Implementing MPI's One-Sided Communications for WMPI

Fernando Elson Mourão and João Gabriel Silva

Universidade de Coimbra, Departamento de Eng. Informática,
Pólo II, 3030 Coimbra, Portugal
elson@dsg.dei.uc.pt, jgabriel@dei.uc.pt

Abstract. One-sided Communications is one of the extensions to MPI set out in the MPI-2 standard. We present here a thread-based implementation of One-sided Communications written for WMPI, an existing Windows implementation of MPI written at the Universidade de Coimbra. This is a major step towards WMPI incorporating the MPI-2 standard, with the further benefit of contributing to the thread safety of WMPI. We discuss the main design decisions associated with the implementation and consider further research work required in this area to improve both the existing implementation and to assess other implementations of One-sided Communications.

1 Introduction

MPI is the de facto standard for message passing, and its acceptance is so wide that the demand for new features increases rapidly. So the MPI Forum released the MPI-2 standard[1] in June 1997. This paper describes the implementation of one of the most important new chapters in the standard, the One-Sided Communications (OSC) chapter. This implementation is an extension to an existing Windows implementation of MPI and represents the first step towards MPI-2 compliance.

This paper is laid out as follows. Firstly section 2 gives background information placing this implementation of OSC into context. In section 3 the implementation is discussed, including major design decisions and performance issues. Following this, section 4 suggests directions for further research and work, and finally section 5 concludes the paper.

2 Background

The implementation of OSC discussed here was done over an existing Windows implementation of MPI, the Windows Message Passing Interface (WMPI) [2, 3].

WMPI is now in the process of being extended to meet the requirements of the MPI-2 standard [1]. The implementation of OSC forms part of this work. Below we give brief details of WMPI, the OSC chapter in the MPI2 standard and an overview of OSC.

2.1 WMPI

WMPI was the first implementation of MPI for computers running the Windows operating system. This implementation was originally based on MPICH [4–7] but it has been tuned and recently Mark Baker showed[8] that WMPI was the fastest Windows implementation freely available. The idea behind WMPI is to take advantage of the evergrowing number of Windows based machines and that purpose has been achieved.

2.2 MPI 2

MPI-2.0, as stated by the MPI Forum, is a set of extensions to the MPI-1.1 standard. These extensions include a defined way of running MPI processes, C++ bindings and thread compliance. However its main new features are discussed in its four main chapters:

- Process Creation and Management is a first simple approach to allow MPI applications to launch more MPI processes during runtime, a feature mostly needed in networks of workstations (NOWs) and clusters of PCs (COPs).
- Parallel I/O concerns the parallel and distributed environments but is outwith the message passing scope.
- Extended Collective Operations are a true extension to the existing collective operations, but also an extension to allow collective operations to cope with Process Creation and Management.
- One-Sided Communications are asynchronous communications that allow one process to specify both the sending and receiving parameters for the message being transferred, hence the name "one-sided". This also means that the remote process involved does not have to explicitly call any MPI function to send or receive the message.

As is evident from their names, only one, Extended Collective Operations, refers to pure message passing. The reasons for this seem to be related to the fact that MPI is being used so widely that there is a need to cover other areas aside from pure message passing. Moreover, nowadays the use of NOWs and COPs for parallel computing is a reality, which seems to have driven the MPI Forum to take into consideration the needs of these types of machines in presenting chapters such as Process Creation and Management.

2.3 One-Sided Communications

The MPI Forum also names OSC function calls as remote memory access (RMA) calls. There is a set of synchronisation functions to control the access to remote processes' memory, and a set of RMA functions to retrieve and put data into a remote process's memory.

For RMAs to be issued a group of processes have to call an initialisation function, `MPI_Win_Create`. There each process states the amount of memory that

is available for remote access, as well as giving a pointer to that space. When the RMAs are finished the processes call `MPI_Win_free` to release the memory and close remote accesses.

Between these two calls any number of synchronisation and RMA calls can be issued. The synchronisation calls open what the standard refers to as **epochs**. Epochs can be **access epochs** or **exposure epochs**. If a process A is issuing RMAs to a process B then process A must have an access epoch open and process B must have an exposure epoch open. There are three types of synchronisation calls that can be used:

Fence (`MPI_Win_fence`) is a global (to the group of processes that initially called `MPI_Win_Create`) synchronisation call which opens both exposure and access epochs in all the processes.

