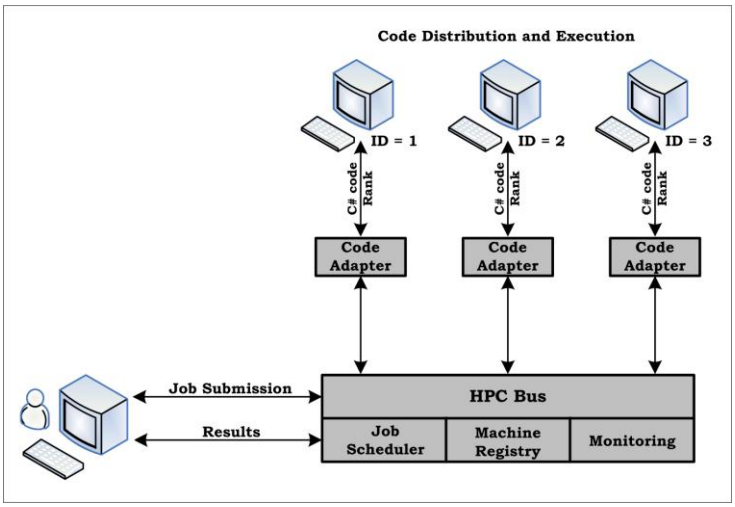

Figure 3. Code Distriubtion in Multilingual Bindings for A-JUMP

4. Multilingual Bindings for A-JUMP Using C/C++

As A-JUMP has been developed on *Apache ActiveMQ*; a Message Oriented Middleware (*MOM*) based on JMS 1.1, it inherits the ability of asynchronous communication between similar applications in a loosely coupled fashion. *ActiveMQ* is not just a JMS broker; it also offers various connectivity modes, thereby acquainting itself as a means of messaging among varied platforms [12]. *ActiveMQ* is a middleware functioning between Java applications. Similarly *ActiveMQ-CPP* is an open source implementation of *CMS*; the C++ Messaging

Service. It is a C++ based middleware, which bears the responsibility to enable communication between C++ applications through any compatible message broker; for our implementation we use *Apache ActiveMQ*.

The reasons for choosing C/C++ include language popularity and library support; platform independence; and interoperability. C language is known for its simplicity, ease of implementation, efficiency, portability, and conciseness. C++ extended the feature set of C for support of development of large programs. Secondly, C/C++ has been equally popular for implementing system software as well as developing application software. Thirdly, the compilers are available for almost all major hardware architectures. The most important reason for choosing C/C++ is availability of ActiveMQ-CPP, which is an open source implementation of CMS that enables message communication between C++ applications. Though the C/C++ developers claim portability as an important feature of the language, but this is not true when we consider portability of applications across different hardware architectures and different operating systems. The applications written in C/C++ cannot be reused under different hardware architecture and OS without recompilation.

Interoperability is an important design goal in adding multilingual bindings to the A-JUMP architecture. The goal is to exchange messages between different applications written in C/C++ and also between C/C++ and other programming languages.

4.1. Implementation Details

In this section, we discussed augmenting the A-JUMP framework by incorporating the C/C++ language bindings. In the first step, we have provided message passing between two C++ applications via A-JUMP JMS broker written in Java. After that, we have incorporated the server and client side APIs for C/C++ for fine-tuning the features available for the developers. Moreover, the inter-language communication is desired as well such that C++ based application could interact with Java based applications.

The cross language protocol used to achieve communications between C++ applications through Java based ActiveMQ broker is *Openwire*. *Openwire* is a native protocol employed by *ActiveMQ*. Message passing could be carried out by putting into use the supported primitive data types under the hood of any of the available message objects including:

- *TextMessage* for string type;
- *ByteMessage* for Java supported byte type;
- *StreamMessage* for handling stream types; and
- *MapMessage* for manipulating a map type comprising of a key/value pair internally.

The AM sends the application information to MA. The MA passes on the message and other necessary information to the CM. The CM examines the message and the information sent along with it. If the sender and receiver application are of same type (i.e., written in same programming language) the message is sent as it is, otherwise the CM does the conversion. The MPI communication calls are provided as APIs for inter process and intra-process communications. C++ APIs have also been developed as part of the enhanced framework.

