
THE UNIVERSITY *of* EDINBURGH

# McMPI – a Managed-code Message Passing Interface Library for High Performance Communication in C#


*Daniel Holmes*

A thesis submitted to

The University of Edinburgh

in fulfilment of the requirements for the degree of

Doctor of Philosophy

October 2012

# Abstract

This work endeavours to achieve technology transfer between established best-practice in academic high-performance computing and current techniques in commercial high-productivity computing. It shows that a credible high-performance message-passing communication library, with semantics and syntax following the Message-Passing Interface (MPI) Standard, can be built in pure C# (one of the .Net suite of computer languages).

Message-passing has been the dominant paradigm in high-performance parallel programming of distributed-memory computer architectures for three decades. The MPI Standard originally distilled architecture-independent and language-agnostic ideas from existing specialised communication libraries and has since been enhanced and extended.

Object-oriented languages can increase programmer productivity, for example by allowing complexity to be managed through encapsulation. Both the C# computer language and the .Net common language runtime (CLR) were originally developed by Microsoft Corporation but have since been standardised by the European Computer Manufacturers Association (ECMA) and the International Standards Organisation (ISO), which facilitates portability of source-code and compiled binary programs to a variety of operating systems and hardware.

Combining these two open and mature technologies enables mainstream programmers to write tightly-coupled parallel programs in a popular standardised object-oriented language that is portable to most modern operating systems and hardware architectures.

This work also establishes that a thread-to-thread delivery option increases shared-memory communication performance between MPI ranks on the same node. This suggests that the thread-as-rank threading model should be explicitly specified in future versions of the MPI Standard and then added to existing MPI libraries for use by thread-safe parallel codes.

This work also ascertains that the C# socket object suffers from undesirable characteristics that are critical to communication performance and proposes ways of improving the implementation of this object.

# Declaration

Except where otherwise stated, I declare that this thesis, and the work herein described, has been undertaken by myself unaided. The work reported has not been submitted in application for any previous degree or qualification.

D. Holmes

5th October 2012

# Acknowledgements

I wish to thank EPSRC for funding this work, without which it would not have happened.

I wish to thank The University of Edinburgh, specifically EPCC, for supplying office space and test hardware for the duration of the project. I also wish to thank the staff of EPCC for their comments, suggestions, encouragements and informative discussions. In particular, I wish to thank my supervisors, Dr Stephen Booth and Dr Judy Hardy, for their technical expertise, wisdom and dedication.

I wish to thank my wife, Toni Collis, for keeping me on the straight and narrow path and for making copious quantities of cake.

Finally, I wish to thank my family for their unwavering optimism.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

In recent years, computing power has been increased by increasing the number of central processing units (CPUs) rather than by increasing the processing speed of each CPU. This trend began with the advent of multi-processor computers and continues today with multi-core processors including the use of graphics processing units (GPUs) for general purpose computing. Exploiting the computing power provided by multiple processing units requires that multiple series of instructions are executed simultaneously in parallel. Each series of instructions can perform independent tasks, i.e. separate tasks that achieve unrelated goals, which may be referred to as high-throughput computing. Alternatively, each series of instructions can perform dependent tasks, i.e. sub-tasks that together achieve a single goal, which is commonly called high-performance computing (HPC).

For some goals, HPC is desirable because it allows the goal to be achieved in less time. For example, a goal that would take years to complete using a single CPU might complete in hours when using thousands of CPUs. This is sometimes referred to as increased capacity.

For some goals, HPC is essential because the goal cannot be completed without it. For example, a goal that requires more memory than can be attached to a single CPU can use multiple CPUs with extra memory attached to the extra CPUs. This is sometimes referred to as increased capability.

HPC provides both the capability to achieve some goals that are otherwise not possible and the capacity to achieve some goals faster than is otherwise possible.

A shared-memory computer is one in which all the memory is attached to all the CPUs meaning that any CPU can directly access any part of the memory without assistance from other CPUs. A distributed-memory computer (or system) is one in which each CPU can only directly access part of the memory in the system. Accessing information in other parts of the memory requires co-operation and inter-process communication.

## 1.1 HPC in Academia and Science

HPC is well established in academia, especially in sciences such as physics. The equations that describe physical systems, from atoms to galaxies, from fluid-flow inside a blood vessel to plasma-flow inside a fusion reactor, can readily be converted into parallel computer programs that exploit thousands of CPUs.

Typical computer systems can attain Tera-scale performance, i.e. of the order of $10^{12}$ operations per seconds, although the fastest systems can attain Peta-scale performance, i.e. of the order of $10^{15}$ operations per seconds. Current challenges include making efficient use of existing Peta-scale computer systems and increasing computing power to Exa-scale performance, i.e. of the order of $10^{18}$ operations per second.

## 1.2 The Commercial Market for HPC

HPC is not well established in industry and business. Many business applications require high-throughput computing rather than HPC, for example to increase the number of customers that can be using a web site at once rather than to increase the responsiveness of the web site by making each operation it performs faster. In addition, writing efficient parallel programs needs a skillset that is rarely found in main-stream computer programmers. However, HPC is becoming more popular, in particular for financial institutions where a greater capacity for quickly performing complex calculations can bring a commercial advantage.

## 1.3 Parallel Computing Methodologies

### 1.3.1 Client-Server and Peer-to-Peer

There are already ways to produce distributed, parallelised code without needing detailed expertise or knowledge of HPC. Two examples are web-server farms and federated databases. Currently popular products include Microsoft Information Server (IIS) or Apache web servers and Microsoft SQL Server, Oracle or MySQL databases. In both of these, someone else (typically a large software company, such as Microsoft or Oracle) has done the majority of the hard work, i.e. all of the parallelisation and communication. In addition, the parallelism and communication is often completely abstracted or hidden from the programmer.

For a web-server environment, the programmer is allowed to produce serial code but is required to guarantee that it is thread-safe. The web server then executes multiple instances of the serial code in response to multiple requests. This is usually used for client-server applications where a graphical user interface queries a centralised business logic application layer but can also be turned to task-farm codes where a master process distributes tasks to multiple, identical workers. This addresses a limited subset of parallel codes but does not include any provision for coupling between the workers or dependencies between the tasks. Database programmers make use of a language such as Structured Query Language (SQL) to specify what they wish to do but not how they wish it to be done. The database is then free to perform the necessary operations in parallel, if that is deemed (by the database itself – not the programmer) to be a good idea. Typically data is stored in tables that could be physically distributed and operations to cross-reference, filter one table with data from another, sort a distributed result-set and so on, will require multiple machines to cooperate in a loosely or tightly coupled manner. However, this complexity is completely transparent to the programmer, i.e. the query remains the same whatever the distribution of the data or of the eventual work-load and even if the algorithm for an operation is altered, for example a parallel sort instead of a sequential sort. SQL is not a general-purpose programming language and only supports a well-defined subset of operations that are useful in querying databases. This makes it easier for a vendor to develop the parallelisation code inside the database but limits its applicability. Whilst web-servers execute code written in a general purpose language the requirement for thread-safety and the lack of provision of intercommunication and cooperation between worker processes once again limits its applicability.

### 1.3.2 Generative Communication

Linda [1], although a fairly old idea, and one that is out of favour at the moment, is a novel approach to parallelisation that is deserving of consideration, especially for non-expert programmers, because of its simplicity of expression. Even complex parallel algorithms can be coded in a straight-forward, easy to comprehend manner. The concept of tuple-space is unorthodox and possibly very difficult to implement efficiently in practice but provides a way to decouple multiple dependent tasks, which allows for a modular approach to design and coding. There are similarities with many other subsequent suggestions and ideas in this field, including distributed shared memory (section 1.3.4), partitioned global address space

(section 1.3.5), parallel languages and compilers (section 1.3.6), as well as agents and actors (section 1.3.7). Tuple-space may be considered as a type of globally shared memory and a "live tuple" is quite similar to an agent or actor. The four extra statements added to the base language to support the necessary operations on tuples must be efficient when compiled into machine-code and executed. This is the same difficulty facing designers and implementers of parallel languages and parallelising compilers.

### 1.3.3 Shared Memory

Programming for shared memory machines is easier than for distributed memory machines because the programmer does not need to choose the optimal distribution for the data to achieve acceptable load-balancing. It is often the case that an optimal data-distribution for one operation is non-optimal for another. Re-distributing data is usually very expensive so a compromise much be reached whereby the overall benefit is maximised. When all the processes have access to all the data all the time, as in shared-memory systems, these considerations become irrelevant and only the work must be portioned out correctly to achieve the best load-balance and therefore the best runtime performance.

A popular method of apportioning the work in shared memory systems is OpenMP [2], which uses compiler directives to indicate areas of the code that should be executed by more than one thread. Typically, the compiler splits the iterations of a loop into groups that are executed by multiple threads simultaneously.

### 1.3.4 Distributed Shared Memory

Unfortunately, hardware restrictions prevent shared-memory systems scaling beyond about 256 processors at the most, with current technology. For larger systems, physically distributing the available memory is the only option and processors must communicate with each other to access non-local data. There have been suggestions that a distributed memory system could be abstracted by a runtime system, software layer or special-purpose hardware so that it appears like a shared-memory system to the programmer.

A system called Munin [3], for example, relaxes the traditional memory coherency rules in order to allow for delayed updates and presents a single virtual memory space with coherency rules similar to cache coherent shared-memory machines. Data movement is still necessary but is handled automatically and transparently by the runtime system. Different

variables are treated differently, using familiar mechanisms such as replication, migration, invalidation and remote load/store. The choice of mechanism is based on the usage type of the data variable, classified by the number of processes that will access the variable with read and write operations. This allows the runtime to choose an appropriate communication algorithm for each variable and optimise the timing of any data movement necessary for each access operation. Good speed-ups (within 5-10% of equivalent hand-coded and optimised message-passing applications) were reported when this approach was implemented on a 16-processor prototype system [4].

### 1.3.5 Partitioned Global Address Space

There are several research projects in progress that are focused on mitigating non-uniform features of modern architectures, in particular intra-node memory access time compared with inter-node in a compute cluster. In general, the abstractions prevalent in most programming languages treat all memory identically and behave as if all memory accesses will take the same amount of time as each other. This has clearly not been the case for many years, e.g. registers, L1, L2 and L3 cache memory are much faster than main memory, but these differences have been hidden from the programmer by performing many operations in hardware including maintaining memory coherency amongst all the various levels of caching, scheduling memory reads and writes in advance of computation operations and using pipelines and streams for multiple contiguous accesses. Continuing to use hardware to simulate fast memory is not feasible as memory resources increase in size and, crucially, become increasingly disparate and fragmented. Introducing a new model that allows the programmer to specify the data-dependencies and therefore the expected best pattern for locality seems like a good idea. Several languages (X10 [5], Titanium [6], UPC [7], D-PGAS [8] and Co-Array FORTRAN [9]) have been developed with this model of partitioning memory into locations in mind.

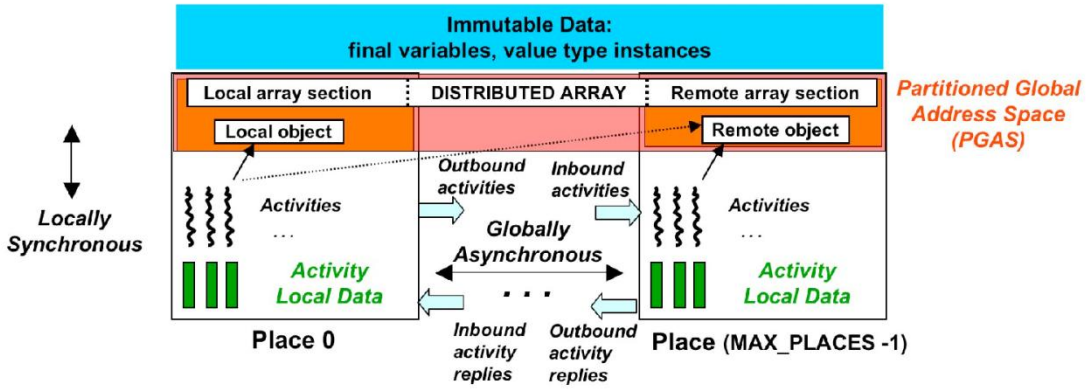


**Figure 1: Overview of X10 Activities, Places and PGAS [5].**

As mentioned in section 1.3.2, the diagram showing an overview of X10 activities, Places and PGAS (reproduced here as Figure 1) demonstrates remarkable similarity to Linda and tuple-space. The restriction in Linda that all shared data must be expressed as tuples is relaxed to allow more general objects and X10 activities do not explicitly have any data associated with them as do live tuples in Linda, although the description of asynchronous expressions and futures in X10 is almost identical to the description of live tuples except for minor differences in semantics and terminology. However, whereas Linda only adds four new tuple-space operations to a serial base language, X10 modifies Java (its base language) by adding new keywords, such as "`place`" and "`finish`", new semantic constructs, such as "`async`" and "`atomic`", new built-in parallel operations, such as "`foreach`" and "`ateach`", and new underlying methods, such as multidimensional arrays and value types. These are just some examples of the changes introduced by X10 and serves to illustrate the steepness of the learning curve associated with a new language, even when it is based on an existing one that is well-known, general-purpose and has a large re-usable framework in addition to its built-in syntax. On the other hand, this proliferation of new syntax does have the distinct advantage of making non-local communication explicitly visible in the code, which should aid the process of performance tuning, both by refactoring arising from source-code inspection (manual or automatic) and by compile-time checks.

### 1.3.6 Other Parallel Languages and Compilers

In addition to the ones mentioned in the previous sections, there are other parallel languages that do not mandate a particular memory-model. An early example, which strictly only allows for shared-memory concurrency rather than distributed-memory parallelism, is COOL [10]. An extension of C++, COOL adds the ability to annotate functions

as parallel (i.e. to be executed asynchronously and therefore concurrently) and to define variables as futures (unresolved until an asynchronous function completes). There are also some parallel languages supporting other programming paradigms, such as ParLog [11] for logic programming. A more recent example is Chapel [12], which promotes the idea of a single conceptual thread of control, like High-Performance FORTRAN (HPF), and includes the concepts of "locale", "domain" and "distribution" to assist in reducing latency by distributing data so that it is co-located with computation. The aim is for the programmer to provide the dependencies between computation and data without requiring a fully detailed communication strategy. This gives the compiler and runtime more freedom to choose an appropriate algorithm, which relies on the system making correct choices.

### 1.3.7 Agents and Actors

Using mobile agents (also known as actors) to simplify the design and implementation of complex distributed systems has been a promising technique for over a decade. Common design patterns for agent programming have been identified and classified [13]. Agents combine ideas of encapsulation from the object-oriented paradigm with a notion of autonomy, such as live tuples from Linda for example, and with freedom of movement so that the computation can be brought to the data it requires. The main aim of this approach is to reduce the complexity of the system, whilst increasing robustness and allowing code re-use, and hence increasing the productivity of its programmers. Personifying agent design patterns with descriptive names like Master, Slave, Messenger and Facilitator promotes intuitive reasoning about them, their role in the system and their interactions with each other. On the other hand, the runtime performance of an agent system is hard to predict because its operation is potentially fluid and adaptive as new agents are created when needed and movement across a network happens whenever an agent decides to change location. This presents challenges with avoiding hot-spots (both computational and in network traffic), contention over limited or locked resources and so on because these properties are not deterministic or easily controlled.

### 1.3.8 Generative Design Patterns and Architectural Skeletons

Design patterns (also known as architectural skeletons or templates and also seen in generic programming), are a very useful tool and have been applied to most programming styles. The principles necessary for a skeletal system to be successful, as suggested in [14] and extended in [15] for example, are desirable for all systems. There are many benefits of

pattern-based design, e.g. re-use of best practice, simple and understandable application structure and speed of prototyping using pre-implemented components. This increases the accessibility of computer programming to novice developers as well as simplifying and standardising the tasks of experienced developers. Several parallel pattern systems have been created, including DPnDP [16], PASM [17] and $CO_2P_3S$ [18], [19], [20].

The last of these, Correct Object-Oriented Pattern-based Parallel Programming System (abbreviated to $CO_2P_3S$ and pronounced "cops"), has been through many evolutionary stages over many years. Its origins can be traced to FrameWorks [21] and its successor Enterprise [22]. Work on $CO_2P_3S$ itself started with defining the programming model [18], continued with a shared-memory implementation in Java [19] and continued further with a distributed-memory implementation, also in Java [20]. The currently available version promises all of the anticipated benefits of pattern-design tools and addresses the three issues that are usually major obstacles, i.e. performance, utility and extensibility. Extensibility, the ability to create new patterns and integrate them into the tool, is provided by Meta $CO_2P_3S$, which describes the new pattern using extensible mark-up language (XML) that can be directly understood by the main graphical user interface (GUI). The utility of $CO_2P_3S$ was tested by implementing the Cowichan Problems [23]. Performance is good on shared-memory systems but is still an issue for distributed memory applications. At least part of the reason for this lack of performance is that the tool generates Java code that uses remote method invocation (RMI) to perform inter-node communication. As noted in section 1.5, RMI is not an efficient communication technology at the moment.

## 1.4 The Language Barrier

The fact that the current state-of-the-art for HPC requires the programmer to use a message-passing library written in FORTRAN or C is a potential barrier to rapid adoption outside academia, i.e. by commercial businesses. It may be argued that commercial businesses include the cost of developing a computer program, and therefore the productivity and training of computer programmers, in decisions such as which programming language to use. Many organisations have chosen object-oriented languages, such as Java and C#, and all code is written in the preferred language. Some interesting features and potential advantages of these two languages are discussed in work on the first version of McMPI [24].

## 1.5 MPI in Java

There have been many attempts to produce a version of MPI for computer languages other than C and FORTRAN, including higher-level languages such as Java. It is useful to include information and performance results from MPI-like libraries in any discussion about creating an MPI implementation for higher level computer languages, especially those considered to be high-productivity languages, because they commonly use the objected-oriented paradigm. Although the MPI standard [25] includes bindings for C++, which can be used for MPI libraries written in object-oriented languages, it is also possible to specify an API that is, to a greater or lesser extent, similar to MPI, e.g. both mpiJava [26] and MPJ [27] define a Java API that is similar to MPI but does not conform to the C++ bindings.

The intent of MPI is to provide a high-performance communication layer so some implementations, whilst useful as a proof-of-concept or for language research, are not used in real codes because of the performance limitations of the language or, more accurately, its runtime environment or static compiler. For example, MPI Ruby [28] uses an interpreter at runtime, which limits the possibilities for optimisation and so even serial code performs poorly.

Of the popular, high-level computer languages in use today, Java stands out as the one that has been used most often to implement MPI. The serial performance of Java code can approach that of equivalent code in C and so it initially seems like a good candidate for an object-oriented, high-productivity implementation of MPI. However, no efficient version has yet been forthcoming, with the best of them demonstrating about approximately quadruple the latency of a reference implementation in a more traditional language, such as MPICH in C.

Some implementations in Java, e.g. JavaMPI [29] and mpiJava [30], delegate to an existing, non-Java, MPI library via Java Native Interface (JNI) calls, which is provided in Java as a mechanism for integrating existing or legacy code that is not written in Java with a Java program. Typically, whilst this approach produces a working library with minimal programming effort, it is also suffers from very poor performance because the Just-In-Time compiler cannot optimise the non-Java code and must assume a worst-case for the operations that external code will perform. Some Java Virtual Machines actually revert to

an interpreted mode for the sections of code that contain JNI calls. This prevents almost all optimisation, including simple tricks such as in-lining and statement re-ordering.

Another serious performance problem arises from the different in-memory representations employed by Java and the non-Java code. This requires that the data be converted and packed into buffers for the duration of each external call and then unpacked and converted back into the Java layout when the external call completes. The impact of this second issue is illustrated by the ping-pong times reported for CCJ [31] versus mpiJava, reproduced as Figure 2. Both for CCJ and mpiJava, the times for the 2-D array are consistently greater than for the 1-D array due to the overhead introduced by differences in memory layout.

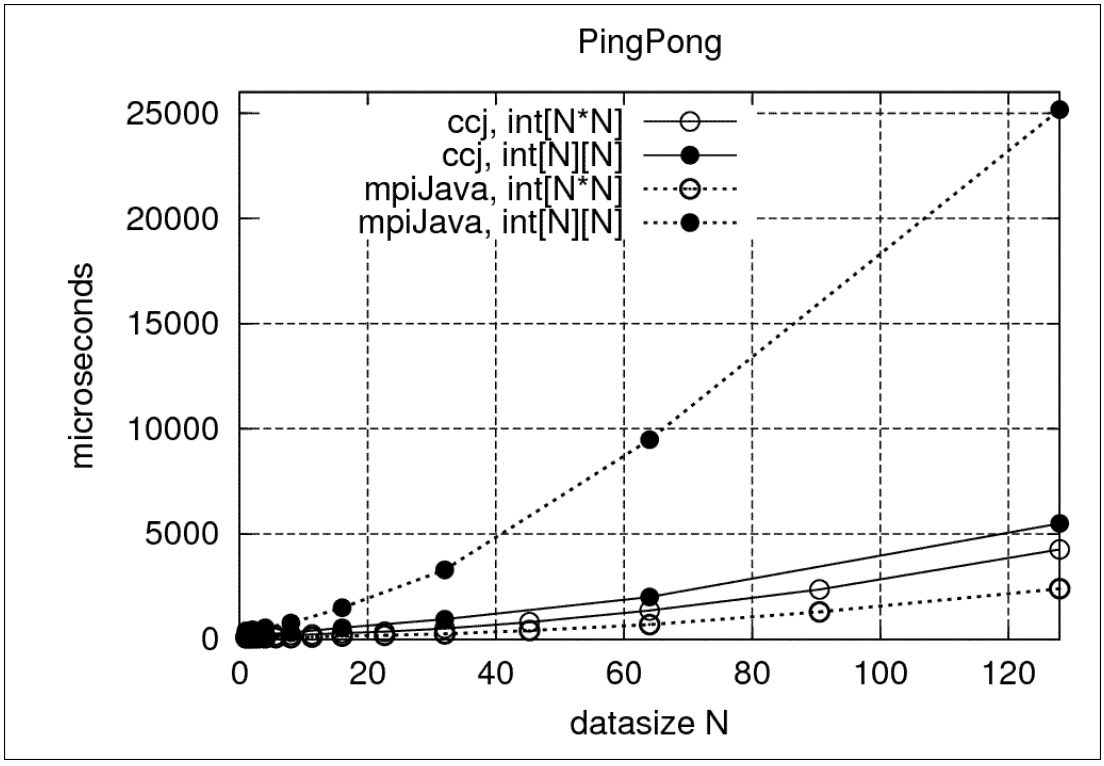


**Figure 2: Ping-pong times for CCJ and mpiJava [31].**

Some pure Java versions follow the MPI standard closely, e.g. JMPI [32] and jmpi [33] whilst others deviate from it in order to provide more object-oriented functionality, e.g. JOPI [34] and CCJ [31]. Although Morin, et al [32] suggest that using the Java implementation of Berkeley sockets would give much better performance results, all of these libraries actually use Remote Method Invocation (RMI) to perform inter-process communication because it is much easier to achieve correct code whilst avoiding low-level complexity. This introduces two major overheads: the first is the serialization and deserialization of the Java objects to

and from the byte-stream that is actually communicated and the second is the RMI software layer itself. These overheads have been tackled by replacing the Sun JDK implementation of RMI with various alternatives, such as KaRMI and Manta RMI, but this does not entirely solve the underlying problems inherent in treating all data as objects and in the memory layout in Java, in particular multi-dimensional arrays being arrays of arrays.

More recent implementations of MPI in Java, e.g. MPJ/Ibis [35], MPJ Express [36] and F-MPJ [37], achieve much better performance (collated and compared by Guillermo, et al [38]) by replacing RMI with a faster communication solution, such as Java sockets (TCP/IP), Java NIO and Java Fast Sockets [39].

## 1.6 MPI in C#

At the time of writing, there have been three attempts to produce an MPI library in C#: two versions of MPI.NET [40] [41], which is a wrapper of an existing MPI library (first LAM/MPI and then MS-MPI [42]), and one called McMPI [24], which is written (by the author of this present work, as an MSc project) in pure C# using .Net Remoting (basically RMI/RPC by another name). Both the first version of MPI.NET [40] and McMPI show poor performance relative to MPI in C but for different reasons. The second version of MPI.NET achieves comparable performance to an external MPI library written in C but retains the disadvantage that it depends on the external MPI library and it also involves complex programming techniques, such as runtime code-generation, that potentially offer some avenues for improvement.

Both versions of MPI.NET use the Platform Invoke (P/Invoke) method, called Java Native Interface (JNI) in Java, to access code in an external library. The first version recognises and addresses the problems that this introduces but does not entirely solve them. When calling 'unmanaged' code from 'managed' code, the external code must be given access to data in memory that is controlled by the Common Language Infrastructure (CLI) garbage-collector. The garbage-collector is normally free to relocate the data to a different physical memory location at any time, which the external code could not anticipate or cope with, causing unpredictable behaviour. To prevent this, the memory must be “pinned” to a fixed location during the external call. This operation was investigated in the development of the first version of MPI.NET but was found to result in an unacceptably large overhead for small messages, although the impact was reduced for larger message sizes. Another possible

source of performance loss is the choice of the Rotor environment and the Shared-Source-CLI (SSCLI). Although free and suitable for installation on a Linux operating system, this CLI is not commercially important to its vendor (Microsoft) and may not be as fully optimised (especially on non-Windows operating systems) as their .Net Framework product which is shipped with Windows operating systems. The performance tests were comparisons between a C application linked to an MPI library written in C running on FreeBSD and a C# application linked to a C# wrapper layer that calls into the same MPI library. This means that any performance issues with the CLI itself would be included in the overheads measured and attributed to the wrapper code. This does not appear to have been noticed or investigated.

The second version of MPI.NET [41] makes use of advanced C# and CLI language features to remove almost completely the overhead seen in the first version. Note, however, that the testing was done on a Windows cluster using the .Net Framework, which may have reduced the expense of the P/Invoke calls and of other operations such as pinning memory. A different MPI library was used but as this was the same for both the C application and the C# application, this should not have affected the relative performance, i.e. the measured overhead of the C# wrapper. Although the C# code should be portable to other operating systems (using an appropriate CLI for that operating system), this does not appear to have been done so no direct comparison between the two versions, e.g. to determine the exact benefit of the code changes, is possible. In addition, one of the advantages of the .Net runtime, in common with the Java Virtual Machine (JVM), is partially lost. Pure C# code is portable to any CLI that obeys the applicable ECMA and ISO standards [43] [44], in the same way as Java code is portable to any JVM. This write-once-run-anywhere ability is compromised by the reliance on external libraries. For novel system architectures (taken here to mean a combination of the hardware and the operating system), the C# code requires only that a new CLI be created but the MPI.NET code additionally requires that the MPI library code be ported to, and optimised for, the new system.

The best way to regain the portability of MPI in C# is to remove the dependence on external library code, i.e. to develop a pure-C# communication library. There are two main methods for achieving this: .Net Remoting and .Net Sockets (an implementation of Berkeley sockets). Whilst .Net Remoting is certainly the quicker and simpler option in terms of coding the library itself, it suffers from very poor runtime performance because of the .Net

Remoting infrastructure, which cannot make many assumptions about the target of each remote method call and must maintain compatibility and security in order to comply with the strict rules for managed code. In particular, the stack-walk checking for security permissions and the use of full-object serialisation could be avoided in many instances for HPC codes but are done regardless by Remoting. Initial investigation work, by the author, into this topic resulted in the first version of McMPI, which shows that a fully managed .Net MPI library is possible and quantifies the performance problems associated with object serialisation and .Net Remoting. However, the initial work on McMPI also indicated that using .Net Sockets has the potential to compete with sockets in C, which is the underlying communication technology used by well-optimised MPI libraries, such as MPICH [45].

This work studies high-performance message-passing communication in a modern high-productivity language by producing a new version of McMPI, written in pure C# and using .Net Sockets to implement point-to-point communication, as defined by the MPI Standard.

# Chapter 2

# The MPI Standard

The requirements for an MPI library are specified by the MPI Standard; version 2.2 is the most recent to be published, although work towards version 3 is currently underway. This section loosely follows the structure of that document, describing the parts of the MPI Standard that are necessary for minimal compliance but also recommending modifications and extensions – suggested by, and evaluated as part of, this research – that facilitate the process of implementing MPI in C#, which is not covered by the standard.

Chapter 1 of the MPI Standard introduces the Message-Passing Interface in terms of motivation, goals, historical context, versions and intended applicability; this information is summarised in section 2.1.

Chapter 2 of the MPI Standard defines various terms and conventions that form a significant part of the functional specification; these terms, and their implications for a C# MPI library, are explored in section 2.2.

Chapter 3 of the MPI Standard specifies the requirements for point-to-point communication, which is necessary for minimal compliance with the standard; these requirements are examined in section 2.3.

Although a significant part of the MPI Standard falls outside the scope of this research, some consideration of the requirements for full compliance with the standard provides helpful guidance when implementing the functionality needed for partial compliance in a way that assists future work. Consequently, various topics from the rest of the MPI Standard are briefly discussed section 2.4.

## 2.1 Introduction to MPI

Message-passing is a parallel programming model that enables communication between processes by copying data between their memory spaces using cooperative operations in each process. Data in the memory address space of one process (such as the result of a calculation, a progress update or a request for more work) is packaged into a message by

that process and sent to a second process. The second process receives the message into memory allocated within its own address space and unpacks it to obtain the data. If a response is required then another message is sent with the roles of the two processes exchanged, i.e. the second packs and sends whilst the first receives and unpacks. This communication paradigm is widely portable because it does not rely on any particular hardware or software and is tolerant of low-speed connections. An implementation can be written for shared-memory machines, distributed-memory machines or hybrid memory architectures such as a network of multi-core workstations and can efficiently use any available communication mechanism. It can support programs written in the fully general MIMD (Multiple Instruction Multiple Data) style as well as the more common SPMD (Single Program Multiple Data) style and is language independent.

The MPI Standard was created at a time when multiple competing message-passing solutions existed and was intended to increase code portability and ease-of-use by harmonising the interface to these disparate message-passing libraries. This involved not only standardising the syntax of method calls but also the semantics of each method call. All the message-passing libraries supported the same basic functionality but there were significant differences in the precise details, such as the actual timing of data movement relative to the send and receive method calls. In many cases these differences allowed the library to be optimised in a way not available to other libraries. One of the goals of MPI was to provide a standardised API with semantics that do not change on different hardware or in different computer languages. A guarantee of thread-safety was included in this goal, which entitles a user to expect identical and predictable behaviour from all compliant MPI libraries.

Most of the other stated goals – efficiency, reliability, convenience and support for a heterogeneous environment – are also user-oriented, with the remaining goal (that the standard should be implementable without changes to the communication or system software) being aimed at both users and implementers. Efficiency includes avoiding unnecessary operations such as memory-to-memory copying and allowing communication to be overlapped with computation or offloaded from the main processor into dedicated hardware when possible. Reliability means that failures are dealt with by the underlying communication system, so missing or garbled messages are automatically resent, duplicate messages are silently ignored and any correct MPI program is guaranteed to complete.

