The Implementation of One-Sided Communications for WMPI II

Tiago Baptista[2], Hernani Pedroso[1], and João Gabriel Silva[2]

[1] Critical Software, S.A.,
Instituto Pedro Nunes, R. Pedro Nunes, 3000 Coimbra, Portugal
hpedroso@criticalsoftware.com
[2] CISUC/Dep. Engenharia Informática
Universidade de Coimbra – PoloII, 3030-097 Coimbra, Portugal
baptista@student.dei.uc.pt, jgabriel@dei.uc.pt

Abstract. This paper describes the implementation of MPI-2 one-sided communications (OSC) for the forthcoming WMPI II product, aimed at clusters of workstations (Windows workstations, for now). This implementation is layered directly on top of the WMPI Management Layer (WML), rather than being on top of the MPI layer and as such can draw more performance from the new features of the WMPI's WML. The major features of this implementation are presented, including the synchronization operations, the remote memory operations and the datatype handling mechanism. Performance benchmarks were taken, comparing the message passing and the one-sided communication models, as well as to compare this implementation with one layered on top of MPI.

1 Introduction

The second version of the MPI standard [1], MPI-2 [2], introduced several new important features. Among others, it introduced the ability to make message passing without the involvement of one of the processes – one-sided communication (OSC). This standard's chapter specifies functions for single sided communication, where a process specifies both origin and target parameters.

Although this functionality can easily be mapped over shared memory systems, its implementation for distributed memory systems (e.g. clusters) is not straightforward. Most of the OSC implementations use the traditional MPI point-to-point functions to do one-sided communication when processes don't share memory [3, 4]. Nevertheless this approach cannot use the internal specific capabilities of the implementation. This paper presents an implementation of the OSC functionality for NT clusters inside the WMPI [5] library. This implementation joins the experience gathered by a first OSC implementation on top of MPI point-to-point functions [3] with the new internal structure of the WMPI library, which allows the use of several communication devices, where some of them may have RMA support.

We begin by describing the background of this implementation and library it belongs to. Next, the specifics of the architecture are presented, the synchronization operations, the remote memory access operations and the datatype handling, followed by performance benchmarks results. Finally, conclusions are presented regarding current and future work.

2 Background

The implementation of One-Sided Communications presented here was created for WMPI commercial implementation. As part of this project, all MPI-2 [2] features are being implemented on WMPI 1.5 – the current MPI implementation – that will then evolve into the WMPI II product.

WMPI was first implemented in 1996 [6], based in the MPICH/p4 [7] implementation. It was the first MPI implementation for Windows™ systems. Since then the library has undergone several improvements and in the last year a complete new structure was created (WMPI 1.5) [5]. This implementation allows the use of several communication devices simultaneously, is thread-safe and was the base for the implementation of the dynamic process creation functionality introduced in the MPI-2 standard [8].

An OSC implementation had already been created on top of WMPI 1.1 [3]. That implementation had a somewhat different methodology as it was layered on top of the MPI point-to-point functions, whereas the new implementation is layered directly on top of WML (WMPI Management Layer) [5], the layer that translates the MPI actions into the library and operating system specific actions. WML is the responsible layer to manage the communications, choose the correct communication device, request to send the messages and verify when new data arrives.

Hence, we were able to design several enhancements to one-sided communications using the new features of WMPI 1.5. One of these new features is the ability to handle multiple devices simultaneously and to have those devices in a separate module. The one-sided communications were as such, implemented to allow direct RMAs for devices that support them. For now, the shared memory device is already implemented allowing these direct RMAs. In the future other devices may follow, like a device for VIA.

