Advanced Architecture for Java Universal Message Passing (AA-JUMP)

Adeel-ur-Rehman[1] and Naveed Riaz[2]
[1]National Centre for Physics, Pakistan
[2]School of Electrical Engineering and Computer Science, National University of Science and Technology, Pakistan

Abstract: *The Architecture for Java Universal Message Passing (A-JUMP) is a Java based message passing framework. A-JUMP offers flexibility for programmers in order to write parallel applications making use of multiple programming languages. There is also a provision to use various network protocols for message communication. The results for standard benchmarks like ping-pong latency, Embarrassingly Parallel (EP) code execution, Java Grande Forum (JGF) Crypt etc. gave us the conclusion that for the cases where the data size is smaller than 256K bytes, the numbers are comparative with some of its predecessor models like Message Passing Interface CHameleon version 2 (MPICH2), Message Passing interface for Java (MPJ) Express etc. But, in case, the packet size exceeds 256K bytes, the performance of the A-JUMP model seems to be severely hampered. Hence, taking that peculiar behaviour into account, this paper talks about a strategy devised to cope up with the performance limitation observed under the base A-JUMP implementation, giving birth to an Advanced A-JUMP (AA-JUMP) methodology while keeping the basic workflow of the original model intact. AA-JUMP addresses to improve performance of A-JUMP by preserving its various traits like portability, simplicity, scalability etc. which are the key features offered by flourishing High Performance Computing (HPC) oriented frameworks of now-a-days. The head-to-head comparisons between the two message passing versions reveals 40% performance boost; thereby suggesting AAJUMP a viable approach to adopt under parallel as well as distributed computing domains.*

Keywords: *A-JUMP, java, universal message passing, MPI, distributed computing.*

Received February 5, 2015; accepted December 21, 2015

1. Introduction

To process computation intensive tasks, the trend of traditional supercomputers got evolved in favour of commodity computing i.e., utilization of cluster of computers in order to achieve the throughput of traditional supercomputers with economical infrastructure cost. This derived model leads us to the fascinating world of parallel computing. On the other hand, in order to solve more complex computing problems, it's still a challenge to provide extensive infrastructure within a single physical cluster. Thus, in order to get such problems untangled in an efficient manner, a robust High Performance Computing (HPC) solution is needed [10].

This paper deals with a parallel code execution framework written in pure Java i.e., it doesn't incorporate any native libraries/directives. The framework primarily supports asynchronous communication mechanism in order to deal with message passing as well as for distribution of code over the cluster. Hence it is named as Architecture Java Universal Message Passing (A-JUMP) [3] framework. A-JUMP launches a new technique of communication in the form of an HPC bus which entails an open-source implementation of Java Message Service (JMS) specification 1.1, namely Active Message Queue (ActiveMQ) by Apache. It supports various communication protocols in order to entertain message passing. Moreover, its representation layer (application) and the business layer (communication) are not tightly coupled giving the advantage of modifying the application without having any need to touch the underlying communication mechanism. The performance of A-JUMP has been calculated based on the standard communication test such as ping-pong latency tests, as well as Java Grande Forum (JGF) and NASA Advanced Supercomputing Division (NAS) benchmark tests. It was observed that its performance looked promising with message sizes smaller than 256K bytes. But when the message size exceeds that limit, the figures get disturbed to a great extent. Based on this observation and information, this research work revolves around the attempt to cope up with the known performance limitation of original A-JUMP implementation in the form of its derived model titled Advanced A-JUMP (AA-JUMP) keeping the basic workflow of the original model intact.

This paper is organized as follows: Our area of focus covering common underlying message passing mechanisms as well as domains of interest are discussed in section 2, basic architecture and workflow of original A-JUMP model is presented in section 3, the advanced A-JUMP (AA-JUMP) explaining the improvement methodology is covered in section 4, one-to-one comparisons between the original and the enhanced model is portrayed in section 5 while section 6 reveals the conclusion and possible future work. As far as the Related Work part is concerned, we have already discussed that in detail in a separate paper published earlier [6]. In that paper, we had come up with a comparative survey report of various Message Passing Interface (MPI) implementations available. In context of the current paper, the desired comparison report could be consulted from the survey paper under section 3.2.