Convenience refers to the existence of appropriate bindings for each target language. Currently the target languages are C, FORTRAN and C++, although the specific bindings for C++ have now been deprecated in favour of the C bindings because these can be used in C++ as well. The aim of this research is to evaluate whether it is possible for C# to become a fully-fledged target language for MPI, on a par with C and FORTRAN. As neither the C nor the FORTRAN bindings are appropriate for C#, the goal of convenience requires the definition of a new set of bindings, discussed in section 2.2.2. Heterogeneous support requires that processes on different hardware platforms can interoperate successfully without changes to the MPI program, although this support is rarely called upon in practice. In part this is because it is likely to have a negative impact on the performance of the MPI library. A superficial difference such as processer clock speed can introduce load-balance issues. More fundamental differences involve extra work in the library, such as reordering bytes during packing and unpacking for big-endian to little-endian conversion. Implementing an MPI library without changing the communication or system software should be a less daunting task for the implementer and should lead to a library that is easier to install and maintain. However it may be possible, with detailed expert knowledge of the target system, to achieve better library performance if such changes are allowed. The MPI standard is specified so that both approaches are possible and acceptable routes to compliance.

The current version of the MPI Standard (MPI-2.2) includes all previous versions and refers to two other documents that will be the basis of future versions. All these documents are available from the MPI Forum [46]. MPI-1.0, the first version of the MPI Standard, embodies the main features needed in a message-passing library. MPI-1.1 and MPI-1.2 are addenda to MPI-1.0 and consist of clarifications and corrections of the original MPI-1.0 document. MPI-1.3 refers to MPI-1.2 after it was combined (largely unchanged) with MPI-2.0. MPI-2.0, the second major version of the MPI Standard, embodies the whole of MPI-1 plus new functionality and bindings for C++. MPI-2.1 and MPI-2.2 are addenda to MPI-2.0 and consist of clarifications and corrections to the MPI-2.0 document. MPI-3.0 is a work-in-progress towards a new version of MPI, which has not yet been released. The Journal of Development (MPI-JOD) is not part of the MPI Standard, although some of the information it contains is very useful to library implementers.

## 2.2 MPI Terms and Conventions

### 2.2.1 Semantic Terms

The MPI Standard uses certain semantic terms in a precise way that defines behavioural characteristics of methods. The exact definitions may be found in chapter 2.4 of the MPI Standard but some comments are included here for convenience.

- *Blocking* – a procedure is blocking if the resources used during the call may be reused immediate after the procedure returns control. This is the normal expectation in procedural computer languages unless the procedure is intended to initiate an asynchronous operation.
- *Non-blocking* – a procedure is non-blocking if it may return before the resources supplied to the procedure may be safely reused, i.e. if the operation may not have completed. The operation is started by the non-blocking call and, when control is returned, may continue in parallel or remain paused until a future call to an MPI procedure. The resources may only be safely reused once the operation completes, which must be explicitly determined by a further MPI procedure, e.g. a call to `MPI_WAIT` that returns or a call to `MPI_TEST` that returns `flag = true`.
- *Local* – a procedure is local if its completion only depends on the local executing process, i.e. no MPI calls in other processes are needed. A local procedure may be part of an operation that is non-local, for example a blocking buffered send is local (it only depends on there being sufficient buffer space available to the local process) but the communication operation it is part of is non-local because it only completes when the other process receives the data.
- *Non-local* – a procedure is non-local if its completion may depend on one or more MPI calls in other processes. Note that a non-blocking `MPI_ISSEND` will return control without depending on other processes but will not complete without another process receiving the data and so it is non-local. An ambiguity arises when the MPI Standard allows either a local or non-local implementation of a procedure, so that a user cannot determine by examining the procedure and its arguments or by any other means, whether a particular call will depend on other processes or not. In this case, the procedure may depend on other processes and is defined to be non-local.

- *Collective* – a procedure is collective if all processes in a process group must call the procedure. A collective procedure can be synchronising (if all processes in the group must start their procedure call before any in the group can return control, e.g. `MPI_ALLREDUCE`) or non-synchronising (if some processes may return control before all others in the group have started their procedure call, e.g. `MPI_BCAST`).

### 2.2.2 Language Binding

In the latest version of the MPI Standard, the language bindings for C++ have been deprecated in favour of the bindings for C. This does not involve loss of functionality because C syntax is a subset of C++ and so all bindings for C can be used in C++. Although C# can be described as being in the same family of languages as C and C++, it is not a superset of either and so the language bindings specified in the MPI Standard cannot be used in C#. The best starting point for a definition of new language bindings for C# is the deprecated C++ bindings because they naturally fit the object-oriented paradigm. In addition, the bindings for C and FORTRAN in MPI frequently require memory addresses as function arguments, whereas the bindings for C++ specify the use of object references in some of these cases. Explicit memory addresses are not allowed in C# unless the code is marked as "unsafe" but object references achieve the desired functionality whilst preserving type-safety and allowing garbage collection to proceed normally. Section 2.4.3 explores this subject in more depth. By convention, this document uses the language-independent definitions from the MPI Standard when discussing MPI functionality and the proposed new C# language bindings (refer to section 3.3.5 for the C# code for these bindings) when discussing the McMPI implementation.

### 2.2.3 Processes

Processes in MPI are a logical demarcation of program units only and do not necessarily correlate with operating system processes or physical processing units. Each MPI process executes a MIMD style code and communicates with other MPI processes via MPI communication procedure calls; they may be single-threaded or multi-threaded and these threads are not tied to a single physical core in a multi-core processor or even to a single processor in a multi-processor machine. Each MPI process is identified by a unique rank within the global communicator, `MPI_COMM_WORLD`. The exception is the null process,

identified by the constant MPI_PROC_NULL, which represents a dummy process in that communication with the null process always succeeds but has no effect.

It is common practice for an implementation of MPI to require a one-to-one mapping between MPI processes (apart from the null process) and operating system processes. As specified in chapter 12.4 of the MPI Standard, a thread-compliant MPI implementation allows a user to create multi-threaded processes and guarantee thread-safe operation of MPI procedures but prevents the threads being individually addressable by rank within MPI. When an implementation provides a unique rank for each thread within each operating system process, there is a one-to-one mapping between MPI processes and operating system threads; in this case, each MPI process is single-threaded. This statement, that an operating system process can contain multiple single-threaded MPI processes, introduces an ambiguity into the MPI Standard, which otherwise reads as though a one-to-one mapping is assumed.

This ambiguity is most apparent during initialisation: every MPI process must call MPI_INIT (or MPI_INIT_THREAD) exactly once. Any thread that makes this call thereby becomes the main thread of an MPI process. A second or subsequent call to MPI_INIT (or MPI_INIT_THREAD) within a single MPI process is defined by the MPI Standard to be erroneous. The assertion that multiple MPI processes can exist within a single operating system process implies that subsequent calls to MPI_INIT (or MPI_INIT_THREAD) must initiate new MPI processes, which is not discussed or specified at the point these functions are defined or, indeed, anywhere else in the MPI Standard.

The statement that these multiple MPI processes within a single operating system process would be single-threaded seems to contradict the definition of the single-threaded thread support level. A call to MPI_INIT_THREAD that specifies MPI_THREAD_SINGLE as the desired level of threading support informs MPI that only one thread will execute. This is intended to allow optimisations such as avoiding the need for locks on static data-structures and allowing the use of library functions that are not thread-safe, e.g. malloc for dynamic memory allocation. If multiple single-threaded MPI processes are initiated by multiple threads then the assumption that only one thread will execute is false for all of them. The optimisations within MPI that are not thread-safe may cause race-conditions or memory corruption.

Irrespective of the allocation of ranks, these MPI processes require a thread support level of at least MPI_THREAD_MULTIPLE because each of them is likely to make calls to MPI functions independently of the others (so cannot guarantee to comply with MPI_THREAD_FUNNELLED) and simultaneously (so cannot to guarantee comply with MPI_THREAD_SERIALIZED).

Care must be taken when calling MPI procedures in multi-threaded situations to avoid them being ambiguous or erroneous. For example, the MPI Standard specifies that any thread within a given MPI process can receive a message sent to that MPI process. If only one MPI process exists for each operating system process then no ambiguity exists (each thread acts as part of the MPI process of its operating system process) although thread synchronisation may be needed to prevent race conditions. For any thread that has successfully called MPI_INIT (or MPI_INIT_THREAD), again there is no ambiguity. However, in an operating system process that contains more than one initialised thread – and therefore more than one MPI process – the results of calling MPI procedures on a thread that has not been individually initialised are not specified by the MPI Standard and so are implementation-dependent. This ambiguity can be avoided in one of three ways, by:

- allowing only one initialised thread in a given operating system process, i.e. only one thread may call MPI_INIT (or MPI_INIT_THREAD); this is the normal approach
- ensuring that all threads in a given operating system process are initialised with an individual call to MPI_INIT (or MPI_INIT_THREAD); the approach used in this work
- requiring that MPI procedures are only called on threads that have successfully called MPI_INIT (or MPI_INIT_THREAD); supported by the library created in this work

The second and third of these approaches are not defined in the MPI Standard but could form the basis for possible extensions to the existing four thread support levels. Ensuring that all threads are initialised and can be addressed by individual ranks suggests a new thread support level named MPI_THREAD_AS_RANK, which is an extension of the existing MPI_THREAD_SINGLE level for when there are multiple initialised threads in an operating system process. Requiring that MPI procedures are only called on initialised threads suggests a new thread support level named MPI_THREAD_FILTERED, which is an extension of the existing MPI_THREAD_FUNNELLED level for when there are multiple initialised threads in an operating system process. These new thread support levels require

a superset of the requirements of MPI_THREAD_MULTIPLE and so they would be listed fifth and sixth, in increasing order of thread support.

Any implementation of MPI that is capable of supporting MPI_THREAD_MULITPLE and is additionally able to support MPI_THREAD_AS_RANK should also be capable of supporting MPI_THREAD_FILTERED. The difference between the two new levels is the existence of threads that are not individually initialised and will never directly use MPI functionality, which must be dealt with as part of support for MPI_THREAD_MULTIPLE.

For MPI_THREAD_FUNNELLED, the MPI Standard states that MPI functionality must only be used by the main thread (the thread that called MPI_INIT). Use of MPI functionality by other threads is defined to be erroneous so that MPI does not have to handle multiple MPI function calls simultaneously and may be able to retain some of the optimisations that are possible for the MPI_THREAD_SINGLE level. However, support for the new MPI_THREAD_FILTERED level requires MPI to handle multiple MPI functions calls simultaneously. Therefore, it is proposed that the new MPI_THREAD_FILTERED level define calls to MPI functions by threads that have not individually called MPI_INIT to be legal and to complete as though issued by one of the individually initialised threads in the operating system process. When there are multiple initialised threads in the operating system process, this definition is ambiguous and implementation-dependent. In these circumstances, the ambiguity should be avoided by filtering all calls to MPI via threads that have been initialised with MPI_INIT. However, when there is only one initialised thread in the operating system process, this definition requires that MPI_THREAD_FILTERED is identical to MPI_THREAD_MULTIPLE. The function MPI_INITIALISED is defined by the MPI Standard to return true if MPI_INIT has been called by any thread in the operating system process. The function MPI_IS_THREAD_MAIN is defined by the MPI Standard to return true if the calling thread has called MPI_INIT. Together these two MPI functions can be used to determine the status of the thread with respect to MPI.

Any implementation of MPI that is capable of supporting MPI_THREAD_FILTERED as defined above would be able to support any of the other five levels because they require only a subset of the functionality. For example, it should be possible to configure such an implementation of MPI so that only one rank is allocated to each operating system process, thus restricting the maximum possible level of thread support to the four existing levels as

defined in the MPI Standard. Existing parallel codes that use MPI would therefore be portable to such an implementation of MPI, although they may not be able to take advantage of the new features offered by these suggested extensions to the MPI Standard.

Codes that comply with this definition of the `MPI_THREAD_FILTERED` level should automatically also comply with the `MPI_THREAD_FUNNELLED` level and, therefore, should be portable to most implementations of MPI.

In this work, the possibility of an operating system process containing multiple MPI processes, each initialised by a different operating system thread and having its own rank, i.e. `MPI_THREAD_AS_RANK` and therefore `MPI_THREAD_FILTERED`, is investigated as an extension to the MPI Standard, which is specified in more detail in section 3.2.7.

## 2.3 Point-to-Point Communication

In MPI, point-to-point communication is two-sided: one side must call a procedure to send a message and the other side must call a different procedure to receive that message. The communication is only complete once both operations are complete. When either the send procedure or the receive procedure (or both) are non-blocking, further procedure calls will be necessary to complete the operation. When calling a send procedure, the intended target MPI process is specified as part of the envelope information; this target is the only MPI process that can receive the message. When calling a receive procedure, envelope information is also required and is used to select the correct message; the source MPI process may be specified explicitly or a wildcard value, which matches any source, may be supplied instead. There are different communication modes, which are variations on this theme. The full specification of point-to-point communication forms chapter 3 of the MPI Standard but the relevant information is detailed in sections 2.3.1 to 2.3.5.

### 2.3.1 Envelopes and Matching

An envelope consists of a communicator, a source and destination, and a tag. Communicators are described briefly in chapter 3, and explained fully in chapter 6, of the MPI Standard. The important characteristics from the point of view of the present work are that communicators provide discrete communication contexts – within which messages can be sent and received without interference from messages in other communicators – and that each communicator operates for an ordered group of MPI processes. Note that the

process group for each communicator need not contain all MPI processes. Each MPI process is identified by its rank within the process group for a given communicator; an MPI process may have no rank or a different rank in each communicator. The MPI Standard requires that a default communicator, called `MPI_COMM_WORLD`, which contains all accessible MPI processes, is provided.

The communicator field in the envelope stores the globally unique identifier for the communicator performing the communication operation. The ranks of the MPI processes within the specified communicator are stored in the source and destination fields of the envelope. The tag is an additional integer field that may be used to distinguish messages from each other; it is often used to indicate the purpose of the message. Special wildcard values are defined for source and tag, which match any source value and tag value, respectively. For a send operation all of the envelope fields must contain actual values in the appropriate valid range; the wildcard values are not allowed. For a receive operation, both the source and tag fields are allowed to contain the applicable wildcard value. The values in envelopes are used to match receive operations to send operations. For an incoming message with an attached send envelope to match a receive operation envelope, the two communicators and two destinations must exactly match, the source in the receive envelope must be the source wildcard value or must exactly match the source in the send envelope, and the two tags must exactly match unless one is the tag wildcard value. The values of the source and tag fields from the message envelope can be retrieved from the status that is returned by the receive operation. The status also contains the length of the message, which indicates how many items of the data type specified in the receive operation were received; the actual number of received items will be less than or equal to the maximum number given by the count argument of the receive operation.

No data type information is necessary in the envelope because both the send and receive operations specify the data type for the message. Data type matching is very strict: the program is defined to be erroneous and its behaviour is undefined if one data type is sent and a different type is received. (Note that derived data types are not considered to be different if they consist of the same data types in the same order, e.g. an array of three integer values is considered to be the same as a structure containing three integer fields.) This makes type conversion unnecessary but many implementations exceed the MPI Standard and provide some automatic type conversion functionality. Note that the MPI

Standard does require inter-operation in heterogeneous environments, including across different languages and involving different hardware. This means that different MPI processes may represent and store the same data type in different formats. The encoding on-the-wire, i.e. in transit between MPI processes, is entirely implementation dependent. Where representation conversion to and from the encoding on the wire is needed for correct inter-operation, it is mandatory.

### 2.3.2 Communication Modes

The MPI Standard defines four communication modes for point-to-point send operations: synchronous, ready, standard and buffered.

The synchronous communication mode is so called because its semantics force the MPI processes involved in the communication to synchronise their execution. In implementation, this is also referred to as the rendezvous protocol because neither MPI process can proceed with its communication operation until the other MPI process starts the matching operation; the first to arrive must wait until the second reaches the rendezvous point. Note that, even while the communication operation is blocked in this manner, the MPI process will not be blocked if a non-blocking procedure call (see section 2.3.3) is used. The protocol suggested in the MPI Standard is for the sender to transfer a notification or request-to-send to the receiver, wait for an acknowledgement or permission-to-send from the receiver (this indicates that a matching receive operation has started), and then transfer message. If the receiver arrives first, i.e. a receive operation is started and there is no notification of a matching send operation, then it simply waits until a matching request-to-send arrives. The synchronous send operation is non-local because it requires a matching receive operation to have started before it can complete. Completion indicates that all data has been read and the user is free to reuse the memory location supplied to the send procedure. In a synchronous send, the message data is only read and transferred once the receiver indicates that a matching receive operation has started; at this point the message data is copied directly between the memory locations specified by the user in the send and receive procedure calls.

When ready mode is used for a send operation, the sender is declaring that a matching receive operation has already been started and so the message can be transferred as soon as possible. In implementation, this is referred to as the eager protocol. This assertion

makes ready send a local operation because it removes the dependency on the receive operation; no work is necessary at the sender MPI process to verify the existence or status of a matching receive operation. In some implementations this has a performance benefit because some of the hand-shake and protocol communication traffic is superfluous and can be avoided. However, if the receive operation has not reached the correct stage when required by the ready send operation, then the program is defined to be erroneous and its behaviour is undefined. In all other respects, the semantics of ready send are identical to synchronous send and, indeed, the MPI Standard allows ready send to be implemented in exactly the same way as synchronous send, i.e. the sender ignores the extra implied information, notifies the receiver via a request-to-send and waits for a permission-to-send acknowledgement. Some implementations revert to this behaviour if the receive operation is not started rather than failing to deliver the message or generating an error.

The buffered send mode wholly decouples the send operation from the receive operation; the message data is copied from the memory location specified by the user into a system controlled buffer space, which immediately allows the user to reuse the memory location supplied to the buffered send operation. The message data is transferred to the receiver from the system controlled buffer space when the receive operation has been started; this is usually achieved via a non-blocking send operation using one of the other send modes (most commonly, the standard mode). The system controlled buffer space must be allocated by the user and provided to the MPI library before the buffered mode send procedure is called. An error will be generated if insufficient buffer space has been provided. As all non-blocking operations are local (see section 2.3.3) and copying to a local buffer does not depend on other MPI processes, the buffered mode send is also a local operation. Buffered mode may exhibit reduced performance compared with ready mode due to extra memory-copies but replaces the latter's requirement for the receive operation to have already been started with the more predictable and controllable requirement for sufficient buffer space. Similarly, buffered mode performance may suffer relative to synchronous mode if there is only a short wait for the receive operation but, when the receive operation is started much later than the send operation, buffered mode allows the sender MPI process to continue execution – including reuse of the memory that stored the message data – without a long wait to synchronise.

The standard send mode allows the MPI library to choose the most appropriate send mode for each send operation in an implementation-dependent manner. Although it is legal in the MPI Standard to implement standard mode in exactly the same manner as synchronous mode, it is typically a hybrid of the other modes. For example, an MPI library may choose to allocate some buffer space itself and use buffered mode for small messages but use synchronous mode for large messages in an attempt to avoid running out of buffer space. If too many small messages are required to be buffered, it may choose to switch to synchronous mode for all subsequent messages until buffer space is freed by the completion of previous send operations. Alternatively, an MPI library may choose to use ready mode for small messages and, if no receive operation has been started, to buffer the data in the receiving MPI process. In this case, synchronous mode may be used for large messages to avoid over-running the buffer space available at the receiver. The ambiguity inherent in allowing the MPI library to choose the implementation means that standard mode send is a non-local operation because it may require a receive operation to have started in another MPI process.

### 2.3.3 Non-blocking Operations

The purpose of most MPI programs is not solely to communicate but to compute; MPI facilitates parallel computation by allowing MPI processes to cooperate, which requires communication. All of the communication modes available to MPI (described in section 2.3.2) restrict the freedom of participating MPI processes to compute by imposing constraints on their execution and resource usage. Each of the modes is available as a blocking procedure call; the resources supplied to the procedure are no longer needed by the operation when it returns control, although the communication is not necessarily complete. There are many situations where this will block the MPI process for considerable time during which it could be doing useful computation work instead. Multi-threaded MPI processes are one way to solve this: the main thread would call MPI procedures and block whilst execution would continue on one or more other threads. However, this significantly complicates the programming model, not least by introducing thread-synchronisation issues. Worker threads share the same memory address space as the main thread and are restricted by the same resource usage constraints; the memory locations supplied to a communication procedure cannot be accessed by a worker thread until the main thread is no longer blocked by that procedure, which requires inter-thread communication, e.g.

locking, signalling or messaging. The MPI Standard defines non-blocking procedures for all communication modes in order to allow MPI processes to continue execution without complicating the user programming model with threads.

Non-blocking operations in MPI split the initiation and completion of a task into two procedure calls rather than, as for blocking operations, having one procedure that initiates the task and notifies of completion by returning control. A non-blocking procedure call returns control as soon as the task is initiated and provides a handle for the task, called a request, which is used in other procedure calls to check the progress of the task or to wait for its completion. The MPI library may implement this functionality using threads managed by the library or by making progress with all outstanding tasks whenever control is passed to the library in subsequent procedure calls. Calling a non-blocking procedure and immediately passing the request it returns to a wait procedure is semantically identical to calling the equivalent blocking procedure.

Each of the communication modes may benefit from potential optimisations depending on the capability of the hardware and on the precise timing of events during execution. In particular, the synchronous send cannot complete until a matching receive operation has been started so a blocking procedure must block the calling MPI process whereas a non-blocking procedure allows the sender to proceed ahead of the receiving MPI process. In addition, a non-blocking buffered send is only required to buffer data when no matching receive-operation has been started by the time the send-complete procedure is called; this may avoid the overhead of memory-to-memory copying. All modes may benefit if the hardware is capable of concurrently performing computation and data-transfer, either memory-to-memory copying or network communication. The non-blocking receive procedure is likely to result in lower overheads because it can be initiated as soon as the receiving MPI process anticipates that a message will be sent. Having a receive operation already started allows the use of the ready send operation, which may be implemented to send message data immediately without a network hand-shake or verification of the existence of the receive operation. For synchronous mode, the permission-to-send acknowledgement can be issued to the sender MPI process as soon as the request-to-send notification arrives, which minimises the delay before transfer of the message data can begin. For buffered mode, the message data may not need to be buffered at all because it can be transferred directly to the receiving MPI process, which removes the overhead

associated with memory-to-memory copying. The standard mode, being a combination of other modes, may be able to benefit from some or all of these optimisations.

A completion procedure is necessary for all non-blocking initiation procedures. This requirement simplifies implementation of the MPI library because any resources allocated as part of the operation, specifically the request, can be de-allocated during the completion procedure. The MPI Standard defines several procedures that can complete non-blocking operations; the user may wait for completion, which blocks execution, or test for completion, which does not block. Furthermore the user may test or wait for a single request or for any, some or all of a list of requests. It is also possible to test for completion of a single request without de-allocating the resources associated with it so that the status information may be queried several times by several application layers. A procedure for de-allocating a request without testing or waiting for completion is also defined for situations when the status information is not needed and the timing of completion of the operation can be determined in some other way.

### 2.3.4 Communication Semantics

The MPI Standard stipulates four general principles for point-to-point communication that must be guaranteed by any valid MPI implementation: ordering, progression, fairness and limitation of resource usage.

Communication operations in MPI are guaranteed to be ordered insofar as they must not overtake each other. Messages sent by a single threaded MPI process to the same destination, which could be matched with particular receive operation in any order, must in fact be matched by receiving MPI process in the order that they were started, i.e. in the order the blocking or non-blocking send procedure calls were made in the sending MPI process. The timing of the send-complete procedure calls in the sending MPI process is not relevant to this ordering requirement. Messages sent by different threads are logically concurrent and so no ordering requirement exists. Similarly, receive operations in a single threaded MPI process, which specify the same source, must be satisfied in the order that they were started, i.e. in the order the blocking or non-blocking receive procedure calls were made. Again, the timing of the receive-complete procedure calls at the receiving MPI process is not relevant to the ordering requirement; also receive operations from different threads are logically concurrent and so are not required to be ordered. This ordering

requirement does not apply to messages or receive operations that specify disjoint matching criteria; if a receive operation does not match the first message that arrives, but does match the second, then the second message will be received before the first message. Similarly, if a sent message does not match the first receive operation, but does match the second, then the second receive operation will be completed before the first.

Communication operations in MPI are guaranteed to progress insofar as they must eventually complete once a matching operation has been started. If a send and a receive operation, which could match each other, are successfully started in the source and destination MPI processes respectively then at least one of them must complete; either the send is matched to a prior receive operation and completes, or the receive is matched to a prior send operation and completes, or they are both matched (to prior operations or to each other) and both complete. Non-blocking operations are deemed to have started successfully if the initiation procedure call returns without generating an error. The progress guarantee is not dependent on the completion procedure for the matched operation: the completion of each side of a communication only requires the matching operation to have started. If both sides of a communication are non-blocking and neither completion procedure is called then neither side of the communication will complete, despite the guarantee of progression.

There is no guarantee in MPI that communication operations will be fair (in fact, there is not even a definition of fairness in the MPI Standard), except for the completion procedures that test or wait for some of a list of requests. These procedures are fair insofar as they guarantee that a request for an operation that could be completed will, if repeatedly passed to one of these completion procedures, be completed. The suggestion in the MPI Standard that these procedures should complete as many pending communications as possible fulfils this definition of fairness. Apart from this exception, MPI is not fair. The MPI Standard states that it is possible and permissible to allow a send or receive operation to remain incomplete, despite multiple matching operations being started, as long as neither ordering nor progression guarantees are violated, i.e. the potentially matching operations are instead matched by other send or receive operations (which maintains progress) that are logically concurrent (which maintains ordering).

There is no guarantee in MPI that communication operations will not exceed resource limitations, although quality implementations will make provision for most situations without generating errors. Pending operations – either sends or receives – will each use a small amount of resources, which should be independent of the message size. The maximum number of pending operations allowed at any one time should therefore be limited but this limit should be large enough to handle normal usage. Buffered mode sends will fail if there is insufficient buffer space available; it is the responsibility of the user to avoid these resource limitation errors. It is legal for an MPI library to provide no buffer space and to rely on the user to attach sufficient buffer space before calling buffered mode send procedures. Standard mode communication may buffer messages if space is available but will not generate errors if there is insufficient buffer space. It is possible to write a program that relies on buffer space being available for correct execution, either to avoid deadlock or to prevent resource limitation errors. However, this is deemed to be an unsafe program, which may or may not work.

### 2.3.5 Persistent Communication Requests

The non-blocking procedures (described in 2.3.3) require the MPI library to create a request for each procedure call. If many similar procedure calls are needed by the program then an optimisation may be achieved by creating the request once and recycling it for each communication. The MPI Standard permits this reuse of requests through persistence – a persistent request is created by a procedure that is identical to one of the non-blocking communication procedures (apart from the name) except that no communication occurs. The persistent request is initially inactive; it becomes active when used to start a communication operation and reverts to inactive when the communication completes, i.e. after a successful call to a test or wait procedure. A subsequent communication operation can be started whenever the request is inactive.

## 2.4 Out of Scope

This work intentionally prioritised the implementation of distributed memory communication because it is generally more difficult to achieve acceptable performance for distributed memory communication than for shared memory communication. Point-to-point communication was prioritised above single-sided communication because the chosen implementation (socket connections) requires both the sender and the receiver to

take an active part in the transfer of data and a message-passing library cannot be MPI compliant without implementing point-to-point communication but can be MPI-1 compliant without single-sided communication. In addition, some functionality that embellishes (i.e. adds local functionality that does not depend on other MPI processes) the "core" point-to-point functionality was excluded from this work. For example, buffered mode send uses one of the other send modes to transfer data (the non-local operation) but requires functions for attaching, managing and detaching buffer space (all local operations).

Chapter 3 of the MPI Standard includes some topics that are part of MPI-1 but are not within the scope of the present work:

- probing and cancelling (see section 2.4.1)
- send-receive operations (see section 2.4.2)
- predefined data types (see section 2.4.3)

Chapter 4 of the MPI Standard deals with derived data types, which pose a particular implementation problem for object-oriented languages, as discussed in section 2.4.3.

The remaining chapters of the MPI Standard relate to MPI-2 functionality and are, therefore, not within the scope of the present work. However, certain topics have influenced the present work and these are discussed in section 2.4.4.

### 2.4.1 Probing and Cancelling

There are situations when information about an incoming message, such its size, is not known in advance. The MPI Standard includes the ability to probe for the next message, i.e. to retrieve the status information (from which the message size, for example, may be determined) without receiving the message itself. The message may then be received with a normal receive operation procedure call. The blocking probe procedure waits until an unmatched message arrives and returns the status information for that message, whereas the non-blocking probe procedure returns immediately, indicates if an unmatched message was found and, if so, provides the status information for it. In a multi-threaded environment, there is no guarantee that a receive started by the same thread immediately after a probe using the same envelope information will receive the message that was discovered by the probe – a different receive operation started by another thread may intervene and receive it instead.

The MPI Standard allows pending communications to be cancelled so that the program can gracefully de-allocate resources committed to them. If the cancelation procedure call is issued too late then the communication will not be cancelled; either the cancelation or the communication succeeds, not both or parts of both. Cancellation is defined to be a local operation, which does not depend on the execution of other MPI processes, even in the case where communication is necessary to achieve the desired results. For example, if a message is transferred to a receiver before the matching operation has been started then cancelling that message involves the sender transferring a cancellation request to the receiver. This must be acted on independently of the MPI process at the receiver; either an interrupt or a thread managed by the MPI library must attempt to cancel the message and, if successful, transfer a cancellation confirmation back to the sender.

### 2.4.2 Send-Receive Operations

Two procedures that combine a send and a receive operation are included in the MPI Standard for the convenience of the user: the send-receive operations must guarantee, through buffering or some other means, to avoid cyclic deadlocks that would otherwise occur if two separate operations, a synchronous send and a receive, were used. For example, a cyclic shift operation, where all processes are logically arranged as a ring and communicate with their nearest neighbours – sending a message clockwise and receiving a message anticlockwise. If all processes send first then none completes because they are all waiting for the destination to receive the message and if all processes receive first then no messages are sent because none have arrived. If all processes instead use a send-receive operation then the communication is guaranteed to succeed.

### 2.4.3 Data Types

The data type of the content of each message in MPI must be specified in both the send and the receive procedure call in order that the MPI library knows how to copy the content from the sender to the receiver correctly. The MPI Standard mandates that MPI libraries must include a predefined data type for each fundamental built-in variable type in the target computer language and that other, more complex, data types can be built from combinations of predefined data types. In C and FORTRAN, detailed knowledge of the memory layout of the message content can be obtained and provided to the MPI library, which stores a type map and assigns a derived data type. When a message is sent (or received) the MPI library uses the type map in the MPI data type to read (or write) the

message directly from (or to) the memory locations referred to by the user code. Full information can be found in chapter 4 of the MPI Standard.