3 One-Sided Communications Architecture

This implementation of one-sided communications was designed always having in mind the new features of WMPI 1.5 and the best way to gain performance and stability using these features. The object-oriented methodology was chosen to design and implement OSC, as it allowed for better layering and encapsulation of functionality. The standard itself is not particularly pointed towards an object-oriented interpretation. However, if one reads it having in mind an object-oriented design, it all begins to look like it had been thought this way. It is rather straightforward to define the architecture in terms of classes, despite the lack of other implementations having taken this path. Take the case of windows and epochs for example. A window has both an access and an exposure epoch. These can be any kind of epoch. Thinking object-oriented we will have the class CWindow that will have two objects of the class CEpoch. This class is the base class for all epoch classes. This way, the class

CWindow won't require the knowledge of what type of epoch is open and the epoch classes can be implemented separately. That way they won't interfere with each other. As the object-oriented paradigm can be implemented in C++, the performance won't suffer.

One of the major architecture decisions taken regards the functions where to block: the epoch open, the remote memory access or the epoch close. The standard allows any of the stated. For this implementation the choice was to block on the RMA, if the target has not yet called the epoch's open function. This decision took primarily into account the fact that the RMA may be a direct access to the remote processes memory and as such it is required that the remote exposure epoch be opened prior to performing the operation.

3.1 The Synchronization Operations

There are three different synchronization methodologies, the fence, the start-post and the lock mechanisms.

Fence. The fence mechanism is the most basic form of synchronization in MPI-2 one-sided communications. Using a call to MPI_Win_fence, access and exposure epochs are opened for all the processes in the communicator of the window. This call is a collective call and as such involves all the processes that share the window. The same MPI_Win_fence function is used both to open and to close the epochs. As this mechanism involves all the processes in the communicator, the synchronization is achieved using a call to MPI_Barrier. As to ensure that all the remote memory accesses issued before this call are complete, every process checks and waits until all pending operations are complete.

Start-Post. For applications where processes communicate only with few other processes, there is no need to include all the processes of the window in each synchronization operation. Using the start-post mechanism, groups of communicating processes can synchronize themselves without involving the other processes of the window. The function MPI_Win_start opens an access epoch to a group of processes it receives as a parameter. This function will never block nor send any message. It will only initialize the access epoch and process any pending posts received by this process. The epoch created will only grant access to a process after having received its post message. Every process of the group passed to MPI_Win_start has a matching MPI_Win_post call. This function will create and initialize an exposure epoch allowing all processes of a group received as a parameter. It will send a post message to every process in the group allowing them to open an access epoch.

To close the access epoch, a process must call MPI_Win_complete. This function will send a complete message to all the processes in the group of this epoch (passed to MPI_Win_start) and checks that all the RMAs issued are complete at the origin. This function will also check if all posts were received prior to closing the epoch, thus ensuring that a subsequent call to MPI_Win_start won't receive a post that was meant for the previous epoch.

To close the exposure epoch at the target, a process must call MPI_Win_wait. No messages are sent. The function waits for the complete messages from all the processes in the group of this epoch (passed to MPI_Win_wait) and checks that all RMAs are complete at this process (the target). All RMAs are sure to be complete at the origin because the complete message was received.

Lock. For applications requiring a truly one-sided synchronization mechanism, the lock/unlock methods were introduced. These methods implement the so-called *passive target* synchronization. That name refers to the ability of one process to exclusively lock another process without the latter having an active part in the process.

By calling MPI_Win_lock, a process can open an access epoch to the process whose rank is passed as a parameter. No explicit exposure epoch is required to be opened at the target. However, an internal exposure epoch is opened to maintain consistency with the other two synchronization methods and to allow the RMAs to be generic, having no knowledge of the type of epoch in use. The lock acquired can be either exclusive or shared. Several processes can lock the same target simultaneously provided that all the calls to MPI_Win_lock are shared. However, if a process is exclusively locked, no other (shared or exclusive) lock can be acquired until the call to MPI_Win_unlock by the origin. The MPI_Win_lock function doesn't block. It only sends the request to the target process. A subsequent call to an RMA will, nevertheless, block until the lock is acquired.

To close the access epoch and release the lock, the origin process must call MPI_Win_unlock. As the MPI_Win_lock didn't block, this methos method must check if the lock is acquired before proceeding with the release. On exit of this function all RMAs must have completed both at the origin and at the target. To accomplish this, first the origin process checks if all RMAs are complete, next a message is sent to the target requesting the release of the lock. At the target, checks are made to ensure all RMAs are complete and sends to the origin the unlock reply message. The origin can now exit from the function.