2. MPI and Java

MPI communication involves data marshalling of primitive types which leads to high message latency. Nonetheless, adoption of Java language under HPC realm has become common due to many of powerful features of Java such as platform independence, portability, purely object oriented, sound memory management, support for multi-threading, security, built-in communication libraries, very rich collection of Application Programming Interfaces (APIs) etc., Hence, numerous attempts have been made for offering Java oriented implementation of MPI. Implementations of message passing libraries under Java mostly possess their individual MPI resembling Java language binding. Such Java implementations are built using techniques including Java sockets, Java Remote Method Invocation (RMI), Java Native Interface (JNI) etc.

2.1. Communication Models for Java HPC

A number of implementations regarding messaging passing libraries under Java exist today. Most of them employ either of the following approaches:

- Use of Java Native Interfaces (JNI).
- Java Remote Method Invocation (RMI).
- Low-level Java Sockets.

Even though none of these low level models directly offer message passing facility but upper layer libraries could be constructed on top of these in order to develop parallel applications under Java. The most significant aspects for HPC models include portability, performance, ease of use, and scalability. Interestingly, neither of these implementations provides all of these important features rather each of them have a subset of those to present.

Moreover, Java language has copious tempting qualities to offer including but not limited to multi-threading support, simplicity, portability, and user-friendly network libraries which attest it as a pleasant option for building HPC architectures. In addition, Java threads could be engaged to develop shared memory programs on multi-core CPUs, e.g., JOMP [4]. Such implementation types haven't been discussed here as Java 7- the new version of Java [8] is offering a built-in feature to develop OpenMP codes by utilizing fork-join model.

2.2. Approaches of Interest

Three major approaches stand out among others:

- *JMS*: Their capability of offering powerful merging regarding various application types permits individual components to be integrated in order to develop seriously scalable, expandable, and trustworthy systems. Today, ActiveMQ [1], because of its built-in features, is considered one of the most appropriate choices to be used as a JMS implementation to perform message passing activities under Java oriented communication models.
- *AMQP*: AMQP [2] is recognized as an open standard, binary protocol for Message Oriented Middleware (MOM) for offering proficient support across various message passing application and data communication patterns. It provides comprehensive functional interoperability among its complying clients and middleware servers in charge of messaging i.e., brokers.
- *Super Sockets (ZeroMQ)* [11]: is a C++ based high-performance messaging library using a socket interface without having to deal with the intricacies of a full-fledged messaging system. Applying Zero Message Queueing (0MQ) framework proves to be very simple as compared to its predecessor models because of being merely a socket library to carry out communications. This approach also achieves scalability by utilizing Pragmatic General Multicast (PGM) protocol which deals with transferring data to multiple end-points by implicitly exercising load balancing over them. Even portability feature could be achieved to certain extent by using 0MQ as it supports several language bindings covering nearly most of the popular languages of today including Java. We have found that 0MQ can prove itself as the most effective Java HPC model for data communication comparing to the other approaches mentioned above.

3. Architecture for Java Universal Message Passing

3.1. Overview

The communication performance is the major concern for High Performance Computing (HPC) community. MPI implementation could benefit from JMS to provide network independent, heterogeneous, and architecture neutral message passing over LAN, WAN, peer-to-peer and Grids. Therefore the evaluation of MPI implementation using JMS becomes an interesting task.

An effort in such direction has been made to develop an MPI framework based on pure Java and JMS to entertain parallel execution. The model is named as Architecture for Java Universal Message Passing (A-JUMP) [3]. Java is chosen as the base programming language for its implementation as today it is the most renowned language for developing platform independent applications [9]. Thus, A-JUMP does not involve any native code or libraries. AJUMP follows MPI 1.2 specifications [5]. It also offers asynchronous communication mode for dealing with message passing as well as in order to distribute the code to be executed over the clusters. It has the provision of supporting various communication protocols to perform message passing. The communication layer of A-JUMP is kept isolated from the underlying application code. This leads the possibility of adoption of various JMS based implementations without touching the application level code. The framework achieves parallel execution of code supporting homogeneous as well as heterogeneous clusters.

3.2. Architecture

The main components of A-JUMP along with their brief description is covered in this section.. A collection of APIs is also shipped with the framework in order to facilitate developing MPI oriented code.

