MPI over uDAPL: Can High Performance and Portability Exist Across Architectures?

LEI CHAI, RANJIT NORONHA AND D. K. PANDA

Technical Report
OSU-CISRC-1/06-TR11

MPI over uDAPL: Can High Performance and Portability Exist Across Architectures? [*]

Lei Chai   Ranjit Noronha   Dhabaleswar K. Panda

Department of Computer Science and Engineering
The Ohio State University
{chail, noronha, panda}@cse.ohio-state.edu

Abstract

Looking at the TOP 500 list of supercomputers we can see that different architectures and networking technologies appear on the scene from time to time. The networking technologies are also changing along with the advances of processor technologies. While the hardware has been constantly changing, parallel applications written in different paradigms have remained largely unchanged. With MPI being the most popular parallel computing standard, it is crucial to have an MPI implementation portable across different networks and architectures. It is also desirable to have such an MPI deliver high performance. In this paper we take on this challenge. We have designed an MPI with both portability and portable high performance using the emerging uDAPL interface. We present the design alternatives and a comprehensive performance evaluation of this new design. The results show that this design can improved the startup time and communication performance by 30% compared with our previous work. It also delivers the same good performance as MPI implemented over native APIs of the underlying interconnect. We also present a multi-stream MPI design which aims to achieve high bandwidth across networks and operating systems. The design also has the potential to be extended easily for fault tolerance purpose. Experimental results on Solaris show that the multi-stream design can improve bandwidth over InfiniBand by 30%, and improve the application performance by up to 11%.

[*]This research is supported in part by Department of Energy's Grant #DE-FC02-01ER25506; National Science Foundation's Grants #CCR-0204429 and #CCR-0311542; grants from Mellanox, Intel, Sun MicroSystems, Cisco, and Linux Networx; and equipment donations from Apple, AMD, Intel, IBM, Microway, Mellanox, PathScale, SilverStorm, and Sun MicroSystems.

1. Introduction

The TOP 500 list [6] of supercomputers has been measuring the capacity of computers to process large scale applications for more than a decade. This list serves as a gauge of the rapid technological changes taking place in this area. Different architectures have made their appearance on the scene from time to time, with clusters becoming the dominant technology. Additionally, different networking technologies have been used for clusters. These networking technologies change over time as the ratio of processing to network speed changes. While the underlying hardware changes, the software packages and interfaces used by different applications has remained largely unchanged, with the message passing interface (MPI) [17] paradigm being the most popular. With the rapid advance in different technologies, having an MPI that can deliver high-performance over all these different architectures, interconnects and operating systems etc. has become crucial.

Native implementations of MPI over different high-performance interconnects currently exists. These include implementations like MPICH-MX for Myrinet [11], MVAPICH for InfiniBand [4], and MPI/Elan for Quadrics [22]. Though all these implementations provide the same interface to the applications, internally they might differ in their designs. They might also differ in the range of applications they may have been tested for. Thus deploying applications over a new network may expose latent problems in the MPI library or application that might not have been seen before. In addition, supporting all possible configurations might burden the application developer. If the application vendor can test his application with a single MPI library which can be moved across different networks and architectures with high-performance, this might potentially reduce the burden.

The user-level direct access transport APIs (uDAPL) defined by the DAT Collaborative attempts to provide a network, architecture and operating system independent interface to the application for communication. This potentially allows applications to seamlessly use different networks as the underlying transport with minimal effort. While new applications may take advantage of the uDAPL interface, a large number of legacy applications exist which are written in MPI. The effort required to port these applications to uDAPL may be significant, given their size and complexity. In some cases, it may be impractical, especially if the expertise needed to rewrite these applications is not available. The easiest way to achieve this with minimal effort is to develop a port of MPI over uDAPL.

While it is possible to design an MPI that uses the uDAPL interface, an important consideration is whether this design can deliver high-performance close to that of the underlying networking system or not. Initial building of MVAPICH2 over uDAPL [12] suggests that it is possible to achieve latency and bandwidth close to that of the native implementation. Due to limitations in the design of MVAPICH2, we choose to revisit the issue of whether the widely deployed MVAPICH can be redesigned to use the uDAPL interface.

In addition to portability, we also choose to explore the issue of *portable high-performance*. This issue comes into play because the uDAPL library may have different performance characteristics on various architectures. This is because the uDAPL library may be designed in a variety of ways. In addition, the uDAPL library depends on the performance of the underlying communication library. Additionally, there are operating system considerations such as memory registration, scheduling and memory allocation strategies which might affect performance. Thus, a design optimized for one architecture may not be able to achieve optimal performance on a different architecture.