4.2. Limitations

When we use ActiveMQ-CPP for communication, the messages are sent as byte streams. The primitive data types are also sent in byte-by-byte fashion and sending primitive data types also involves complexity. Because of message communication in the form of byte

streams the communication performance is degraded. The messages size becomes inversely proportional to the communication performance; the larger the message size the worse is communication performance. This slow communication performance is somewhat compensated with execution performance of C/C++ applications. The applications written in C/C++ especially the CPU bound applications show better execution performance than Java applications, and the overall performance is improved.

Another limitation we faced working with C/C++ is portability across different hardware architectures and hybrid operating systems. The application written and compiled under one hardware platform and operating system does not run on different hardware architecture and OS without recompilation.

5. Performance Measurements and Results

The performance measurement analysis of the implementations is presented using the Code Speed-up Performance Analysis; and Communication Performance Measurement.

5.1. Test Environment

For performance measurement tests, a cluster of eight single core machines with 3.2 GHz CPU has been used. These machines are installed with Windows XP operating system (with Service Pack 3) and one (01) GB of main system memory (RAM; the installed memory is DDR2 667 MHz). All machines have single copper based Gigabit LAN interface. The TCP window size is default on all of the machines.

5.2. Test Results

5.2.1. Code Speedup Performance Measurement: For code speed-up, we have used bubble sort because it offers worst case sorting time. This test has provided valuable information to study the pattern of code execution and communication. For comparing A-JUMP (C/C++) performance we have used two other implementations, i.e. MPICH and A-JUMP (Java). For C# implementation, we have compared the performance results with original A-JUMP (Java) only. The test results for various data size for C/C++ are presented in Table I, and for C# in Table II. Fig. 4, Fig. 5, Fig. 6, and Fig. 7 illustrate the performance comparison for each data set discretely for C/C++ implementation. Fig. 8 indicates the code speed-up factor for these results using A-JUMP (Java) as baseline.

Fig. 9, Fig. 10, Fig. 11, and Fig. 12 illustrate the performance comparison for each data set discretely for C# implementation. Fig. 13 indicates the code speed-up factor for these results using A-JUMP (Java) as baseline.

Table I Code Speedup Performance Results A-JUMP (C/C++)

No. of elements	Time (in ms) for bubble sort		
	A-JUMP	MPICH	A-JUMP (C/C++)
10K	62	97	541
100K	5062	9424	6003
200K	20250	37629	16420
500K	127032	235000	74068