3.2 The Remote Memory Access Operations

The remote memory access operations were implemented as generic, i.e., the type of epoch in use is not known by the function. This was easily accomplished due to the object-oriented paradigm followed.

There are three different situations that can occur when calling an RMA: the datatype is not contiguous, the datatype is contiguous and the device supports direct RMA or the datatype is contiguous and the device does not support direct RMA. To gain performance wherever possible, the implementation takes into account the kind of operation being performed and acts accordingly. Even so, there is some functionality that is common to all three. First the function checks the current epoch to ensure that the RMA can proceed (e.g. the matching post has been received), and then it verifies if the block to transfer fits the target location.

Now there are the three different options. If the datatype is not contiguous, the data must be packed and the datatype sent along with the packed data. If the datatype is contiguous and the device does not support RMA, there is no need to pack the data or send the datatype. A message is sent using the memory location of the RMA as the send buffer. If the datatype is contiguous and the device does support RMA, no message is sent. A direct call to the device is made to process the request.

The Put is implemented with none (the case of direct RMA) or one message exchange and the Get with none or two messages. The Accumulate is implemented the same way as Put but no direct RMA is possible. It would require two direct RMAs to perform the Accumulate and this would be less efficient than sending one message. The datatype is always needed to perform the operation at the target.

3.3 Datatype Handling

Datatype creation functions are local functions, which build local representations of datatypes, which may be unknown in other processes of the computation. Opposite to the point-to-point functions, in one-sided communication, the receive parameters are specified in the origin process. One of the parameters to specify is the receiving datatype, however there is no guarantee that the specified datatype is available at the remote process.

This new OSC implementation has some different conditions than the previous one, it is developed for homogeneous environments and may use communication devices that allow for remote memory access. These two new conditions simplify the situation when the receive datatype is contiguous and the communication device has support for RMAs. In this case the un-marshal and marshal of the data is done locally and the result buffer is copied directly to the target process's memory. If the send datatype is equal to the remote datatype (and is contiguous), then the user data buffer can be directly copied to the remote process without further computation (this is the best case).

Nevertheless if the receive datatype is non-contiguous it is preferable to pack the data and send it to the remote process, there the data should be unpacked and copied into the target memory region.

If the communication device does not have RMA support, then the communication has to be performed over the traditional message passing mechanisms, and the target process has to know the receive datatype to un-marshal the data. In this case, we use the same scheme as in the case of the previous OSC version. We pack the datatype information and send it along with the data.

The last two scenarios have penalties due to the local characteristics of the datatypes. Our future development will address this issue using the datatype caching [9] and signatures [10]. Instead of sending the datatype along with the data, only its signature is sent. If the remote process has a datatype with the same signature, then the data is un-marshaled using that datatype, otherwise the remote process has to request the datatype. If the datatype is not present at the remote site, then the operation has a considerable penalty, however the most common case is that the user has specified the datatype in the remote process. If the datatype has to be sent to the remote process, techniques to efficiently pack it, like the one presented in [11], will be used.

4 Performance

To benchmark this implementation, we compared the current with the previous OSC implementation [3]. Although the underlying programming models differ, some comparisons were also made between the MPI 1 Pt2Pt and MPI 2 OSC performance results. The program used to benchmark was a third party application, the Pallas' MPI Benchmarks [12]. This is the only available set of benchmarks that support one-sided communications. The runs were performed on a Windows 2000 dual PIII (for shared memory) and on a Windows 2000 PIII (for TCP) connected by a switched 100Mb/s Ethernet. All the results refer to two processes. The time is the average of 1000 operations synchronized with MPI_Win_fence. The benchmark has both an aggregated and a non-aggregated version. On the aggregated benchmark, the synchronization happens at the start and finish of the 1000 operations, hence being called twice. On the non-aggregated the synchronization happens twice for every operation. The aggregated results were chosen. This would be the normal operation of a real world OSC application.

Figure 1 shows the results gathered for shared memory and figure 2 shows the results for TCP.