In any object-oriented language (including C#), obtaining detailed information about the data contained in an object and reading or modifying that data directly breaks the principle of encapsulation. The object-oriented design principle of encapsulation requires that, as far as possible, data should be owned by objects and kept private within those objects. Public accessors for the data (either getters or setters or both) may be provided by the object to allow controlled access to the private data. This allows the object to have some control over when data is read and, more importantly, changed.

A message passing library must, at some point, have access to the data which comprises the message. Allowing the library code to read the data directly from the memory locations referred to by the owning object breaks the encapsulation of that object because the library code must know about the internal structure and layout of the data within the owning object. Whilst a suitable representation format for this structure could be devised, the whole idea of publicising the internal structure of an object is contrary to the encapsulation principle. In particular, calculated properties (where the value of the property is not stored in the object but calculated each time the value is requested) cannot be obtained by a direct memory read.

Furthermore, to ensure that the message passing library gets a consistent state when reading the internal data, the owning object must somehow prevent the data from changing during the read operation. The timing of a direct read is controlled by the message passing library not the owning object and so it cannot be predicted or protected without some form of notification or locking mechanism. For example, the library could call a method to instruct the owning object to enter a read-only state, then perform the read and finally call a method to allow the owning object to return to its normal state. Locking the object in this way protects the direct read operation but impairs the ability of the object to respond to other threads that may call its methods or query its properties and so is undesirable. In addition, this approach is undesirable because it restricts the types of object that the message passing library can interact with to those that can be instructed to enter a read-only state, i.e. they must implement an interface that extends this functionality.

In C#, the direct read approach is further complicated by garbage collection (a background process that releases unused resources back to the operating system), which might move objects in memory to make better use of resources. The memory address of a field within such an object may change dynamically during the execution of the program in a manner which is difficult to predict. Obtaining and using memory addresses are therefore considered to be unsafe operations and require the code to be marked “unsafe” and compiled with a flag that specifies that unsafe code is allowed to execute. Such unsafe code requires that the program is given special permission to execute by the administrator of the machine on which is it to be executed.

To comply with encapsulation, internal data must be read and exposed by the owning object itself – the message passing library must call a method or query a property of the owning object. This immediately raises the question of how the message passing library determines which method or property to call for each object that contains message data.

One approach, called serialisation, is commonly used in object-oriented languages whenever the internal state of an object must be transferred from one storage location to another. In serialisation, the object implements a method that navigates its data structure and outputs the state of the object in a standardised format. For de-serialisation, the object implements a constructor method that interprets this state information and produces a new object that has the same internal state as the original serialised object. Typically remote procedure calls use serialisation for parameter objects: the input parameter objects may be serialised and transmitted to the called procedure, which de-serialises them, and the output parameter objects may be serialised and transmitted back to the caller, which de-serialises them. In addition, objects can be persisted to permanent storage using serialisation and later read from permanent storage using de-serialisation.

However, serialisation is a time-consuming operation and requires a large buffer because it collects and stores full type information about the object as well as the data values contained by the object. In addition, de-serialisation is a time-consuming operation because it interprets full type information about the object and constructs a new object from that information. In addition, de-serialisation always constructs a new object – it cannot be used to update or modify the object supplied to the receive operation. This restriction would force a change to the MPI specification of the receive operation (the user would no longer

supply a target for the message data) and the status information (an extra field or function would be needed so that the user can retrieve the received object). Furthermore, for an object to be serialised, it must implement an interface to specify the serialisation method, which would restrict the types that could be sent and received using MPI.

An alternative to a common interface would be to provide delegates or function-pointers to the MPI library for use during the send and receive operations. The timing of the call to this delegate function is still not controlled by the owning object but at least the owning object performs the actual read and write operations and so can decide what constitutes a consistent state and whether locking is necessary or not. Both de-serialisation and this alternative of calling a delegate to modify the target object, allow the possibility that the user function will throw an exception (i.e. generate an error), which the MPI library must be able to handle gracefully. Specifically, mechanisms must be put in place to prevent such exceptions from terminating internal library threads or leaving the MPI library in an unresponsive or unknown state.

Another option would be for user code to pack the data for the message before invoking the send operation and for user code to unpack the data for the message after the receive operation is complete. The message passing library could extend only the ability to send an array of bytes, or a similar buffer type, and require the user code to provide the message data in this generic form. The library code would comply with the encapsulation principle because it no longer needs to know anything about the owning object and does not control the timing of the data read operation. Each object is free to decide how to pack and unpack its data (including, but not limited to, serialisation and de-serialisation). However, extra memory copies are necessary. During the send operation, the data must be read from the source memory locations and written to a buffer by user code, then read from the buffer and sent to the target by the message passing library. During the receive operation, the data must be received from the source and written to a buffer by the message passing library, then read from the buffer and written to the target memory locations by user code. This approach is similar to the paradigm used by PVM [47] and is equivalent to sending a message with the MPI data-type `MPI_PACKED`, which is defined in the MPI Standard as a byte-buffer that has already been packed by the user code before the send operation begins and will be unpacked by user code in the destination MPI process after the receive operation is complete.

One problem with using the `MPI_PACKED` data type is type-safety. Most implementations of MPI are not strictly type-safe because no type information is sent as part of the message but the MPI Standard does mandate representation conversion, where necessary (discussed in section 2.3.1). When using the `MPI_PACKED` data type, MPI will not perform representation conversion because it does not know enough information about the content of the packed buffer. It will faithfully transfer the bytes in the buffer from the source to the destination but it is left to the user code at the sending MPI process and receiving MPI process to perform any conversions necessary.

The buffer address, count and data type parameters for communication functions and replaced in McMPI with a single object, `System.ArraySegment<byte>`, which is a standard .Net generic class representing part of an array of bytes. It consists of fields, called `Array`, `Offset` and `Count`, which provide a reference to the byte array, the offset into that array of the start of the described segment and the count of how many bytes are contained in the described segment.

### 2.4.4 Other Topics

Collective communication is defined, in chapter 5 of the MPI Standard, as communication that involves one or more groups of MPI processes. Collective communication procedures must be called by all MPI processes in the group or groups involved (as defined by the communicator used). They are all blocking and the messages they generate are independent of all point-to-point messages. Apart from the barrier operation, they may or may not be synchronising, depending on the implementation.

A field has been included in the message envelope (see section 3.2.6) that identifies the type of collective operation. This field is included for future use; currently it is always set to zero, representing “not collective”, i.e. a normal point-to-point message. Reserving this field in the message envelope and including it in the matching algorithm will simplify future development of collective operations by allowing them to be built from point-to-point messages. Setting the field to a non-zero value separates collective messages of each type from each other and from normal point-to-point message without duplicating each communicator for each type of collective operation.

Chapter 12 of the MPI Standard (External Interfaces) describes the interaction of MPI with other parts of the system. In particular, chapter 12.4 describes the interaction of MPI with threads, which section 2.2.3 discusses in detail.

# Chapter 3

# The Design of McMPI

This chapter studies the design and implementation options for a new message-passing library (to be called McMPI, an acronym for Managed-code Message-Passing Interface) written in pure C#, i.e. including only ISO standardised language features and specifically excluding calling external library functions via Remote Procedure Calls (RPC, called .Net Remoting in C#).

An object-oriented style was embraced for software design, including layered and modular design as well as the use of structured analysis diagrams, universal modelling language (UML) diagrams and common software design patterns [48] to illustrate key design concepts. Design patterns are used in software engineering, especially object-oriented programming, to solve common creational, structural or behavioural problems by using recognisable proven templates. Once a design pattern has been identified as appropriate, the template solution suggests suitable relationships and interactions between classes and objects that can be implemented with confidence. Documenting the design patterns used in an application can aid communication between programmers and designers about the code as well as making the code easier to understand and maintain. In addition, a test-driven approach was adopted for implementation, with a particular emphasis on performance tests.

The requirements in the MPI Standard can be grouped into three layers: an interface layer, a protocol layer and a communication layer. The interface layer presents an external interface to the programmer, usually called an application programming interface (API), which conforms to the MPI Standard and maps the many external function calls to the few internal operations necessary to implement them. The protocol layer implements the fundamental operations needed for message-passing according the semantics defined in the MPI Standard. The communication layer hides the complexity of having different communication methods by wrapping each of them in a communication module that conforms to an abstract device interface (an ADI) which ensures that all communication modules operate identically, even when the actual communication devices do not. This layered design is illustrated in Figure 3. The interface layer has been split into three areas of

functionality and the communication layer has been divided to show three types of communication method, or delivery mechanism.

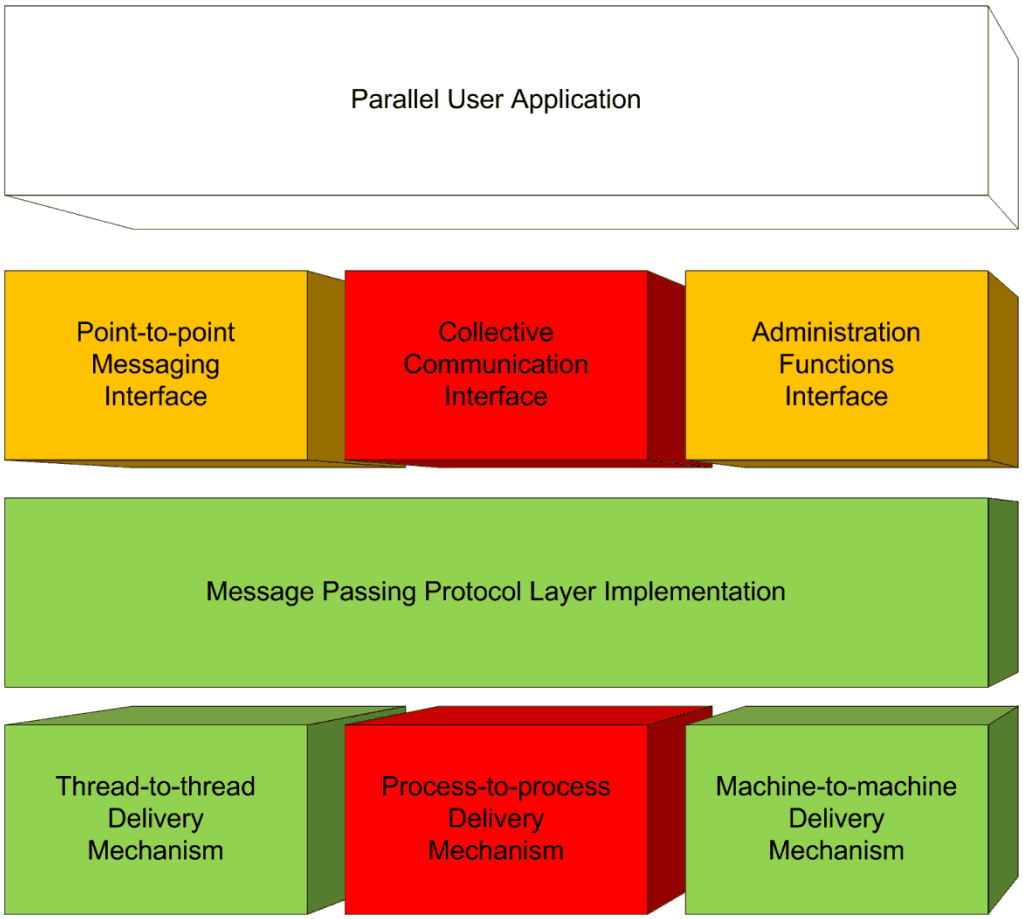


**Figure 3: schematic of a layered design for an MPI-1 library**

Message-passing is aimed at facilitating efficient co-operation between multiple processes in disparate locations. The fundamental operation is to transmit data from one location to another as fast as possible. All the other functionality within a message-passing library can be considered as an overhead that adds desirable features, or increases usability, but ultimately slows down the operation itself. From this point of view, the machine-to-machine module of the communication layer should be coded and tuned first because, if it cannot be implemented efficiently in the chosen programming environment, then any message-passing library built from it will not be fast enough to be viable. This suggests a 'bottom-up' process for creating the communication layer – first optimising the communication method itself and then adding the complexity of the abstract device interface. Once a successful communication layer is built, the next programming task

should be the protocol layer, finally followed by the interface layer. The descriptions for both of these layers follow a ‘top-down’ process.

Section 3.1 discusses the communication layer: section 3.1.1 examines the implementation of a machine-to-machine delivery mechanism using TCP socket objects and section 3.1.2 explains the design of the abstract device interface. Section 3.2.6 establishes a thread-to-thread delivery mechanism that bypasses much of the communication layer. Although a process-to-process delivery mechanism is outside the scope of the present work, section 6.2 suggests a possible approach.

Section 3.2 constructs an object-oriented design for the protocol layer: section 3.2.1 investigates four communication scenarios, 3.2.2 recommends three functional units, sections 3.2.3 and 3.2.6 specify the linkage to the communication layer, and sections 3.2.4 to 3.2.9 evaluate different implementations of several necessary components. Sections 3.2.6 and 3.2.9 contain example performance-tuning exercises, including the use of de-compilation into common intermediate language (CIL) to investigate how the output from the C# compiler can be optimised by changing coding practices.

Section 3.3 studies the interface layer: section 3.3.1 presents the overall design for the layer, including an object model that is compatible with standard .Net programming practice and sections 3.3.2 to 3.3.4 explore the implementation of the point-to-point messaging interface, with C# bindings for MPI point-to-point functions.

Section 3.4 reviews the design and implementation options selected for McMPI.

# 3.1 The Communication Layer

The requirements for the communication layer are that it unifies the various available communication methods by providing a common ADI for each one and that it makes certain guarantees, in particular the reliability of delivery, the ordering of messages and the preservation of message boundaries. Most communication methods support one or more of these delivery guarantees but very few support them all. The design of the protocol layer is much simpler if the communication layer can, by design, guarantee the appropriate delivery behaviour (this is explained in section 3.2.4). Data-types for the message content are not relevant to the communication layer because all data-types are stored in memory as bytes, so it should be possible to convert any message data into a linear array of bytes.

Conversion between an array of bytes and other data-types adds usability rather than functionality and so it should be done in the interface layer. In the MPI Standard, messages include an envelope and variable-length data.

Therefore, the minimum functionality that must be provided by each module in the communication layer is to deliver, reliably and in the correct chronological order, containers that are represented as a fixed-length one-dimensional byte array and a variable-length portion of a one-dimensional byte array.

If the communication layer must support multiple communication modules then a mechanism is needed for selecting the best communication module for each message. The communication layer will have to maintain some form of directory or routing table that stores the abilities of each communication method that is registered with it. This data-store will be used to determine which method is best for each destination, so either there must be exactly one method for each destination, with any attempt to register another method for the same destination causing an error, or some form of cost metric must be specified or calculated. In many cases, a system administrator could supply sufficient information and configure exactly one communication method for each destination. On the other hand, the developer of each communication layer module will have a good idea of the cost of the communication method it uses relative to other methods (and modules) already in existence. Another approach would be to dynamically test each method to determine its cost, either once during installation of the software or periodically according to a schedule or triggered by some event. There is a parallel here with network router hardware algorithms. The various methods of specifying a fixed mapping between communication methods and destinations are equivalent to configuring a network router with static routes, whereas dynamically testing the cost of each method to each destination is more akin to normal, adaptive network routing behaviour.

Section 3.1.1 considers the construction of a communication module that uses TCP sockets as its communication device.

Section 3.1.2 recommends an object-oriented design for an abstract device interface that encapsulates communication modules allowing the protocol layer to locate and operate the appropriate communication device whilst preventing any direct dependency between objects and classes in the protocol layer and those in the communication layer.

Section 3.1.2.3 introduces the data-store that holds both the registry of all communication modules and the mapping between them and possible message destinations.

### 3.1.1 The TCP Sockets Module

The implementation of each module in the communication layer depends on the communication method it will employ and the functionality already guaranteed by that communication method. A shared-memory communication method will be faster than one that involves a network, but to cope with distributed memory architecture machines, at least one network-based communication method must be developed and tuned. Current choices include an Ethernet network over commodity copper wires, e.g. for a cluster architecture machine, and purpose-built interconnection fabrics, such as InfiniBand and Myrinet. Whilst the hardware and device drivers for these are very different, the programming interface in C#, i.e. the `System.Net.Sockets.Socket` object, is identical. Currently, Ethernet network connections are the most common method of connecting computers together, at least for short distances, e.g. a Local Area Network (LAN). In the November 2011 top 500 list [49], Gigabit Ethernet is reported to be the interconnect family used in 224 of the top 500 supercomputers. The next most common interconnect family is Infiniband with 209 systems. The hardware for Gigabit Ethernet is ubiquitous and very cheap compared to the other options, so it is a good candidate for the first communication layer module.

The Microsoft Developer Network (MSDN) documentation for the socket object and related topics uses the terms ‘synchronous’ and ‘blocking’ in ways that are likely to cause confusion with the MPI semantic terms described in section 2.2.1. Throughout the MSDN documentation, the terms ‘synchronous’, and its converse ‘asynchronous’, refer to the behaviour of the method being executed, i.e. whether, when the method returns control, the operation is complete, or might still be incomplete, respectively. These are similar to the MPI terms ‘blocking’ and ‘non-blocking’, respectively.

In the MSDN documentation on sockets, the terms ‘blocking’ and ‘non-blocking’, refer to the behaviour of the socket itself, i.e. whether, when a task cannot be performed immediately, the socket blocks execution, or fails to complete the task, respectively. The socket may fail to complete a send-task in one of three ways: if a non-blocking socket operation is already in progress then subsequent non-blocking operations will generate an

error (an `InProgress` error); if the network is busy and there is no buffer space available then the socket will generate an error (a `WouldBlock` error); if only some of the data can be sent or buffered immediately, then some of the data will not be sent and further socket operations must be initiated to complete the task. This behaviour is different to the behaviour of MPI functions, which will queue a task if it cannot be performed immediately instead of generating an error or only performing part of the task. In addition, error-handling in C# is slow because it requires an error object to be created, which contains a lot of information and so involves a lot of processing time. Therefore, all sockets should use blocking mode because using a socket in non-blocking mode results in behaviour that is not appropriate to a high-performance communication library.

Thus, all subsequent uses of the terms 'blocking' and 'non-blocking' will refer to the MPI semantic terms. The term 'asynchronous' is not used as an MPI semantic term; its meaning will refer to the MSDN definition, i.e. a synonym for the MPI semantic term 'non-blocking'. The meaning of 'synchronous' should be clear from the context, i.e. synchronous send (one of the four modes defined in the MPI Standard for point-to-point send operations) or synchronous method (used throughout the MSDN documentation – a synonym for the MPI semantic term 'blocking').

#### *3.1.1.1 Network Protocol*

The Transmission Control Protocol (TCP) includes many features designed to cope with an unreliable network so, when the physical links are almost 100% reliable, it is not the most appropriate network protocol. In TCP, transmitted data is broken into small packets that are labelled with sequence numbers; received data is acknowledged by sending the sequence numbers back to the source. Missed packets can be identified and are re-transmitted. Duplicate packets can be identified and ignored. The speed of the link is regularly adjusted based on the rate of problem packets. All these features require processing time, so TCP is slower than simpler network protocols. However, it has become ubiquitous because it is appropriate for use in large, unreliable networks, such as the Internet, where re-transmission, flow-control and many other features of this protocol are very important. For many applications, TCP over Ethernet is not only the default choice for communication but the only option to be given serious consideration.

The main desirable network protocol feature that is missing from TCP is that of maintaining message boundaries; the low-level paradigm provided is one of a stream of bytes. The User Datagram Protocol (UDP) is a network protocol that does preserve message boundaries. Data from each separate send operation is packaged as a datagram. Each receive operation unpackages data from a single datagram. However, UDP is unreliable because it does not guarantee delivery or protect against duplicated delivery #ref_rfc768#. This may be acceptable for some applications, such as streaming video, which does not require 100% reliability because individual video frames can be lost without spoiling the experience of the viewer. However, point-to-point communication in MPI must be reliable: each send operation must be matched with an appropriate receive operation.

Another network protocol supported by the C# `Socket` object is called RAW. It is not really a network protocol at all but simply exposes the underlying transport mechanism. For example, with an Ethernet network, the RAW protocol expects valid Ethernet frame headers to be provided as part of the transmitted data. Its main purpose is to build test implementations of network protocols that are not supported by C#, including well-known protocols with publicly available specifications, such as Stream Control Transmission Protocol (SCTP), and private protocols for proprietary or unusual hardware. The network protocol used by MPICH is SCTP but this is not yet supported by the C# `Socket` object. At least one project, called SCTP.NET (#ref#), that aims to add support for this network protocol to C# is underway.

The network protocol chosen for the first communication module in McMPI is TCP over Gigabit Ethernet because it is ubiquitous, the hardware for it is cheap, and there is built-in support for it both in .Net and in most operating systems. In addition, it is anticipated that the overhead of TCP can be mitigated in HPC systems using a TCP offload engine (TOE), which processes the TCP headers using dedicated hardware in the network interface card (NIC) rather than requiring CPU processing.

#### *3.1.1.2 Communication Device*

There are three objects within the .Net Framework that provide network communication using sockets: the `NetworkStream` object, the `TCPClient` object and the `Socket` object.

The `NetworkStream` object presents a standard stream interface to the programmer that fits naturally with the TCP paradigm, which is a stream of bytes. The stream interface has the advantage of being simple to use but it also has the disadvantage that it only supports blocking methods for communication operations. The `Write` and `Read` methods provided by the `NetworkStream` object internally use the blocking send and receive methods provided by the `Socket` object. The MPI Standard includes non-blocking operations, so a mechanism is needed for preventing a blocking method on the `NetworkStream` object from blocking the MPI process.

The `TCPClient` object is a wrapper for a `Socket` object. Its advantage is that it simplifies the creation of a socket that will use TCP by automatically supplying the appropriate arguments for protocol and address family. However, the `TCPClient` object, like the `NetworkStream` object, only supports blocking methods. Unlike the `NetworkStream` object (which provides a stream interface) the `TCPClient` object works with byte arrays.

The `Socket` object exposes all of the functionality supported by the operating system's socket interface to the hardware. This allows it to support non-blocking communication methods. It also allows the programmer to send control codes to the hardware that modify its low-level functionality.

The `Socket` object is the most versatile of the three discussed here but it is also the most complicated to use. Blocking communication methods can be used to implement non-blocking operations – as explained in section 3.1.1.5 – however, using the non-blocking communication methods supported by the `Socket` object may yield better performance. Thus, the `Socket` object was chosen as the communication device for the first communication module in McMPI.

#### *3.1.1.3 Sending and Receiving Data*

For TCP, the `Socket` object provides three methods that send data (`Send`, `BeginSend` and `SendAsync`) and three methods that receive data (`Receive`, `BeginReceive` and `ReceiveAsync`). The `BeginSend` and `BeginReceive` methods follow the standard syntax in C# for non-blocking methods. The MSDN documentation states that the `SendAsync` and `ReceiveAsync` methods were introduced because they re-use system resources more efficiently and so yield better performance. Thus, the `BeginSend` and `BeginReceive` methods were not evaluated for this work. The other methods provided

by the Socket object, in particular SendPacketsAsync, are not considered here because they are designed for use with datagram protocols, such as UDP, rather than stream protocols, such as TCP. Data sent with any of the send methods can be received with any of the receive methods, so there are four potential combinations.

The Send method is synchronous, i.e. it will block execution until data has been transmitted to the destination. The Send method can accept a single byte array or multiple byte arrays as input arguments and will only return control once all the data in all the byte arrays has been transmitted or buffered internally by the socket. The Receive method is also synchronous, i.e. it will block until data has been received from the source. The Receive method can accept a single byte array or multiple byte arrays as output arguments but may return control before all of these byte arrays are filled with data. It will not return control until at least one byte of data has been received but, if the total amount of data transmitted from the source and internally buffered by the socket is less than the total amount of data needed to fill all the output byte arrays, then it will receive all the available data and return the number of bytes that were successfully received. To fill the rest of the byte arrays, more data must be received; the Receive method must be called again with arguments that specify the unfilled portions of the byte arrays.

The BeginSend and BeginReceive methods follow the standard .Net Framework syntax pattern for producing an asynchronous method from a synchronous method, in this case from the Send method and the Receive method, respectively. The BeginSend method initiates a send operation but does not complete it. Control is returned immediately along with an object (which implements the IAsyncEvent interface) that must be passed to the EndSend method to complete the send operation. The BeginReceive method initiates a receive operation but does not complete it. Control is returned immediately along with an object (which implements the IAsyncEvent interface) that must be passed to the EndReceive method to complete the receive operation. This pattern is similar to the non-blocking pattern in the MPI Standard: for example, the MPI_ISEND function initiates a send operation and returns a request structure, which is later passed to the MPI_WAIT function to complete the operation. This pattern requires that a new IAsyncEvent object (or request structure) is created for each asynchronous (or non-blocking) operation. Object creation (and structure allocation) can be slow operations and so should be avoided, if possible, for example by reusing an

existing object (or structure). The MPI Standard achieves reuse of request structures by defining persistent requests (see section 2.4.2 and section 3.3.3). The `SendAsync` and `ReceiveAsync` methods allow reuse of the `SocketAsyncEventArgs` object (which is the equivalent of the `IAsyncEvent` object).

The `BeginSend` and `SendAsync` methods are both asynchronous equivalents to the `Send` method, i.e. the data will be sent in exactly the same manner as for the `Send` method but the execution of the calling thread will not be blocked. The `BeginReceive` and `ReceiveAsync` methods are both asynchronous equivalents to the `Receive` method, i.e. the data will be received in exactly the same manner as for the `Receive` method but the execution of the calling thread will not be blocked. When calling the `BeginSend` (or `BeginReceive`) method, a call-back method is specified that will be called when data has been sent (or received). The call-back method will be passed a new `IAsyncEvent` object in order that the send operation (or receive operation) can be completed by calling the `EndSend` (or `EndReceive`) method. When calling the `SendAsync` (or `ReceiveAsync`) method, a `SocketAsyncEventArgs` object is supplied that specifies a call-back method that will be called when the operation is complete. No new object is needed in this case and there is no equivalent of the `EndSend` (or `EndReceive`) method; the `SocketAsyncEventArgs` object will be passed to the call-back method and it will contain the information about the completed operation. If the send (or receive) operation completes synchronously during the `SendAsync` (or `ReceiveAsync`) method then the call-back method is not called and the `SocketAsyncEventArgs` object contains the information about the completed operation immediately.

As stated in section 3.1.1.1, splitting data into packets and verifying the delivery of each packet account for a significant proportion of the overhead of TCP. One large packet will involve less overhead than two small packets, even when the total number of bytes transmitted is the same. For this reason, TCP implementations use the Nagle algorithm #ref#, which delays transmission of small amounts of data in order that multiple small chunks of data can be combined into fewer large chunks of data. When data is frequently being sent and received in small amounts, this algorithm increases bandwidth usage at the expense of latency. For MPI, this algorithm can cause problems; in particular, a small write may be delayed until the recipient acknowledges a previous write, which occurs either

when the recipient transmits data in response (typically causing a delay of one round-trip time) or after a long timeout has expired (typically causing a delay an order of magnitude greater than a round-trip time). The `Socket` object provides the `NoDelay` property to turn off the Nagle algorithm, which results in all data being sent as soon as possible even when that will result in multiple small TCP packets.

The eager and transfer protocol messages (refer to section 3.1.2 for more information) consist of header information (a small byte array buffer created by the protocol layer) and message data (a different byte array buffer created by the user). In a multi-threaded environment, the two parts of the eager and transfer protocol messages must be transmitted in a single socket method call to avoid corruption caused by another thread inserting an unrelated protocol message between the header and message data. For small messages, combining these two buffers into a single buffer will result in a fewer packets being created and should result in faster transmission to the destination. For large messages, combining these two buffers into a single buffer is a costly procedure because it involves copying the header information and the message data. Although none of the three send methods allow granular control of packet boundaries, all of them allow multiple byte array buffers to be specified in a single method call, which preserves the integrity of the protocol message whilst avoiding the cost of combining the two constituent buffers, and should result in faster transmission to the destination.

#### *3.1.1.4 Detecting the Arrival of Data*

The `Socket` object provides three ways to obtain information (such as whether data has arrived) about the status of the socket; the `Available` property, the `Poll` method, and `Select` method. This information can be used to predict the behaviour of calls to send and receive methods – in particular, whether the call will block.

The `Available` property returns the number of bytes that have been received and internally buffered by the socket and are therefore available to be read. If the `Available` property returns zero then no data is available and a call to `Receive` will block until data arrives; otherwise a call to `Receive` will return immediately having copied some or all of the available data bytes into the specified byte array buffers. If the `Available` property returns zero then a call to `ReceiveAsync` will return true indicating that it will complete asynchronously by calling the `Completed` call-back method when data arrives; otherwise

a call to `ReceiveAsync` may complete synchronously and so return false indicating that the `Completed` call-back method will not be called.

The `Poll` method returns the status of a socket. The mode argument controls which aspect of the socket status is queried: `SelectRead` queries whether the socket has data available for a receive operation, `SelectWrite` queries whether the socket has buffer space available for a send operation, and `SelectError` queries whether the socket has detected an error condition. The `Poll` method will block execution until status of the socket allows it to return true or until the specified number of microseconds has elapsed (or indefinitely, if a negative number of microseconds is specified). For example, assuming the `Poll` method is called with `SelectRead`, if data is available then it will true immediately, if data arrives within the specified number of microseconds then it will return true at that time, but if there is still no data available when the timeout expires then it will return false.

The `Select` method returns the status of multiple sockets. It is passed three lists of sockets: those that must be checked for readability, those that must be checked for writeability, and those that must be checked for an error condition. It modifies the three lists by removing sockets that are not in the appropriate state, i.e. where the `Poll` method would return false. The `Select` method will block execution in the same manner as the `Poll` method. Only one socket in any of the three lists must be in the correct state to cause the `Select` method to return control before the timeout expires.

#### *3.1.1.5 Building a Non-blocking Operation from a Blocking Method*

For the purposes of implementing point-to-point message passing in an MPI library, the `Send` and `Receive` methods are problematic because they may block execution and the MPI Standard requires some communication functions to be non-blocking.

A blocking MPI function can be implemented simply by calling the non-blocking equivalent and then waiting for the non-blocking function to complete before returning control. However, the converse – implementing a non-blocking function using a blocking one – is not as simple. The MPI Standard allows the non-blocking initiation function to test whether the operation can be completed immediately and, if not, to record the details of the operation and rely on future function calls to complete it. Calls to unrelated MPI functions give the MPI library an opportunity to test whether any pending operations can be

completed. In addition, the MPI Standard requires that a completion function is called for every non-blocking operation that is started. The completion function is either blocking, such as `MPI_WAIT`, or must be called repeatedly until it indicates that the operation has completed, such as `MPI_TEST`. Thus, a non-blocking function can be implemented using a blocking function if there is a non-blocking test that reliably detects when the blocking function can complete immediately, i.e. when it will not have to block.

The `Poll` and `Select` methods, with a timeout value of zero microseconds, offer the possibility of a non-blocking test of whether a blocking `Send` method, or a blocking `Receive` method, will complete immediately or will have to block. TCP sockets only connect two nodes together. However, an MPI library must connect many nodes together and will so need to check the status of many sockets. Thus, the `Select` method, which can check multiple sockets in each call, is more suitable than the `Poll` method, which is limited to checking only one mode for one socket in each call.