In this paper, we take on the challenge of designing a high-performance MPI over uDAPL. The rest of the paper is organized as the following: In Section 2, we introduce the background for this work. We explore several design alternatives for the basic design in Section 3. Following that, a multi-stream MPI design is presented in section 4 for achieving high performance across different architectures. We evaluate our design on Linux and Solaris in section 5 and 6 respectively, and describe the related work in section 7. Finally, in section 8, we provide our conclusions and future work.

2. Background

In this section we introduce the background knowledge of uDAPL, MVAPICH and MVAPICH2, and popular networks and operating systems in the area of high-performance computing (HPC).

The emerging uDAPL (User Direct Access Programming Library) standard [14] defines a set of transport-independent, platform-independent Application Programming Interfaces that exploit the RDMA (remote direct memory access) capabilities of next-generation interconnect technologies such as InfiniBand, the Virtual Interface Architecture and iWARP. uDAPL is defined by DAT Collaborative [1].

MVAPICH [4] is a high-performance MPI-1 implementation over InfiniBand. It is an implementation of MPICH [17] ADI2 layer. MVAPICH is implemented on top of Verbs Level Interface (VAPI), developed by Mellanox Technologies. It uses eager protocol for small messages and rendezvous protocol for large messages as shown in Figure 1. The detailed design is discussed in [20]. MVAPICH2 [21] is an MPI-2 implementation over InfiniBand, which is also on top of VAPI. It was implemented based on MPICH2 [8] RDMA channel. Starting from version 0.9.0 the design has been moved to MPICH2 ADI3 layer. MVAPICH and MVAPICH2 are currently being used by more than 280 organizations worldwide (in 30 countries).

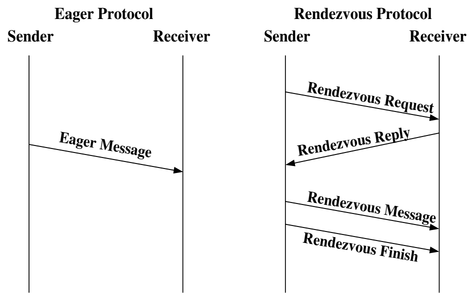


Figure 1. MVAPICH Eager and Rendezvous Protocols

Modern interconnect technologies in the high-performance computing area support RDMA operations as well as send/receive operations, and allow MPI programs to deliver low latency and high bandwidth. These networks include InfiniBand [3], Myrinet [11], Quadrics [22], Ammasso Gigabit Ethernet [7], etc. While similar in semantics, these networks have syntactically very different native APIs from each other, but they all support uDAPL interface. As high performance computing advances rapidly, more and more operating systems start to support these high speed networks. Linux has been supporting various networks for a long time. Solaris is also active in this area. It starts to support InfiniBand since Solaris 10 and the transport layer is IBTL [24]. Recently MS Window also joined the HPC arena. In this paper we focus on Linux and Solaris.

3. Design Issues

In this section we address various issues in designing MPI over uDAPL. We present a single-stream design in this section and extend it to multi-stream design in the next section.

3.1. Overall Design

Our design is adapted from MVAPICH. As can be seen from Figure 2, MVAPICH has four major components: connection management, communication channels, progress engine and memory management. On the other side, uDAPL provides various services. To design a high-performance MPI over uDAPL, we need to map the MVAPICH components onto uDAPL services in an efficient manner.

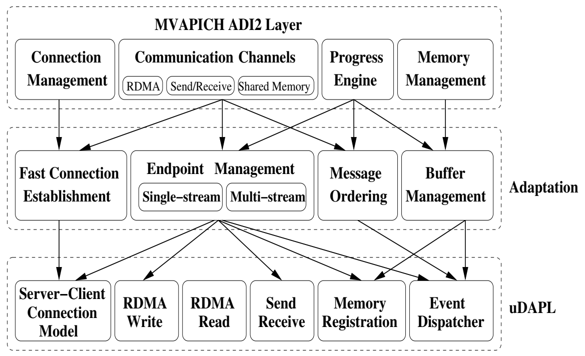


Figure 2. Overall Design for MVAPICH-uDAPL

This work is more challenging compared with our previous work of designing an MPI-2 over uDAPL through MPICH RDMA channel [12]. First of all, MVAPICH utilizes multiple communication channels (RDMA, send/receive, and shared memory) which makes the design more complicated, whereas in our previous work only RDMA channel is used. Secondly, we would like to design a more efficient connection management component because connection establishment time becomes critical as clusters scale to really large sizes. In addition, our goal is to design a high-performance and portable MPI not only for Linux but also for other operating systems such as Solaris, thus optimization for other operating systems is also a big challenge.