- *Scheduler*: The responsibility of the Scheduler component is to offer submission of incoming jobs to the cluster resources available.
- *Monitoring*: The scheduler collects information from the Monitoring component. The purpose of the Monitoring component is to witness the dynamic information regarding the computing resources included in the framework. It also has to keep track of number of busy or free computing resources.
- *Registry*: This component has to maintain a list of resources that are introduced or eliminated from the cluster in a dynamic fashion. The information stored about each registered machine includes number of processors, cores per processor and volume of overall system.
- *JMS*: The JMS component serves as the backbone for communication in A-JUMP. The communication could be inter-process, monitoring and circulation of Registry related information. JMS is employed in order to have the business and communication layers of the implementation separated.
- *Output Controller*: The prime purpose of this component is to establish the synchronization among the machines, gather job outputs from every machine and transmit it towards the initiating machine. Again all this is achieved through JMS.
- *Code Migrator*: This component deals with distributing and executing jobs over the network. Once a resource is found free for accepting a job, Scheduler passes it on to that machine. The class file corresponding to the job is loaded via Code Migrator. The component level illustration is provided under Figure 1. Afterwards, the Java Virtual Machine (JVM) on the destination machine loads the class file received for its execution. The output of the job is then forwarded to the user machine. The fundamental send() and receive() routines are also provided under communication libraries provided by the framework in pure Java.
- *Communication APIs*: A-JUMP is also comprised of a communication API in order to facilitate Peer-to-Peer (P2P), selective as well as collective communication. The selective and collective paradigms could be mapped to multi-cast and broad-cast communication strategies. Due to JMS supervising the communication and distribution workflows, A-JUMP achieves that in an asynchronous fashion [3].

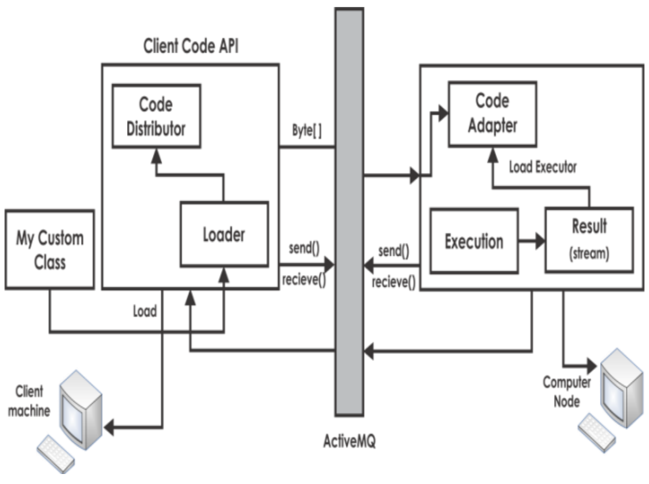

Figure 1. Component level diagram of code migrator/ execution.

3.3. Workflow

First of all; an end user loads a Java class via client APIs shipped by the framework related to the Code Migrator module. The class is then transformed into byte code and then forwarded to the Scheduler component making use of JMS for its submission over the appropriate and available resources on the cluster. Each of the machines assigned the job gets a separate code replica and starts running it in parallel fashion. The message passing is performed in an asynchronous mode of communication because of JMS. The Scheduler components keeps the job queued temporarily until any free resources are identified for its execution. Once identified, the job is sent to the corresponding resources. The cluster administrator is capable for increasing or decreasing the number of jobs according to the situation of resources available. The relationships between the components constituting A-JUMP is shown under Figure 2.

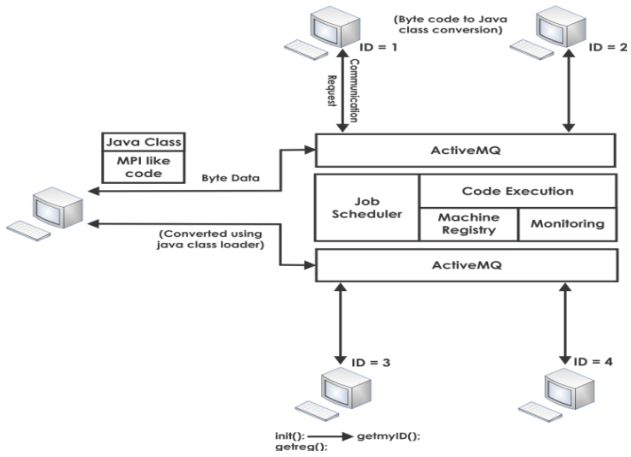

Figure 2. Component level diagram A-JUMP.