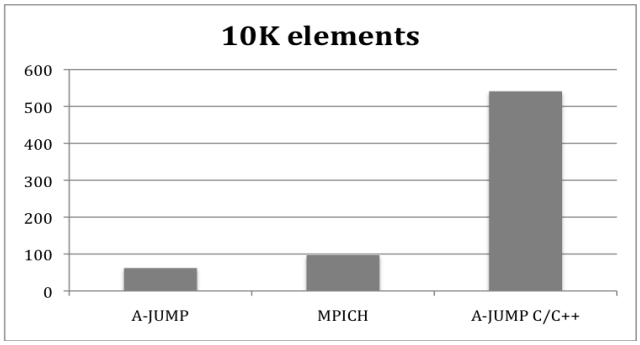

Figure 4. Execution Time (in ms) for 10K Elements

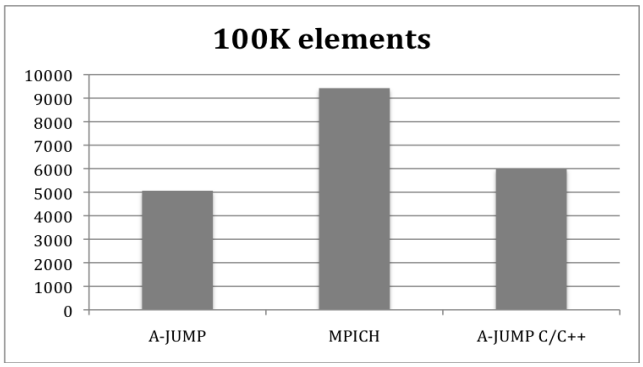

Figure 5. Execution Time (in ms) for 100K Elements

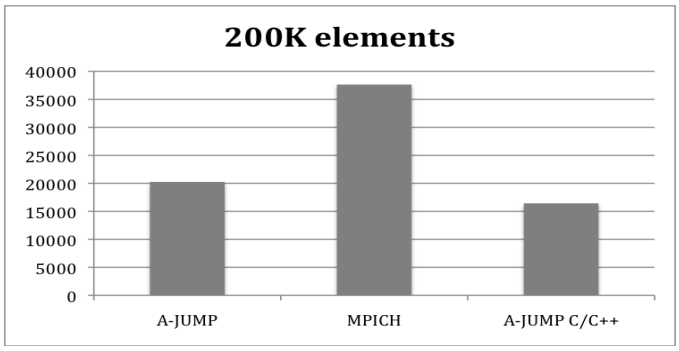

Figure 6. Execution Time (in ms) for 200K Elements

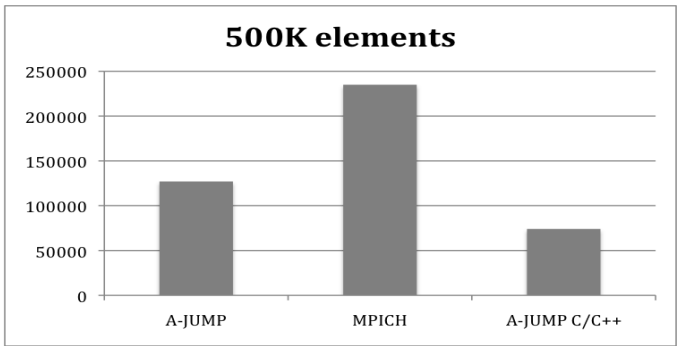

Figure 7. Execution Time (in ms) for 500K Elements

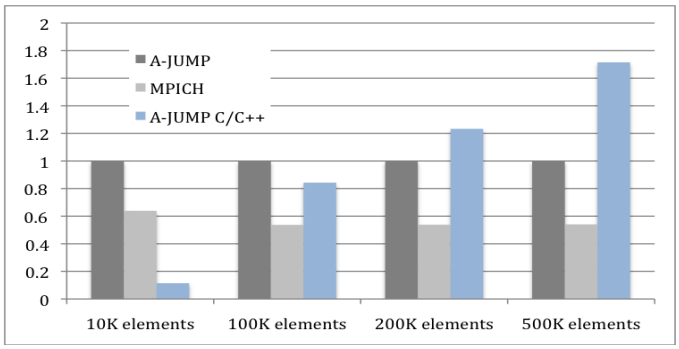

Figure 8. Speedup Gains A-JUMP (C/C++) (Higher is better)

For the first two datasets (i.e. 10K and 100K elements) the performance of A-JUMP C/C++ implementation is not good as compared to other two models. This is due to the startup/initialization delays. As the dataset size increases better performance is observed. The initialization delays are compensated mainly due to the fact that code execution performance of C/C++ implementation is better than the Java implementation. For the larger datasets the A-JUMP C/C++ implementation comprehensively beats the other two implementations.

These results demonstrate that A-JUMP (C/C++) has performed better as compared to other code execution models. Despite the initialization overheads the code speedup performance of C/C++ implementation is much better for larger data sets. The performance comparison of C# implementation is presented in Table II below.

Table II Code Speedup Performance Results A-JUMP (C#)

No. of elements	Time (in ms) for bubble sort	
	A-JUMP	A-JUMP (C#)
10K	62	156
100K	5062	26350
200K	20250	107468
500K	127032	678984