We discuss a single-stream based design in this section in detail, and move to a multi-stream based design in the next section.

3.2. Efficient Connection Establishment

MPI assumes a fully connected topology and uDAPL currently only supports Reliable Connection (RC) service, which implies that every process needs to establish a connection with every other process at the initialization phase. We use the server-client model provided by uDAPL for connection establishment.

We proposed a thread based approach in our previous work [12]. In this approach, every process acts as a server for processes who have higher ranks and acts as a client for processes who have lower ranks. Every process first creates a server thread. The server thread persistently listens on the *Public Service Point (PSP)*, which is a listen handle used for detecting incoming connection requests. Then every process issues connection requests to all its corresponding servers. A connection is established between two processes as soon as the server thread accepts a request. The server thread exits once all the connections are established. In this way connections can be established concurrently among all the processes.

As we did more research on connection management, we found several disadvantages of the thread based approach. First, thread creation and switching overhead makes this approach less efficient than we would expect. Second, thread creation and execution functions may not be portable across different operating systems and C libraries. In this context, we propose a threadless connection establishment approach, which maintains the concurrency but makes it more efficient and portable.

Figure 3 shows the basic idea of the approach. Suppose we have *n* processes, then we need to establish *n*(n-1)/2* connections. We can do so in only *(n-1)* steps. In step 1, every process issues a connection request to its left neighbor, then listens on *PSP* to accept the request from its right neighbor. In step 2, every process issues a connection request to the process with *rank = myrank-2*, and listens on *PSP* to accept the request from the process with *rank = myrank+2*. So on and so forth, at the end of the *(n-1)th* step, all the connections will be established. Experimental results show that this simple approach works quite efficiently.

3.3. Multi-channel Communication

After initialization, a connection has been established between every two processes. In the single-stream design only one connection is established for a pair of processes. But to achieve high communication performance, we can use multiple channels for communication. Here we define a channel as a type of communication operation. We use RDMA and send/receive channels for inter-node communication and shared memory channel for intra-node communication as in MVAPICH. To use uDAPL RDMA and send/receive operations (or *Data Transfer Operations (DTOs)*) we post descriptors, which are the encapsulation of communication information, to *End Points (EPs)*, which are the abstraction of the local part of a connection. Once a DTO completes, it generates a completion event to the *Event Dispatcher (EVD)*.

uDAPL requires the memory to be registered with the *Interface Adapter (IA)* before it can be used for communication. To remove the memory registration time from the critical communication path, we use a set of pre-registered RDMA buffers for small messages. In the case when the RDMA buffers are consumed, small messages will go to the

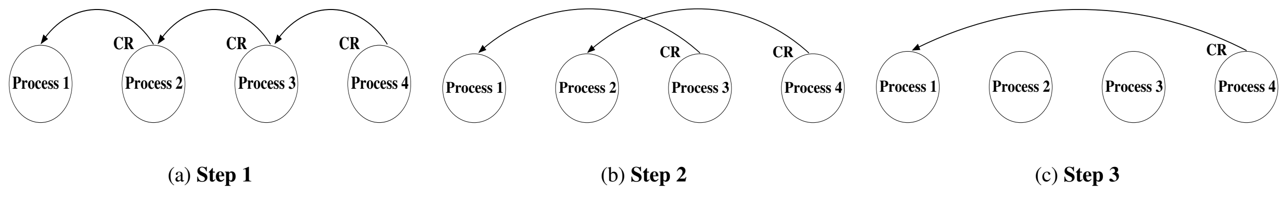


Figure 3. Threadless Connection Establishment Scheme. CR: Connection Request

send/receive channel. Large messages are also sent through the RDMA channel as long as there is sufficient amount of memory that can be registered. Buffers for large messages are registered on the fly.

Processes poll the shared memory buffers, RDMA buffers, and the EVDs to discover incoming messages. Message ordering within one channel is ensured by the uDAPL specification, but there may be out-of-order messages among different channels. Assigning every message a *sequence number* solves this problem. Out-of-order messages are left in the channel without further processing.