When checking whether data can be read from a socket, these methods indicate that at least one byte of data is available. This means that a call to the `Receive` method will not block. However, it may return one byte. If less data is read than is required for a receive operation then the MPI library can modify the information about the pending operation and rely on future function calls to complete it. Alternatively, the `Available` property can be examined before a call to the `Receive` method to make sure that the amount of data desired is actually present.

When checking whether data can be written to a socket, the `Poll` and `Select` methods indicate that at least one byte can be written without causing the socket to block. However, there is no way to determine exactly how much data can be written to the socket, so there is no way to guarantee that a call to the `Send` method will not block.

#### *3.1.1.6 Read- and Write-threads*

A blocking method blocks the thread that makes the method call but does not block other threads. It is possible to arrange for all `Send` method calls to be made by a thread dedicated to write operations (a write-thread). Similarly, it is possible for all `Receive` method calls to be made by a thread dedicated to read operations (a read-thread). In this way, the main thread is not blocked by calls to blocking socket methods and is free to return control to the MPI program.

A read-thread would call the `Select` method, passing it a list of all the sockets that connect to other MPI processes, then call the `Receive` method for each of the sockets that the `Select` method indicates has data available, and then repeat, i.e. call the `Select` method again. The timeout used for the `Select` method calls can be a large number of microseconds because the read-thread is allowed to block until data arrives. Using the `Select` method in this manner allows a single read-thread to service multiple sockets.

A write-thread would need a list of buffers that are to be sent to other MPI processes along with the associated destination locations, i.e. the identity of the socket to be used for each transfer. The write-thread would call the `Select` method, passing it a list of only the sockets that have data ready to be sent, and then call the `Send` method for each of the sockets that the `Select` method indicates is writeable. It will repeat these steps for the entire life-time of the program. As mentioned in section 3.1.1.5, using the `Select` method in this manner suffers from the disadvantage that the `Send` method may block even for a writeable socket when the amount of data being sent exceeds the amount that the socket can currently accept.

Another approach to threading is to use the `System.Threading.ThreadPool`, which is a collection of background worker threads provided to any C# application that uses it. Any operation can be submitted as a work item to the thread pool and will be executed asynchronously by one of the worker threads from the thread pool. Each send and receive operation can be wrapped in a simple work item method that calls the `Send` or `Receive` method on the appropriate socket object. The worker threads will then execute the work items without blocking the main thread. However, there is no guarantee that one work item will finish before the next is started so several `Send` and `Receive` method calls for the same `Socket` object may be in progress on different worker threads at the same time. This is not advisable because the `Socket` object is not guaranteed to be thread-safe: two simultaneous `Send` methods may result in corruption of the data from the send buffers when it is written into the socket; two simultaneous `Receive` methods may result in corruption of the data when it is read from the socket into the receive buffers.

The asynchronous socket methods, `SendAsync` and `ReceiveAsync`, are executed by separate threads that are created and managed by the system, in a similar manner to the

`System.Threading.ThreadPool` approach described above. However, whereas the work items in a `ThreadPool` may execute simultaneously and interfere with each other, the asynchronous socket method calls are guaranteed to be executed sequentially when needed to prevent corruption of the data being sent or received. In addition, whereas dedicated communication threads need to check the status of multiple sockets using the `Select` method, the asynchronous socket methods queue the tasks until the appropriate socket is in the correct state; the data will be sent or received when sufficient buffer space or incoming data becomes available.

#### *3.1.1.7 Summary*

A `Socket` object using TCP over Ethernet is a versatile communication device, offering blocking and non-blocking communication operations as well as several ways to obtain the status of the socket, which can be used to implement non-blocking communication using the blocking communication operations.

For this work, three ways of sending data are assessed:

- using the `Send` method in the main thread
- using the `Send` method in a write-thread
- using the `SendAsync` method

These are combined with five ways of receiving data, which are:

- using the Receive method in the main thread
- using the Receive method in a read-thread
- using the Select method followed by the Receive method in the main thread
- using the Select method followed by the Receive method in a read-thread
- using the ReceiveAsync method

The results of evaluating these fifteen combinations of send and receive operations are presented in section 5.2.1 (for loopback sockets) and section 5.3.1 (for network sockets). In addition, the four of these fifteen combinations that demonstrated the best communication performance were used to create communication modules for McMPI, following the design defined in section 3.1.2.

### 3.1.2 The Abstract Device Interface

The abstract device interface separates the protocol and communication layers by de-coupling the classes and objects in one from those in the other, which allows each layer to be developed, modified and extended independently from the other. It forms a contract between the two layers that promises specific capabilities and behaviour expected from each layer by the other. In particular, the goal of the abstract device interface is to allow different communication devices to be used by the protocol without code changes and without re-compilation.

The overall design chosen for the ADI follows the abstract factory design pattern [48], as depicted in Figure 4. The abstract factory design pattern is a creational pattern: it specifies the template for a way of creating objects indirectly, i.e. without the creator directly calling the constructor of the object to be created. In this pattern, the client is only aware of (i.e. directly dependent on) the abstract factory and the abstract product interfaces. This allows not only the implementation but also the type (or class) of the factory and product objects to vary independently from the client object.

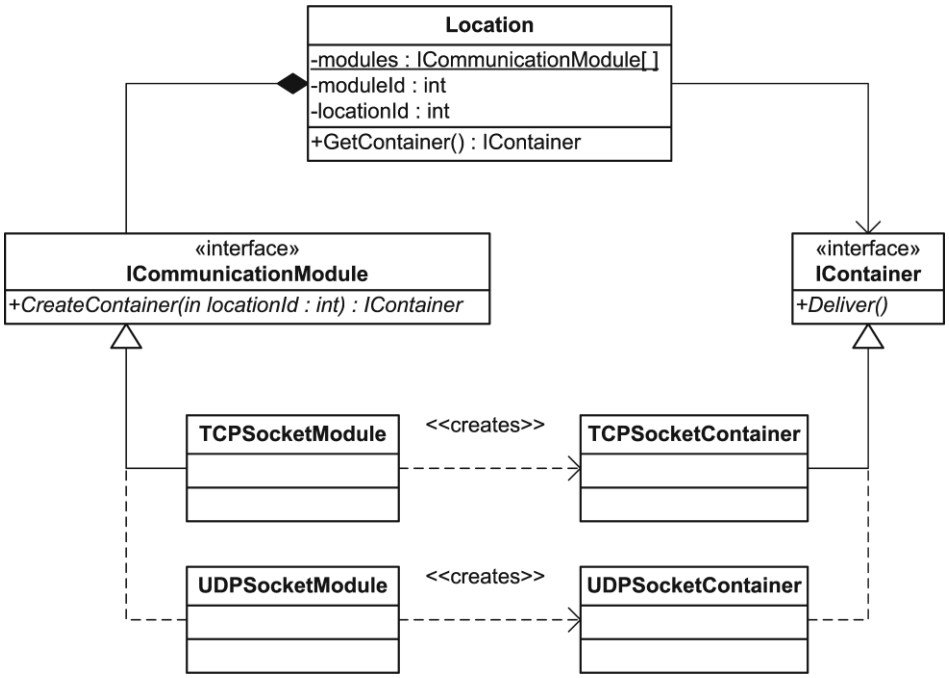


**Figure 4: the design for the ADI follows the abstract factory design pattern; the `TCPSocketModule` and `TCPSocketContainer` classes form the communication layer, the `ICommunicationModule` and `IContainer` interfaces form the ADI, and the `Location` class is part of the protocol layer.**

For the present work four different combinations of `Socket` object methods were selected for evaluation as communication modules in McMPI. The objects and classes in the protocol layer must be able to operate any of these communication modules (as well as any future modules), which requires that all communication modules implement a common interface. This interface, called `ICommunicationModule`, is discussed in section 3.1.2.1.

As discussed in section 3.1, the information to be sent and received by the communication layer is a container, which is made up of a fixed-length one-dimensional byte array (the header) and a variable-length portion of a one-dimensional byte array (the data). Each communication module may require a different implementation for its container objects. However, the protocol layer must be able to interact with all types of container, so all containers must implement a common interface. This interface, called `IContainer`, is discussed in section 3.1.2.2.

The client suggested by the abstract factory pattern is the `Location` class. This class is part of the protocol layer but its primary purpose is to interact with the ADI. Thus, it is discussed in section 3.1.2.3.

The `ICommunicationModule` and `IContainer` interfaces suggested by the abstract factory pattern form the ADI. Together they introduce a level of indirection that avoids unwanted direct dependencies between the protocol layer and the communication layer. This allows new types of communication modules, with new types of containers, to be added to the system without requiring code changes in the protocol layer or in other communication modules and containers.

#### *3.1.2.1 The `ICommunicationModule` Interface*

The abstract factory pattern suggests that the `ICommunicationModule` interface should only contain a single parameter-less method, called `CreateContainer`, which creates and returns an object that implements the `IContainer` interface.

However, each communication module should be able to handle communication with many other locations. In the case of TCP socket communication (as in the present work) this will be achieved by creating and maintaining multiple connections. This requires that a parameter be added to the `CreateContainer` method to specify the location to which the container will be delivered.

#### *3.1.2.2 The `IContainer` Interface*

When sending large messages, the ideal sequence of actions is to copy the data directly from the user send buffer into the communication device in the sending process and directly from the communication device into the user receive buffer in the receiving process. The specifications of the ready mode send and synchronous mode send in the MPI Standard ensure that this ideal is possible by guaranteeing that the receive buffer is available to the MPI library before the message data from the sending process arrives. The ready mode guarantees this by simply declaring the program to be erroneous if it is not. The synchronous send mode guarantees it by sending a notification (effectively a request for permission to send) and waiting for an approval (which is only sent when the receive buffer is available) before sending the message data. The buffered mode send allows the library to copy the data into a buffer in the sending process and send that buffer using synchronous mode semantics. Thus, the receiving process can respond to a buffered send in the same manner as for a synchronous send, i.e. it can guarantee the incoming data is copied directly from the communication device into the user receive buffer. The standard mode send does not guarantee that the user receive buffer is available before the message data arrives in all cases. If, as suggested by the MPI Standard, standard mode is implemented using synchronous mode semantics for large messages but by sending the data immediately (as in ready mode) for small messages, then the data for small messages may arrive before the user receive buffer is available.

Locating the correct user receive buffer (or providing a temporary buffer, if needed) is not within the responsibilities of the communication layer. The protocol layer is responsible for matching remote send operations (i.e. incoming message data) with local receive operations (i.e. user receive-buffers or temporary buffers). The detail of how this step is performed is not relevant to the communication layer but is set out as part of the specification of the protocol layer in section 3.2.2.3 – in brief, the protocol layer attempts to locate a user receive buffer; if one is not found then a temporary buffer is created and used instead. This division of responsibility can be achieved by including with the message data some header information that is generated by the protocol layer in the sending process and interpreted by the protocol layer in the receiving process. The communication layer in the receiving process delivers the header to the protocol layer, which responds with the correct buffer into which to receive the data. Thus, although the communication layer in the sending process must transmit the header and message data as one piece, the

communication layer in the receiving process must be aware of the partition that separates the header portion from the data portion.

In practice, the container objects are defined by classes that are specific to particular communication modules, e.g. the `TCPSocketContainer` class is the container for the `TCPSocketModule`.

The solution adopted for McMPI is to define the `IContainer` interface, which explicitly partitions the data to be transmitted into protocol header and message data. All container classes implement this interface so that the protocol layer can interact with them. The author of a specific container class is then free to choose any suitable implementation.

For example, a communication module based on UDP would not be able to transmit data byte arrays larger than the maximum datagram size permitted in the UDP implementation. In this case, the `UDPSocketContainer` in the sending process would break each large data byte array into multiple smaller datagrams for transmission and the `UDPSocketContainer` in the receiving process would reassemble them into the large data byte array. As TCP can handle transmissions of arbitrary size, the `TCPSocketContainer` can keep the data byte array intact.

Each container provides a non-blocking `Deliver` method, which causes the container to deliver itself using the information provided by the communication module during its creation. In particular, the `TCPSocketModule` gives each `TCPSocketContainer` that it creates a reference to the `Socket` object that is connected to the intended destination. The `TCPSocketModule` is responsible for connecting `Socket` objects to destinations and for selecting the correct `Socket` object for a given destination but it is not responsible for delivery of data. The `TCPSocketContainer` is responsible for delivery of data using the `Socket` object provided to it; it has no knowledge of how the `Socket` was created or connected to the destination.

#### *3.1.2.3 The `Location` Class*

Having one communication module per connection (destination or source) will be too wasteful of memory, especially for the `TCPSocketModule` as it requires a different socket for each connection. As the number of MPI processes is scaled up on larger systems, the number of socket objects needed is increased proportionally. One communication

module can handle multiple connections by creating multiple socket objects. The method to get a container object must be parameterised with the identification of the connection that is required, i.e. the location identifier. The location identifier can be as simple as the integer index in the internal array of connections but it should be encapsulated inside a `Location` object to isolate the rest of the application from the precise details. The `Location` object can also contain a reference to the communication module that should be used for the location it represents and provide a method that gets a container from the communication module by passing the internal location identifier to it. This design follows the flyweight design pattern [48] with the following designations:

- the `Location` object is the flyweight factory
- the `ICommunicationModule` is the generic type of the flyweight objects
- the `TCPSocketModule` is an example of a concrete flyweight object

The `Location` class contains an array of `ICommunicationModule` objects, either only those that have been used by the application so far (commonly called lazy initialisation) or all those that could be used by the system (created from initialisation information during application start-up). Each `Location` object stores two integers: the module identifier (the index of the appropriate `ICommunicationModule` object in the modules array) and the location identifier (used by the communication module to identify the appropriate connection or location, e.g. for the `TCPSocketModule`, it is the index of the appropriate `Socket` object in the sockets array).

## 3.2 The Protocol Layer

The protocol layer may be seen as the intermediary between the interface layer, which interacts with the user, and the communication layer, which interacts with the network hardware. The instructions from the user consist of send and receive operations, i.e. providing data that must be transmitted to another MPI process and requesting data that must be transmitted from another MPI process. The communication layer handles the transmission of data to and from other MPI processes. The protocol layer is responsible for coordinating with other MPI processes. It must decide what to do with outgoing message data – whether to notify the destination MPI process then wait for approval (for synchronous mode messages) or to send it with the message envelope without waiting for approval (for ready mode messages). It must decide what to do with an incoming

notification – whether to approve the transfer of the message data (if a matching receive request exists) or to store it (if there is, as yet, no matching receive request). It must decide what to do with incoming message data – whether to complete the matching receive request (if there is one) or to store it (if there is not).

These decisions are diagrammatically represented and analysed from two perspectives: the logical design as four communication scenarios, using data-flow diagrams (in section 3.2.1), and the physical design as three functional units, using pseudo-code (in section 3.2.2).

Section 3.2.3 explains the linkage between the protocol layer and the communication layer.

Subsequent sections analyse the benefits of different methods of implementing various necessary components: data storage (in section 3.2.4), thread safety (in section 3.2.5), local delivery (in section 3.2.6), envelope matching (in section 3.2.8), and encoding the message header (in section 3.2.9).

Section 3.2.7 presents a possible extension to the threading model defined in the MPI Standard, which allows the optimisation used for local delivery to be applied to delivery between MPI 'processes' that are threads in the same operating system process.

### 3.2.1 Four Communication Scenarios

The data-flow diagrams (DFDs) in the following four sub-sections depict two MPI processes cooperating in a point-to-point communication scenario where, in each case, the process on the right of the diagram sends a message (using the standard send mode defined in the MPI Standard) to the process on the left of the diagram, which receives the message. In the first two of these scenarios, the message is small, i.e. the size of the message data is less than a threshold value (called the eager limit). In the last two of these scenarios, the message is large, i.e. the size of the message data is greater than the eager limit. The exact value of the eager limit should be determined from performance data measured, for example, when the library is installed on a particular machine. When the message size is below the eager limit, the eager protocol is used to send it, i.e. the message data is transmitted with the message envelope as part of the initial protocol message and without obtaining permission from the destination MPI process. When the message size is above the eager limit, the rendezvous protocol is used to send it, i.e. only the message envelope is transmitted in the initial protocol message; the data is transmitted when the destination

MPI process gives permission by responding with an approval protocol message. Within each pair of scenarios, the first assumes the receive operation is later than the send operation and the second assumes the send operation is later than the receive operation.

The rendezvous protocol described in the last two scenarios is also used to implement the synchronous mode send defined in the MPI Standard. Thus, the diagrams in Figure 7 and Figure 8 could equally well depict a synchronous mode send of a message of any size instead of a standard mode send of a large message. The eager protocol described in the first two scenarios is also used to implement the ready mode send defined in the MPI Standard. The diagram in Figure 6 could equally well depict a ready mode send of a message of any size instead of a standard mode send of a small message. However, the first scenario is erroneous for ready send as the receive operation is not started before the send operation. Thus, the diagram in Figure 5 cannot depict a ready mode send.

Data-stores are shown as two parallel horizontal lines, above and below the name of the data-store. There are two data-stores that are used for both the eager and the rendezvous protocol: the Request Queue stores receive operations that have not yet been matched with a send operation; the Unexpected Queue stores incoming messages that have not yet been matched with a receive operation. In addition, there are two data-stores that are only used for the rendezvous protocol: the Pending Queue stores the message data for outgoing messages until permission to transfer it is obtained from the destination; the Matched Queue stores receive operations that have been matched with a send operation until the message data is transferred from the source.

The communication layer is shown as rectangle and the boundary of the operating system process is shown as a vertical dashed line. Data-flows that cross the operating system process boundary must be sent by the communication layer in one process and received by the communication layer in the other process.

Tasks within the protocol layer are shown as circles with no border. Tasks that are not part of the protocol layer are shown as circles with a border: a white border signifies that the task provides an input data-flow to the protocol layer; a black border signifies that the task accepts an output data-flow from the protocol layer. When implemented, these tasks will be the points at which the protocol layer interacts with the interface layer (described in section 3.3).

### *3.2.1.1 The Eager Protocol with a Late Receive Scenario*

The send operation for the eager protocol is straightforward: an eager protocol message, containing the message envelope and the message data, is given to the communication layer for delivery to the destination MPI process. The communication layer is reliable so the message will definitely arrive and the send operation can complete with no further action in the sending MPI process. For a blocking send operation this means that it can return control and for a non-blocking operation it means that the request object returned by the send operation can be marked complete.

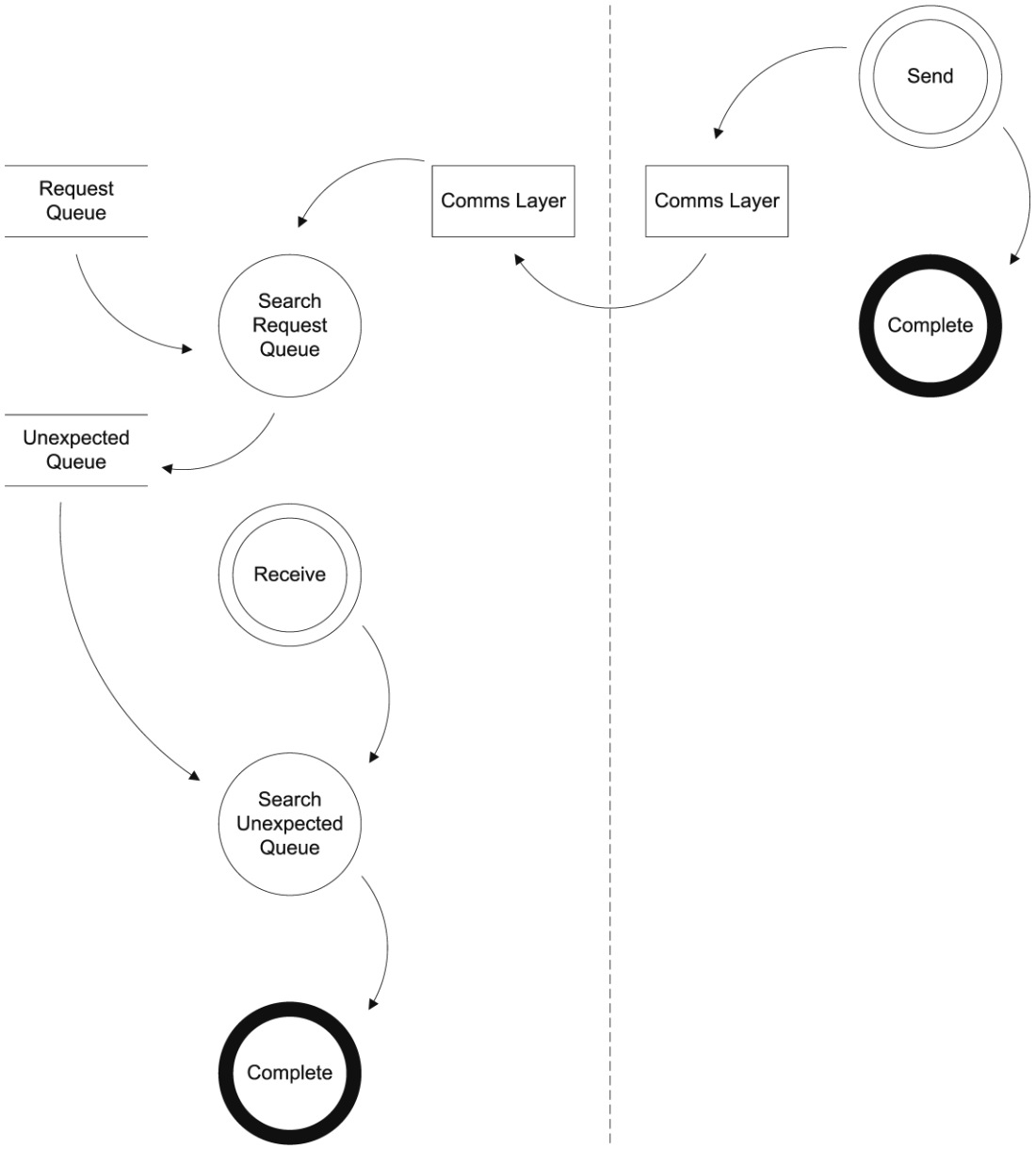


**Figure 5: DFD showing the eager protocol with a late receive.**

The communication layer in the receiving MPI process takes delivery of the eager protocol message and passes it to the protocol layer, which looks for a receive operation that matches the message envelope by examining the request queue. The request queue stores the envelope information from all receive operations that have not yet been matched with incoming messages. In this scenario, the receive operation is later than the send operation, so the request queue will be empty (or, more generally, will not contain a receive request that matches the incoming eager message). As the incoming eager message cannot be matched immediately it is stored in the unexpected queue.

When a receive operation is started, it first looks for a message that matches its envelope information by examining the unexpected queue. In this scenario, a matching message has been stored in the unexpected queue and so the search is successful. The eager protocol message is removed from the unexpected queue, the message data is copied from it into the receive buffer, and the receive operation can complete without further action in the receiving MPI process. For a blocking receive operation this means that it can return control and for a non-blocking operation it means that the request object returned by the receive operation can be marked complete.

#### *3.2.1.2 The Eager Protocol with a Late Send Scenario*

In this scenario, the receive operation is started before the send operation or, more precisely, the receive operation examines the unexpected queue before the matching message from the sending MPI process arrives. In this case, the search for a matching message is unsuccessful and the envelope information from the receive operation is stored in the request queue. A non-blocking receive operation can return the request object (marked as incomplete) at this point, which allows the MPI process to do other work. A blocking receive operation cannot return control yet; it must wait until the operation is complete.

The send operation in this scenario is identical to the one in the previous scenario because the relative timing of the send and receive operations only makes a difference at the receiving MPI process. Thus, the send operation simply passes an eager protocol message to the communication layer and completes with no further action in the sending MPI process.

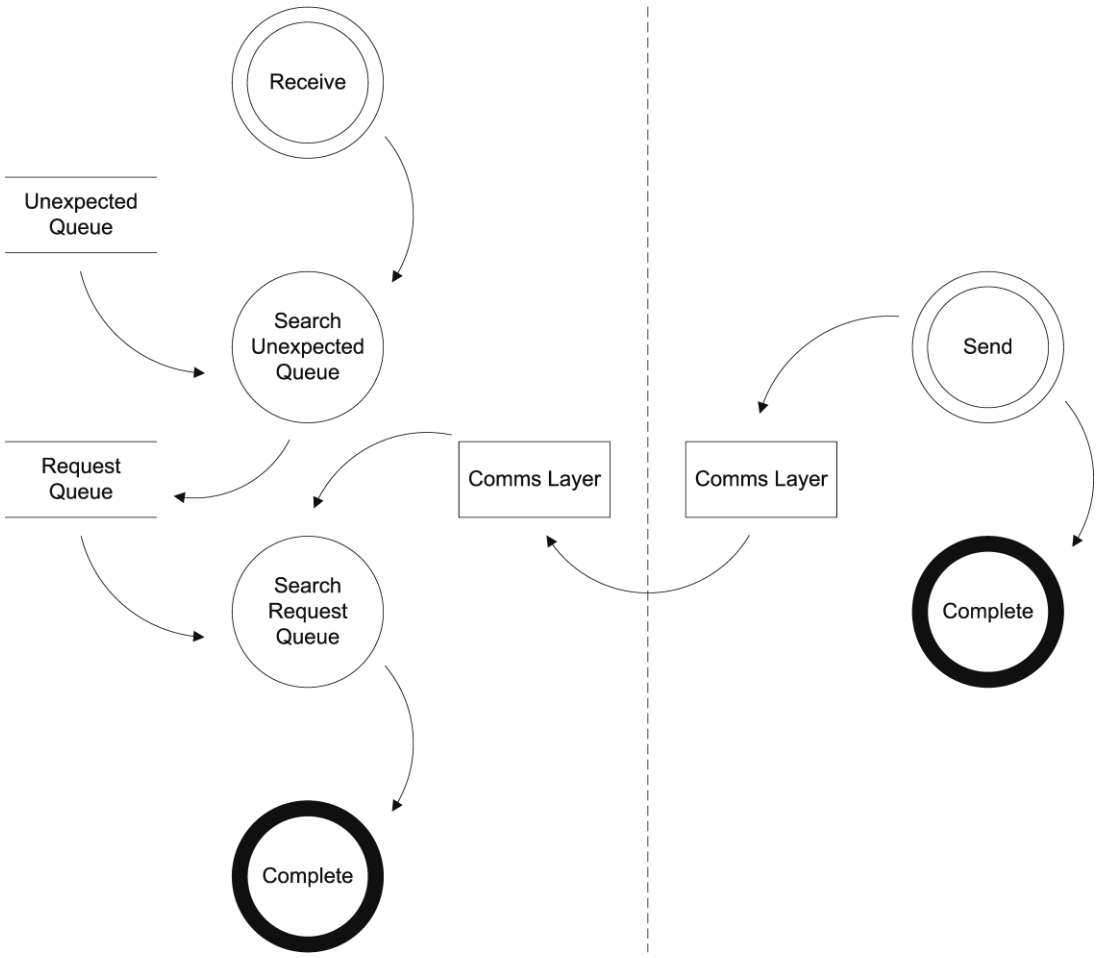


**Figure 6: DFD showing the eager protocol with a late send.**

As in the previous scenario, the communication layer in the receiving MPI process takes delivery of the eager protocol message and passes it to the protocol layer, which looks for a receive operation that matches the message envelope by examining the request queue. In this scenario, the search is successful because the send operation is later than the receive operation. The matching request is removed from the request queue, the message data is copied from the eager protocol message into the receive buffer, and the receive operation can complete without further action in the receiving MPI process.

#### *3.2.1.3 The Rendezvous Protocol with a Late Receive Scenario*

The send operation for the rendezvous protocol simply notifies the receiving MPI process that a message is available to send: a notification protocol message, which contains the message envelope but no message data, is given to the communication layer for delivery to the destination MPI process. The message data is stored in the pending queue. A non-blocking send operation can return a request object (marked as incomplete) at this point,

which allows the MPI process to do other work. A blocking send operation cannot return control yet; it must wait until the operation is complete.

The communication layer in the receiving MPI process takes delivery of the notification protocol message and passes it to the protocol layer, which looks for a receive operation that matches the message envelope by examining the request queue. In this scenario, the receive operation is later than the send operation, so the request queue will not contain a receive request that matches the incoming notification. As the incoming notification cannot be matched immediately it is stored in the unexpected queue. The initial response to the arrival of a notification protocol message is, therefore, identical to the initial response to the arrival of an eager protocol message.

The receive operation in this scenario begins in the same way as for the eager protocol scenario: it looks for a message that matches its envelope information by examining the unexpected queue. In this scenario, a matching message envelope has been stored in the unexpected queue and so the search is successful. The notification protocol message is removed from the unexpected queue but it contains no data so further action is needed. An approval protocol message is passed to the communication layer for delivery to the MPI process that sent the notification and the information known so far is stored in the matched queue where it can be retrieved when the data arrives. A non-blocking receive operation can return a request object (marked as incomplete) at this point, allowing the MPI process to do other work, but a blocking receive operation cannot return control yet; it must wait for the message data to arrive.

The communication layer in the sending MPI process takes delivery of the approval protocol message and passes it to the protocol layer, which locates the message data in the pending queue. A transfer protocol message, which contains the message data, is passed to the communication layer for delivery to the receiving MPI process. The communication layer is reliable so the message data will definitely arrive and the send operation can complete with no further action in the sending MPI process. For a blocking send operation this means that it can return control and for a non-blocking operation it means that the request object returned by the send operation can be marked complete.