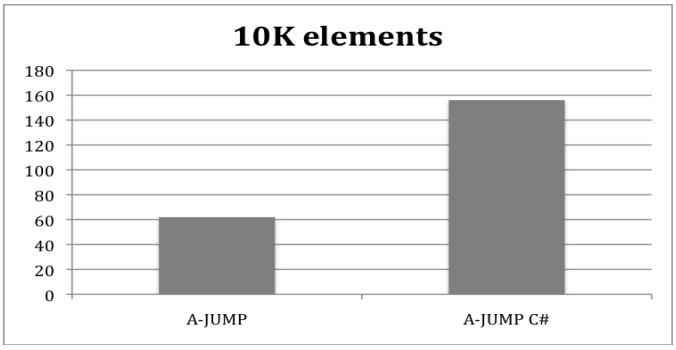

Figure 9. Execution Time (in ms) for 10K Elements

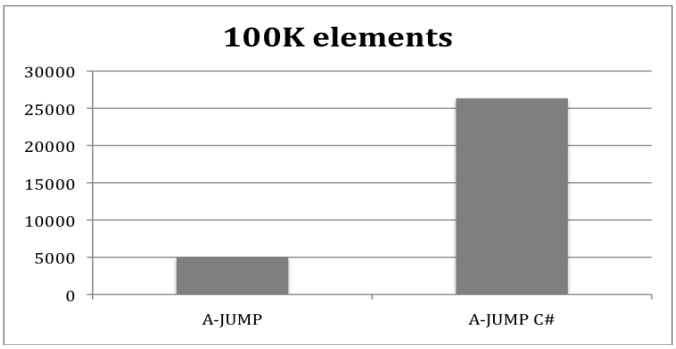

Figure 10. Execution Time (in ms) for 100K Elements

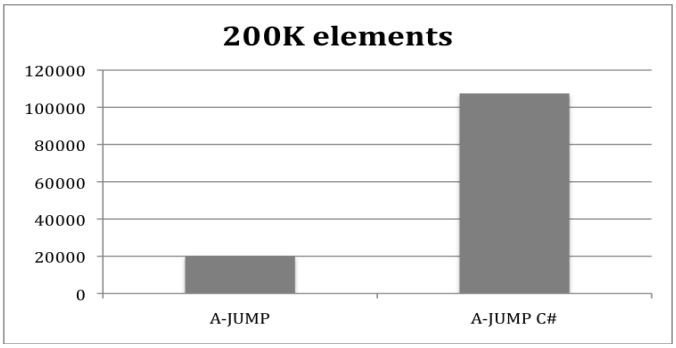

Figure 11. Execution Time (in ms) for 200K Elements

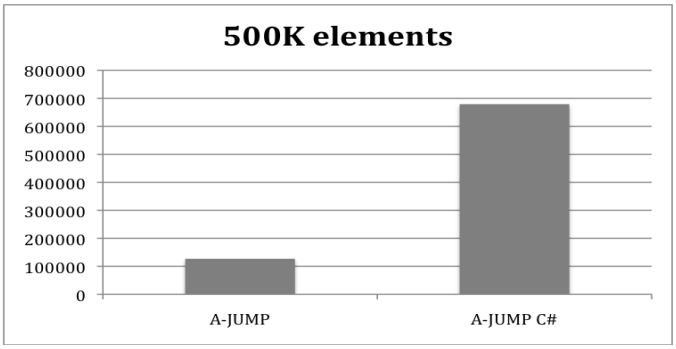

Figure 12. Execution Time (in ms) for 500K Elements

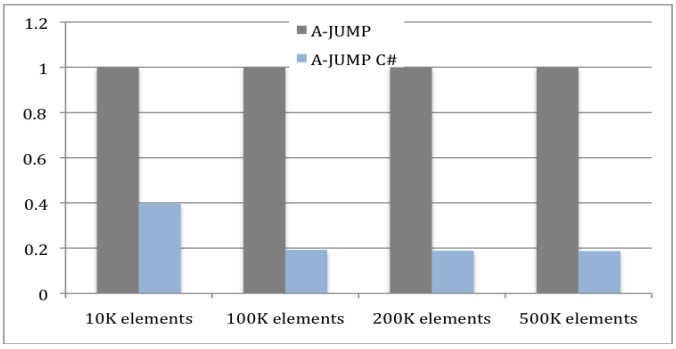

Figure 13. Speedup Gains of A-JUMP (C#) (Higher is better)

For all data sizes, the performance of A-JUMP (Java) is better than A-JUMP (C#) implementation. We anticipate that this poor performance is a result of the initialization and communication overheads involved with ActiveMQ NMS libraries. We further see that as the data sizes grow the performance is degraded and this degradation is also linear.

5.2.2. Communication Performance Measurement (Ping Pong Latency): Ping Pong communication benchmark is well-established method to measure the communication latencies of a given framework. It sends messages back and forth between two processes. It is based on blocking MPI send() and receive() mechanism. The test provides latency results on unidirectional sending and receiving of messages. It is repeated 1000 times for various message sizes. Fig. 14 contains the comparison between A-JUMP (Java), MPICH and A-JUMP (C/C++). The message latency time of A-JUMP (C/C++) is almost similar to A-JUMP and MPICH for smaller message sizes (up to 8K). As the message size increases the latency increases as well. As A-JUMP (Java) also shows higher network latencies for larger message sizes. Another delay factor is the interoperability between different languages at the message broker level. Fig. 15 contains the comparison between A-JUMP (Java), MPJ Express, MPICH and A-JUMP (C#). It is evident from the figure that the A-JUMP (C#) has the highest network latencies and they are increasing with the increase in the message size.