4. Advanced Architecture for Java Universal Message Passing

4.1. Why AA-JUMP

The original A-JUMP implementation depicts promising results when compared with their predecessor models like P2P-MPI, MPJ Express, MPICH etc., for message passing, but this happens for a small-to-medium amount of data transfers only i.e. up to 256KB. However, if the message size goes beyond that level, the performance gets severely hampered [3]. It was claimed that the performance limitation is coming from the HPC bus, which in turn comprises of a JMS based middleware i.e., ActiveMQ [1]. Till present, all of the previous A-JUMP implementations employed ActiveMQ for incorporating the JMS specifications into the framework. Also, ActiveMQ has been identified to be the source of the performance limitation observed [7].

This observation leads us towards a direction chasing which, we could hit and adjust the area for boosting up the performance figures reported by the current A-JUMP models. Hence, in order to address the issue, one has to hunt for a newer and superior message framework in lieu of ActiveMQ, which has the ability to provide evidence for better performance metrics.

4.2. Why ZeroMQ Could be Potential Choice

ZeroMQ can be reckoned as a middle level messaging system as it integrates the performance and flexibility of low level messaging systems with the simplicity and ease-of-use of a high-level one.

This research work deals with seeking a high-performance messaging framework and ZeroMQ is particularly eminent because of that feature. It is claimed to be much faster comparing with most of AMQP oriented messaging systems. Some of the reasons for its top notch performance are; lacking the overhead of a full-fledged protocol/system such as AMQP, provision to use efficient transport schemes such as reliable multicasting, ability to deal with smart message batching [11].

Upon choosing a model with some eagerly desired feature, one should not take for granted the shortcomings of the model if comprised of other major features shared by such type of frameworks. So, in our case we were wary regarding the simplicity and scalability features more than other aspects. Fortunately enough, ZeroMQ also offers those features while keeping intact its prime trait-performance. Consequently, when combing all these features, ZeroMQ unanimously wins the contest and looks perfect choice for employment under A-JUMP.

4.3.Comparing Active MQ and ZeroMQ

Although the leading purpose of both the middleware i.e., ActiveMQ by Apache and ZeroMQ by iMatix Corp. is similar but still various differences could be observed between both these models when explored technically. Therefore, it would be a nice idea to present one-to-one comparison between the two approaches. Table 1 below is serving the job for us.

Table 1. Comparison summary of ActiveMQ and ZeroMQ.

Area/ Models	**ActiveMQ**	**ZeroMQ**
Implementation	Pure Java	C/C++
Communication	JMS	Super Socket
Topology	Message Broker	Broker/Broker-less
Performance	Promising	Much Faster and lightweight
Deployment	Non-Trivial, stand-alone process	No dedicated process requirement
Monitoring	Third-party web console	Non-trivial; could be done using legs or network monitoring
Message Persistence	KahaDB	No built-in support

4.4. Integerating ZeroMQ with A-JUMP

Acquaintance with ZeroMQ revealed at least two feasible approaches to deal with it to address our need as identified below:

- Completely replacing the ActiveMQ layer by a ZeroMQ layer.
- Introducing a ZeroMQ wrapper over ActiveMQ intelligently.

The wiser approach to go with would be by making best use of both the middleware features together somehow.

Consequently, one should try to modify our communication layer in such a way that for initial setup of queues, connections, producers and consumers, message persistence, monitoring etc, ActiveMQ should be serving the client application, like we have been using it since the inauguration of the original A-JUMP framework. However, in order to deal with message passing between the business processes, ZeroMQ should come into action. The modification is proposed only for data transfer part of the workflow as it is this piece of code of ActiveMQ that would be mostly responsible for the higher message latencies we were getting. Thus, we have attempted to adopt the ZeroMQ Wrapper oriented approach. For making our choice able of being implemented, we should first of all keep in mind that we need to address a framework (i.e., A-JUMP) which is purely Java based. On the other hand, ZeroMQ is a C++ based model but fortunately enough, it offers support for several language bindings which are currently popular including Java. Ultimately for our model to get into work, we need to setup ActiveMQ, ZeroMQ based implementation along with Java binding of ZeroMQ on top of that on every machine intending to get involved as an execution node under the available cluster resources.

4.5. AA-JUMP Architecture

Most of the structure suggested by the original A-JUMP is kept intact as described earlier under section 4.4. The only change which is made is under the message passing part of the ActiveMQ layer.