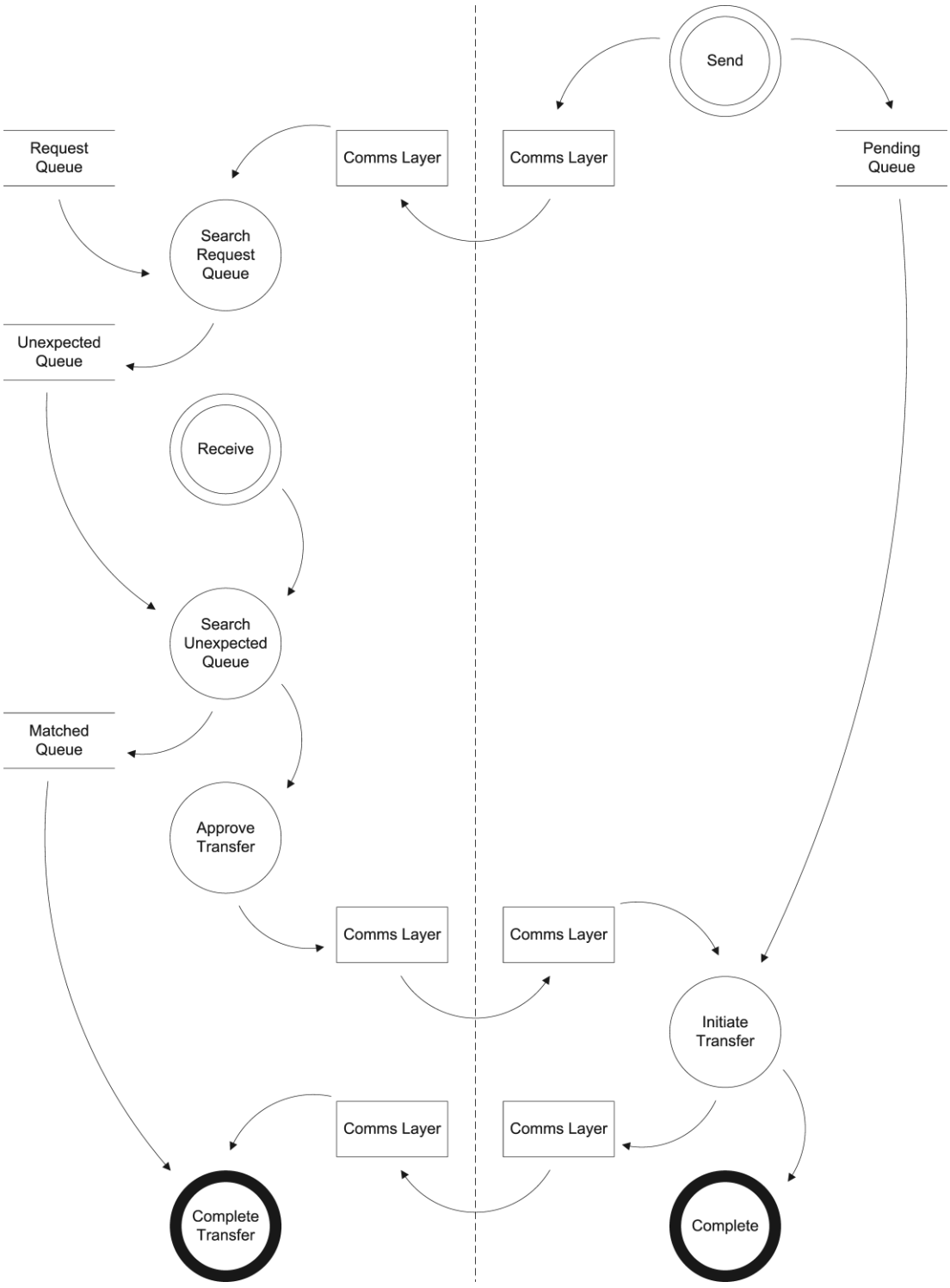


**Figure 7: DFD showing the rendezvous protocol with a late receive.**

The communication layer in the receiving MPI process takes delivery of the transfer protocol message and passes it to the protocol layer, which locates the information (about the notification and the receive operation) stored in the matched queue. This information is removed from the matched queue, the message data is copied from the transfer protocol

message into the receive buffer, and the receive operation can finally complete. For a blocking receive operation this means that it can return control and for a non-blocking operation it means that the request object returned by the receive operation can be marked complete.

#### *3.2.1.4 The Rendezvous Protocol with a Late Send Scenario*

All of the steps necessary for this scenario are identical to steps in the other scenarios and have been described in previous sections.

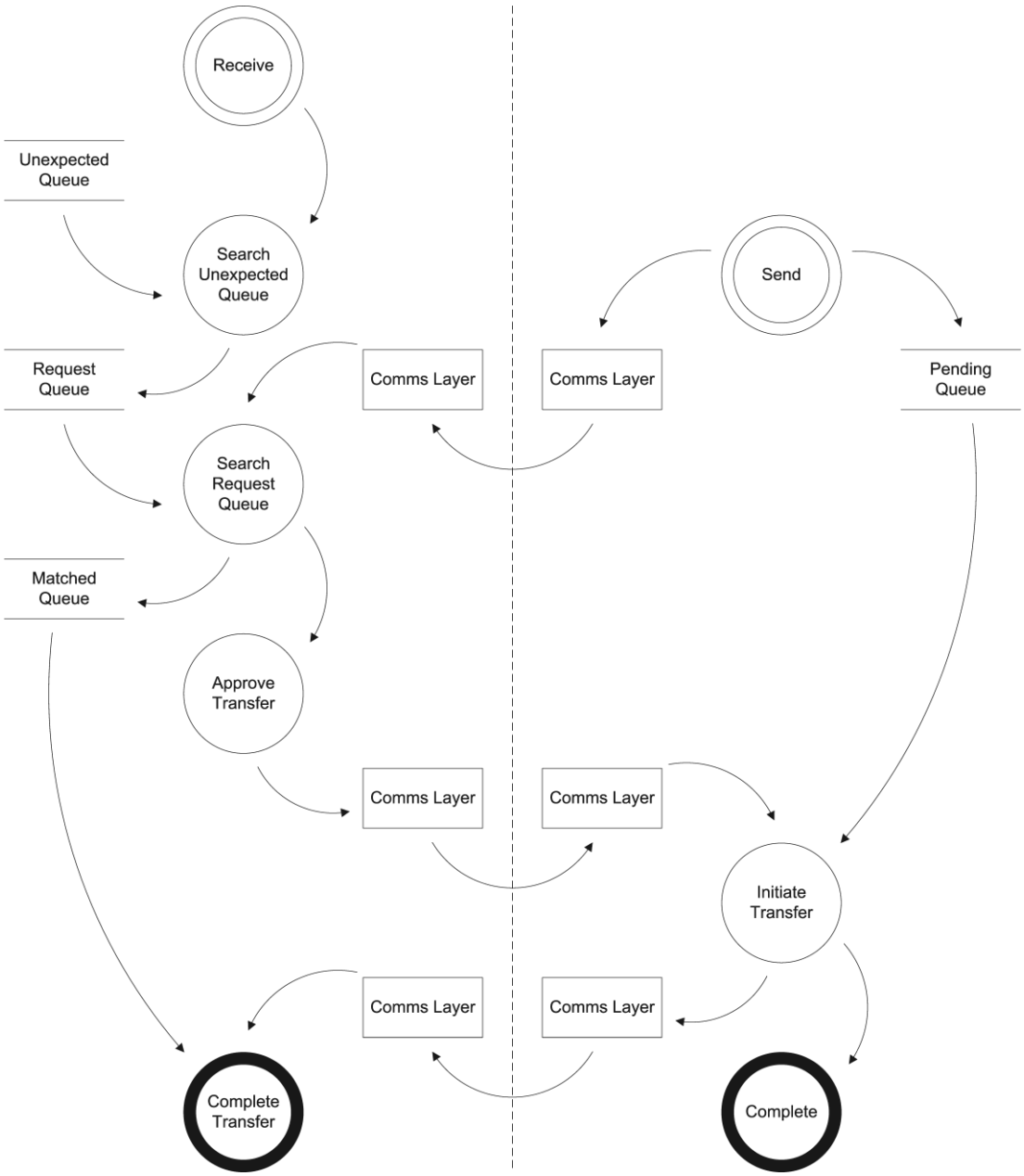


**Figure 8: DFD showing the rendezvous protocol with a late send.**

Both parts of the send operation are identical to the rendezvous protocol with a late receive scenario (section 3.2.1.3). Initially, a notification protocol message containing the message envelope is passed to the communication layer whilst the message data is stored in the pending queue. When the approval protocol message arrives, a transfer protocol message containing the message data, which has been removed from the pending queue, is delivered to the receiving MPI process. The receive operation begins the same way as the eager protocol with late send scenario (section 3.2.1.2). It examines the unexpected queue, fails to find a message that matches and stores its envelope information in the request queue. When the notification protocol message arrives, the envelope information for the matching receive operation is removed from the request queue but, because the notification contains no data, further action is needed – just as in the rendezvous with late receive scenario (section 3.2.1.3). An approval protocol message is passed to the communication layer and the information known so far is stored in the matched queue until a transfer protocol message arrives from the sending MPI process and allows the receive operation to complete.

## 3.2.2 Three Functional Units

The tasks depicted in the scenarios in section 3.2.1 can be split into three functional units according to the trigger for the task: section 3.2.2.1 examines a send operation initiated by the user via the interface layer, section 3.2.2.2 examines a receive operation initiated by the user via the interface layer, and section 3.2.2.3 examines the arrival of a protocol message sent by another MPI process via the communication layer. The pseudo-code in this section does not include locks to protect the data-stores – it is assumed that this functionality is encapsulated within the data-stores themselves. Section 3.2.5 discusses the protection of data-stores with locks.

### *3.2.2.1 The Send Functional Unit*

Each send operation must decide whether to use the eager protocol or the rendezvous protocol. This decision is made by the interface layer: a ready mode send will always use the eager protocol, a synchronous mode send will always use the rendezvous protocol and a standard mode send can use either protocol, depending on the message size. The eager protocol requires that messages are sent in their entirety in the first protocol message, so the message is passed to the communication layer immediately upon receipt from the interface layer. The rendezvous protocol requires that permission is obtained before the

message data is sent, so it must be stored until the receiver approves the transfer. Thus, the send operation in the protocol layer may be described by the following pseudo-code:

```
IF (protocol is eager) THEN
  pass the entire message to the communication layer
ELSE //protocol is rendezvous
  store the message in the pending queue
  pass a request for permission to the communication layer
ENDIF
```

#### *3.2.2.2 The Receive Functional Unit*

The receive operation may happen before or after a matching initial protocol message (IPM) from the sender arrives. This can be determined by attempting to match the receive operation with all the initial protocol messages that have already arrived but have not yet been matched, i.e. those in the unexpected queue. If a match is not found then the receive operation must be stored in the request queue until it can be matched with an incoming IPM. If a match is found then the next action depends on the type of the IPM. For an eager message, all the data is delivered in the initial protocol message so the receive operation can be completed immediately. For a rendezvous message, only a message envelope (representing a request for permission to send the data) is delivered in the initial protocol message, so the next action is to send an approval protocol message. Thus, the receive operation in the protocol layer may be described by the following pseudo-code:

```
FOR EACH initial protocol message IN the unexpected queue
  IF the receive operation matches the unexpected IPM THEN
    STOP FOR EACH
  ENDIF
END FOR EACH
IF no match was found THEN
  store the receive operation in the request queue
ELSE IF the matching IPM is an eager protocol message THEN
  complete the receive operation
ELSE //matching IPM is a notification protocol message
  store the receive operation and IPM in the matched queue
  pass an approval to the communication layer
ENDIF
```

#### *3.2.2.3 The Data Arrival Functional Unit*

The data arrival functional unit performs different tasks depending on which of the four types of protocol message has arrived.

An eager protocol message is an IPM that contains both a message envelope and the message data. The message envelope must be checked against each unmatched receive operation in the request queue in turn until a match is found or the end of the queue is reached. If no match is found then the IPM must be stored in the unexpected queue. If a matching receive operation is found then it can be completed by copying the message data into the receive buffer. This is described by the following pseudo-code:

```
FOR EACH unmatched receive operation IN the request queue
  IF the receive operation matches the incoming IPM THEN
    STOP FOR EACH
  ENDIF
END FOR EACH
IF no match was found THEN
  store the incoming IPM in the unexpected queue
ELSE
  complete the receive operation
ENDIF
```

A notification protocol message is an IPM that contains a message envelope but does not contain message data. Just as for an eager protocol message, the envelope must be checked against each unmatched receive operation in the request queue in turn until a match is found or the end of the queue is reached. If no match is found then the IPM must be stored in the unexpected queue. If a matching receive operation is found then the IPM and the receive operation must be stored in the matched queue and an approval protocol message must be passed to the communication layer for delivery to the sending MPI process. This is described by the following pseudo-code:

```
FOR EACH unmatched receive operation IN the request queue
  IF the receive operation matches the incoming IPM THEN
    STOP FOR EACH
  ENDIF
END FOR EACH
IF no match was found THEN
  store the incoming IPM in the unexpected queue
ELSE
  store the receive operation and IPM in the matched queue
  pass an approval to the communication layer
ENDIF
```

An approval protocol message will always relate to a pending message that has been stored until permission to send is obtained. The arrival of an approval protocol message permits

the data for the associated message to be sent to the receiving MPI process as a transfer protocol message. This is described by the following pseudo-code:

```
locate the message in the pending queue
pass a transfer protocol message to the communication layer
```

A transfer protocol message will always relate to a receive operation that has been matched with a notification protocol message and stored in the matched queue until the data is obtained. The arrival of a transfer protocol message permits the associated receive operation to be completed by copying the message data into the receive buffer. This is described by the following pseudo-code:

```
locate the receive operation and IPM in the matched queue
complete the receive operation
```

### 3.2.3 The Bridge Design Pattern

All of the protocol messages share a common interaction pattern with the communication layer: each protocol message obtains a container, customises its contents and instructs it to deliver itself. This relationship can be formalised by deriving all specific protocol message classes from a common abstract `ProtocolMessage` class that contains all of the shared code. This design follows the bridge design pattern [48] shown in Figure 9.

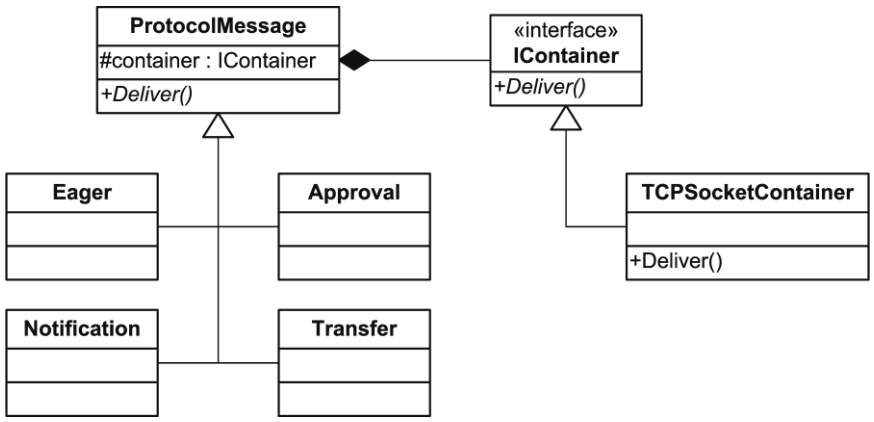


**Figure 9: the design for the linkage between the protocol and the communication layers follows the bridge design pattern. The ProtocolMessage class is the abstraction, its four derived classes are refined abstractions, the IContainer interface is the abstract implementer and the TCPSocketContainer is a concrete implementer.**

The container objects needed by protocol message objects to send messages are created via the flyweight design pattern using a `Location` object (as explained in section 3.1.2.3). The container objects needed by the communication layer to receive messages are created by the communication modules during the initialisation of the program. A receiving

container object uses a call-back method, or event, e.g. called `ReceivedData`, to pass itself to the protocol layer when data has been read. During the start-up of the library, an initialising object will pass a reference for this call-back method (called a delegate in C#) to the constructor of each communication module, which will in turn pass it to each container object it creates to receive data. The call-back delegate, or event, is exposed via the `IContainer` interface so that it is supported by all types of container objects. In this way, a common bi-directional relationship is created between specific protocol message objects and specific container objects without either being directly dependent on the other.

### 3.2.4 The Singleton Design Pattern

There are four data-stores identified during the design of McMPI in section 3.2.1 and 3.2.2: the request and unexpected data-stores (which support the matching of receive operations with send operations), plus the pending and matched data-stores (which support the delaying of data transfer for synchronous mode messages and large standard mode messages).

The request and unexpected data-stores require identical functionality to each other. The unmatched receive operations and unexpected initial protocol messages (which represent unmatched send operations in remote locations) must be matched in a strict sequence. Each new receive operation must be matched with the first (chronologically) unmatched send operation, i.e. initial protocol message in the unexpected data-store, that has suitable envelope information. Each new send operation, i.e. initial protocol message arriving from a remote MPI process, must be matched with the first (chronologically) unmatched receive operation. The chronological ordering element of the sequence required by the MPI Standard is most easily facilitated by storing the unmatched receive operations and unexpected initial protocol messages in chronological order, i.e. in a first-in-first-out (FIFO) queue. However, the requirement that the envelope information must match means that the correct order of matching is not strictly chronological and so objects may not be removed in exact first-in-first-out queue order. The C# queue objects (either `System.Collections.Queue` or `System.Collections.Generic.Queue<T>`) only allow the head item of the queue to be removed, which is not suitable for this situation where any item may be the first item that matches and must be removed. A linked-list has the properties suitable for both the request and unexpected data-stores. If new items are always added to the tail of the linked-list then the chronological sequence

will be maintained, with the head being the first item to have arrived and the tail being the most recent item to have arrived. A linked-list can be traversed from head to tail and any item can be removed whilst maintaining the chronological order of the remaining items. There is no object in the .Net Framework that implements a singly-linked-list but the `System.Collections.Generic.LinkedList<T>` object implements a doubly-linked-list of objects of type `T`, which is also suitable. Adding and removing an item are $\Theta(1)$ operations, i.e. the time taken is independent of both the number of, and the size of, items in the list.

For the unexpected data-store, the discussion above makes the assumption that the communication layer maintains the order of send operations, i.e. the initial protocol messages arrive at the receiving MPI process in the same chronological order in which the send operations were issued at the sending MPI process. This requires either that the order of the send operations is maintained throughout the delivery mechanism or that the order can be restored at the receiving MPI process, e.g. from embedded message sequence numbers. One problem with reassembling the order of send operations is that no matching can take place beyond the first gap in sequence numbers, even if later messages have arrived at the receiver and would match a receive operation and none of the send operations that will eventually fill in the gap will match that receive operation. A complex algorithm that cleans up the unmatched receive operations once the sequence of initial protocol messages is completed would need to be designed and implemented. On the other hand, requiring that initial protocol messages arrive at the receiver in the correct order simplifies the receiver algorithm but requires that both the sender and the communication layer maintain the order. This restriction poses little problem for the sender because calls to the send operation are sequential within each thread of execution, meaning that the initial protocol message for one send operation will be passed to the communication layer before another send operation can begin in the same execution thread. The restriction may pose a bigger problem to the communication layer if the underlying communication method does not inherently guarantee ordering, e.g. UDP over Ethernet, however this desirable feature can be added to the underlying communication method within the communication layer itself. All the complexity of adding and checking sequence numbers so that ‘future’ messages are buffered until previous messages arrive is then encapsulated and only affects the implementation of specific communication layer modules rather the design of protocol layer.

The unexpected data-store must be able to contain items that represent eager protocol messages and items that represent notification protocol messages. All objects in C# are strongly-typed, which means that the type information for all objects is known at the time of compilation of the code. Objects that refer to other objects must specify type information for the referenced objects. A particular consequence of this restriction is that all the objects referenced by (that is, stored in) a collection object must be of the same type. For example, a `System.Collections.Queue` object represents a first-in-first-out (FIFO) queue of objects of type `System.Object`. All objects in C# inherit from `System.Object`, so this queue can contain any type of object. However, the programmer is encouraged to be more specific about the type of object that can be contained in a collection by using generic classes, such as `System.Collections.Generic.Queue<T>` where `T` is the type of the object that can be contained in the queue. This means that eager and notification protocol messages should be represented either by the same class or by two classes that implement an interface or inherit from an abstract class, e.g. an initial protocol message class. This class hierarchy is shown in Figure 9, where all four specific protocol message classes inherit from a generic protocol message abstract class.

The communication layer delivers protocol messages from the protocol layer in one MPI process to the protocol layer in another MPI process without needing to know the specific type of each protocol message. This suggests that all four protocol messages should be represented either by the same class or by four classes that implement an interface or inherit from an abstract class, i.e. a protocol message class. Classes in C# can only inherit from one other class, so the initial protocol message abstract class should inherit from the protocol message abstract class. The bridge design pattern linking the protocol and communication layers (see section 3.2.3) calls for an abstract class from which all specifically types of protocol messages inherit rather than an interface or separate classes. The inheritance hierarchy of protocol message classes is not important to the bridge design pattern. Only the top-level `ProtocolMessage` abstract class contains code that interacts directly with the communication layer.

The pending and matched data-stores also require identical functionality to each other. The items must be easy to identify and remove but no particular ordering is required, so a data-structure representing an indexed collection of key-value pairs would be appropriate, such

as `System.Collections.Generic.Dictionary<Tkey, Tvalue>` with `Tkey` being the type of the key, i.e. the unique identifier of each item, and `Tvalue` being the type of the item itself. For the pending data-store the key will be a message sequence number and the value will be a transfer protocol message object that contains the data for the pending message. For the matched data-store the key will be a receive operation sequence number and the value will be a combination of a receive operation and a notification protocol message. To guarantee uniqueness, the sequence numbers used as keys must be generated on the machine that holds the data-store; this is why the pending data-store (which holds data for outgoing, locally created, messages) uses a message sequence number and the matched data-store (which holds locally created, receive operations – modified with information from an incoming notification protocol message) uses a receive operation sequence number.

The details of how the data-stores are declared, created and structured (and protected in a multi-threaded environment, see section 3.2.5) should be isolated from the rest of the system by encapsulating them and providing methods and properties to access the objects they contain. This can be achieved using the singleton design pattern [48]: the data-stores are held in a static class (i.e. a class where all its members must be static and which cannot be instantiated into objects because it does not contain an instance constructor). This is one of several variants of the singleton pattern; another variant involves a class with a private instance constructor (so that no object instances can be created) and static properties and methods; a third variant involves a class with a static factory method that creates an object instance, if one does not already exist, or returns the existing object instance, if one has already been created. The factory method variant is used when the singleton must implement an interface (which can only be achieved with instance members). It also allows lazy initialisation of the singleton object, i.e. its creation can be delayed until it is first needed. The private constructor variant is used when the singleton must inherit from a non-static base class. The static class variant is most appropriate here because the singleton does not need to inherit from a base class or implement an interface and lazy initialisation is not required. The static methods required in the data-store singleton class are:

- An `AddToPending` method, which adds a key-value pair to the pending data-store
- An `AddToMatched` method, which adds a key-value pair to the matched data-store

- A `RemoveFromPending` method, which removes and returns the value for a particular key from the pending data-store
- A `RemoveFromMatched` method, which removes and returns the value for a particular key from the matched data-store
- A `SearchUnexpected` method, which looks for an initial protocol message that matches the receive operation passed to it by sequentially checking each item in the unexpected data-store. If a matching message is found then it is removed from the unexpected data-store and returned. If not then the receive operation is added to the request data-store.
- A `SearchRequest` method, which looks for a receive operation that matches the initial protocol message passed to it by sequentially checking each item in the request data-store. If a matching request is found then it is removed from the request data-store and returned. If not then a second parameter, `probeOnly`, determines if the message should be added to the unexpected data-store, or not. The `probeOnly` parameter is included to support the two stage delivery semantics of large eager protocol messages, where the communication layer defers reading the data until the protocol layer supplies a receive buffer. If the message were added to the unexpected data-store unconditionally then it would be possible for an incomplete eager protocol message to be matched by another thread with a receive operation that would then return a corrupted receive buffer to the user.

### 3.2.5 The Lock Design Pattern

There are four levels of thread support specified in the MPI Standard: `MPI_THREAD_SINGLE`, `MPI_THREAD_FUNNELED`, `MPI_THREAD_SERIALIZED` and `MPI_THREAD_MULTIPLE`. It is assumed that McMPI will be running in a multi-threaded environment and may therefore be accessed by multiple threads, i.e. it must support the `MPI_THREAD_MULTIPLE` threading model. This requires that the four data-structures described in section 3.2.4 must be protected from potential corruption due to multiple threads accessing them simultaneously. In particular, write operations (such as add and remove) must not overlap with each other. In some cases, reading an element from a data-store may also fail if a write operation is in progress at the same time. If the maximum level of threading support required was any of the three lower levels then no protection would be necessary as no simultaneous access could take place.

The first step is to create a single static data-store for each purpose, i.e. one data-store for unmatched receive operations, one for unexpected initial protocol messages, one for pending messages, and one for matched receive operations. Each of these can then be protected by taking a mutual exclusion lock on a data-store before any operation that uses it and releasing that lock when the operation is complete. This conforms to the lock design pattern [48] and results in all accesses to each data-store being serialised. This approach has two obvious problems.

Firstly, having individual locks for the unexpected and request data-stores introduces the possibility of dead-lock: both locks can be taken by different threads that will then wait for each other to release the other lock. A thread that is responding to an incoming initial protocol message will first lock the request data-store (to look for a matching receive operation) and then lock the unexpected data-store (if there is no matching receive operation). A thread that is responding to a receive operation will first lock the unexpected data-store (to look for a matching initial protocol message) and then the request data-store (if there is no matching initial protocol message). The deadlock can be avoided by always taking both locks in the same order or by combining the two locks into a single lock, which is taken whenever either data-store is to be accessed.

Secondly, insisting that a lock be taken before any type of data-store access means that the lock must be taken even when it is actually not necessary: there may be no contention between threads or the operations may not logically depend on each other. For example, the messages in the pending message data-store do not logically depend on each other and are not ordered; the only reason for the lock is to maintain the integrity of the referencing system in the data-store itself. Thus, some read and write operations may be delayed until other unrelated operations complete.

The effects of serialising access to these data-stores might be alleviated by partitioning them to reduce the possibility for contention. There are several ways to partition these data stores, each with advantages and disadvantages. For example, each thread could have its own private pending data-store. This has the advantage that each of these pending data-stores is only accessed by one thread and only one operation at a time can be performed on each one so no lock is necessary. However, in a multi-threaded environment it cannot be guaranteed that an incoming approval protocol message will be handled by the correct

thread, i.e. by the thread that holds the appropriate transfer protocol message in its pending data-store. The disadvantage of switching thread far outweighs the advantage of avoiding the lock. As another example, there could be one partition of each data-store for each remote MPI process. This has the advantages that there may be fewer items in the data-store (so sequential searches are faster) and that a lock on one data-store does not need to block any other data-store (so multiple similar operations can proceed concurrently). The disadvantage of this partitioning technique is resource usage, specifically the number of locks needed to protect each partition of each data-store individually.

The data-store singleton class encapsulates all accesses to the data-stores and so it could easily be modified to use mechanisms of controlling concurrency, other than a mutual exclusion lock, without affecting the rest of the code.

### 3.2.6 Bypassing the Communication Layer for Local Delivery

In certain special-case situations, delivery of a message can be a local procedure (as defined in the MPI Standard, i.e. a procedure whose completion only depends on the local executing process). The special-case situations always include delivery to and from the null MPI process and delivery to and from the same MPI process that sent the message, i.e. self-delivery. Local delivery does not require communication with any other MPI process and can therefore entirely bypass the communication layer. For delivery to or from the null MPI process, the communication completes immediately and successfully with no work being done. For delivery to the same MPI process that sent the message, the send operation has direct access to the data-stores in the receiving MPI process and so it can perform the work in the data arrival functional unit. If the receive operation has already been initiated then the send operation can immediately complete the communication and mark both the send operation and the receive operation as complete. If the receive operation has not already been initiated then the send operation can immediately add the initial protocol message to the unexpected data-store in the receiving MPI process.

### 3.2.7 Extending the MPI Threading Model

The MPI Standard describes (in section 12.4) the requirements that should be met in order for an implementation of MPI to be thread-compliant. This description states that individual threads are not addressable by rank; a rank identifies a process and any thread within that process can receive messages sent to that rank. In the rationale for this design choice, an

alternative threading model is presented – where each MPI ‘process’ is a thread within a process. This “thread-as-rank” model is dismissed as being not thread-compliant, by definition; the MPI ‘processes’ are single-threaded.

However, the thread-as-rank threading model could instead be viewed as a possible extension to the MPI Standard – a fifth threading model (as discussed in section 2.2.3). From the perspective of users of the MPI library, the threading model would be similar to `MPI_THREAD_SINGLE`: only one thread would be addressed by each rank. From the perspective of the implementer of the MPI library, the threading model would be similar to `MPI_THREAD_MULTIPLE`: multiple threads can make MPI calls simultaneously requiring that internal data-structures be protected from race-conditions.

None of the existing MPI threading levels captures the full detail of the implications of assigning an individual rank to each thread. In particular, both the library user and the library implementer must write thread-safe code because each MPI ‘process’ will share static data-structures with other threads in the same process, i.e. with other MPI ‘processes’. Internally to the MPI library, most of the issues arising from the thread-as-rank threading model are already tackled by supporting the `MPI_THREAD_MULTIPLE` threading level. The internal data-structures are accessible by all threads in the process because they are implemented using the singleton design pattern (see section 3.2.4) and these data-structures are protected from multiple threads accessing them simultaneously by the lock design pattern (see section 3.2.5). Care is needed during initialisation of the MPI library so that each process reserves the correct number of ranks for the number of threads that will be assigned ranks, i.e. the number of MPI ‘processes’ that will be contained in that process. Thread-local storage must be used to identify the rank assigned to each thread – either by storing a unique thread identifier in thread-local storage (which can be used to look up the appropriate rank for a particular communicator) or by storing an array of ranks in thread-local storage (one per communicator). The first of these options is implemented in McMPI: the `McMPI.Init_thread` method (the C# binding for `MPI_INIT_THREAD`) sets a private thread-static variable, called `threadId`, by incrementing a private static variable, called `numThreads`, in a thread-safe manner. This implementation is shown in the following code extract, which also demonstrates the implementation for two methods defined by the MPI Standard: `McMPI.Initialised` and `McMPI.Is_Thread_Main` (the C# bindings for `MPI_INITIALISED` and `MPI_IS_THREAD_MAIN`).

```
public static class McMPI
{
  private static int numThreads = 0;

  private static bool isInitialised = false;

  [ThreadStatic]
  private static int threadId;

  [ThreadStatic]
  private static bool isInitialisedThread = false;

  // read-only access provided for the Communicator class
  internal static int ThreadId { get { return threadId; } }

  public static bool Initialised
  { get { return isInitialised; } }

  public static bool Is_Thread_Main
  { get { return isInitialisedThread; } }

  public static int Init_thread(string[] args, int required)
  {
    threadId = Interlocked.Increment(ref numThreads) - 1;

    if (required > (int)ThreadSupportLevel.THREAD_MULTIPLE)
      // create other local threads (implementation not shown)
    else
      maxThreads = 1;

    // initialise current thread (implementation not shown)

    isInitialisedThread = true;

    // wait for other local threads to enter Init_thread
    while (Thread.VolatileRead(ref numThreads) < maxThreads)
      Thread.SpinWait(10);

    isInitialised = true;

    if (required > (int)ThreadSupportLevel.THREAD_MULTIPLE)
      return (int)ThreadSupportLevel.THREAD_FILTERED;
    else
      return (int)ThreadSupportLevel.THREAD_MULTIPLE;
  }
}
```

Features from the `System.Threading` namespace (part of the .Net Framework and therefore accessible to all .Net languages, including C#) are used to aid thread-safe programming. The `ThreadStatic` attribute places the static variable in thread-local storage for each thread. The `Interlocked.Increment` method increments a variable and returns its new value as an atomic operation. The `Thread.VolatileRead` method assumes that the variable will be modified by other threads and so invalidates local cached copies before reading its value.

All threads will have a `threadId` of `0` (zero) – unless they call `McMPI.Init_thread` after at least one other thread has also called `McMPI.Init_thread`. Therefore, this implementation will appear to be identical to `MPI_THREAD_MULTIPLE` when there is only one initialised thread per operating system process: all threads in the process will have the same rank as each other.