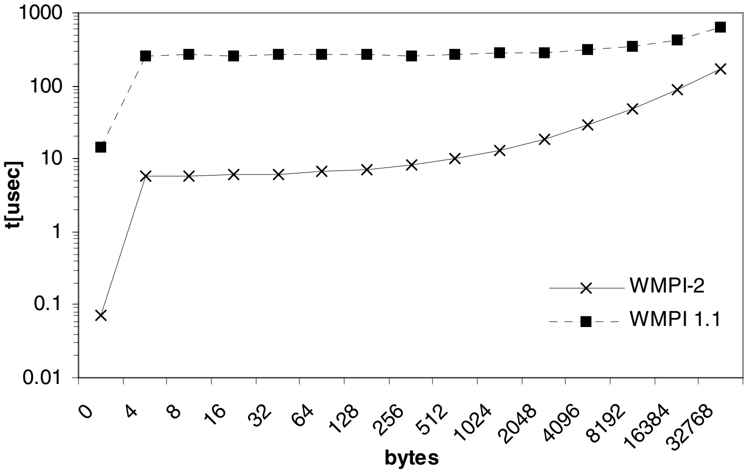

Fig. 1. Unidir_Get in WMPI II OSC vs WMPI 1.1 OSC (shmem)

As can be observed, the performance of the WMPI II OSC implementation is faster than that of WMPI 1.1 on shared memory. This is mainly due to the use of direct RMAs on the shared memory device. The performance for TCP is also better on the new implementation, except for some oscillations that are still occurring at this stage. So far tests have shown that the problem lies in the utilization of the winsock library. We are investigating this issue to discover the cause and solve the problem.

n figure 3 we show the results for the comparison between the MPI 1 and MPI 2 Pallas' benchmarks, using shared memory. We compare the unidir_get to the pingpong and the bidir_get to the pingping. The results show that the performance is very similar, with the one-sided communications taking the lead up to some extent. This shows that this implementation of OSC brings no overhead to the communication operations and can, with a properly implemented synchronization scheme, bring many enhancements to distributed applications.

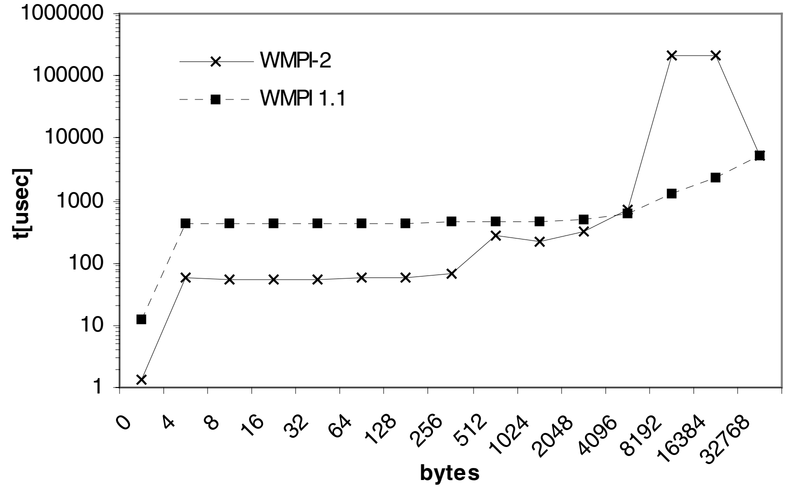

Fig. 2. Unidir_Get in WMPI II OSC vs WMPI 1.1 OSC (tcp)

