Asynchronous Communication in Java over Infiniband and DECK*

Rodrigo da Rosa Righi, Philippe O. A. Navaux
Universidade Federal do Rio Grande do Sul
{rrrighi, navaux}@inf.ufrgs.br

Márcia Cristina Cera, Marcelo Pasin
Universidade Federal de Santa Maria
{cera, pasin}@inf.ufsm.br

Abstract

Java is becoming an attractive and easy to use programming language. It provides two systems for distributed computing, RMI and sockets, which describe a synchronous communication over TCP/IP. These Java core features may not be the best choice for cluster computing environments, since they do not provide high performance, a critical factor in this scenario. In this paper it is presented the development of Aldeia System, a library proposal to provide asynchronous communication in Java for cluster programming. Aldeia has been deployed on SAN hardware through Infiniband and DECK high-speed substrates. This paper shows the rationale for Aldeia’s creation, its structure and encouraging results in synchronous and asynchronous approaches.

1. Introduction

Clusters have been widely used as an architecture to achieve high performance computing[1]. They are a controlled and dedicated environment with low error rate, where nodes exchange messages across the network in order to solve parallel applications. Thus, a cluster can be assembled as a system area network (SAN)[9, 17] that focuses on low latency, low CPU overhead and high bandwidth communication. To achieve these features, SAN software can use asynchrony concept in message passing[4]. Additionally, SAN hardware can reach a gigabit bandwidth and present low latency in microseconds order.

Beside network infrastructure, another issue in cluster computing is the programming language adopted to develop parallel applications. Concerning this, we can observe the growing of Java usage in clusters[10, 14]. Java provides better and easier mechanisms to write multi-threading applications and also offers a remote method invocation (RMI) and sockets systems for distributed computing. RMI and sockets perform synchronous communication between Java processes over TCP/IP protocol. Commonly, cluster applications use standard TCP/IP protocol for inter-nodes communication, even though TCP/IP imposes software penalties, such as copies and checksums computation. These features can turn TCP/IP out for high performance environments[8].

In this context Aldeia System[6] was developed aiming to overlap embedded Java distributed systems. Aldeia exploits some SAN cluster features, such as high-speed hardware and optimized communication techniques. Its facilities are presented through an intuitive Java programming interface, providing easier manners to write SAN applications. Aldeia’s main purpose is to take profit from asynchronous communication. This idea explores concurrency between computation and communication, increasing application flexibility and efficiency. Besides this, asynchrony concept is a powerful mechanism to hide network latency, especially when used over TCP/IP. Aldeia applications can execute on a SAN cluster and over TCP/IP networks. Nowadays, Aldeia has been used over Infiniband[16] and over the platforms supported by DECK[2] environment.

This paper presents the Aldeia system and is organized as follows. Section 2 describes cluster communication evolution. Section 3 shows some Java communication libraries for cluster programming. Section 4 presents the Aldeia, explaining its creating rationale, its structure and implementation. In section 5 Aldeia’s evaluation is shown. Section 6 presents a conclusion with Aldeia’s main contributions.

2. Cluster Networking Evolution

Generally, the standard cluster communication system is composed by Ethernet (802.*) hardware and uses TCP/IP as network protocol. Despite its wide usage and attractive costs, this combination imposes high communication latency. TCP/IP is a software protocol that spends CPU cycles for its computations and was developed for wide and local area networking bringing up reliable communication[8]. It imposes headline checksums, message copying, flow control and interruption handling to implement its communication facilities. Moreover, there are overheads related with user-kernel levels context switching for each network inter-
action. In a low error rate environments, most of TCP/IP features are not desired since they can bound applications performance. To fill this gap SAN infrastructure is used.

SAN networks present optimized circuits and software techniques aiming to reach high speed communication. They can use lightweight user-level protocols, DMA-based message passing and asynchronous communication[9]. SAN hardware may implement itself some protocol tasks such as checksums, data routing and data segmentation. This idea is important since it does not spend CPU cycles, that now can be used to calculate other application’s operations. Currently, SAN usage is increasing[17], helping to reduce its final price.