Supporting the new `MPI_THREAD_AS_RANK` and `MPI_THREAD_FILTERED` threading support levels, proposed in section 2.2.3, allows all MPI 'processes', i.e. threads, in the same process to use the same local delivery mechanism as self-delivery. It is hoped that one of the strengths of McMPI will be its ability to deliver thread-to-thread messages via shared variables, i.e. completely within the process that hosts both threads rather than resorting to out-of-process shared memory, which would cross the process boundary twice.

### 3.2.8 Envelope Matching

There are two main reasons for the design of McMPI to include an `Envelope` class. Firstly, there is sufficient similarity between the envelope information for send and receive operations to justify using the same storage structure. Secondly, encapsulating the storage and processing of envelope information allows the implementation of the matching algorithm to be changed without affecting the rest of the code.

All send and receive operations defined in the MPI Standard require the user to specify some pieces of envelope information – a communicator, a source rank, a target rank and a tag. For a send operation, all four values are available – the source rank is implicitly determined by the identity of the sending MPI process and the other values are supplied in the send function call. For a receive operation, only communicator and target rank (implicitly determined by the identity of the receiving MPI process) are definitely available.

The source rank and tag can be special wildcard values, as in the C and FORTRAN language bindings, or omitted entirely, as in the C++ language bindings.

Messages are selectively received by stipulating that the receive operation envelope must match the send operation envelope, i.e. the communicator and target rank must be identical and, if specified by the receive operation, the source and tag must also match. In recognition of the fact that the collective operations defined by the MPI Standard can be built from point-to-point messaging – and to include a degree of future-proofing – the `Envelope` class also includes an indication of the type of collective operation (zero for "not collective", i.e. point-to-point, with all other values reserved for future use). The following code shows the declarations of the fields within the `Envelope` class:

```
public partial class Envelope
{
  protected CommunicatorId communicatorId;
  protected Rank targetId;
  protected CollectiveConstants collectiveType;
  protected WildcardConstants wildcard;
  protected Rank sourceId;
  protected int tag;
}

public partial class Request : Envelope { }
```

The fields are declared as `protected` in order that the `Request` class will inherit them and be able to use them, for example to provide status information when the operation is complete. The non-blocking send and receive operations return a `Request` object, defined by this `Request` class.

When searching the request queue for a `Request` object that matches the `Envelope` object from an incoming message, each pairing must be checked for compatibility. Similarly, when searching the unexpected queue, a `Request` object must be checked against the `Envelope` object from each of the unmatched initial protocol messages. These can both be achieved by a `Matches` function on the `Envelope` class, as follows:

```
public partial class Envelope
{
  static bool Matches(Envelope message, Request request)
  {
```

```
    if (message.targetId != request.targetId)
      return false;
    if (message.communicatorId != request.communicatorId)
      return false;
    if (message.collectiveType != request.collectiveType)
      return false;
    if ((request.wildcard & WildcardConstants.SourceWildcard)
        != WildcardConstants.SourceWildcard)
      if (message.sourceId != request.sourceId)
        return false;
    if ((request.wildcard & WildcardConstants.TagWildcard)
        != WildcardConstants.TagWildcard)
      if (message.tag != request.tag)
        return false;
    return true;
  }
}
```

This function is small, self-contained and used frequently, so it is a good candidate for optimisation, in particular using an intermediate language dis-assembler tool, such as ILDasm.exe, which is supplied with the Microsoft Windows Software Development Kit.

The Common Intermediate Language (CIL) code, produced by compiling and then dis-assembling this C# code, repeatedly includes the following two instructions:

```
  ldc.i4.0
  ret
```

The first of these loads a constant, which is a 4-byte signed integer, with a value of zero. The second returns control from the function to the calling code. Together they are the translation of the repeated "`return false`" statements in the C# code. All but one of these repetitions could be removed by combining all the separate conditional expressions into a single conditional expression as follows:

```
public partial class Envelope
{
  static bool Matches(Envelope message, Request request)
  {
    if ((message.targetId != request.targetId) ||
        (message.communicatorId != request.communicatorId) ||
        (message.collectiveType != request.collectiveType) ||
        ((request.wildcard & WildcardConstants.SourceWildcard
                           != WildcardConstants.SourceWildcard)
```

```
          && (message.sourceId != request.sourceId)) ||
         ((request.wildcard & WildcardConstants.TagWildcard
                            != WildcardConstants.TagWildcard)
          && (message.tag != request.tag)))
       return false;
     return true;
   }
}
```

This version performs the same task and the CIL code size is reduced to 98 bytes from 106 bytes but the C# code is less readable. A simple test program that times a million calls to the function shows that the runtime performance of the second version is better than that of the first. On a computer with 2.5GHz processors (the server test system described in section 4.3.1), the time-per-call for the first version varies between 5.2ns (when the target ranks do not match) and 9.5ns (when both source and tag are specified in the `Request` object and the two envelopes match). On the same test system, the time-per-call for the second version varies between 1.7ns and 5.2ns. In both cases, the time-per-call of the function varies depending on which fields fail to match, if any, because both versions use short-circuit logic. For example, if the target ranks do not match then both functions return false without checking the remaining fields because the result of the function is already known.

The second version can be improved slightly further by noticing that the bitwise-and "`&`" operations on the wildcard values cause unnecessary implicit conversion instructions `conv.u1` to be included in the CIL. In addition, the comparison of the result of the bitwise-and to the wildcard constant causes an unnecessary load instruction `ldc.i4.2` to be included in the CIL.

```
ldfld valuetype WildcardConstants McMPI.Envelope::wildcard
ldc.i4.2
and
conv.u1
ldc.i4.2
beq.s
```

The conversion is included because the bitwise-and instruction only operates on 4-byte integer values and produces a 4-byte integer result. This is then converted to an unsigned byte because the `WildcardConstants` enumeration inherits from `byte` so that is the actual type of the two operands in the C# code. The conversion is unnecessary because the

unsigned byte is implicitly converted back to a 4-byte integer by the conditional branch instruction `beq.s` that uses it. It can be avoided by allowing the compiler to treat the input values as 4-byte integers (by casting to `int` in the C# code) so that the result is allowed to remain as a 4-byte integer.

The second load of the constant "2" (the actual value represented by the enumeration constant `WildcardConstants.SourceWildcard` in the C# code) is unnecessary because, if the result of the bitwise-and operation is not "2", then it must be zero. The actual values of the wildcard enumeration constants are chosen so that they represent a bitmask with only one bit set. A bitwise-and operation with one operand having only a single bit set will evaluate to zero, if that bit is not set in the other operand, or the same non-zero value as the bitmask. Thus, because of the careful choice of the actual enumeration values, the C# code can be changed to "`==0`", which removes the need to load the constant a second time. The enumeration is defined internally to the library code and is marked with the C# `flags` attribute, which indicates that the bit-pattern of the values is important and allows the compiler to perform bitwise operations on them.

Thus, in the third version of the C# code for the `Matches` function, the conditions involving source and tag are changed to the following:

```
((((int)request.wildcard &
   (int)WildcardConstants.SourceWildcard) == 0)
  && (message.sourceId != request.sourceId)) ||
((((int)request.wildcard &
   (int)WildcardConstants.TagWildcard) == 0)
  && (message.tag != request.tag)) ||
```

The CIL produced by compiling and then dis-assembling this version is smaller, with a code size of 94 bytes, and is faster, with a time-per-call varying between 1.3ns and 5.2ns.

If the storage of the information inside the `Envelope` object is changed, it is possible to create a fourth version of the `Matches` function as follows:

```
public partial class Envelope
{
  private long firstChunk;
```

```
  private long secondChunk;
  private long bitMask;

  static bool Matches(Envelope message, Request request)
  {
    if ((message.firstChunk != request.firstChunk) ||
        ((message.secondChunk & request.bitMask)
         != request.secondChunk))
      return false;
    return true;
  }
}
```

The `firstChunk` field is an 8-byte signed integer that contains the collective type, the communicator identifier and the target rank (in bytes 2, 3-4 and 5-8, respectively – the first byte is not used). The `secondChunk` field contains the source rank and the tag (in bytes 1-4 and 5-8 respectively) but is combined with the `bitMask` field before storing. The `bitMask` field contains a bitmask that determines which bytes in the `secondChunk` field will be significant for comparisons. If the `Envelope` object represents envelope information for a send operation then both source and tag must be specified and the bitmask will select all bytes, i.e. it will have all bits set to 1. If the `Envelope` object represents a receive operation and source is not specified then the bitmask must not select the first four bytes, i.e. every bit of the first four bytes of the bitmask will contain 0. Similarly, if the tag is not specified the bitmask must not select the last four bytes, i.e. every bit of the last four bytes of the bitmask will contain 0.

For two envelopes to match, the `firstChunk` fields must match exactly because all three of the constituent values must match exactly. In addition, the significant parts of the `secondChunk` fields – as selected by the `bitMask` fields – must match exactly.

This version almost completely obscures the meaning of the operation but it produces CIL with a code size of 39 bytes and its time-per-call varies between 2.0ns and 2.9ns.

### 3.2.9 Message Header

The header information for protocol messages must be copied from the variables that hold the strongly-typed information into a byte array suitable for transmission by the communication layer. The header information is stored in, and copied from, an `Envelope`

object created by the send operation. The byte array is created by, and accessed via, the container object provided by the appropriate communication module for the destination.

In C#, the standard way to copy information from an object to a byte array is serialisation, which is performed by a formatter object and outputs a standard stream that completely describes the layout of the object to be serialised including all the data stored by that object. The `BinaryFormatter` object is the most efficient of the formatter objects provided in C# because it produces a binary stream, i.e. a stream of bytes, rather than a stream of XML, for example. The binary stream from the formatter object can be written to a pre-existing byte array using a `MemoryStream` object. Serialising an `Envelope` object by wrapping the target byte array with a `MemoryStream` object and using a `BinaryFormatter` object to perform the serialisation, outputs 251 bytes (of which 15 bytes are the data from the fields in the `Envelope` object and 236 bytes are the detailed type information needed to reconstruct the object) and takes 19920.6ns (an average of one million calls to the serialisation method).

Another approach that uses streams is to attach the `MemoryStream` object to a `BinaryWriter` object. This allows only the data from the fields in the `Envelope` object to be written, resulting in 15 bytes being written in 438.7ns.

There are several other ways to implement the copy operation that do not use streams.

The `GetBytes` method from the `BitConverter` class converts a single variable with a primitive C# type to an array of bytes, which it returns. The `BlockCopy` method from the `Buffer` class can then be used to copy the contents of each of these small byte arrays into the target byte array. This method results in 15 bytes being written in 165.6ns.

The data can be copied into the target array without using built-in objects to help: the value of each byte can be calculated with bitwise operations and type-casting. The collective type variable is already a single byte so the first available byte in the target array can be set directly. The communicator identifier is a 2-byte integer. Casting this to a byte retrieves the least significant byte, which can then set into the target array. Bitwise shifting the communicator identifier to the right by eight bits and then casting to a byte retrieves the most significant byte, which can then be set into the target array. The same “shift right and

cast to byte" approach can be used for the other variables. This method results in 15 bytes being written in 13.3ns.

This research is focused on using safe code, i.e. code that can be verified as type-safe via compile-time analysis. However, C# provides mechanisms for including unsafe code, such as obtaining the memory address for a variable, performing pointer arithmetic, casting pointers and de-referencing pointers. To operate on managed variables in an unsafe manner, the memory management services must be temporarily suspended for that variable, i.e. that variable must be pinned to a particular memory location using the `fixed` statement. It is also possible to dictate the memory layout of the fields in an object by specifying a byte-offset for each field. These features of C# suggest two alternative implementations, called `WriteBytes1` and `WriteBytes2` in the code below.

```
partial class Envelope
{
  [StructLayout(LayoutKind.Explicit)]
  partial struct EnvelopeHelper
  {
    [FieldOffset(1)]
    private CommunicatorId communicatorId;
    [FieldOffset(3)]
    private Rank targetId;
    [FieldOffset(0)]
    private CollectiveContants collectiveType;
    [FieldOffset(7)]
    private Rank sourceId;
    [FieldOffset(11)]
    private int tag;

    unsafe void WriteBytes(byte[] target, ref int offset)
    {
      fixed (byte* pTarget = &target[offset])
      {
        *(EnvelopeHelper*)pTarget = this;
      }
      offset += 15;
    }
  }
  unsafe void WriteBytes1(byte[] target, ref int offset)
  {
    (new EnvelopeHelper(this)).WriteBytes(target, ref offset);
  }
```

```
  unsafe void WriteBytes2(byte[] target, ref int offset)
  {
    fixed (byte* pTarget = &target[offset])
    {
      *pTarget = (byte)collectiveType;
      *(short*)(pTarget + 1) = (short)communicatorId;
      *(uint*)(pTarget + 3) = (uint)targetId;
      *(uint*)(pTarget + 7) = (uint)sourceId;
      *(int*)(pTarget + 11) = tag;
    }
    offset += 15;
  }
}
```

In the code listing above, the `WriteBytes1` method from the `Envelope` class creates an `EnvelopeHelper` structure and delegates the task of copying data to it. This delegation is necessary because all pointers in C# point to the memory address of a structure, not an object. The `WriteBytes` method from the `EnvelopeHelper` structure pins the target byte array into memory by creating a byte-pointer to one of its elements in a `fixed` statement. It then casts that byte-pointer so that it points to an `EnvelopeHelper` structure, de-references it and sets it to the current structure (denoted by "`this`" in the code). The assignment operator performs a byte-by-byte copy of the memory referenced by the right operand into the memory referenced by the left operand. This results in 15 bytes of envelope information being written in 21.9ns.

The second unsafe implementation demonstrated in the code listing above, uses the `WriteBytes2` method from the `Envelope` class. This method pins the target byte array in the same way as before but then copies each field individually. For each envelope field, the byte-pointer to the first available byte in the target array (`pTarget`) is incremented by the appropriate number of bytes, cast to the appropriate type, de-referenced and set to the value of that field. This results in 15 bytes of envelope information being written in 6.9ns.

The implementation chosen for McMPI is the "shift right and cast to byte" method. This implementation is used, not just for the `Envelope` class, but throughout the code, wherever strongly-typed variables must be converted to or from bytes. The primary reason for this choice is that the byte order for transmission can be controlled and standardised, even when the memory architectures of the source and target machines are different, i.e.

even when one is little-endian and the other is big-endian. In addition, it is the fastest of the implementations that use safe code and its performance is of the same order as the implementations that use unsafe code.

## 3.3 The Interface Layer

The intention for McMPI is to be an MPI-like library. In particular, it is intended that the syntax used in the external API is recognisable as being based on the functions defined in the MPI Standard. There is no language binding for C# defined in the current version of the MPI Standard, although the deprecated C++ bindings can be used as a helpful guide. In general, where there is a direct conflict between the MPI Standard syntax and the standard syntax found in C#, the MPI syntax has been chosen for McMPI.

Section 3.3.1 documents the object model that is exposed to code outside McMPI.

Section 3.3.2 describes the conflict and the chosen resolution between how asynchronous operations are syntactically presented in the MPI Standard and standard C#.

Section 3.3.3 examines in detail the implementation of the `Request` object.

Section 3.3.4 discusses the implementation of communicators and ranks, which are defined by the MPI Standard, using locations, which are defined by the `Location` class described in section 3.1.2.3.

### 3.3.1 The Façade Design Pattern

The protocol layer is designed to support the core functionality of point-to-point message passing, as defined in the MPI Standard. It is a fully functional communication library but its internal interface – the `ProtocolMessage` classes (section 3.2.3) and the `Location` class (section 3.1.2.3) – is very different to the functions defined in the MPI Standard, even in the C++ bindings. The requirement for the interface layer is to present an MPI-like API to users of the McMPI library and translate between that API and the interface presented by the protocol layer. This design follows the façade design pattern [48].

The object model that forms the API is similar to the C++ bindings in the MPI Standard and contains the following objects and classes:

- The `Communicator` class contains static field that returns a `Communicator` object called `World`, which contains, and assigns a rank to, all the MPI processes accessible by the currently running parallel program. It is marked as a `partial` class in order that, in future, further methods can be separately coded, compiled and included without modifying the code in this class (see section 6.2 for suggested further work).
- The `Communicator` objects contain instance methods that perform communication operations, which comprise three of the four blocking send modes `MPI_SEND`, `MPI_RSEND`, and `MPI_SSEND`, their non-blocking equivalents `MPI_ISEND`, `MPI_IRSEND`, and `MPI_ISSEND`, and their persistent equivalents `MPI_SEND_INIT`, `MPI_RSEND_INIT`, and `MPI_SSEND_INIT`, as well as methods for blocking, non-blocking and persistent receive operations `MPI_RECV`, `MPI_IRECV`, and `MPI_RECV_INIT`.
- The `Request` objects returned by the non-blocking communication methods on the `Communicator` class are defined by the `Request` class and include `MPI_WAIT` and `MPI_TEST`.
- The `PersistentRequest` objects returned by the persistent communication methods on the `Communicator` class are defined by the `PersistentRequest` class and include `MPI_START` and `MPI_STARTALL`.

### 3.3.2 Presenting Asynchronous Operations in the API

There is a standard syntax pattern in C# for asynchronous methods: in addition to the synchronous method, e.g. `Example`, two new methods, prefixed with `Begin` and `End`, are provided, e.g. `BeginExample` and `EndExample`. The `Begin`-prefixed method returns an object (which implements the `IAsyncReturn` interface) that must be supplied to the `End`-prefixed method. Section 3.1.1.3 analyses a practical example of the standard C# syntax pattern for asynchronous methods including the similarities to, and the differences from, the MPI Standard syntax for non-blocking functions. The C# standard syntax requires resources to be allocated for the `IAsyncReturn` object even if it carries no information useful to the user. It also requires a separate method call for completing each operation. The MPI Standard requires resources to be allocated for the `Request` structure but mitigates this overhead by allowing persistent requests. It also requires a function call for completing each operation but mitigates this overhead by providing completion functions that complete multiple operations in one function call.

The solution adopted is to follow the MPI Standard syntax pattern for non-blocking functions when defining asynchronous methods in McMPI. For example, in the MPI Standard the `MPI_ISSEND` function returns a request structure that may be passed as a parameter to the `MPI_WAIT` or `MPI_TEST` functions. Thus, in McMPI the `Issend` method returns a `Request` object that may be passed as a parameter to the `Wait` or `Test` methods. Similar to the C++ bindings in the MPI Standard, in McMPI the `Issend` method is an instance method of the `Communicator` object and the `Wait` and `Test` methods are instance methods of the `Request` object. The `Request` class also contains static methods for completing or testing multiple requests in a single call, e.g. `Waitall` and `Testall`.

### 3.3.3 Requests

Internally, the `Request` object in McMPI uses an operating system thread synchronisation object, a `System.Threading.ManualResetEvent` object, to implement waiting and testing for completion of a non-blocking operation. The `ManualResetEvent` class exposes a `WaitOne` instance method, which performs the task required by the `MPI_WAIT` method. It also exposes static methods, `WaitAll` and `WaitAny`, which perform the tasks required by `MPI_WAITALL` and `MPI_WAITANY`, respectively. Testing for completion can be achieved by supplying a timeout value of zero to the appropriate waiting method.

If the `Request` class inherits from the `ManualResetEvent` class then an array of `Request` objects can be used unchanged and passed directly to the static `WaitAll` and `WaitAny` methods. However, this exposes the `Request` objects to unwanted manipulation by external code: a `Request` object could be passed to the `SignalAndWait` static method inherited from the `ManualResetEvent` class, which would signal waiting threads that the operation represented by the `Request` object is complete. For a non-persistent request this may lead to one communication operation being corrupted, either because a send buffer is written to by external code before it is fully read from and sent by McMPI or because a receive buffer is read from by external code before it is fully received and written to by McMPI. A persistent request becomes inactive when an operation completes and may be started again. If a `PersistentRequest` object (discussed in section 3.3.3.3), derived from the `ManualResetEvent` class via the

`Request` class, were to be signalled prematurely then multiple communication operations could be corrupted.

If the `Request` object holds a reference to a `ManualResetEvent` as an instance field value then the methods that wait or test for completion of multiple requests must build an array of `ManualResetEvent` object references by extracting the values of this field from each `Request` object. This adds a small amount of processing time to every multiple-request completion method call but avoids the potential for unwanted manipulation. It also allows the reference to the operating system thread synchronisation object to be released before the `Request` object is destroyed, which may assist with garbage collection, especially of critical system resources.

Thus, the choice made for McMPI is that `Request` objects hold a reference to their `ManualResetEvent` object in an instance field.

#### *3.3.3.1 Freeing Resources*

In McMPI, there is no need for the `MPI_REQUEST_FREE` function that is defined in the MPI Standard, neither in the `Request` class nor anywhere else. This is because the aim of that function is achieved through the garbage collection service in C#. Any object in C# that is no longer referenced by active code is marked as ready for garbage collection, i.e. its resources can be freed and reused. A background service in the common language runtime monitors the pressure on resources, such as memory, and collects garbage either when the application is idle or when the application is close to exhausting a system resource. To use this service efficiently, care must be taken to release references to unused objects as soon as possible. Within McMPI, when the operation represented by a `Request` object is complete, a single reference to it is returned to the user and all internal references to it are released. Its lifetime is then determined by the user and the garbage collection service. Instead of the user calling a method to mark the request as ready to be freed, i.e. an equivalent to the `MPI_REQUEST_FREE` function, the user needs only to set any in-scope references to the `Request` object to `null`.

#### *3.3.3.2 Status*

In McMPI, there is also no need for an equivalent to the `Status` structure that is defined in the MPI Standard. This is because the information it should contain is accessible directly from the `Request` object. The main reason for combining the `Request` and `Status`

structures into a single `Request` object is that, although advantage might be gained by keeping them separate in languages like C and FORTRAN, keeping them separate in C# is more likely to be a disadvantage.

The MPI Standard specifies the C and FORTRAN bindings so that the `Status` structure is optional and, when used, the memory address of a pre-existing structure is always passed as an argument. This allows an MPI library to avoid allocating memory for `Status` structures entirely and, conditionally, to avoid processing time for setting `Status` values.

In C#, a separate status could be an object or a structure. Structures in C# are held and used "by value" not "by reference": a variable with a structure type contains the data values for that type rather than a reference to the memory location of the data values, as for an object. The memory allocated for a structure-typed variable, enough to hold all the constituent values, is allocated when the variable comes into scope, i.e. at the point of its declaration. When passed as a normal argument to a method, the entire contents of the input structure is copied into a local variable, which is an entirely separate structure. When passed by reference, the input structure is "boxed", i.e. an object is created to wrap the structure and a reference to that object is passed to the method. Thus, every method call involving a structure incurs either a structure-allocation cost or an object-creation cost.

The option of a separate status object is also unappealing. All objects incur an object-creation overhead, both processing time and memory for the object itself. One combined object will incur lower overheads, both processing time and memory, than two objects that, between them, contain the same data values as the combined object. It is possible to define a `RequestWithStatus` class, which inherits from a `Request` class that does not contain status information. However, this introduces other problems. In order to return a `RequestWithStatus` object (when the status will be required later) but a `Request` object (when the status will not be required), two separate methods must be declared. Because the signature of overloaded methods in C# cannot vary only by return type, these two methods must differ by more than just the return type, i.e. either the names of the methods must be different or the argument list must be different. This unnecessarily complicates the API.

Another problem arises when the internal code is choosing whether or not to set the status information. The data-stores hold `Request` objects; the library code would not know

whether a particular `Request` object is actually a `RequestWithStatus` object unless it uses reflection, i.e. unless it examines the meta-data for the object, which is a relatively slow operation. A better solution would be to include a `SetStatus` method in the `Request` class, which is over-ridden in derived classes, such as `RequestWithStatus`. The `SetStatus` method in the `Request` class would do no work but the `SetStatus` method in the `RequestWithStatus` class would set the status information fields. This approach requires that the `SetStatus` method in the `Request` class has the same arguments as the over-ridden method and so the `Request` class must contain information that should be encapsulated within the `RequestWithStatus` class.

The problems with the design and implementation of a separate status structure or object, in conjunction with the likelihood that no significant advantage would be gained, confirms that the best choice is to combine the status information with the `Request` class and to set the status field values unconditionally for all `Request` objects.

#### *3.3.3.3 Persistent Requests*

In the MPI Standard, the functions that create a persistent communication operation return a `Request` structure in the C and FORTRAN bindings but a `Prequest` structure in the C++ bindings. Both can be achieved in McMPI by deriving the `PersistentRequest` class from the `Request` class so that a `PersistentRequest` object can be used wherever a `Request` object is required. It can also provide additional functionality, such as the `Start` method (the C# binding for the `MPI_START` function), which initiates the persistent communication operation. In particular, some or all of the requests supplied to the static completion functions in the `Request` class, e.g. `Waitall`, can be `PersistentRequest` objects rather than `Request` objects. In addition, the `Wait` and `Test` instance methods are inherited from the `Request` class and apply to `PersistentRequest` objects without needing extra code in the `PersistentRequest` class.

### 3.3.4 Communicators, Ranks and Locations

In the MPI Standard, the destination for a send operation is specified by a communicator and a rank. In McMPI, the protocol layer distinguishes possible destinations by their location, represented by a `Location` object (discussed in section 3.1.2.3).

In both the C++ language bindings in the MPI Standard and the proposed C# bindings in the McMPI API, the communicator is the object that executes the send method call and the rank is passed as an integer argument. The `Communicator` object must store enough information to create or retrieve the correct `Location` object so the protocol layer can use it to inform the communication layer where to deliver the message.

The simplest approach is for the `Communicator` object to store an array of `Location` object references, with the index in the array indicating the rank of the MPI process at that location. In McMPI, duplicates are allowed so that one location can support multiple MPI processes, e.g. the location can be an operating system process containing multiple threads, each of which is a single-threaded MPI process. This requires memory to store the array of `Location` object references, which will increase proportionally with the size of the system, i.e. the number of MPI processes. In addition, forming a new communicator, e.g. by splitting an existing one, requires the creation of a new 1-D array and copying some or all of the location references from the existing communicator to the new one. This will scale proportionally with the size of the communicator.

Usually, the communicators required by a code are created during the initialisation phase at the beginning of the program although, rarely, new communicators may be created repeatedly during the execution of the program, for example when re-distributing the data for a more efficient data-decomposition strategy. Initialisation usually involves reading data from permanent storage media such as hard disk drives and distributing the data throughout the system, which are very slow operations compared with a memory copy. A dynamic re-distribution of the data during the main execution loop is a very expensive operation and would only be attempted if the benefit of the new data-decomposition were expected to outweigh the cost. Therefore, the disadvantage that the time taken to create a new communicator depends on the size of the system is mitigated by the infrequency of this operation.

### 3.3.5 The API for MPI Functions Implemented by McMPI

The public interface for the MPI functions defined and implemented in McMPI is shown in the following C# code, which has been extracted from the McMPI source code.

```
using System;

namespace System.HPC.MPI
{
 public static class MPI
 {
  static bool Initialised { get; }

  static bool Is_thread_main { get; }

  static void Init       (string[] args);
  static int  Init_thread(string[] args, int required);

  static void Finalise();
 }

 public class Communicator
 {
  // provision of default communicators
  static Communicator Null  { get; }
  static Communicator Self  { get; }
  static Communicator World { get; }

  // communicator properties
  Rank Rank { get; }
  Rank Size { get; }

  // blocking sends: standard, ready and synchronous modes
  Request  Send(ArraySegment<byte> data, Rank dest, int tag);
  Request Rsend(ArraySegment<byte> data, Rank dest, int tag);
  Request Ssend(ArraySegment<byte> data, Rank dest, int tag);

  // non-blocking sends: standard, ready and synchronous modes
  Request Isend (ArraySegment<byte> data, Rank dest, int tag);
  Request Irsend(ArraySegment<byte> data, Rank dest, int tag);
  Request Issend(ArraySegment<byte> data, Rank dest, int tag);

  // persistent sends: standard, ready and synchronous modes
  PersistentRequest  Send_init(
    ArraySegment<byte> data, Rank dest, int tag);
  PersistentRequest Rsend_init(
    ArraySegment<byte> data, Rank dest, int tag);
  PersistentRequest Ssend_init(
    ArraySegment<byte> data, Rank dest, int tag);

  // blocking receives: with and without source and tag
  Request Recv(ArraySegment<byte> data                  );
```

```
 Request Recv(ArraySegment<byte> data, Rank src         );
 Request Recv(ArraySegment<byte> data,           int tag);
 Request Recv(ArraySegment<byte> data, Rank src, int tag);

 // non-blocking receives: with and without source and tag
 Request Irecv(ArraySegment<byte> data                  );
 Request Irecv(ArraySegment<byte> data, Rank src        );
 Request Irecv(ArraySegment<byte> data,           int tag);
 Request Irecv(ArraySegment<byte> data, Rank src, int tag);

 // persistent receives: with and without source and tag
 PersistentRequest Recv_init(
   ArraySegment<byte> data                  );
 PersistentRequest Recv_init(
   ArraySegment<byte> data, Rank src        );
 PersistentRequest Recv_init(
   ArraySegment<byte> data,           int tag);
 PersistentRequest Recv_init(
   ArraySegment<byte> data, Rank src, int tag);
}

public class Request
{
 // status information
 Communicator Communicator { get; }
 Rank         Source       { get; }
 int          Tag          { get; }
 int          Count        { get; }

 // completion operations
 void Wait();
 bool Test();
}

public class PersistentRequest : Request
{
 // activation operations
 static void Startall(
   IEnumerable<PersistentRequest> requests);
 void Start();
}

// data type alias
public enum Rank : uint { }
}
```

In general, this API follows the C++ bindings defined in the MPI Standard as closely as possible for the C# language. Some notable exceptions to this rule are as follows:

- The `MPI` namespace is nested within the existing namespace hierarchy defined by standard .Net classes.
- The pre-defined communicators are declared as static members of the `Communicator` class rather than constants within the `MPI` namespace because C# does not allow declarations of constants to appear directly within a namespace.
- The C# bindings for the blocking receive operation `MPI_RECV` (i.e. the `Recv` methods) return a `Request` object so that the status information (which has been combined with the `Request`) can be retrieved. This `Request` object will be complete when the `Recv` method returns control: a call to `MPI_WAIT` will return immediately and `MPI_TEST` will return `true`.
- The data type of all send and receive buffers is `ArraySegment<byte>` rather than an un-typed pointer, e.g. `void*`, or an un-typed object reference, e.g. `Object`. Section 2.4.3 discusses this change in detail.

## 3.4 A Review of the Design and Implementation

This section reviews the design and implementation of McMPI by following a send operation and a receive operation step-by-step from the initial calls by user code until the return of the completion notifications to user code.

The example send operation presented in this section is a non-blocking synchronous-mode send or, equivalently, a non-blocking standard-mode send of a large message. Both of these send operations call for a rendezvous protocol message sequence, i.e. a notification protocol message is transmitted and, once an approval protocol message is received, a transfer protocol message is transmitted. The message is received by a non-blocking receive operation in a different operating system process, which forces the use of the socket-based communication module. The send operation is “late”, as defined in section 3.2.1 by the “Late Send” scenarios.