4. Multi-Stream MPI Design

The design described in section 3 delivers high performance for various networks on Linux. However, when we moved on to InfiniBand on Solaris, we found out that the MPI-level bandwidth was not as high as we expected. Further study reveals that it is because Solaris does not allow one pair of *EPs* to take up all the bandwidth due to some QoS concern. To the best of our knowledge there is no easy way to overcome this limit. In order to get the desired bandwidth for MPI applications, we present a multi-stream design for Solaris in this section. This framework can also easily be extended to serve other purposes later such as fault tolerance.

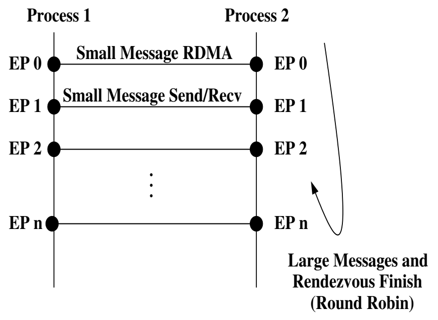


Figure 4. Multi-stream MPI Design

The basic design idea is to use multiple pairs of *EPs* between every two processes to achieve high aggregate bandwidth as shown in Figure 4. It is to be noted that multi-stream is different from multi-channel described in section 3.3 in the sense that multi-channel refers to different types of operations while multi-stream refers to different connections. Different channels can utilize the same connection and the same channel can utilize different connections. There are various issues associated with this design that we discuss in detail below.

4.1. Memory Management

As we mentioned in section 3.3, we use a set of pre-registered buffers for small messages going through RDMA channel. Now we have multiple *EPs*, one important question is: should all the *EPs* share the same set of RDMA buffers or should they have separate buffers? We choose to use the shared buffers for three reasons. Firstly, using shared buffers saves memory space. Otherwise, the memory usage grows quickly as the number of nodes and number of *EPs* increase. Secondly, it reduces polling overhead. To discover incoming messages, the process needs to poll RDMA buffers from time to time. If we have multiple sets of RDMA buffers, the process needs to poll multiple buffers for each connection, which adds unnecessary overheads. And thirdly, it simplifies the design for flow control. If we have multiple sets of RDMA buffers we need to manage flow control for every set which also adds overhead.

4.2. End Point Selection Policy

One critical issue in our design is how to decide which message goes to which *End Point*. This policy can affect the overall performance. In this section we discuss how we deal with small and large messages respectively. The scheme is illustrated in Figure 4.

In the current design, small messages are sent eagerly through the RDMA channel. Once the RDMA buffers are consumed the messages go to the send/receive channel. These two channels are quite different in terms of *EP* selection. For send/receive, if a process posts a *send* to some *EP*, the receive must have a *receive* already posted on the corresponding *EP*. Otherwise the *send* fails. This requires that the receiver knows in advance which *EP* the sender is going to use. This information is not available with the current scheme. One alternative is to let the receiver post *receives* on every *EP*, but this adds memory and time overhead. So we choose to use a fixed *EP* for the send/receive channel. For RDMA, however, there is no corresponding operation at the remote side, so the sender can choose any *EP* to use. But experimental results show that using multiple *EPs* hurts small message bandwidth, so we also use a fixed *EP* for small message that goes through RDMA channel.

Large messages are sent using rendezvous protocol. We use a round robin policy to select *EPs* for large messages. In this way nonblocking MPI calls will be able to saturate the network bandwidth. However, there are three control messages associated with each rendezvous message that we must take care of: *rendezvous request, rendezvous reply*, and *rendezvous finish*, as shown in Figure 1. The first two can be dealt as normal small messages using the policy specified above. We need to pay special attention to *rendezvous finish*, because the receiving of *rendezvous finish* indicates the corresponding rendezvous send has finished, therefore, we must make sure the *rendezvous finish* is not received before the rendezvous message has completely arrived. We make sure of this by letting the sender select the *EP* before it sends the *rendezvous request*, and use the selected *EP* for both the actual message and the *rendezvous finish*. The order of operations posted on the same *EP* is ensured by the uDAPL specification. The sender also informs the receiver which *EP* it is going to use through *rendezvous request*, and the receiver can post *receive* on the corresponding *EP* for *rendezvous finish*. We force all the *rendezvous finish* messages to go through the send/receive channel, because otherwise the receiver doesn't know whether it should post *receive* or not since it has no means to know which channel (RDMA or send/receive) the *rendezvous finish* message is coming from.