We can expose some SAN high performance communication technologies examples, such as SCI, Quadrics, Myrinet and Infiniband. Infiniband Architecture (IBA)[13, 16], especially, is a standardization initiative for SAN networks and defines an infrastructure that makes possible asynchronous communication in a high-speed switched fabric. IBA development process was based on Virtual Interface Architecture (VIA) protocol and uses queue pairs (QP) as an user endpoint interface. The programmer just puts communication request descriptors in a sending or receiving QP queue to describe his asynchronous interactions. Another system that takes profit from SAN features is DECK environment[2]. It operates over high speed interconnections, such as Myrinet, SCI and VIA. Furthermore, DECK supports also standard TCP/IP communication.

3. Exploiting Cluster Programming in Java

Java has emerged as a powerful language for applications development making the task of writing complex systems easier. Although Java oriented object profitability advantage, its performance is an important and much commented issue, especially in cluster computing. Java is interpreted by default, using bytecode representation for its executable files. Its interpreted execution makes its performance worse than compiled languages, like C. In order to improve its performance when running only in an interpreted way some efforts have been done to get a faster Java environment. Examples of these initiatives are: java-to-binary translators and JIT (Just in Time) compilers. This last one has been integrated in newer Java virtual machines for runtime performance enhancements. Maassen et al[11] presented that it is already possible to reach Java application execution performance similar to native machine compiled one.

Java presents two systems for distributed computing which describe different network interaction styles: RMI and sockets. Both perform TCP/IP synchronous communication which can decrease parallel applications efficiency. Intending to profit from asynchronous communication semantic (nonblocking), there are some Java libraries for cluster programming such as ProActive, JavaSymphony and Ibis. ProActive[3] makes high level asynchronous RMI through future objects and wait by necessity capacity. It implements RMI over standard Java RMI (implemented using sockets), Jini and Ibis. JavaSymphony[7] RMI can either be blocking or not and the communication between two processes is built on the top of Java RMI. Ibis[15] offers a message passing interface based on sending and receiving ports abstraction. Its ports can be configured to handle asynchronous calls and Ibis can operate over Java sockets, MPI and Panda library[11] (using UDP or GM/Myrinet).

In a general analysis, these related systems can use Java sockets for TCP/IP message passing. It would be good to eliminate TCP/IP overhead for Java communication within a cluster, at least when a SAN infrastructure is available. Reimplementing Java sockets, definitely, is not a new idea. We can expose libraries such as those that rewrite Java sockets for TCP/IP asynchronous communication. An example of this is the DECK library, that prescribes to use VIA protocol[5] or SCI[14]. But none of them describes an asynchronous communication directly.

4. Aldeia Communication Library Proposal

The first issue in Aldeia’s development is its programming interface. We gave priority to Java RMI and sockets well established interfaces, once they represent the most commonly used systems in Java programs. However, none of them describes a direct asynchronous communication. RMI system is a large package involving naming services, dynamic class loading and network communications. On the other hand, sockets are simple because they deal only with connection and bytes transferring between processes. Beside this, reimplementing Java sockets is a straightforward solution that can improve Java RMI. One possible improvement in Java sockets would be using its interface to offer asynchronous communication directly. However, the existing programs would stop working when sockets native semantics are changed.

Aldeia library was proposed to join asynchronous communication on high performance cluster using a well known Java programming interface. Following this idea, it reimplements Java TCP/IP sockets to offer asynchronous message passing over SAN high speed interconnection. In this context, the main Aldeia focus is over Infiniband infrastructure. The most used Infiniband interface is VAPI[12], which is a sophisticated and asynchronous user-level native communication library. Aldeia presents an adapter to use this low level library. The VAPI asynchrony advantage is just used in the sending operations, which are always buffered to guarantee that the existing sockets applications will be executed with Aldeia correctly.

Proposing Java sockets communication in order to work with asynchronous semantics over Infiniband is the Aldeia’s main idea. But it also incorporates other SAN advantages.
through DECK environment usage. Aldeia over DECK is made through another adapter on VAPI level, intending to reuse its sophisticated API. Using DECK facilities, Aldeia can operate over standard networks (TCP/IP), beyond the set of SAN environments that DECK natively supports.

4.1. Modular Structure

Aldeia reimplements the sockets classes package to work with VAPI library, written in C language (compiled to native code). In this way, JNI (Java Native Interface) system was employed as an interface between Java and the native code. Aldeia was divided in modules which make its development easier. Figure 1 shows Aldeia’s modules organization and the interactions among them.