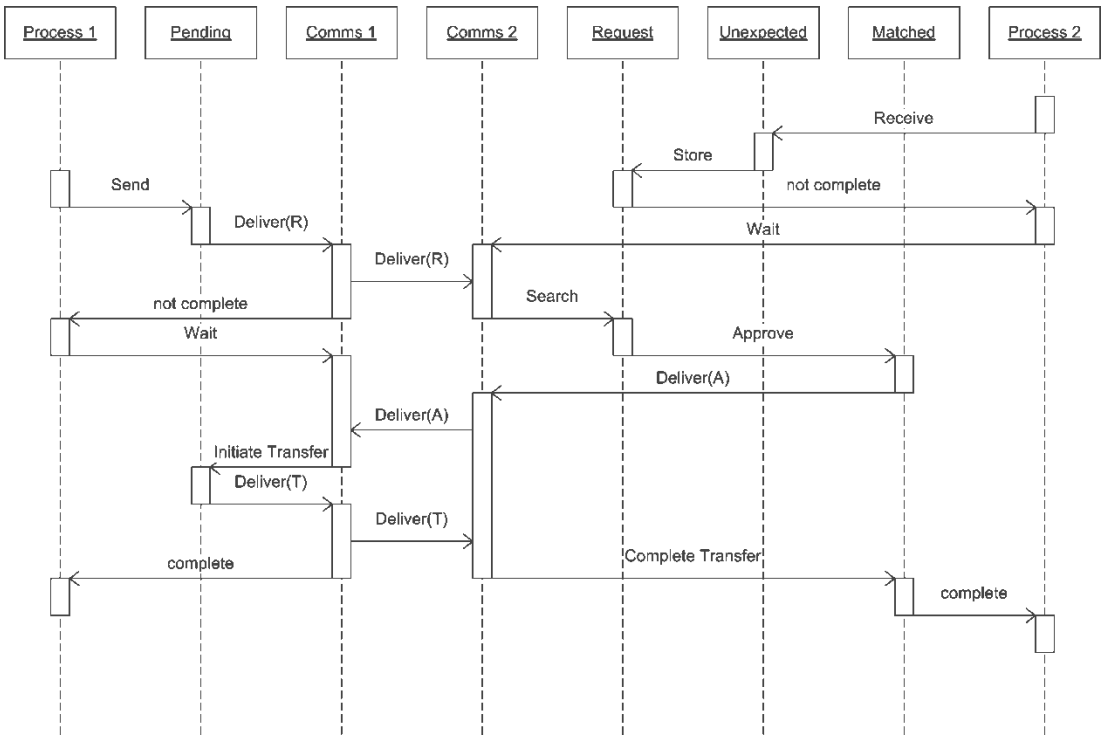


**Figure 10: UML sequence diagram for process-to-process rendezvous protocol with a late send.**

The receiving MPI process (shown in Figure 10 as "Process 2") issues a call to the `Irecv` method from a particular `Communicator` object. That method creates a `Request` object from the envelope information and the user receive buffer supplied as arguments. It then searches the unexpected messages data-store, discovers there is no matching initial protocol message and stores the `Request` object in the `Request` data-store. It then returns a reference to the `Request` object back to user code.

The `Request` object can be tested by calling its `Test` method, which will test the internal `ManualResetEvent` object and return `false` because the operation has not yet completed. Alternatively, as shown in Figure 10, the user code can wait for the receive operation to finish by calling the `Wait` method on the `Request` object, which will wait for the internal `ManualResetEvent` object to be set to the signalled state.

The sending MPI process (shown in Figure 10 as "Process 1") issues a call to the `Issend` method from a particular `Communicator` object. The `Issend` method creates a `Request` object from the envelope information and the user send buffer supplied as arguments. It creates a `NotificationProtocolMessage` object, passing the

`Request` object as an argument, stores it in the pending data-store, calls its `Deliver` method, and returns a reference to the `Request` object back to user code.

The `Request` object can be tested by calling its `Test` method, which will test the internal `ManualResetEvent` object and return `false` because the operation has not yet completed. Alternatively, as shown in Figure 10, the user code can wait for the send operation to finish by calling the `Wait` method on the `Request` object, which will wait for the internal `ManualResetEvent` object to be set to the signalled state.

The constructor for the `NotificationProtocolMessage` object inherits from the constructor for the `InitialProtocolMessage` class, which in turn inherits from the `ProtocolMessage` constructor. Together these constructors determine the `Location` object that represents the destination for the message, obtain a container object from the appropriate communication module and customise that container by writing the envelope information from the `Request` object into the header byte array and setting a reference to the user send buffer.

The `Deliver` method from the `NotificationProtocolMessage` object is inherited from the `ProtocolMessage` class, which delegates to the `Deliver` method of the container object. The container is a `TCPSocketContainer` object. Its `Deliver` method uses a `Socket` object (provided by the `TCPSocketModule` object when it created this container) to transmit the header and data buffers to the communication layer of the destination process.

During initialisation of the `TCPSocketModule` in each operating system process that uses McMPI, when the socket connections between operating system processes are being created, a `TCPSocketContainer` object is also created for each socket. Each of these containers is used to receive protocol messages for a particular `Socket` object. The `TCPSocketModule` calls the `Receive` method on each of these `TCPSocketContainer` objects. The `Receive` method uses the `Socket` object to receive protocol messages from the communication layers of other processes. When data is received, the `TCPSocketContainer` calls the `Receive` method again, so that a subsequent protocol message can be received.

The NotificationProtocolMessage transmitted by the communication layer of the sending process is received by one of these receiving TCPSocketContainer objects in the communication layer of the receiving process. When it arrives a call-back method in the TCPSocketContainer object is called, which in turn calls the IncomingContainer method from the ProtocolMessage class and then calls the Receive method from the container.

The IncomingContainer method creates a protocol message using the information in the container object passed as an argument and calls the LocalDeliver method from the newly created NotificationProtocolMessage object.

The LocalDeliver method from the NotificationProtocolMessage object searches the Request data-store for a matching Request object. In this example, a matching Request object is found and removed from the Request data-store. The LocalDeliver method calls the SetStatus method from the Request object (passing it the envelope information from the NotificationProtocolMessage) and adds the Request object to the matched data-store. It then gives permission to the sending process to transfer the data by creating an ApprovalProtocolMessage object and calling its Deliver method, which proceeds exactly as for the NotificationProtocolMessage (except in the opposite direction).

When a copy of the ApprovalProtocolMessage is created by the IncomingContainer method from the ProtocolMessage class in the sending process, its LocalDeliver method retrieves the Request object stored in the pending data-store, creates a TransferProtocolMessage from it and calls its Deliver method. The constructor for the TransferProtocolMessage customises the container object by writing header information into the header byte array but it also sets the Data field in the container to reference the user send buffer and sets a completion delegate in the container to a call-back in the Request object. When the TCPSocketContainer object has transmitted the protocol message, this call-back method is called. It sets the internal ManualResetEvent object to the signalled state – releasing the user thread that is waiting for this event in the Wait method and allowing it to return control back to user code. The send operation is now complete.

When the receiving `TCPSocketContainer` object in the receiving process reads the header information for the transfer protocol message, it calls the `IncomingContainer` method from the `ProtocolMessage` class. This then creates a copy of the `TransferProtocolMessage` object; the constructor retrieves the `Request` from the matched data-store and sets the `Data` field in the container to reference the user receive buffer. On returning control to the `TCPSocketContainer`, the `Socket` object is directed to read data directly into the user receive buffer rather than the header byte array. When the read operation completes, the `IncomingContainer` method is called again. This time the `TransferProtocolMessage` constructor recognises that the `Data` field of the container is already set and calls the call-back method from the `Request` object that sets the internal `ManualResetEvent` object to the signalled state – releasing the user thread that is waiting for this event in the `Wait` method and allowing it to return control back to user code. The receive operation is now complete.

# Chapter 4

# Testing and Evaluation Methods

## 4.1 Communication Patterns

The communications patterns described in sections 4.1.1 and 4.1.2 are based on benchmark tests from the Intel MPI Benchmark (IMB) suite [50]. Primarily they are designed to measure various performance characteristics of an MPI library on an HPC system; however, the correctness of the MPI library is a pre-requisite for guaranteed successful completion of these patterns. The communication patterns described in section 4.1.3 are new patterns, designed as part of this work, to test the correctness of the algorithm that matches point-to-point messages, i.e. send operations, with receive operations. They can also be used to evaluate the performance of the implementation of that algorithm. Together these benchmarks are intended to test both correctness and performance of point-to-point communications using an MPI library.

### 4.1.1 The Ping-Pong Pattern

In the Ping-Pong pattern, a message is sent from one node to another (called the ping message, or simply, the ping), which then replies with a message back to the first node (called the pong message, or simply, the pong). The first node measures the time taken for the complete round trip. It is important that the receipt of messages is done in a blocking manner, i.e. the receive operation for the ping message must be complete before the send operation for the pong message is started. This may be achieved either by using a blocking receive method call or by waiting for a non-blocking receive method to complete. It is common practice to use a blocking method call for sending messages as well, although this is not critical.

With MPI commands, the first node obtains a timestamp, then calls `MPI_SEND` followed by `MPI_RECV` and then obtains a second timestamp. The second node calls `MPI_RECV` followed by `MPI_SEND`. Subtracting the first timestamp from the second gives the round-trip time. It is usual to quote half the round-trip time as the latency for the particular message size. This usage of the term latency should not be confused with the usage

meaning ‘time until the first byte arrives’. Similarly, the bandwidth for a particular message size is usually quoted as the amount of message data transferred divided by the round-trip time. This is not the same as the usage meaning ‘the instantaneous rate of data transfer’.

The Intel MPI Benchmark (IMB) documentation distinguishes two variants of the PingPong pattern: ‘PingPong Specific Source’, in which the `MPI_RECV` parameters specify the source for each message to be received rather than using a wildcard, and ‘Multi-PingPong’, in which all MPI processes communicate in pairs rather than just two processes communicating with each other.

As no optimisations regarding source have been included in McMPI, specifying the source should have no effect on the performance of this library. However, both uses of `MPI_RECV` (using a wildcard for the source argument in the PingPong pattern versus the actual rank of the sending MPI process in the PingPong Specific Source pattern) should be tested for correctness.

Expanding the PingPong pattern so that all MPI processes communicate in pairs, as in the ‘Multi-PingPong’ pattern, is usually done to investigate the bisectional bandwidth of the hardware communication system rather than to stress the MPI library itself. Careful choice of the pairings of processes can expose bottlenecks in the network such as an overloaded network switch, where it cannot handle data arriving at all its inputs at maximum rate and starts to discard some of it making all routes unreliable, or an oversubscribed link, where too many routes rely on the same physical cable and the bandwidth must be shared amongst them. These hardware tests are not directly relevant to this work because all the testing of McMPI compares the performance of different software MPI implementations on the same communication hardware rather than comparing the performance of a single software MPI implementation on different communication hardware.

However, McMPI is multi-threaded, with each thread taking on the role of an individual MPI process but sharing internal structures, specifically the message and request queues, with other threads within the actual OS process. This means that all active threads within the OS process will attempt to match their incoming messages with all requests in the single request queue for that OS process, irrespective of which thread issued the request; similarly, all active threads within the OS process will attempt to match their requests with all messages in the single unexpected queue for that OS process, irrespective of which

thread is the target of each of those messages. Sharing access to internal structures between threads requires thread-synchronisation, such as locking. Therefore, having more active threads within an OS process should affect the performance of the McMPI library code. Thus, both of these Ping-Pong pattern variants are important in comparative testing.

### 4.1.2 The Ping-Ping Pattern

The Ping-Ping pattern is similar to the Ping-Pong pattern except that the order of the send and receive operations on the second node is reversed; both nodes send first and then receive. In this pattern it is critical that the messages are sent in a non-blocking manner to avoid dead-lock – if both processes wait for their send operation to complete then neither can proceed to their receive operation and so neither can complete the send operation of the other. The Ping-Ping pattern is intended to test the bi-directional efficiency of the communication method, although it can also be used to test for certain aspects of correctness; in particular, the non-blocking nature of the `MPI_ISEND` method is critical to avoid dead-lock.

In the IMB documentation, the time measured for the Ping-Ping pattern is not a round trip (as it is for the Ping-Pong pattern). Instead, each repetition is timed individually, i.e. each node obtains a timestamp, calls `MPI_ISEND` followed by `MPI_RECV` and obtains a second timestamp. This timing method implicitly assumes that the communicating nodes maintain perfect synchronisation during the Ping-Ping pattern. However, it is very unlikely that this will always be true in practice.

A schematic time-step diagram was developed in order to visualise asynchronous communication patterns. The diagram conventions are an adaptation of the UML Sequence diagram. Each column represents a single communicating node or process and is labelled with the rank of the MPI process. Each row represents a single time-step, i.e. 1 unit of time, with the start of the communication pattern at the top of the column. Each box represents a single iteration of the communication pattern being studied and contains the constituent MPI commands. Blocking operations are prefixed by a closing brace and subsequent time-steps contain only a closing brace until the time-step that the blocking operation completes and returns control. The numbers to the right of each pair of columns (1, 2, 3 and -1, -2, -3) represent the delivery time for each message with positive numbers for left-to-right (i.e. a message sent from the process depicted on the left and to be received by the process

depicted on the right) and negative numbers for right-to-left. By convention, each library operation takes 1 unit of time and delivery takes 3 units of time, including the 'send' time-step and the 'receive' time-step. In practice, the delivery time is likely to be 50 units of time for socket-based communication but 3 units of time was chosen because it reduces the size of the schematic diagram and it is the minimum value that retains the same characteristics as the more realistic ratio. Developing and using this diagraming technique has proven to be valuable for visualising and reasoning about complex asynchronous communication scenarios.

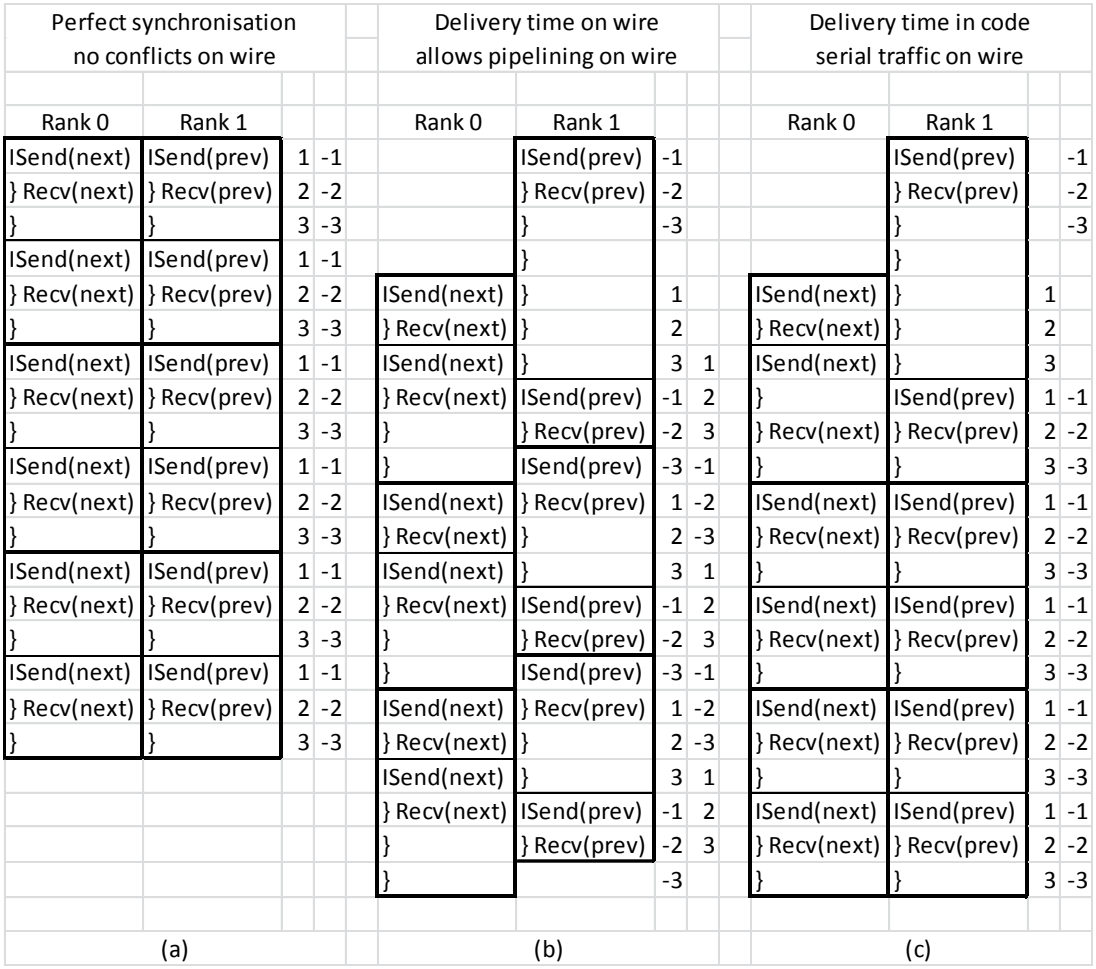


**Figure 11: Schematic time-step diagrams for the Ping-Ping pattern with assumptions that (a) perfect synchronisation is maintained throughout, (b) send operations can be pipelined (c) no two send operations can overlap.**

The three schematic time-step diagrams in Figure 11 show the outcomes of three different scenarios. In Figure 11(a), the normal assumption of perfect synchronisation is applied. All `MPI_RECV` operations are early, i.e. are initiated prior to data arrival, and complete as

soon as the data arrives. All iterations take 3 units of time (the delivery time), with the send and receive operations overlapping the communication, which is assumed to be perfectly bidirectional.

In Figure 11(b) and Figure 11(c), rank 0 is late starting by 4 units of time. At time-step 7, rank 0 issues a second send operation before the delivery of the first one has completed. This introduces a modelling decision: whether the second send should complete in one time-step as normal (shown in (b)) or should wait for delivery of the previous send operation and complete on the following time-step (shown in (c)). If the delivery time represents the transmission time within the network then the second send operation can complete in 1 unit of time as normal, i.e. the network is ready to accept more data in the time-step immediately after the first send operation. This is likely to be true for small messages. Alternatively, if the delivery time represents the time to write data to the network and there is insufficient buffer space to copy the message for later transmission, the second send operation would need to wait for the first one to complete. This is likely to be true for large messages.

It can be seen from Figure 11 that pipelining messages, as in (b), can distort the timing of iterations. If one `MPI_RECV` operation must wait an 'extra' time-step for its matching data to arrive (so the batch takes 4 units of time rather than 3 units in the perfect synchronisation scenario) then the next iteration will start one time-step 'late' and its `MPI_RECV` operation will be able to complete one time-step 'early' (so the iteration only takes 2 units of time). However, if two iterations of the Ping-Ping pattern are timed together as a single unit of work, then both nodes participate in two simultaneous round-trips, one self-initiated and the other peer-initiated. In all three scenarios modelled here, the combination of two consecutive iterations (excluding the first two iterations) always takes 6 units of time, i.e. twice the time of a single 'perfect' iteration, which suggests this will give more consistent and realistic iteration times than timing individual iterations.

With MPI commands, each node obtains a timestamp, calls `MPI_ISEND` followed by `MPI_RECV`, repeats these two calls and then waits for the non-blocking sends to complete, using `MPI_WAIT`, before obtaining a second timestamp. Subtracting the first timestamp from the second gives the 'two round-trips at once' time. This time may be halved for comparison to times produced by the strict interpretation of the IMB Ping-Ping pattern and

to the round-trip times from the Ping-Pong pattern. For a perfect software implementation of bidirectional behaviour on a perfectly bidirectional hardware system, the Ping-Ping time should be identical to the Ping-Pong time, even though two round-trips have been performed simultaneously. The 'bandwidth for a particular message size' for Ping-Ping should therefore be exactly twice that for Ping-Pong because twice the quantity of message data is transferred in the same amount of time. Differences between these expected values and the measured values are partly due to inefficiencies in the implementation of the MPI library and partly due to inefficiencies in the hardware. Comparing the differences for one MPI library with the equivalent difference for another MPI library excludes most of the effect due to the hardware system and reveals the relative efficiency of the implementations of bidirectional behaviour.

Just as for Ping-Pong, the IMB documentation distinguishes two variants of the Ping-Ping pattern: 'PingPing Specific Source', in which the `MPI_RECV` arguments specify the source for each message to be received rather than using a wildcard, and 'Multi-PingPing', in which all MPI processes communicate in pairs rather than just two processes communicating with each other. The observations regarding the Ping-Pong variants in section 4.1.1 also apply here: specifying the source for all `MPI_RECV` operations allows scope for optimisations in the MPI library and expanding the test so that multiple threads in each OS process are communicating with each other stresses the thread-synchronisation mechanism that protects the shared internal structures of the matching algorithm. Thus, both of these Ping-Ping pattern variants are important.

### 4.1.3 InOrderTags and ReverseOrderTags

The `InOrderTags` pattern is the converse of the `ReverseOrderTags` pattern. In both of these patterns, a number of messages, e.g. 10 or 100, are sent from one node to another, which responds with the same number of messages but only after all the incoming messages have been received. These patterns test the MPI matching algorithm, both for correctness and for performance. Correctness is tested by checking the data in each message.

Each outgoing message from the first node contains different data; the second node copies the received data buffers into its outgoing return messages in the order that it (the second node) matches the incoming messages. The first node checks the received data buffers

against the outgoing data, again in the order that it (the first node) matches the incoming messages. In both cases, `InOrderTags` and `ReverseOrderTags`, the data in the return messages should be identical to the data in the outgoing messages for the first node. A further check can be performed by the second node: it can check the data for matched incoming messages against a table of values that duplicates the sending order for the first node – in the `InOrderTags` case the order should match whereas in the `ReverseOrderTags` case the order should be reversed. For higher message counts per batch, these patterns also test for the presence of limits on the number of messages and requests that can be in progress simultaneously.

The sending algorithm for both patterns can be described by the following pseudo-code:

```
FOR tag = 1 TO n
  requests[tag] = MPI_ISEND(…, tag, …)
END FOR
MPI_WAITALL(requests)
tag = 0
MPI_RECV(…, tag, …)
```

The receiving algorithm for the `InOrderTags` pattern can be described by the following pseudo-code:

```
FOR tag = 1 TO n
  requests[tag] = MPI_IRECV(…, tag, …)
END FOR
MPI_WAITALL(requests)
tag = 0
MPI_SEND(…, tag, …)
```

The receiving algorithm for the `ReverseOrderTags` pattern can be described by the following pseudo-code:

```
FOR tag = n TO 1
  requests[tag] = MPI_IRECV(…, tag, …)
END FOR
MPI_WAITALL(requests)
tag = 0
MPI_SEND(…, tag, …)
```

The workload of a particular node will be identical for both of these communication patterns, except for differences arising from forcing the order of matching to be as efficient

as possible for the `InOrderTags` pattern and as inefficient as possible for the `ReverseOrderTags` pattern. In each pattern, the same number and size of messages will be transferred in each direction and the same code to generate and check the message data will be executed at each process. The differences in the workload for these two patterns will be limited to the amount of processing needed for the different number of attempted matches between the envelopes for incoming messages and the envelopes for receive operations. The time taken by one batch of $n$ messages (i.e. $n$ in each direction) can be split into two components: the time taken by attempted matches and the time taken by everything else. The latter component, the time taken by non-matching code, will be identical for the two patterns. The first component will be a different function of $n$ for each communication pattern, depending on the number of unsuccessful attempted matches.

The implementation of the MPI matching algorithm in McMPI, detailed in section 3.2.8, relies on manipulating queues that are protected by locks. Each operation that reads from, or writes to, a queue first obtains permission by taking a lock then manipulates the queue and finally releases the lock, allowing another operation to take the lock and proceed. This approach serialises queue accesses, i.e. no two queue operations can overlap. In practice, this means that each successful receive operation will involve at least two serialised queue manipulations: the receive operation must examine the queue of unmatched messages and the incoming message must examine the queue of outstanding receive operations.

A receive operation may happen first, i.e. it may take the queue-lock at a time when the unmatched messages queue does not contain a message with an envelope that matches the receive operation. In this case, the receive operation envelope will be checked against every message in the unmatched queue, if any, and must then be stored in the outstanding receive operations queue before the queue-lock is released.

Alternatively, a receive operation may happen second, i.e. it may take the queue-lock at a time when the unmatched messages queue does contain a matching message. In this case, the receive operation envelope will be checked against every message in the unmatched queue up to, and including, the matching message. The matching message must then be removed from the unmatched queue before releasing the queue-lock.

Similarly, an incoming message may arrive before or after a matching receive operation. When the receiving MPI process is handling the arrival of the message envelope, the

queue-lock may be taken at a time when the outstanding receive operations queue does not contain a matching receive operation. In this case, the incoming message must be stored in the unmatched messages queue before the queue-lock is released. Alternatively, when the outstanding receive operations queue does contain a receive operation that matches, it must be removed before the queue-lock is released.

Thus, every receive operation and every incoming message involves taking the queue-lock, performing a number of attempted matches (zero or more, depending on the length of the appropriate queue), either storing an element in, or removing an element from, a queue, and releasing the queue-lock. The locking, storing or removing, and unlocking steps entail a fixed cost per receive operation and per incoming message. The time taken for them is the same for both communication patterns and is not proportional to $n$, so it is included in the second term mentioned above: the non-matching term.

In the `InOrderTags` pattern, the message tags are set so that matching occurs in the order of dispatch of the messages, e.g. sequentially increasing for both messages and receipts. When the unmatched messages queue is examined by a receive operation, it will either be empty or the first message will match the receive operation – no unsuccessful matches will be needed. Similarly, when a message envelope arrives, the outstanding receive operations queue will either be empty or the first receive operation will match – again, no unsuccessful matches will be needed. This arrangement of tags, therefore, results in the minimum number of attempted matches – one successful and none unsuccessful per received message – so there are $n$ successful matches per batch (where $n$ is the number of messages sent in the batch).

In the `ReverseOrderTags` pattern, the message tags are set so that matching occurs in the opposite order to the order of dispatch of the messages, e.g. sequentially increasing for messages and sequentially decreasing for receipts. The $i^{th}$ message will match the $(n+1-i)^{th}$ receive operation and the $j^{th}$ receive operation will match the $(n+1-j)^{th}$ message (where $n$ is the number of messages sent in the batch). As McMPI uses a single queue for all unmatched incoming messages and a single queue for all outstanding receive operations, the ordering guarantee in MPI (see section 2.3.4) means that the $i^{th}$ message will be unsuccessfully matched with the first $(n-i)^{th}$ receive operations before it is successfully matched with the $(n+1-i)^{th}$ receive operation. The $j^{th}$ receive operation

will be unsuccessfully matched with the first $(n-j)^{th}$ messages before it is successfully matched with the $(n+1-j)^{th}$ message. These are equivalent ways to count the number of matches: only one should be used to avoid double-counting. The total number of matches is the sum for all messages (or receives), i.e. $\sum_{i=1}^{n}(n+1-i)$. This arrangement of tags, therefore, results in the maximum number of attempted matches – equal to $n(n+1)/2$ per batch, where $n$ is the number of received messages; all but the last one for each message are unsuccessful, so there are $n(n-1)/2$ unsuccessful and $n$ successful matches per batch.

Taking $t_s$ to be the time for a successful match, $t_u$ to be the time for an unsuccessful match and $t_{other}$ to be the non-matching time, then the batch time for InOrderTags, $t_i$, and the batch time for ReverseOrderTags, $t_r$, are

$$t_i = t_s n + t_{other}$$

$$t_r = t_s n + t_u\left(\frac{n(n-1)}{2}\right) + t_{other}$$

The difference between these batch times is independent of both $t_s$ and $t_{other}$ because these terms cancel each other:

$$\Delta t = t_r - t_i = \left(t_s n + t_u\left(\frac{n(n-1)}{2}\right) + t_{other}\right) - (t_s n + t_{other}) = t_u\left(\frac{n(n-1)}{2}\right)$$

Thus, $t_u$ may be determined from measurable values as follows:

$$t_u = \frac{2\Delta t}{n(n-1)} = \frac{2(t_r - t_i)}{n(n-1)}$$

# 4.2 Test Codes

## 4.2.1 NetPIPE in C

The latency for TCP sockets in C was measured with a modified version of NetPIPE [51] [52]. The original source code for NetPIPE was modified to use the high-precision timer from the Windows kernel32.dll driver and to output sufficient information about the timing to enable the first and second sextile values to be calculated via subsequent data analysis. This modified code was compiled and linked into three executable programs:

- with the TCP module and no MPI library (for the TCP socket tests)
- with the MPI module and the MPICH2 [53] library (for the MPICH2 tests)
- with the MPI module and the MS-MPI [42] library (for the MS-MPI tests)

### 4.2.2 NetPIPE in C#

The latency for a TCP socket in C# was measured using a test program that performs the same communication and timing as the modified version of NetPIPE. Both programs use the blocking `Send` method to send data and the blocking `Receive` method to receive data.

### 4.2.3 Pattern Test in C#

The McMPI tests required a new test program to be created. Written entirely in C# and linked with McMPI, it implements the communication patterns described in section 4.1 and produces timing information in the same format as the other two test codes, which simplifies subsequent data analysis. The timing is performed using the high-precision timer from the Windows kernel32.dll driver, just as in the other two test codes.

## 4.3 Test Systems

Testing was performed on three different systems, which are described in detail in sections 4.3.1, 4.3.2 and 4.3.3. The shared memory server (subsequently referred to as the server machine) is owned by the author of this work. The distributed memory desktops (subsequently referred to as the desktop machines) are owned and supplied by EPCC. For both of these systems, administrator access rights were available allowing the installation of third party software, such as MPI libraries. Both MPICH2 and MS-MPI were installed on the server machine and both were configurable to support shared memory communication or communication using loopback socket connections.

Although both MPICH2 and MS-MPI were installed on the desktop machines, only MPICH2 was configurable to allow distributed memory communication. The MS-MPI library configuration requires that the credentials of all the Windows users on all of the machines that create MPI processes are identical. This can easily be achieved with a common domain controller but is not possible with local Windows user accounts. The security policy within the University of Edinburgh restricts the use of machines where administrator rights have been granted to a segment of their network that does not contain a domain controller. Thus, MS-MPI on the desktop machines could not be configured for multiple machine

communication without breaching the corporate security policy and no tests using MS-MPI on the desktop machines were possible.

The HPC compute cluster (subsequently referred to as the cluster machine), is owned and operated by the University of Oxford; a user account was kindly granted to the author along with a generous allocation of resources. The system administrators for the cluster machine had already installed and configured MS-MPI but had not installed MPICH2. There was no possibility of installing additional libraries on this system due to the security policy and terms of use. Therefore, no tests using MPICH2 on the cluster machine were possible.