4.3. Ensuring Message Ordering

As described in section 3.3, the receiver can handle the ordering of the messages coming from different channels based on *sequence number*, and within each channel the ordering is guaranteed by the uDAPL specification when only one *EP* is used. However, in the case of multiple *EPs* the above mechanism is not sufficient, because the *rendezvous finish* messages can go to any one of the *EPs* which leads to out-of-order messages within the send/receive channel. To deal with this problem we still use the *sequence number* based approach, and put the out-of-order completion events into a *"wait for process"* queue. Whenever the MPI calls need to make progress, we need to check the *"wait for process"* queue in addition to polling shared memory buffers, RDMA buffers, and the event dispatcher to process appropriate messages.

5. Performance Evaluation on Linux

In this section we evaluate our single-stream design of MPI over uDAPL on Linux. We first evaluate the performance of MVAPICH-uDAPL over InfiniBand. The evaluation includes the threadless connection establishment scheme and the communication performance of MVAPICH-uDAPL in terms of both micro-benchmarks and applications. We compared the performance of MVAPICH-uDAPL with that of MVAPICH2-uDAPL proposed in our previous work, and also with MVAPICH-VAPI. We also present the performance of MVAPICH-uDAPL over Ammasso Gigabit Ethernet.

Experimental setup for InfiniBand: We use an 8-node EM64T cluster. Each node has dual 3.4 GHz Intel Xeon processors and 2GB main memory. The nodes are equipped with MT25208 HCAs with PCI Express interfaces. An InfiniScale MTS2400 switch is used to connect all the nodes. The uDAPL library is from Mellanox IBGD version 1.8.0.

Experimental setup for Ammasso Gigabit Ethernet: We use an 8-node IA32 cluster. Each node has dual Intel Xeon 3.0 GHz processors and PCI-X 133 MHz bus. The uDAPL library is dapl-1.2 from Ammasso.

5.1. Threadless Connection Establishment

In this experiment we measured the total time spent on connection establishment in MPI_Init. We add a barrier before and after the connection establishment procedure so that when the result is printed out we know all the connections have been established. From Figure 5 we can see that the threadless connection establishment scheme proposed in this paper performs 30% better than the thread based approach. This is because the overhead associated with thread creation and switching is eliminated.

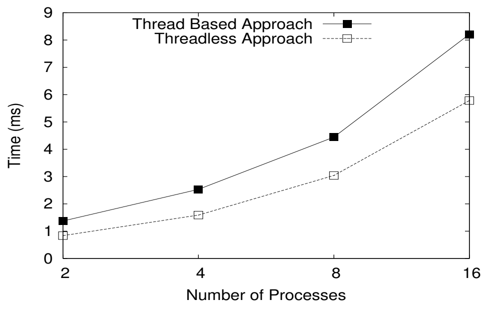
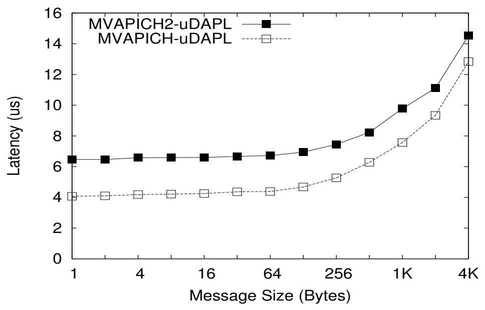


Figure 5. Time Spent on Connection Establishment

5.2. Latency and Bandwidth over InfiniBand

We measured the basic latency and bandwidth of MVAPICH-uDAPL and compared it with the performance of our previous work. Figure 6 shows the results. The small message latency of MVAPICH-uDAPL is 4.07 microseconds, which is 33% better than our previous work. Bandwidth is improved by up to 25%, and the peak bandwidth is 962 MB/s (MillionBytes/sec).



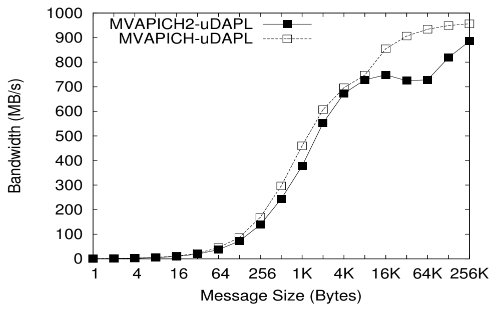


Figure 6. Latency and Bandwidth over InifiniBand on Linux

5.3. Application Performance over InfiniBand

We also conducted experiments using some of the NAS parallel benchmarks [10] and some of the benchmarks in Fluent [16]. We compared the performance of MVAPICH-uDAPL with MVAPICH-VAPI which is known to be one of the best MPI implementations over InfiniBand. From Figures 7 and 8 we can see that MVAPICH-uDAPL and MVAPICH-VAPI perform comparably, which means MVAPICH-uDAPL is able to deliver high performance for real applications.