The user application is located on the top of Aldeia’s structure. Java application can use an available RMI module. The RMI module is not part of Aldeia’s implementation, although it is possible that different sockets-based RMI, such as Ibis, be integrated on it easily. This integration occurs through Aldeia’s fully compatible Java sockets implementation, called Aldeia Sockets (look at Figure 1). In other words, through Java polymorphism Aldeia can provide a basic asynchronous programming interface that can be used transparently not only in RMI systems, but in any distributed system based on standard Java sockets package.

Aldeia Sockets are written using Java and have a set of basic sockets classes, only responsible for handling connection between processes and a reliable communication channel to send and receive bytes. Aldeia Sockets classes have native methods to provide the low level network functionalities. The VAPI Adapter module represents a dynamic native library that has the implementation of each native method located in Aldeia Sockets classes. Aldeia modules above VAPI Adapter are written using Java, while this module and those under it are implemented in C language. VAPI Adapter implements functions for connection and message passing using VAPI interface. Thus, it is possible just linking the VAPI Adapter with the VAPI library implementation directly to establish communication over Infiniband networks.

Aiming to use other SAN platforms, the MicroVAPI library (μVAPI) was integrated in Aldeia’s structure. This library is located between VAPI Adapter and DECK (see Figure 1). It rewrites a VAPI subset functions, needed in VAPI Adapter, and has the same VAPI semantics and syntaxes. MicroVAPI goal is to adapt VAPI calls that have been done by VAPI Adapter to use DECK implementation. Using MicroVAPI, we can pass messages over networks supported by DECK keeping the same Aldeia interface. Nowadays, MicroVAPI operates over DECK, but it can be rewritten to any other communication library, such as UDP, MPI or Elan/Quadrics. Thus, Aldeia offers a simple framework in which new communication substrates to serve an asynchronous sockets-based Java interface can be easily added.

4.2. Application Programming Interface

Aldeia presents four classes: AldeiaServerSocket, AldeiaSocket, AldeiaOutputStream and AldeiaInputStream. The first two classes make processes connection and the others treat message passing. Figure 2 shows Aldeia’s classes interactions. An AldeiaServerSocket object is created to provide server applications. This class extends ServerSocket and sets up the connection with a client. Commonly, the AldeiaServerSocket object is created passing the TCP/IP port on its constructor. Its main method is accept() which returns an endpoint connection object. On the other hand, AldeiaSocket class is used by client applications and acts as an endpoint connection. This class extends Socket and its main method is connect(), that makes connection between two endpoints. Moreover, the connection can be established through AldeiaSocket constructor, passing it the server machine and TCP/IP connection port. Aldeia uses this port to create an auxiliary Java TCP/IP socket in its connection process. This auxiliary socket is used to exchange informations needed by VAPI and DECK endpoint connections and is closed after this initial procedure.

The AldeiaSocket class has two methods for message passing: getOutputStream() and getInputStream(). The first one returns an AldeiaOutputStream object that acts as sending communication channel. This object’s class extends OutputStream, which is used in standard Java sockets. The second method returns a receiving communication channel, represented by AldeiaInputStream class. This class extends InputStream from Java sockets.

Data transfer communication channel changes sending semantic to offer asynchrony concept. AldeiaOutputStream class presents two main methods: write() and flush(). The first one is employed to send bytes following asynchronous, or nonblocking, semantic. Network communications can be made concurrently with
the sender thread operations. Planning full portability between standard Java sockets and Aldeia applications, sending operations must copy data to internal buffer in order to prevent data overlap. The flush() method describes a blocking semantic and waits for all pending sending operations until they are received on the other endpoint completely. After its call, it is assured that all sending requests were done. Also aiming compatibility with sockets, flush() is called automatically when the sending channel is destroyed or when this channel is closed (using close() method).

Data receiving follows the same synchronous idea used in Java sockets. AldeiaInputStream publishes two main methods: read() and available(). The first method is used for synchronous data receiving. This kind of communication is interesting to preserve existing sockets applications execution, once the receiver can need the incoming value in a subsequently operation. The available() method returns the amount of data already available locally on the receiving channel without network interaction. This situation happens because Java sockets can work with any data size at each network interaction (stream concept), where the sender can transfer more data than the receiver is waiting.

4.3. Implementation

Intending to adapt the stream concept with message passing libraries efficiently, Aldeia uses MTU (Maximum Transfer Unit) framing and a registered buffer to describe its communication model. Memory registering is necessary to enable direct VAPI DMA-based communications between user application and the network. Using MTU technique, a communication request can need several network interactions, depending on the message size. The receiver process does not read more than one MTU at each network interaction, eliminating memory overflow problems. Beside this guarantee, other factors that stimulated this Aldeia communication approach (MTU + buffer) are the memory registering operations economy, made only in communication channels creation, and because Infiniband also uses MTU. In addition, Aldeia uses send-receive reliable connection (RC) transport service[16] for Infiniband message passing.