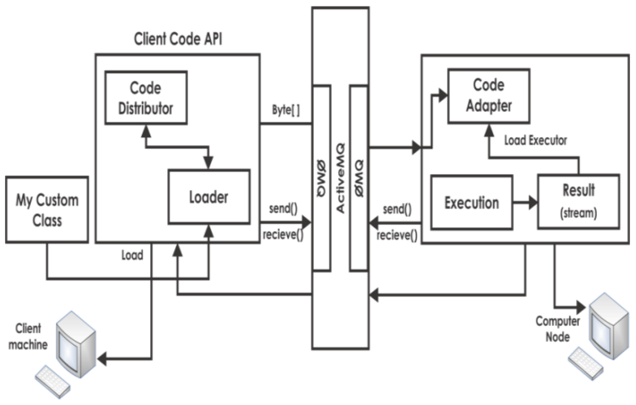

Figure 3. Code execution component of AA-JUMP.

Here the execution of the client application on the sender side is delegated to ZeroMQ which performs the actual transportation of data behind the scene, with its corresponding ZeroMQ layer on the receiver end receives the data sent over the framework and transfers the data back to neighboring ActiveMQ layer which in turn returns the data to the receiving process.

The Code Execution component of AA-JUMP is illustrated in Figure 3.

Basic workflow remains coherent with the original A-JUMP. However, the only difference is introduced at the data communication level during the code migration/execution step where now an additional ZeroMQ layer is introduced where the control transfers between the two middleware on both the ends of the transfer. In other words, in this new implementation, the send() and receive() function calls being utilized by ActiveMQ have been wrapped up by the send() and receive() counterparts provided by ZeroMQ framework which do not use JMS by default rather it provides an abstraction layer on top of traditional low level socket API with customized and enhanced features support.

5. Performance Analysis

5.1. Test Environment

The computational machines used in performing the tests comprised of a cluster of nodes having 2x2 quad-cores Intel Xeon CPU @ 3.16GHz, 16 GB RAM (2GB/Core), 16 GB RAM and a Gigabit interconnect. The OS running on them is Scientific Linux SL release 5.3 (Boron). The machines have been running with no adjustments made under default configurations for TCP Window size, as well as no optimizations performed under hardware/OS. The test results have been obtained manually on the aforementioned machines while the corresponding graphs have been plotted using MS-Excel tool.

5.2. Code Execution Performance

We have compared AA-JUMP with the original A-JUMP by evaluating it against Embarrassingly Parallel (EP) benchmark performance considering Class A.

As could be noticed in Table 2 below, both approaches are offering competing figures. The reason behind is that the proposed AA-JUMP didn't eliminate ActiveMQ layer from its implementation and all the initialization is still done by ActiveMQ, that's why we have to bear with that and performance gained due to introduction of ZeroMQ could not make it dominant as it plays only data communication oriented role in the whole schema.

Table 2. EP Results for AA-JUMP.

Number of CPUs	Times(s)	
	A-JUMP	AA-JUMP
2	39.75	38
4	21.7	19.5
8	15.55	9.95

5.3. Communication Performance

This category of measurement is performed using ping-pong test and network throughput calculations. In case of ping-pong test of communication, a variety of message sizes under the range from 1KB to 1 MB were transferred among two processes executing across two independent machines. Figure 4 depicts the ping-pong latency comparison between A-JUMP and AA-JUMP while network throughput comparison of the two models could be observed under Figure 5.

From Figure 4, it is obvious that in terms of network communication, AA-JUMP wins over its previous implementation as the test mainly emphasize on the data transfer activity; which is the responsibility of ZeroMQ in the proposed AA-JUMP model unlike its parent AJUMP model where ActiveMQ was solely responsible for its communication layer. Figure 5 depicts that unlike original A-JUMP, the proposed model is consuming consistent network bandwidth across a large range of message sizes.

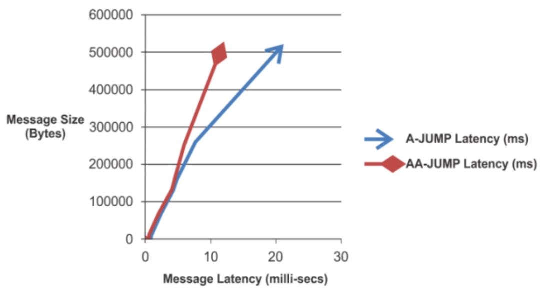

Figure 4. Ping-pong latency for A-JUMP vs AA-JUMP.