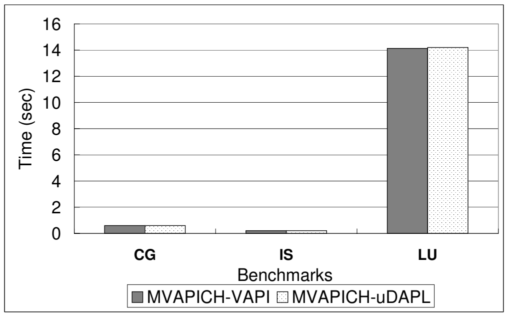


Figure 7. CG, IS, and LU on 16 Processes (Class A)

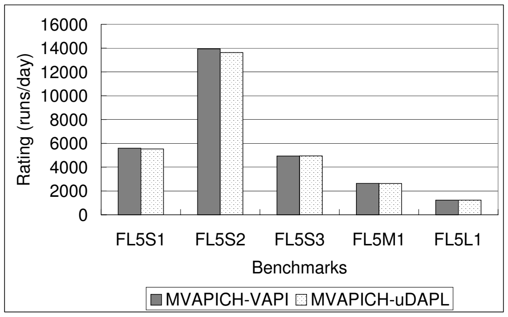


Figure 8. Fluent Benchmarks on 16 Processes

5.4. Performance over Ammasso Gigabit Ethernet

In this section we evaluate the latency of MVAPICH-uDAPL over Ammasso, which is an RDMA-enabled Gigabit Ethernet. We compared with MPICH-iWARP which is an MPI implemented over Ammasso RDMA transport semantics. From Figure 9 we can see that MVAPICH-uDAPL and MPICH-iWARP perform almost exactly the same. For small messages MVAPICH-uDAPL performs even slightly better due to the efficient design. This experiment demonstrates the *portable high performance* of our design.

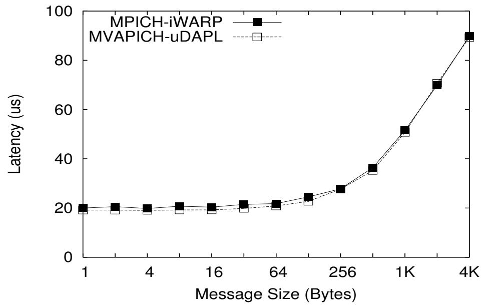


Figure 9. Latency Comparison over Ammasso Gigabit Ethernet

6. Performance Evaluation over InfiniBand on Solaris

In this section we present the performance evaluation of our multi-stream MPI for Solaris and compare with the single-stream design. We first present the micro-benchmark results, followed by application results.

Experimental setup: We use an 8-node Opteron cluster. Each node has dual 2.2 GHz processors and 2GB main memory. They are equipped with MT23108 HCAs with PCI-X 133MHz interfaces. A SilverStorm 3032 switch is used to connect all the nodes. The operating system used is Solaris 10, which has uDAPL over IBTL with the distribution.

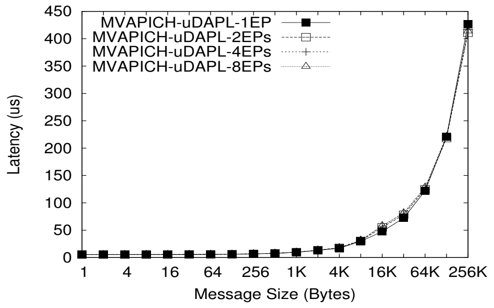


Figure 10. Latency over InfiniBand on Solaris

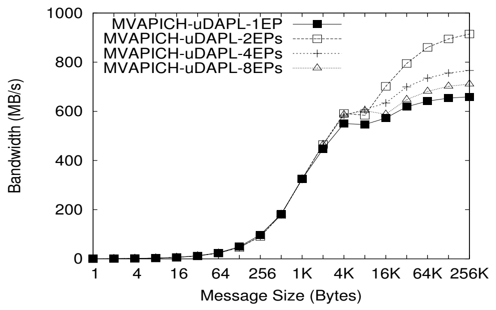


Figure 11. Bandwidth over InfiniBand on Solaris

6.1. Latency and Bandwidth

We first evaluate the ping-pong latency and bandwidth on Solaris. Figure 10 shows that the small message latency of MVAPICH-uDAPL on Solaris is 5.37 micro-seconds, and using multiple *EPs* doesn't hurt the latency. Figure 11 shows that the multi-stream design can dramatically improve the bandwidth. We find that two *EPs* gives the best performance. It improves the peak bandwidth from 659 MB/s to 914 MB/s compared with the single-stream design. We also find that if we use more than two *EPs* the bandwidth performance decreases as the number of *EPs* increases. This is because as we use more and more *EPs*, the MPI takes more and more resources and leaves less resources available for the MPI program.