After defined the communication technique, data segmentation writing was analyzed, if it would either be located in VAPI Adapter (native code) or in Aldeia Sockets (Java code). The native code is more efficient than Java, but each JNI call can be an onerous task. Looking for high performance communication, three message passing prototypes were developed. They analyze data segmentation locale, the amount and the parameters in each JNI call. The sending operation of these prototypes is described below.

- Prototype 1 - Data segmentation control is performed in Java. Communication channel calls a native method to pass each data block (minor or equal to a MTU), carrying it as a method argument. Data block is converted and copied to registered sending buffer in native code;
- Prototype 2 - Data segmentation follows in Java space. Here it is employed a java.nio.ByteBuffer object matched to the native buffer. This object can write in this buffer directly. Neither the transferring of data block as a native method parameter, nor its copy to buffer in native code is required;
- Prototype 3 - Data segmentation is carried over VAPI Adapter. Sending methods have only one line, where a native one is called, passing it full memory region as parameter. In native code data conversion and division are made, beside its copy to registered sending buffer.

All prototypes were implemented and tested with Java primitive data types message passing and through Java serialization classes. Beside this, different message sizes, as well as MTU values, were analyzed. As a result evaluation, prototype 1 got the worst performance in all the cases, because it imposes Java segmentation, data copies in Java and data block passage through native method. Another bad performance issue is the several times that a native method must be called to pass messages bigger than one MTU.

Prototypes 2 and 3 present a similar performance evaluation. Prototype 2 neither imposes data passing in native method nor its copy to registered buffer. However, it makes Java data segmentation using ByteBuffer object, which is implemented handling Java exceptions and controlling native memory access in each method. On the other hand, prototype 3 passes all data in a native method, but data segmentation is optimized and faster, because arithmetic pointer technique can be used to treat contiguous data blocks efficiently. Beside this, this prototype can reach an advantage when messages larger than one MTU size are involved. Based on this analysis, Aldeia adopts the mechanism used in prototype 3.

Aldeia uses two data structures in asynchronous sending channel: a circular registered buffer and a circular list of...
communication descriptors. Circular buffer is allocated and registered on the sending channel creation with large size (at present 10 MBytes, the greatest registered buffer supported by VAPI). Each sending requisition is segmented in MTU-based subdivisions, where each one is copied sequentially in sending buffer and assigned to a specific descriptor. As an increasing amount of new descriptors for asynchronous communication is being produced, the oldest ones are concurrently consumed for VAPI communication mechanism. Aldeia data transmission channel only blocks when there are not free reused descriptors or when an available buffer memory is not enough. This occurs because a scenario where asynchronous transmission faster than communication processing can exist. Both sending channel structures are shown in Figure 3. In this figure MTU of 1024 bytes is used and we can analyze sending requisitions segmentation when the message is bigger than one MTU.

The data receiving channel implementation is easier than the sending one. It uses only one MTU-sized registered buffer and a single descriptor. It employs synchronous semantics that make possible to reuse this descriptor which has a field that is always completed with MTU value for network interactions. After each communication is performed, this field is filled with the real size sent by the other endpoint. Following this idea, data receiving channel can read more data (bounded by one MTU) than requested in Java and then it must store these bytes for next requisitions.

5. Aldeia System Experimental Evaluation

Aldeia experimental testbed was the LabTeC cluster which is composed by 20 Pentium III 1.1 GHz nodes. Each cluster node has 1 GByte main memory and Intel Pro Gigabit Ethernet network adapter. The equipment that connects them is a 3Com Fast Ethernet switch. Beside standard network, LabTeC also has 4 Infiniband nodes, with 2 Gbps Mellanox network cards linked through 10 Gbps switch. Each node uses Linux Kernel 2.4.26, DECK 2.3.1 and VAPI 0.98. In our tests, only TCP/IP DECK version was adopted, because it is the more stable and tested version. We used Blackdown JVM 1.4.2 and all the executions use its JIT runtime compiler.