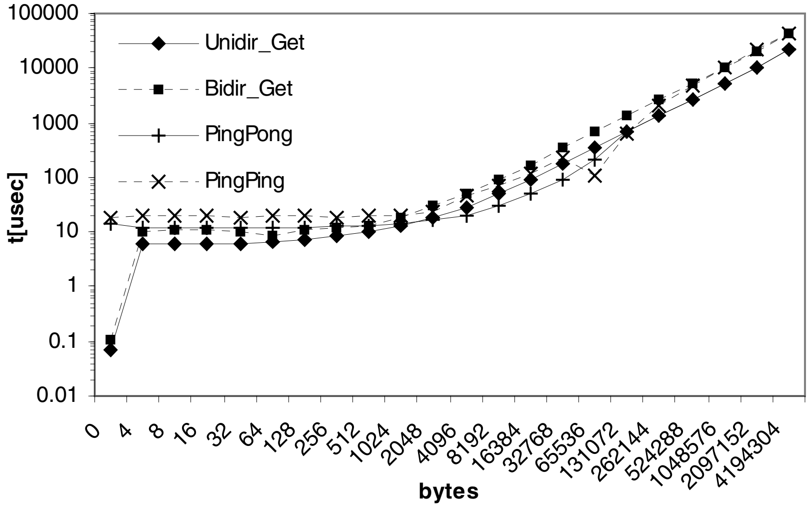

Fig. 3. Comparison between Unidir_Get/Bidir_Get with PingPong/PingPing on shared memory

5 Conclusion

This implementation of OSC is in the testing and tuning phase, but still, some very interesting performance results were shown. The decision to layer this implementation directly on top of WMPI's WML allowed to effectively explore all the new features of WMPI and of the underlying Operating System. Due to the object-oriented paradigm followed by this implementation, all this was implemented easily and any future enhancements to the WML or Operating Systems will be straightforward to explore.

Due to the inexistent loss in performance from double sided to single sided operations, users can try different programming models, using OSC, without fearing loss of performance.

References

1. Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J.: MPI – The Complete Reference, Volume 1, The MPI Core. MIT Press, second edition (1998)
2. Gropp, W., Huss-Lederman, S., Lumsdaine, A., Lusk, E., Nitzberg, B., Saphir, W., Snir, M.: MPI – The Complete Reference, Volume 2, The MPI Extensions. MIT Press (1998)
3. Mourão, F., Silva, J.G: Implementing MPI's One-Sided Communications for WMPI. Proc. of 6th European PVM/MPI Users' Group Meeting, pp 231-238, Springer-Verlag, Barcelona, Spain (September 1999)
4. University of Notre Dame: LAM-6.3 release notes. (1999) http://www.mpi.nd.edu/lam/
5. Pedroso, H., Silva, J.G.: An Architecture for Using Multiple Communication Devices in a MPI Library. Proc. of 8th High-Performance Computing and Networking Europe, pp. 688-697, Springer-Verlag, Amsterdam, The Netherlands (May 2000)
6. Marinho, J., Silva, J.G.: WMPI – Message Passing Interface for Win32 Clusters. Proc. of 5th European PVM/MPI User's Group Meeting, pp. 113-120 (September 1998)
7. Gropp, W., Lusk, E., Doss, N., Skejellum, A.: A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. Parallel Computing Vol. 22, No. 6 (Sptember 1996)
8. Pedroso, H., Silva, J. G.: MPI-2 Process Creation & Management Implementation for NT Clusters. Proc. of 7th European PVM/MPI Users' Group Meeting, pp. 184-191, Springer-Verlag, Balatonfüred, Hungary (September 2000)
9. Booth, S., Mourão, E.: Single sided MPI implementations for SUN MPI. In Supercomputing 2000 (2000) http://www.sc2000.org/techpapr/papers/pap.pap182.pdf
10. Gropp, W.: Runtime Checking of Datatype Signatures in MPI. Proc. of 7th European PVM/MPI Users' Group Meeting, pp 160-167, Springer-Verlag, Balatonfüred, Hungary (September 2000)
11. Träf, J.,Hempel, R., Ritzdorf, H., Zimmermann, F.: Flattening on the Fly: Efficient Handling of MPI Derived Datatypes. Proc. of 6th European PVM/MPI Users' Group Meeting, pp 109-116, Springer-Verlag, Barcelona, Spain (September 1999)
12. Pallas GmbH: Pallas MPI Benchmarks – PMB. (April 2001) http://www.pallas.de/pages/pmb.htm
13. Critical Software S.A. – WMPI. http://www.criticalsoftware.com/wmpi