6.2. Application Performance

From section 6.1 we see that the multi-stream MPI can largely improve MPI-level bandwidth for large messages compared with the single-stream design. To get a comprehensive understanding of how the improvement translates to MPI application performance, we conducted experiments using application level benchmarks. CG and IS are selected from the NAS parallel benchmarks, because these two benchmarks mainly use large messages [15]. We also use a benchmark called PSTSWM [5] which is a parallel spectral transform shallow water modeling program and is bandwidth sensitive. We compared the performance of the multi-stream design with the single-stream design. Since from the latency and bandwidth study described in section 6.1 we know that two *EPs* perform the best, in the application level study we use two *EPs* for multi-stream.

The total execution time is shown in Figure 12. For CG and IS, Class A benchmarks are used, and they were running on 2 and 4 processes. We used a small number of processes for CG and IS, because as the number of processes increases the message size decreases, which makes the benchmarks not bandwidth sensitive any more, thus out of our interest. From the results we can see that the multi-stream design can improve the performance of CG by 11% and 8% on 2 and 4 nodes respectively, and improve the performance of IS by 8.6% and 4.6% on 2 and 4 nodes.

We conducted the experiment for PSTSWM on 8 processes. The result shows that the multi-stream design improves the performance by 7%.

To further study where the benefit comes from, we profiled the MPI functions used in CG and the time spent in each function using DTrace, which is a comprehensive dynamic tracing framework for Solaris [2]. Profiling results show that the benefit mainly comes from the reduced time spent in the progress engine. That is because large messages in nonblocking sends can be pushed out more quickly now.

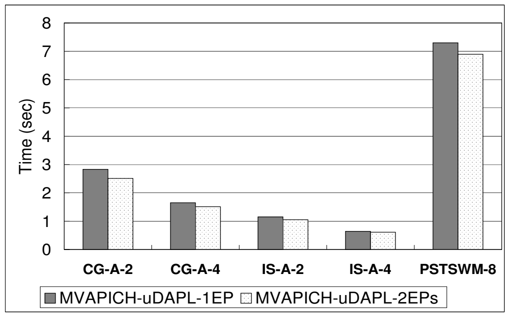


Figure 12. Application Total Execution Time

7. Related Work

There are several MPI implementations in the literature that aim for portability across networks. We proposed an MVAPICH2-uDAPL design based on MPICH RDMA channel in [12]. Scali MPI [23] and Intel MPI library [18] also support uDAPL, but unlike MVAPICH-uDAPL they are not open source software. MPICH-Madeleine [9] is another portable MPI which is based on the MadeleineIII communication library.

In addition to the MPI implementations above, there are also various MPIs that aim for high performance for particular networks, such as MVAPICH [4] and MVAPICH2 [21] (InfiniBand), MPICH-MX (Myrinet), Quadrics

MPI (Quadrics), MPI/GAMMA [13] (Gigabit Ethernet), MPICH-iWARP [7] (iWARP), etc. Most of these implementations are based on MPICH layered structure, and can be ported to different lower-layer programming interfaces. But they cannot be used directly across networks. Further, the designs of these MPIs are highly optimized for the specific networks, and may not deliver portable high performance when ported to other networks.

Using multi-rail networks to overcome bandwidth bottleneck on InfiniBand is discussed in [19] and [25]. The authors discussed various design issues and presented performance benefit. But they mainly focused on InfiniBand and Linux while the design proposed in this paper is more general with respect to networks and operating systems.

8. Conclusions and Future Work

In this paper, we have explored the design alternatives for a high performance implementation of MVAPICH over the uDAPL communication interface. The design goals include *true* portability across networks, architectures and operating systems. In addition to portability, we have also explored mechanisms for *portable high-performance*. These mechanisms include a multi-stream design and a scalable threadless based startup. Evaluation in terms of micro-benchmarks over the uDAPL interface on Solaris shows an improvement in large message bandwidth of up to 30%. This improvement in bandwidth also translates into improved application performance of up to 11%. In addition to an improvement of up to 30% in startup time, it is also more scalable. In addition, evaluation over other networks like Ammasso showed that our implementation is comparable in performance to the native implementation.