5.1. Bandwidth and Communication Time

A simple micro-benchmark, called synchronous Ping-Pong, was implemented to measure Aldeia’s communication channel performance directly. Server and client network interaction can be seen in Figure 4. After several Ping-Pong interactions (i) between the endpoints, each partial result time represent the time spent to transfer one d sized message. Each d sized message final time is obtained through arithmetic average of partial times. In this application there are two features that can vary: message size and MTU. The first one can adopt 20 values between 0 and 10000 bytes, with an incremental factor equal to 500. This upper bound value is the JVM limit that Java sockets can operate without exception launching. MTU size can vary in the following values: 512, 1024 and 2048 bytes. These are the set values supported by VAPI. Aldeia’s MTU size is automatically passed to VAPI in the connection procedure.

The graph in the Figure 5 shows Aldeia’s communication time using DECK/TCP. The first metric that can be achieved is the latency, estimated in 64 μs. We may verify that all different MTU scenarios reached similar communication times. For instance, Aldeia configuration using MTU equal to 2048 bytes achieves 1080.3 μs when it works with 10000 bytes, while other 512 and 1024 MTU versions get 1095.7 and 1087.4 μs respectively for the same message size. About Aldeia over DECK/TCP bandwidth, showed in graph within Figure 6, we can analyze that 2048 MTU version achieved 70.5 Mbps when carrying 10000 bytes sized messages. This behavior without large disparities between MTU sizes may be understood since TCP/IP performs data buffering and acts with a retransmission timer[8].

Aldeia over Infiniband/VAPI communication time graph is presented in Figure 7. In this graph, it can be clearly viewed that Aldeia application presents different behaviors depending on the MTU size that it uses. However, all versions reached similar latencies, that is measured in 29.7 μs when has been used 512 bytes in MTU parameter. We can also observe that up to 500 bytes passage practically all MTU versions obtained the same performance. Using this message size, all MTU versions use one descriptor and perform the same amount of network interactions. With message size upper than 500 bytes, the graph’s lines take different ways. The fastest version uses MTU equal to 2048 and
reached 198.2 μs when 10000 bytes are transferred.

We can observe a stair behavior in VAPI time communication graph (see Figure 7). After each MTU multiple, the communication time goes up because another descriptor and message passing interaction are needed. For example, 85.6 μs transmission time is required when is used MTU of 2048 bytes and message size equal to 4000 bytes. In this transmission are used two communication descriptors. However, when 4500 bytes are sent three descriptors are used and 116.7 μs are observed. After this heavy acceleration time, communications use the same descriptor until the next one is required. Using the same descriptor, stair’s platform is slightly growing, because the message size is also increasing, requiring larger copies to the registered buffer.

Aldeia over Infiniband bandwidth can be seen in Figure 8. The version with MTU of 2048 bytes achieved 384.1 Mbps when working with 10000 bytes. At each MTU multiple it can also be observed that the bandwidth decreases. This occurs because another descriptor with few data is used. Aldeia over VAPI different behaviors happened because as MTU decreases, more network interactions are required. Beside this, VAPI does not perform memory copies and performs data transferring exactly with requested size.

It was also tested Ping-Pong application with standard Java sockets. This evaluation showed that Java sockets achieved 1085 μs to send 10000 bytes. Using the same message size, Aldeia reached 1092 μs in transferring operation. We consider this a good result, because MicroVAPI performs threads and mutual exclusion management, moreover DECK/TCP overhead. Note that Ping-Pong organization (Figure 4) does not profit from asynchrony.

It was also implemented Ping-Pong using VAPI (full native code). In this evaluation, VAPI reached only 512.3 Mbps when 10000 bytes sized messages are sent. This
bandwidth was achieved because VAPI imposes high overhead in its polling for completion communication functions (one to test and another to wait for completed descriptors). Both functions make a very slow system call in order to get network results. This issue was also observed in Liu et al[9], where VAPI tests required large overhead in completion notification functions. Aldeia over VAPI reached 75 percent of 512.3 Mbps, since when it sends 10000 bytes with MTU equal to 2048 bytes, 5 descriptors and message copying are needed. Verifying this result, we consider Aldeia communication mechanism performance satisfactory.

Observing different Java systems for SAN environments, we analyze other two systems indirectly. Taboada et al[14] presented that their SCI sockets reimplementation achieved 9 μs for latency. They use an optimized SCILib library for message passing, but their implementation does not provide asynchronous communication. In a higher level, Ibis[15] presents RMI over GM/Myrinet substrate. Ibis RMI in this scenario reached 42.2 μs for latency in a null RMI call.