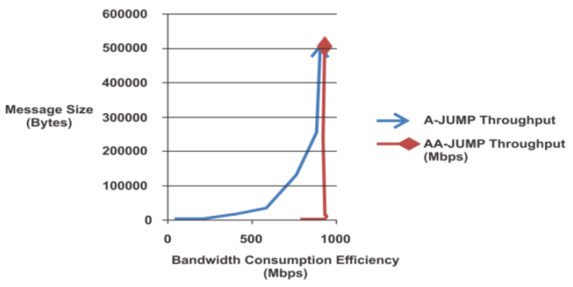

Figure 5. Network throughput measurements (Higher is Better).

6. Summary and Conclusions

A-JUMP was built to enable MPI based java applications to achieve inter-process communication and code distribution over a cluster. But it was observed that upon increase in message size up to or over 256K, A-JUMP exhibits degraded performance in terms of increased message latency and low network throughput.

In quest of finding the solution to this limitation, we have launched its advanced version known as AA-JUMP which seems to resolve the persisting issue to a reasonable extent such that around 40% performance boost is successfully achieved under message passing. The mission is accomplished by adding up an additional layer of ZeroMQ messaging library in order to entertain message transfer on top of ActiveMQ middleware implemented by the original version.

As the proposed AA-JUMP framework is employing ZeroMQ on top of ActiveMQ, it experiences a linear overhead during its message passing. If one has to get rid of that overhead, a complete replacement of the communication layer should be aimed. In future we look forward to further increase the performance by completely replacing the ZeroMQ with a pure Java version of ZeroMQ thereby likely to avoid the overhead being faced due to an additional ZeroMQ Java binding layer.

References

[1] ActiveMQ homepage, *available at http://activemq.apache.org*, Last Visited, 2014.

[2] AMQP homepage, *available at http://www.amqp.org*, Last Visited, 2014.

[3] Asghar S., Hafeez M., Malik U., Rehman A., and Riaz N., "A-JUMP, Architecture for Java Universal Message Passing," *in Proceedings of the 8th International Conference on Frontiers of Information Technology*, Islamabad, 2010.

[4] Bull J. and Kambites M., "JOMP- an OpenMP-Like Interface for Java," *in Proceedings Of ACM Java Grande Conference*, San Francisco, pp. 44-53, 2000.

[5] Carpentar B., Fox G., Hoon S., and Lim S., "Mpijava 1.2: API Specification," *available at www.open-mpi.org/papers/mpi-java-spec*, Last Visited, 2014.

[6] Hafeez M., Asghar S., Malik U., Rehman A., and Riaz N., "Survey of MPI Implementations," *in Proceedings of International Conference on Digital Information and Communication Technology and Its Applications*, Dijon, pp. 206-220, 2010.

[7] Hafeez M., Asghar S., Malik U., Rehman A., and Riaz N., "Secure Peer to Peer Message Passing using A-JUMP,*" in Proceedings of The International Symposium on Grids and Clouds and the Open Grid Forum Academia Sinica*, Taipei, pp. 1-9, 2011.

[8] Java 7 homepage, available at https://jdk7.java.net, Last Visited, 2014.

[9] Judd G., Clement M., and Snell Q., " Distributed Object Group Metacomputing architecture," *Concurrency and Computation*, vol. 10, no. 11-13, pp. 977-983, 1998.

[10] Munir E., Ijaz S., Anjum S., Khan A., Anwar W., and Nisar W., "Novel Approaches for Scheduling Task Graphs in Heterogeneous Distributed Computing Environment," *The International Arab Journal of Information Technology*, vol. 12, no. 3, pp. 270-277, 2015.

[11] ZeroMQ website, *available at http://www.zeromq.org*, Last Visited, 2014.

Adeel-ur-Rehman has received his M.S. degree in Computer Sciences from Shaheed Zulfikar Ali Bhutto Institute of Science and Technology (SZABIST), Pakistan. His areas of interest include software development methodologies, parallel and distributed computing, and information security.

Naveed Riaz has received his PhD degree in Computer Engineering from Graz University of Technology, Austria.. His areas of interest include parallel and distributed computing, Digital Image Processing and Theoretical Computer Sciences.