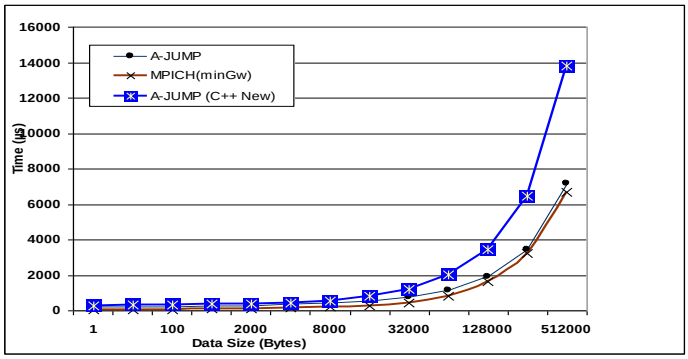

Figure 14. PingPong latencies (µs) for A-JUMP (C/C++)

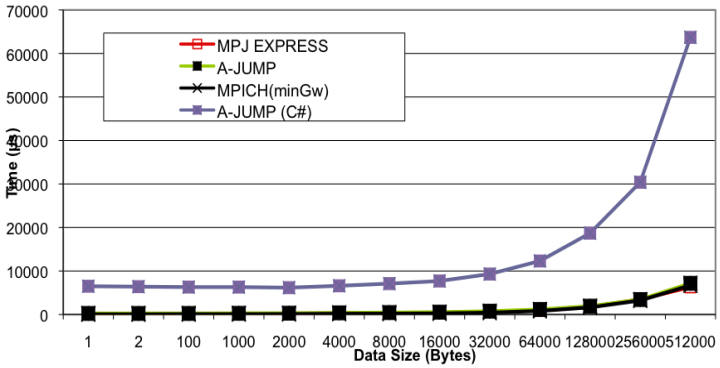

Figure 15. PingPong latencies (µs) for A-JUMP (C#)

5.2.3. Communication Performance Measurement (Network Throughput Measurement): Network Throughput Measurement shows how efficiently A-JUMP C/C++ and C# implementations consume available network bandwidth. The results have

demonstrated that the values for network throughput for A-JUMP (C/C++) are comparable to other implementations up to the message sizes of 8K, see Fig. 16. With message sizes bigger than 8K the network bandwidth consumption degrades. For A-JUMP C# implementation the results for network performance are shown in Fig. 17. The results demonstrate very poor network bandwidth consumption by A-JUMP C#, which is almost ten times lower than the other compared models.

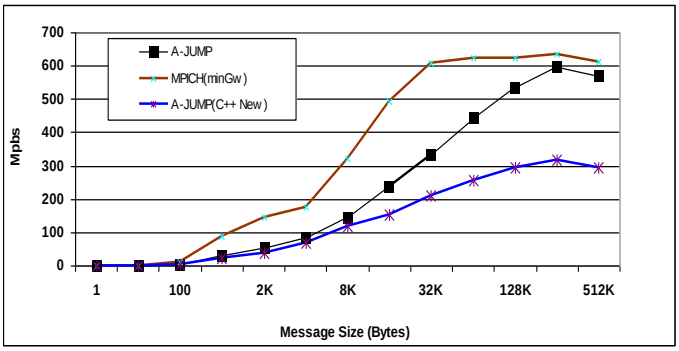

Figure 16. Network Throughput for A-JUMP (C/C++)

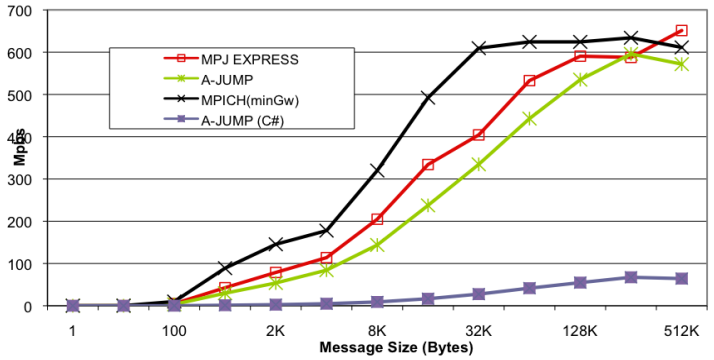

Figure 17. Network Throughput for A-JUMP (C#)

6. Conclusion and Future Directions

In this research work, we augmented the A-JUMP framework by adding multilingual bindings using C/C++ and C# programming languages. These implementations have been integrated with the existing framework. Message passing has been demonstrated between applications written in different programming languages including C/C++ and C#. Moreover, APIs for these languages have also been incorporated. The performance results for ping pong (network latency) and code speed-up for different language implementations have demonstrated that initialization/communication overheads introduced by the underlying libraries reduce the communication performance. We further noticed that the communication performance for larger message sizes is degraded which is inherited from original A-JUMP implementation written in Java. In future, we plan to measure the communication performance between C#, C/C++ and Java applications and also add Python language bindings to A-JUMP.