5.2. Asynchronous Communication

Aiming to demonstrate Aldeia’s asynchronous communication benefits, another specific micro-benchmark was built (see algorithm in Figure 9). This application describes an unidirectional communication and its main feature is getting two times: synchronous and asynchronous. The first one is reached after $\text{flush()}$ method and the other one before this call. Thus, we can measure the time that the sender can gain with asynchronous communication (not involving networking time). Synchronous time represents the instant after the receiver received all $i$ messages completely. In this application, the final times are the average of several executions. Beside this, only MTU of 2048 bytes is employed, because this is the best choice for DECK/TCP and VAPI.

$$\text{client:} \begin{array}{l}
\text{Get initial time} \\
\text{repeat } i \text{ times:} \\
\text{write}(d) \\
\text{Get end asynchronous time} \\
\text{flush()} \\
\text{Get end synchronous time}
\end{array} \text{ server:} \begin{array}{l}
\text{repeat } i \text{ times:} \\
\text{read}(d)
\end{array}$$

Figure 9. Asynchronous simple application

Figure 10 shows results of Aldeia over DECK/TCP. We can observe asynchronous communication advantage, hiding TCP/IP overhead latency. The sender spent only 91.2 μs launching asynchronous requisitions when 10000 bytes were transferred. With the same message size, the sender must wait for nearly 1000 μs until the receiver processes all the network interactions in synchronous communication style. Aldeia over VAPI asynchronous behavior is shown in Figure 11. Using asynchronous approach, the sender required 79 μs to launch a message with 10000 bytes. It must wait for 212.3 μs to complete communication synchronously. In this figure we can see that the difference between synchronous and asynchronous times in Infiniband test is lower than DECK/TCP one, because both VAPI and Infiniband hardware are faster than TCP/IP and Ethernet.

Aldeia asynchronous communication model can be useful in applications that perform sending operations followed by a local processing phase. In a master-slave example, the master could send messages to its slaves asynchronously and then execute some local operations immediately, improving application performance. Verifying the result graphs in this subsection, we can confirm Aldeia asynchronous advantage in TCP/IP overhead hiding. Moreover, the proposed system can overlap VAPI conclusion notification functions overhead with sender thread operations when Aldeia uses asynchronous message passing.
6. Conclusion

The overhead from synchronous communication is a limiting factor in Java usage for high performance computing. Concerning this, we developed an asynchronous library proposal, called Aldeia. Aldeia is a communication system for cluster programming that profits from asynchrony concept and uses Java sockets-based interface to offer its facilities to the user, giving him an already known interface. Sockets applications can use Aldeia’s asynchronism without needing to be rewritten, once Aldeia performs data buffering in its network interactions. Beside this, Aldeia can use SAN infrastructure, eliminating TCP/IP overhead in data traffic in these cases. The proposed system offers an alternative for Java programming joining asynchronous and faster communication in a well established interface.

Although some Java sockets reimplementations have been done, for example, to operate over VIA[5] or SCI[14], none of them is asynchronous. Our differential approach is an asynchronous-based communication model and the possibility to present user level Infiniband communication using Java classes. Aldeia makes possible Infiniband message passing through sophisticated VAPI communication library usage. Aldeia also has a simple framework to join other communication systems, presenting them in a Java interface. It incorporates a MicroVAPI library to its structure which has basic functions for asynchronous message passing and nowadays was written using DECK. Aldeia also hides VAPI and DECK low level complexities and can be useful for performance improvements in any sockets-based systems.

Aldeia over Infiniband/VAPI achieved 29.7 μs for latency and 384.1 Mbps for bandwidth. We concluded that this bandwidth performance is reached because VAPI completion notification functions perform onerous systems calls to get network results. This observation was also found in Liu et al[9]. Aldeia over DECK/TCP presented a latency of 64 μs and bandwidth of 70.5 Mbps. We can see in asynchronous graphs that Aldeia can offer concurrency between computation and communication. In Aldeia’s asynchronous communication model, sending calls do not block and networking time is not involved. The programmer can organize his application to execute sending operations followed by processing phases, aiming to use full Aldeia capabilities.

In the next experiments we will analyze Aldeia over Myrinet through DECK usage. In future work we could test Ibis performance over Aldeia. In this context, we will have RMI over Infiniband. Finally, another future work is to analyze Aldeia asynchrony profitability in real applications.

References