Start/Post (`MPI_Win_start`, `MPI_Win_complete`, `MPI_Win_post`, `MPI_Win_wait`) are two pairs of synchronisation calls that open and close an access epoch and a corresponding exposure epoch on a group of processes. The accessing processes call start and complete, which respectively open and close an access epoch, while the targeted processes have to call post and wait to respectively open and close an exposure epoch.

Lock (`MPI_Win_lock`, `MPI_Win_unlock`) is a one-to-one synchronisation call that opens an access epoch at the calling process and an exposure epoch at the given target.The exposure epoch is opened without the target process having to call any synchronisation call or even being aware of its memory being accessed.

The RMA calls are:

- `MPI_Get` to read data to remote processes.
- `MPI_Put` to write data to remote processes.
- `MPI_Accumulate` to write data to remote processes but using an operation over the existing data.

To avoid repeating the standard refer to the MPI 2.0 Standard document for further details.

3 Implementation

This implementation is a first prototype and improvements are expected. In particular since many of the implementation options were tightly restricted. As well as the standard's requisites there was already a fully running implementation of MPI which had not been planned to satisfy the needs of OSC. Thus some options taken were driven by the fact that it would not pay off to undertake certain changes to the existing code. The most relevant ones relate to the asynchronous agent to handle the requests, and the issue of datatypes handling. This section discusses the most important implementation options taken and the reasons for them.

3.1 Synchronisation Model

The standard states that OSC follows a loose synchronisation model. For that the synchronisation function calls should only block when strictly necessary. However the standard allows an implementation to block on all synchronisation calls if desired. The implementation discussed here behaves as follows:

- The fence call blocks when it is closing an epoch.
- The start call blocks if any of the processes in the group has not yet closed a previous epoch from the calling process.
- The complete call only blocks if there are RMAs requests waiting for a reply.
- The post call does not block.
- The wait call blocks until all processes in the accessing group call complete.
- The lock call only blocks if the target process is the local and there is a lock being held already. Locks to remote processes do not block under any circumstance.
- The unlock call blocks until all the issued RMAs receive a reply and until the lock epoch is closed at the target process.

This behaviour is more complex to implement than blocking all calls, but copes better with network latency in COPs. Moreover if all calls were to block then the whole purpose and advantage of loose asynchronous communications would be lost.

3.2 To Thread or Not to Thread

In NOWs and COPs there is no native support for RMAs, so an asynchronous agent is required to handle requests. This can be achieved either by using specific hardware or by implementing it with software. There are several ways of implementing it depending on the system it is being implemented for. The two approaches we considered for OSC were:

- All MPI calls check for asynchronous OSC requests.
- Use a separate software agent such as a thread or a dedicated process.

The first option would require all MPI calls to check if a request for RMAs had been issued and if so the request would be dealt with. This option requires less dramatic changes to the existing code, but it is easy to see that if a targeted process does few calls to MPI the performance is affected. Scalability is poor and the method is prone to process starvation and deadlocks. It also might cause delay on simple MPI calls if a reasonable number of OSC requests are on hold and have to be processed.

The second option could be implemented using a process or using threads. If a process were to be implemented then a large amount of data would have to be shared between this process and the MPI processes. Thus interprocess communication mechanisms such as semaphores and shared memory would have to be

used intensively. These mechanisms, along with context switching between processes, are very expensive in terms of performance. Thus threads were considered to be a better option.

Having decided to use threads a second decision was required: to use only one thread per process or one thread per window. In the first case one thread would serve all windows of a given MPI process. In the second case each window which the process creates has a thread associated with it. Our conclusions were that one thread per process could easily become a bottleneck, is obviously more complex and in certain ways defeats the purpose of using threads.

In the implementation scheme used, a thread is created in the `MPI_Win_create` call each time a new window is created. The thread is destroyed by the `MPI_Win_free` call.

3.3 Datatypes

The datatypes handling functions are not required to be global operations. The WMPI implementation relies on this fact to make datatype handling calls local.

Considering that the internal data representation can differ from machine to machine, sending or receiving data using OSC becomes an issue. When using regular MPI send and receive calls this is not a problem because when data is received the local datatype is used to un-marshal the data. However when using OSC the datatype is unknown at the receiving end. Both datatypes (send and receive) are given as parameters at the process issuing the RMA. Thus the controller thread has no information about what datatype to use to marshal or un-marshal the data.