We would like to explore how the overall framework can be extended to provide features like network based fault-tolerance. In addition, we would like to explore how technologies like PCI-Express coupled with Single Data Rate (SDR) and Double Data Rate (DDR) InfiniBand will impact the *portable high-performance* design. Understanding how these designs can be portably extended to other networking technologies like Myri 10G might help us understand the limitations and potential enhancements to our design. Finally, we would like to examine how extensions to the uDAPL layer itself such as the addition of support for unreliable datagram can impact the scalability of our MPI design.

Software Distribution: The basic design proposed in this paper has been incorporated into the open source software MVAPICH version 0.9.6. It can be downloaded from [4]

References

[1] DAT Collaborative. http://www.datcollaborative.org.

[2] Dtrace. http://www.sun.com/bigadmin/content/dtrace/.

[3] InfiniBand Trade Association. http://www.infinibandta.com.

[4] MPI over InfiniBand Project. http://nowlab.cis.ohio-state.edu/projects/mpi-iba/.

[5] Parallel Spectral Transform Shallow Water Model. http://www.csm.ornl.gov/chammp/pstswm/.

[6] Top 500 Super Computer Sites. http://www.top500.org/.

[7] Ammasso, Inc. http://www.ammasso.com.

[8] Argonne National Laboratory. MPICH - A Portable Implementation of MPI. http://www-unix.mcs.anl.gov/mpi/mpich.

[9] Olivier Aumage and Guillaume Mercier. MPICH/MADIII: a Cluster of Clusters Enabled MPI Implementation. In *CCGrid*, 2003.

[10] D. H. Bailey and et al. The NAS parallel benchmarks. volume 5, pages 63–73, Fall 1991.

[11] N. J. Boden, D. Cohen, et al. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, pages 29–35, Feb 1995.

[12] L. Chai, R. Noronha, P. Gupta, G. Brown, and D. K. Panda. Designing a Portable MPI-2 over Modern Interconnects Using uDAPL Interface. In *Euro PVM/MPI*, 2005.

[13] G. Ciaccio and G. Chiola. GAMMA and MPI/GAMMA on Gigabit Ethernet. In *7th EuroPVM-MPI*, 2000.

[14] DAT Collaborative. uDAPL: User Direct Access Programming Library Version 1.2. http://www.datcollaborative.org/udapl.html, July 2004.

[15] Ahmad Faraj and Xin Yuan. Communication Characteristics in the NAS Parallel Benchmarks. In *IASTED International Conference on Parallel and Distributed Computing and Systems*, 2002.

[16] Fluent Inc. http://www.fluent.com/.

[17] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI, Message Passing Interface Standard. Technical report, Argonne National Laboratory and Mississippi State University.

[18] Intel Corporation. http://www.intel.com/cd/software/products /asmo-na/eng/cluster/mpi/index.htm.

[19] J. Liu, A. Vishnu, and D. K. Panda. Building Multirail InfiniBand Clusters: MPI Level Designs and Performance Evaluation. In *SuperComputing*, 2004.

[20] J. Liu, J. Wu, and D. K. Panda. High performance RDMA-based MPI implementation over InfiniBand. *Int'l Journal of Parallel Programming*, In Press, 2005.

[21] Network-Based Computing Laboratory. MPI over InfiniBand Project. http://nowlab.cis.ohio-state.edu/projects/mpi-iba/index.html.

[22] Fabrizio Petrini, Wu chun Feng, Adolfy Hoisie, Salvador Coll, and Eitan Frachtenberg. The Quadrics Network: High Performance Clustering Technology. *IEEE Micro*, 22(1):46–57, January-February 2002. Available from http://www.c3.lanl.gov/ fabrizio/papers/ieemicro.pdf.

[23] Scali Inc. http://www.scali.com/.

[24] Sun Microsystems. Solaris 10 Reference Manual Collection, man pages section 7: Device and Network Interfaces.

[25] A. Vishnu, G. Santhanaraman, W. Huang, H. W. Jin, and D. K. Panda. Supporting MPI-2 One Sided Communication on Multi-Rail InfiniBand Clusters: Design Challenges and Performance Benefits. In *HiPC*, 2005.