References

[1] Al Geist et al., "MPI-2: Extending the Message-Passing Interface", Workshop 01 Programming Environment and Tools, LNCS, January 2006, pp. 128 – 135.

[2] S. Asghar, M. Hafeez, U.A. Malik, A. Rehman, and N. Riaz, "A-JUMP, Architecture for Java Universal Message Passing", 8th International Conference on Frontiers of Information Technology (FIT), December 21-23, 2010, Islamabad, Pakistan. doi>10.1145/1943628.1943662

[3] G. Burns, R. Daoud and J. Vaigl, "LAM: An Open Cluster Environment for MPI" Ohio Supercomputer Centre, Columbus, Ohio, 1990.

[4] W. Gropp, E. Lusk, N. Doss and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," Parallel Computing 22 (1996), pp: 789-828.

[5] R. L. Graham et al., "OpenMPI: A High-Performance, Heterogeneous MPI," International Conference on Cluster Computing, 2006.

[6] Microsoft MPI (MS MPI) [Online], available at: http://msdn.microsoft.com/en-us/library/bb524831(v=vs.85).aspx. (Accessed May 2011).

[7] T. H. Kaiser, L. Brieger and S. Healy, "MYMPI – MPI Programming in Python", in Proceedings of the International Conference on Parallel and Distributed Processing Techniques USA, June 2006.

[8] P. Miller, "pyMPI – An Introduction to parallel Python using MPI," UCRL-WEB-150152, September 2002.

[9] L. Dalcin, R. Paz m M. Storti and J. D. Elia, "MPI for Python: Performance improvements and MPI-2 extensions", J. Parallel Distributed Computing 68 (2008), pp. 655 – 662.

[10] J. Willcock, A. Lumsdaine, and A. Robison, "Using MPI with C# and the Common Language Infrastructure", Concurrency and Computation: Practice & Experience, 17(7-8), June/July 2005 pp. 895–917.

[11] B. Carpenter (2002) MPJ specifications (mpiJava 1.2: API Specification) homepage on HPJAVA [Online], available at: http://www.hpJava.org/reports/mpiJava-spec/mpiJava-spec/mpiJava-spec.html. (Accessed May 2011).

[12] B. Snyder, R. Davies and D. Bosanac, "Active MQ in Action", 1st ed., Manning Publications, March 2011.

[13] Mono [Online], available at: http://www.mono-project.com. (Accessed May 2011).

[14] Microsoft .Net Framework [Online], available at: http://www.microsoft.com/net. (Accessed May 2011).

Authors

Usman Ahmad Malik obtained his M.S. degree in Computer Sciences from Shaheed Zulfikar Ali Bhutto Institute of Science and Technology (SZABIST). He is working at National Centre for Physics (NCP) with Advanced Scientific Computing (ASC) group. His research interests include parallel and distributed computing, and information security.

Sajjad Asghar obtained his M.S. degree in Software Engineering from Shaheed Zulfikar Ali Bhutto Institute of Science and Technology (SZABIST). He is currently doing his PhD with University of New Brunswick, Canada. He is working at National Centre for Physics (NCP) with Advanced Scientific Computing (ASC) group. His research interests include parallel and distributed computing.

Mehnaz Hafeez obtained her M.S. degree in Software Engineering from Shaheed Zulfikar Ali Bhutto Institute of Science and Technology. Her research interests include parallel and distributed computing.

Adeel-ur-Rehman is an M.S. Degree candidate in Computer Science from Shaheed Zulfikar Ali Bhutto Institute of Science and Technology (SZABIST). He is working at National Centre for Physics (NCP) with Advanced Scientific Computing (ASC) group. His research interests encompass parallel and distributed computing.

Naveed Riaz received his Ph.D. from Graz University of Technology (2008) in Software Engineering and a M.S. in Software Engineering (2005) from National University of Sciences and Technology (NUST), Pakistan. His research interests include Model-based and Qualitative Reasoning, Theorem Proving, Verification and Validation, Test Pattern Generation, Parallel Computing, Real-Time Systems and Software Engineering.