The solutions found to this problem are:

- Pack and send the needed datatype information with each request.
- Change datatype handling function calls to become global operations.
- Use datatype caching to improve performance over the first option presented here.

The solution implemented was the first of these, due to the fact that the other two required further research and effort which was beyond the project's scope. A more detailed discussion on the last two options is presented in section 4.

3.4 Performance

As stated before performance was not the priority for this project. Although some benchmarking was planned it could not be performed.

The planned benchmarking was to be done using third party benchmarking applications. However no suitable applications were found. This can be explained by the fact that there are few implementations of OSC and those that do exist are not in the public domain.

A first formal analysis suggests that the results are likely to be below that expected for a high performance library. The most obvious reason for lower performance is the datatype information sent with each RMA request. A less obvious reason is that the actual RMA requests are sent and processed individually. However an algorithm that takes advantage of the loose synchronisation model could improve the performance of the current implementation. The following section describes some of these as a subject of further research.

4 Further research

This section highlights areas where further research and work on the OSC implementation is needed. While this list is not exhaustive we believe it covers the major issues.

4.1 Datatypes Handling

As discussed in section 3.3 handling datatypes proved to be a matter of concern in implementing OSC. Datatypes are local to MPI processes but in OSC MPI processes need to know about datatypes belonging to other processes.

At the moment each time an RMA is issued the required datatype information is sent with the request. However we suggest two approaches to improve this solution: global IDs for datatypes and caching of datatype information.

Global IDs: If the datatypes were identified by a global ID then the problem no longer exists, as all processes will have the required information. However to achieve this all current MPI datatypes handling calls would have to become collective. There are three main disadvantages to this option:

- It does not follow the standard.
- It adversely affects performance.
- Under dynamic process creation the propagation of global IDs and datatypes' information to the new processes has to be done, i.e. we still have the same problem.

Cache of Datatypes Information: If the datatype information currently sent with each RMA request were cached by the controller thread, then subsequent RMAs would not need to send the information again. This approach exploits locality of reference in long running high performance applications. Additionally it does not require changes to the current WMPI implementation and is an extension to the OSC implementation presented here.

4.2 RMA Grouping

There is potential to improve the handling of RMA requests. Instead of issuing RMA calls individually, grouping them could cope better with latency and low bandwidths. Further research is needed to find the optimal grouping scheme, or to develop an adaptative algorithm to suit the needs of an application at a given time. For instance, one issue to be considered is the tradeoff between the size of the message and the time required to process the number of RMAs in the message.

4.3 Benchmarking

Benchmarks are fundamental to high performance libraries. They are not only a way of assessing improvements but more importantly to spot areas that need improvement. For OSC we consider that benchmarking should concentrate on the synchronisation calls as these are the ones that require more processing and data checking.

5 Conclusion

In this paper we have described the implementation of One-Sided Communications for WMPI. The first results are satisfying, but the lack of proper benchmarking applications restricted the project work. Further work is required in the area, in particular to develop benchmarks, assess the effects of grouping RMA accesses and to improve datatype handling.

References

1. Message Passing Interface Forum, "MPI-2: Extensions to the Message-Passing Interface", June 1997.
2. Marinho, José, Silva, João Gabriel: WMPI - Message Passing Interface for Win32 Clusters, in Proc. of 5th European PVM/MPI Users' Group Meeting, pp. 113–120, September 1998.
3. José Marinho: Realização Prática da Norma MPI para Redes de Computadores Pessoais, MSc thesis, August 1996, Universidade de Coimbra
4. William Gropp, Ewing Lusk, "MPICH Working Note: Creating a new MPICH device using the Channel interface - DRAFT", ANL/MCS-TM-000, Argonne National Laboratory, Mathematics and Computer Science Division
5. William Gropp, Ewing Lusk, "MPICH ADI Implementation Reference Manual - DRAFT", ANL-000, Argonne National Laboratory, August 23, 1995
6. Ralph Butler, Ewing Lusk, "User's Guide to the p4 Parallel Programming System", Argonne National Laboratory, Technical Report TM-ANL-92/17, October 1992, Revised April 1994
7. W. Gropp, E. Lusk, N. Doss, and A. Skjellum: "A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard", Pre-print MCS-P567-0296, July 1996.

8. Mark Baker, "MPI on NT: The Current Status and Performance of the Available environments", in Proc. of 5th European PVM/MPI Users' Group Meeting, pp. 63–73, September 1998.