### 4.3.1 Shared Memory Server

- Number of Nodes: 1 Customised Armari Magnetar Server
- CPUs per Node: 2 Intel Xeon E5420
- Threads per CPU: 4 Quad-core, no hyper-threading
- Core Clock Speed: 2.5GHz Front-side bus 1333MHz
- Level 1 Cache: 4x2x32KB One data plus one instruction per core
- Level 2 Cache: 2x6MB One per pair of cores
- Memory per Node: 16GB DDR2 667MHz
- Network Hardware: Gigabit Ethernet x2 Intel 82575EB Gigabit Ethernet
- Operating System: Windows XP Professional 64bit with SP3 version 5.2.3790

### 4.3.2 Distributed Memory Desktops

- Number of Nodes: 2 Dell OptiPlex GX620 x64 Workstations
- CPUs per Node: 1 Intel Pentium 4 family 15 model 4 stepping 3
- Threads per CPU: 2 Single-core, with hyper-threading
- Core Clock Speed: 3.2GHz Front-side bus 800MHz
- Level 1 Cache: 1x16KB One per CPU
- Level 2 Cache: 1x2MB One per CPU
- Memory per Node: 4GB DDR2 533MHz
- Network Hardware: Gigabit Ethernet x1 Broadcom NetXtreme 5751 Gigabit
- Operating System: Windows 7 Enterprise 64bit with SP1 version 6.1.7601

### 4.3.3 HPC Compute Cluster

- Number of Nodes: 18 Dell PowerEdge 2900
- CPUs per Node: 2 Intel Xeon 5130 Family 6 model 15 stepping 6

- Threads per CPU: 2 Dual-core, no hyper-threading
- Core Clock Speed: 2.0GHz Front-side bus 1333MHz
- Level 1 Cache: 2x2x32KB One data plus one instruction per core
- Level 2 Cache: 1x4MB One per CPU
- Memory per Node: 4GB DDR2 533MHz
- Network Hardware: Gigabit Ethernet x2 Broadcom BCM5708C NetXtreme II GigE
- Operating System: Windows Server 2008 Standard x64 with SP2 version 6.0.6002

# 4.4 Output and Analysis

The output from all the test methods described in this chapter is simply the time taken for each batch of iterations to complete. This was measured by obtaining two timestamps, one before each batch began, the second after it had finished, and subtracting the first from the second. On each test system, the timing mechanism with the highest precision of those available was used and the function used to obtain a timestamp was itself timed to ensure that it was sufficiently fast to be suitable. The reported resolution for the high-precision timer in C# (the `System.Stopwatch` class), does not indicate the precision of the clock; it states the number of clock-ticks per second rather than the smallest possible interval between two consecutive timestamps. Obtaining a timestamp involves a P/Invoke system call, which is slow relative to many other built-in commands, and may be slower than the event being timed. As a general rule, the batch size was increased until the batch time was at least 2 orders of magnitude greater than the timer time, which allows timing figures to be quoted to 3 significant figures. In some of the tests a batch containing a single iteration was sufficient but for some of the tests a batch size of 10 iterations was required.

## 4.4.1 Mean and Standard Deviation

The timings do not follow a normal distribution so the mean and standard deviation cannot adequately describe the shape of the data. The distribution has a definite lower bound (the theoretical connection speed) but no upper bound. Any number of influences can increase the time taken for a single batch and produce extreme positive outliers, which introduces a systematic tendency for the mean and standard deviation to over-estimate the centre and spread of the distribution.

### 4.4.2 Median and Mode

Other standardised methods to summarise a dataset include the median and the mode. Intuitively, the median should be a better metric than mean to use for comparison of datasets from two different codes because it attaches less importance to infrequent outliers. However, both mean and median attempt to locate the centre of the distribution, which is not the quantity of greatest interest for a performance comparison of communication codes.

In isolation, the performance of a communication code is commonly modelled as a linear function of message size: $L(n) = t_s + t_b n$ where $L(n)$ is the time for a message of size $n$ to be delivered (the latency), $t_s$ is the start-up time and $t_b$ is the transfer time per unit of data (commonly one byte). This model is a simplification of a real system in that it assumes that the start-up time and transfer time per data unit are the same for all messages and that no external influences will delay the message delivery. In practice, all modern operating systems perform background-tasks, such as hardware interrupts, time-slicing and management of virtual memory. When comparing two such codes it may be assumed that many of the influences that produce outliers will affect both codes equally and so, if removed, would not affect the comparison. On the other hand, including all the outliers may result in two similar measurements of the noise of the system environment. A more appropriate metric might measure and compare the "best repeatable" performance for each code.

Repeatability suggests that the mode (i.e. the most common value) should be considered. However, with a continuous variable, such as time, most values in the dataset will be singletons – even if the time of one batch is almost identical to the time of another it is unlikely to be exactly the same. Each value, i.e. each measurement of batch time, is therefore unlikely to occur more than once and the simply counting method used to determine the mode is unhelpful. This can be mitigated by binning – grouping together similar values in a series of ranges to form a frequency histogram – but this technique relies on choosing the size of the ranges very carefully: too narrow produces too many similarly sized peaks, whereas too broad produces too little precision.

### 4.4.3 Minimum and Most Frequent Minimum

It is common practice when attempting to measure latency to quote the minimum time of all measurements taken. Whilst this has the advantage of being as close as possible to the theoretical latency, it suffers the disadvantage of excluding all other information about the distribution. The spread of values is important for assessing one code against another. A code that can achieve a lower latency than another may not be considered "better" if it only achieves this infrequently.

To compare the performance of two communication codes, perhaps the most useful estimate is the "most frequent minimum time" – a concept that is, unfortunately, far less robust. The approach adopted by [54] provides a suggestion that may be useful as a working definition. They quote the first sextile, i.e. the $25^{th}$ value out of 150 measurements that have been sorted in ascending order. There is little explanation given as to why this choice was made other than the intention "to avoid distortions due to timing outliers" and the assurance that "this value is statistically better than the mean or the median" because they "obtained minimal least square errors with this value".

### 4.4.4 Error Bars

It is unusual for error bars to be given on plots of latency or bandwidth; even the paper referenced above, which applies the least square error method to check the models of message latency and bandwidth, does not quote any estimate of the error. In the absence of established best practice in this area, it was decided to use error bars to indicate the spread of data. Specifically, lower bound of the error bar is taken to be the minimum value because this is, commonly, the only value quoted and including the minimum in some form allows comparison to established literature. The choice for the upper error bar is less straight-forward. Options that were considered and rejected include:

- The $5^{th}$ sextile, e.g. the $125^{th}$ value from a dataset of 150 sorted in ascending order.
- The mean.
- The median, e.g. the $75^{th}$ value from a dataset of 150 sorted in ascending order.
- The upper bound of a confidence interval, such as 68% or 95%.

The confidence interval idea is based on the observation that, for a normal distribution, standard deviation can be used to estimate a confidence interval. The symmetric interval

$(-z\sigma, z\sigma)$ is expected to contain approximately 68% of values for z=1 and approximately 95% of values for z=2. The reasons for rejecting this choice are the same as those for rejecting average and standard deviation: the distribution is not normal and the interesting quantity is not the centre of the distribution. In fact, the idea of constructing a confidence interval for the theoretical minimum latency that lies within the range of measurements is fundamentally flawed because any measured value is at least equal to, but most likely greater than, this quantity.

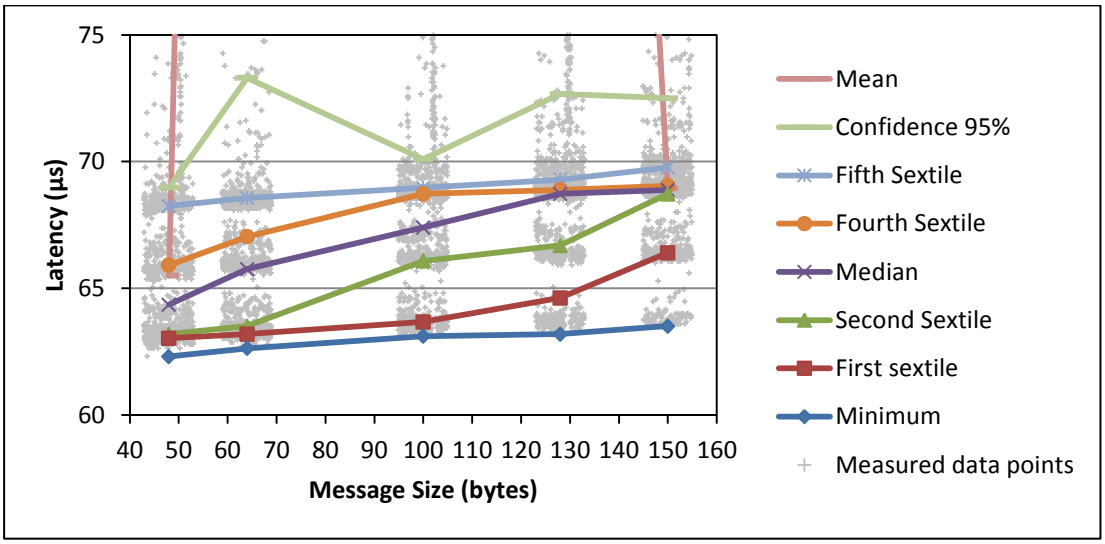


**Figure 12: a typical message latency data set and various suggested summarising metrics (the same data set is used in Figure 45 and Figure 59)**

The measured data points shown in Figure 12 are a typical data set from measuring latency. This data set is presented in two different ways in Figure 44 and Figure 59. The data points have been spread horizontally for visual clarity: all the data points in the first 'column' are actually measured for the same message size, i.e. 48 bytes; similarly the other columns show data for message sizes 64, 100, 128 and 150 bytes. It is clear from Figure 12 that the distribution of data values is complex and cannot be adequately described by a standard model such as a normal distribution. The three mean values that are off the scale are 196µs, 145µs and 146µs because the data for each of these message sizes include a few data values of approximately 39,000µs (5, 1 and 3 values that are greater than 37,000µs).

The 'error-bars' used in this work are not intended to show a measure of the error associated with the measurement of a quantity with a fixed value but rather are intended to provide some information about the distribution associated with the measurement of a quantity that has a variable value. In particular, the intent is to provide more information

about the spread of values in the first (lowest) group of data points. Currently, accepted practice is to plot only the minimum value, which specifies the extreme low edge of the first group of data but discards all other information about that group along with all information about the rest of the data set.

It can be seen from Figure 12 that the first sextile usually picks out a value that is within the first group of data values, unless there are too few data points in that group (as for message size 150 bytes, for example). It is therefore deemed to be a reasonable metric to indicate the location of the first group. The second sextile tends to pick out a value that is either in the first group of data values or in the second group. When the second sextile value is within the first group, the first and second sextile values are usually very close together, which may be interpreted as the first group being relatively dense, i.e. the measurement is relatively repeatable. When the second sextile value is in the second group, the difference between first and second sextile is greater, which may be interpreted as the first group being less dense, i.e. the measurement is less repeatable.

The third sextile, i.e. the median, usually picks out a value that is either in the second group of data values or the third group, although for message size 48 it picks out a value in the first group. Arguably, this metric contains more information about the entire distribution but carries obscures more detailed information about the first group of data values. Similarly, the fourth and fifth sextile values, as well as the 95$^{th}$ percentile value include even more information about the entire distribution but do not give useful information about the first group of data values.

Thus, the option chosen for the upper error bar is as follows:

- The 2$^{nd}$ sextile, e.g. the 50$^{th}$ value from a dataset of 150 sorted in ascending order. This provides additional information about the distribution of the values that are above, but still in the neighbourhood of, the quoted value.

# Chapter 5

# Results

This chapter presents performance measurements for McMPI along with comparison measurements for MPICH2 and MS-MPI, obtained using NetPIPE. In addition, baseline measurements, obtained using TCP sockets directly, i.e. just transmitting raw data rather than message-passing as defined by the MPI Standard, are presented.

Sections 5.1, 5.2 and 5.3 deal with three different communication situations: thread-to-thread, process-to-process and machine-to-machine. They each follow a similar structure:

- Firstly, if applicable, a baseline is established by presenting the performance of TCP sockets in both C and C#.
- Secondly, the Ping-Pong performance of all versions of McMPI is summarised.
- Thirdly, the Ping-Pong performance of McMPI is compared with MPICH2 and MS-MPI.
- Fourthly, the performance of other communication patterns, such as Ping-Ping, is used to investigate particular characteristics of the McMPI library.
- Fifthly, results are presented of investigating performance-critical settings that affect all versions of the McMPI library, such as affinity and socket buffer size.

## 5.1 Thread-to-Thread Delivery

As none of the libraries in this section use sockets for communication, no baseline for socket-only communication is applicable.

Section 5.1.1 presents a summary of Ping-Pong performance results for all versions of McMPI using the thread-to-thread delivery mechanism, i.e. bypassing the communication layer (as described in section 3.2.6).

Section 5.1.2 compares the performance of the best version of McMPI with MPICH2 and MS-MPI communicating process-to-process via shared-memory.

Section 5.1.3 compares the performance of Ping-Ping and Ping-Pong using McMPI, to investigate the bi-directional efficiency of thread-to-thread communication.

Section 5.1.4 compares the performance of In-Order-Tags and Reverse-Order-Tags using McMPI, to investigate the performance of the matching algorithm.

Section 5.1.5 investigates the effect of processor affinity on shared-memory performance.

### 5.1.1 Summary of the Performance of All Versions of McMPI

The various versions of McMPI differ only in their implementation of the TCP socket communication module. The thread-to-thread delivery mechanism bypasses the communication layer entirely, so all versions of McMPI should exhibit identical performance characteristics. Unexpectedly, the versions of McMPI that use the `ReceiveAsync` socket method to receive data from other processes have much higher thread-to-thread latency than other versions, as can been seen from Figure 13. One possible explanation for this is that calling `ReceiveAsync` creates and activates the thread-pool of IO completion threads and these background threads are scheduled by the operating system in sequence with the main thread, increasing the time between useful time-slices in the main-thread and therefore lengthening the elapsed time for each task without increasing the amount of work done for each task. As predicted by this hypothesis, the lowest latency is achieved by the version that uses fewest threads. However, further work is needed to verify this.

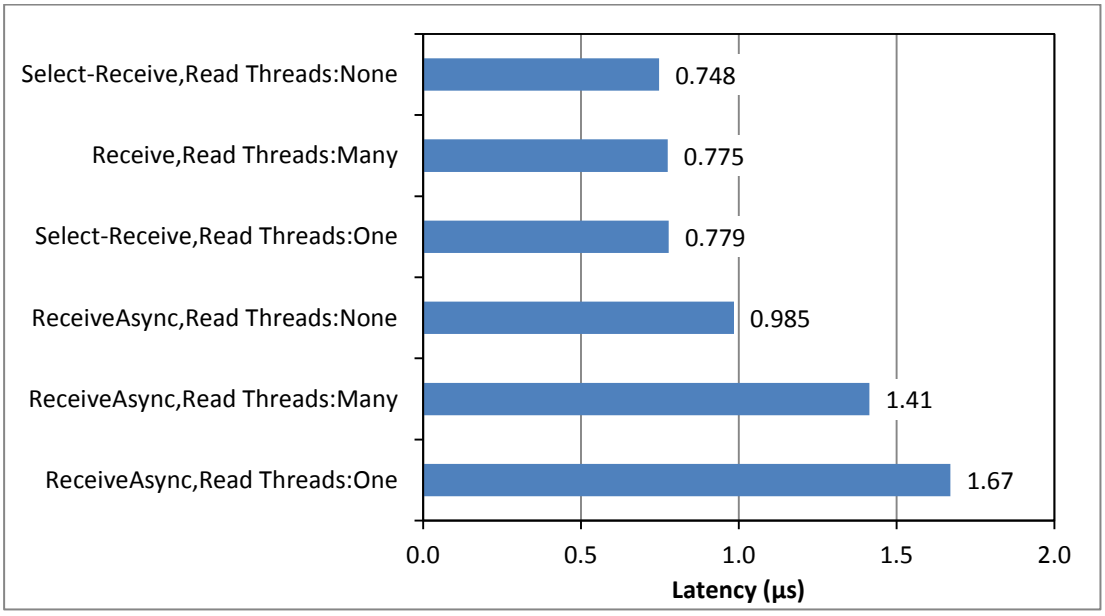


**Figure 13: a summary of the lowest thread-to-thread ping-pong latency for all tested message sizes (between 1 byte and 1MB) for each version of McMPI.**

The difference in bandwidth between the versions of McMPI, shown in Figure 14, is also unexpected but may be explained in a similar manner to the latency differences. The version with the greatest bandwidth is also the version that does not create extra threads.

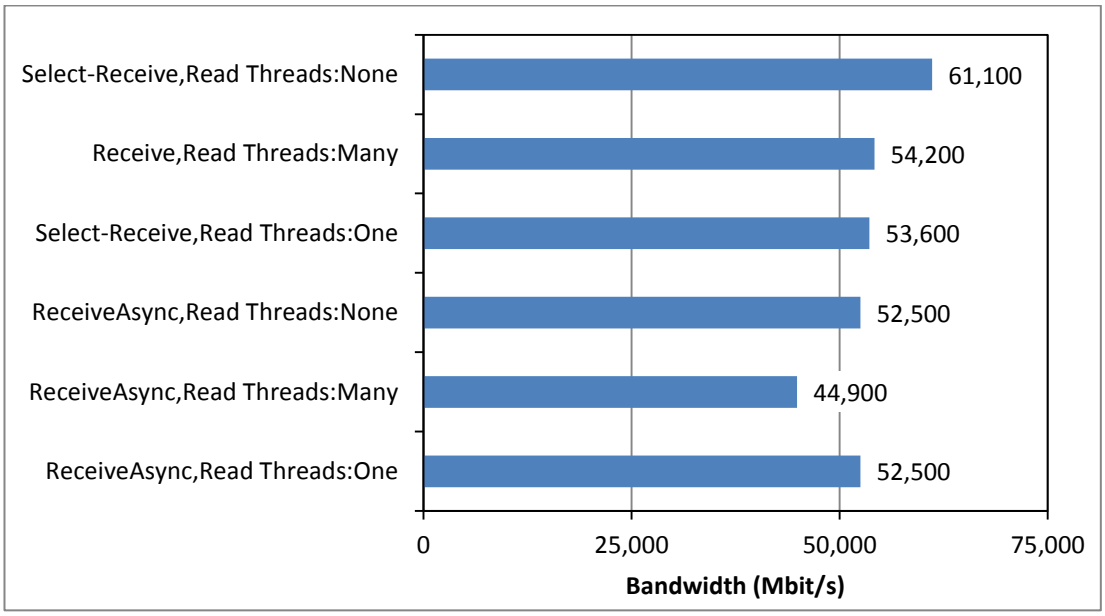


**Figure 14: a summary of the highest thread-to-thread ping-pong bandwidth for all tested message sizes (between 1 byte and 1MB) for each version of McMPI.**

### 5.1.2 Comparison of the Performance of McMPI with MPICH2 and MS-MPI

Figure 15 and Figure 16 show the latency in microseconds of Ping-Pong message-passing with messages of size between 1 byte and 16KB (16*1024 bytes) for McMPI, MS-MPI and, where possible, MPICH2. For Figure 15, the test was performed on the Server machine described in section 4.3.1, i.e. a Windows XP Pro machine with two quad-core 2.5GHz Xeon processors and 16GB of DDR2 memory. For Figure 16, the test was performed on a single node of the Cluster machine described in section 4.3.3, i.e. a Windows Server 2008 HPC Edition machine where each node contains two dual-core processors and 8GB of memory. Note that the Cluster machine does not have MPICH2 installed.

Figure 17 and Figure 18 show the bandwidth in Mbit/s (million bits per second) of Ping-Pong message-passing with messages of size between 4KB and 1MB (1*1024*1024 bytes) for McMPI, MPICH2 and MS-MPI on the Server machine and the Cluster machine, respectively.

In all three test codes, messages are sent and received in 1,500 timed batches of two round-trips each. Therefore, latency is one quarter of the batch time and bandwidth is the message size in bits divided by the latency. The timer used is the high performance counter from the Microsoft Windows kernel32.dll driver. For MPICH2, the channel configuration setting was set to `ssm` to allow shared memory communication and the standard mode MPI send function was used. For MS-MPI, the default settings, which allow shared-memory communication, were not changed and the standard mode MPI send function was used. For McMPI, the configuration was set so that two MPI processes would execute as threads within a single OS process and the eager limit was set to zero to force the use of the rendezvous protocol and to 1MB to force the use of the eager protocol. For thread-to-thread communication in McMPI, there is no measurable difference between times obtained from the eager protocol and those obtained from the rendezvous protocol so the eager protocol times were chosen. As discussed in section 4.4, each latency data value is the first sextile of the 1500 batches (i.e. the 250$^{th}$ value after sorting ascending), each downward error-bar extends to the minimum of the 1500 batches and each upward error-bar extends to the second sextile of the 1500 batches (i.e. the 500$^{th}$ value).

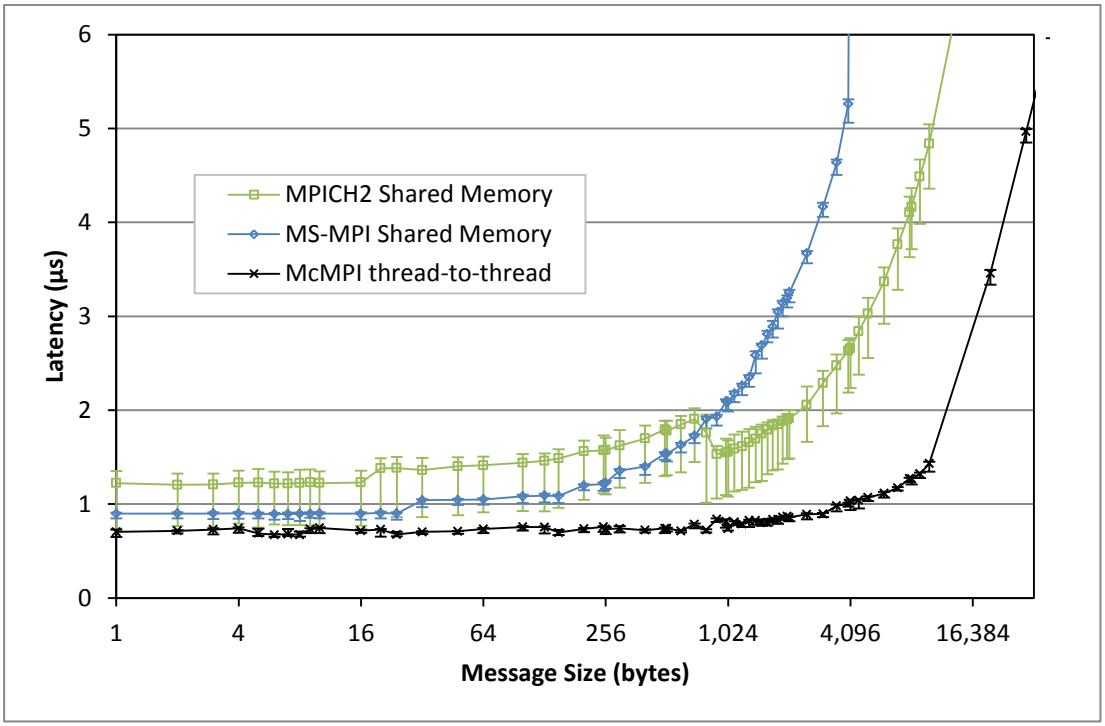


**Figure 15: Ping-pong latency for shared-memory message-passing on the Server machine. Data-points represent the "most-frequent-minimum" latency, i.e. the first sextile latency, from 1500 batches; each batch times 2 round-trips. Error bars represent the minimum and second sextile latency measurements. Both MPICH2 and MS-MPI use their shared-memory module, whereas McMPI uses thread-to-thread delivery.**

The variability of the latency measured for MPICH2, which can be seen by the size of the error-bars in Figure 15, is likely due to the lack of processor affinity. There is no processor affinity setting in MPICH2: a mask may be set by a job manager (no job manager was installed on the Server machine) or by the parallel program itself (which requires changes to the source code of NetPIPE) but neither of these were done. The increase in performance for the MPICH2 code between 700 bytes and 900 bytes is unexplained but may be due to a change of communication protocol. The default settings for MS-MPI on the Server machine appear to set a processor affinity mask for all participating operating system process so that they are restricted to one physical processor core each: monitoring system activity during the test shows that the MPI processes do not migrate between cores and the error-bars are small in Figure 15. However, the variability of the latency for MS-MPI on the Cluster machine, seen in Figure 16, demonstrates that a processor affinity mask is not set by default in this case. For McMPI, a processor affinity mask was explicitly set in the library code to restrict the single operating system process to two physical processor cores (one per thread, i.e. one per MPI process). On both machines the processor cores chosen by the affinity mask share an L2 cache and do not handle system interrupts.

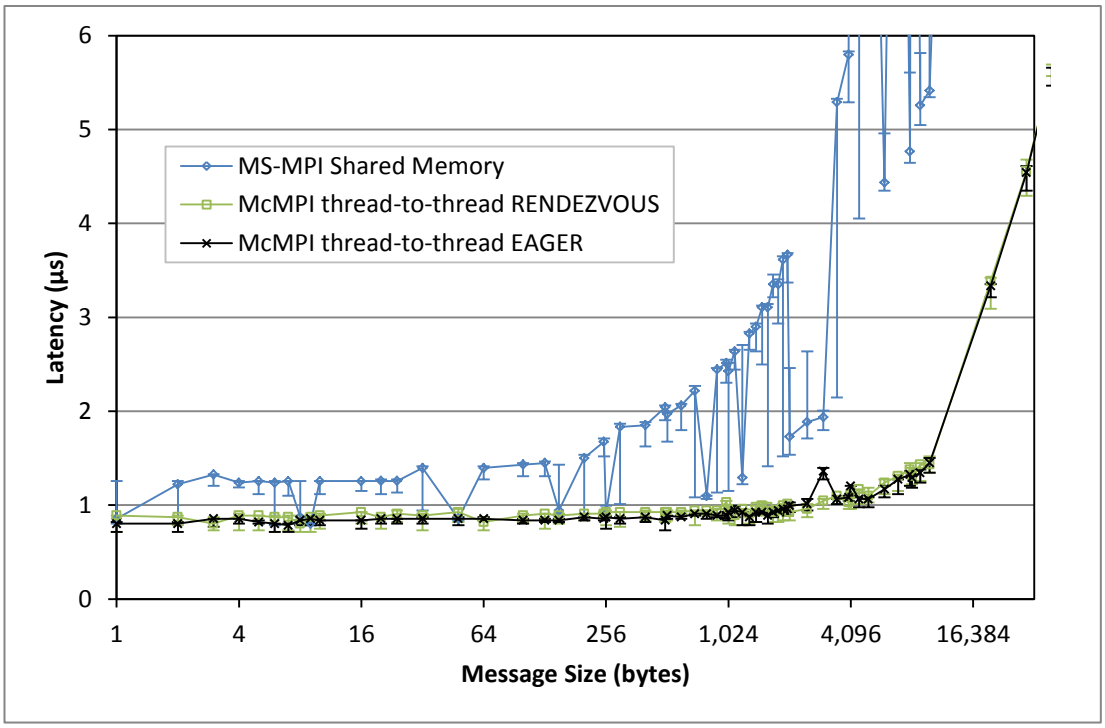


**Figure 16: Ping-pong latency for shared-memory message-passing on the Cluster machine.**

Both the latency and bandwidth (up to 256KB) of McMPI for thread-to-thread communication are better than MPICH2 and MS-MPI. This may be due to the lower overheads involved in using shared-variables (in McMPI) as opposed to shared-memory (in MPICH2 and MS-MPI). In McMPI, the queue of requests in the target MPI process can be searched without transferring data because it is a shared variable. If a matching request is found then the data is copied directly from the sending buffer to the receive buffer. In MPICH2 and MS-MPI, the message must be copied into shared memory before the target MPI process can access it. The message data must then be copied a second time from shared memory into the receive buffer. This extra overhead may explain both higher latency and lower bandwidth of MPICH2 and MS-MPI compared with McMPI. The drop in bandwidth for McMPI for messages sizes above 256KB is unexplained.

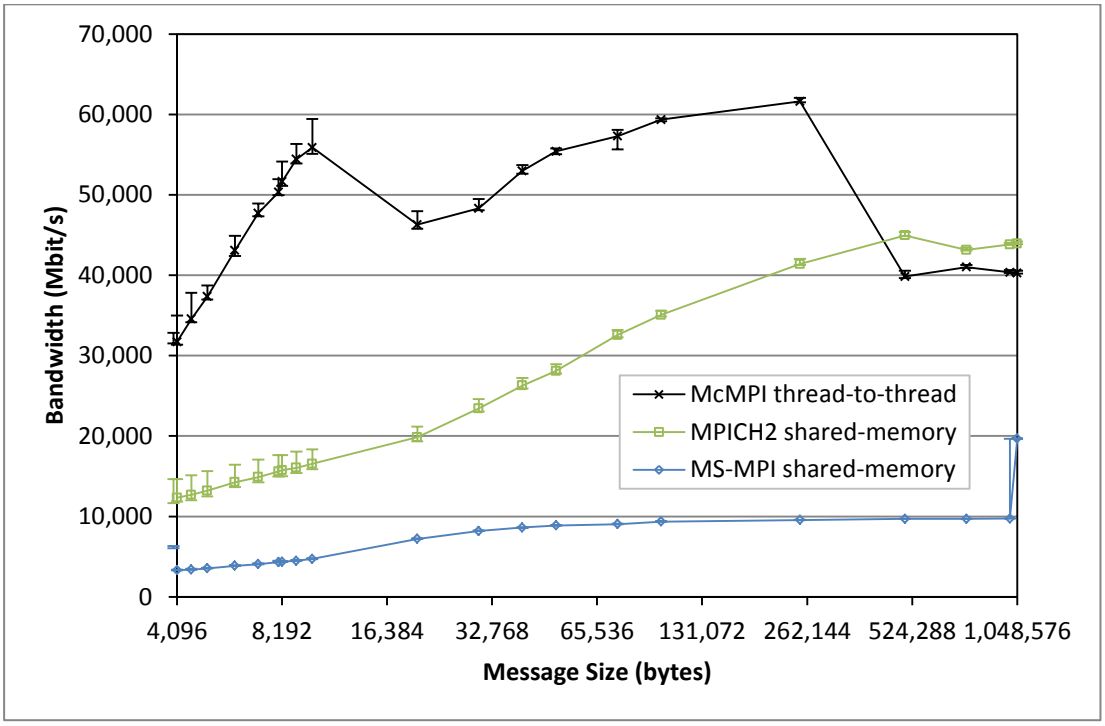


**Figure 17: Ping-Pong bandwidth for shared-memory message-passing on the Server machine. Data-points represent the "most-frequent-maximum" bandwidth, i.e. the fifth sextile, from 1500 batches; each batch times 2 round-trips. Error bars represent the maximum and fourth sextile bandwidth measurements. Both MPICH2 and MS-MPI use their shared-memory module, whereas McMPI uses thread-to-thread delivery.**

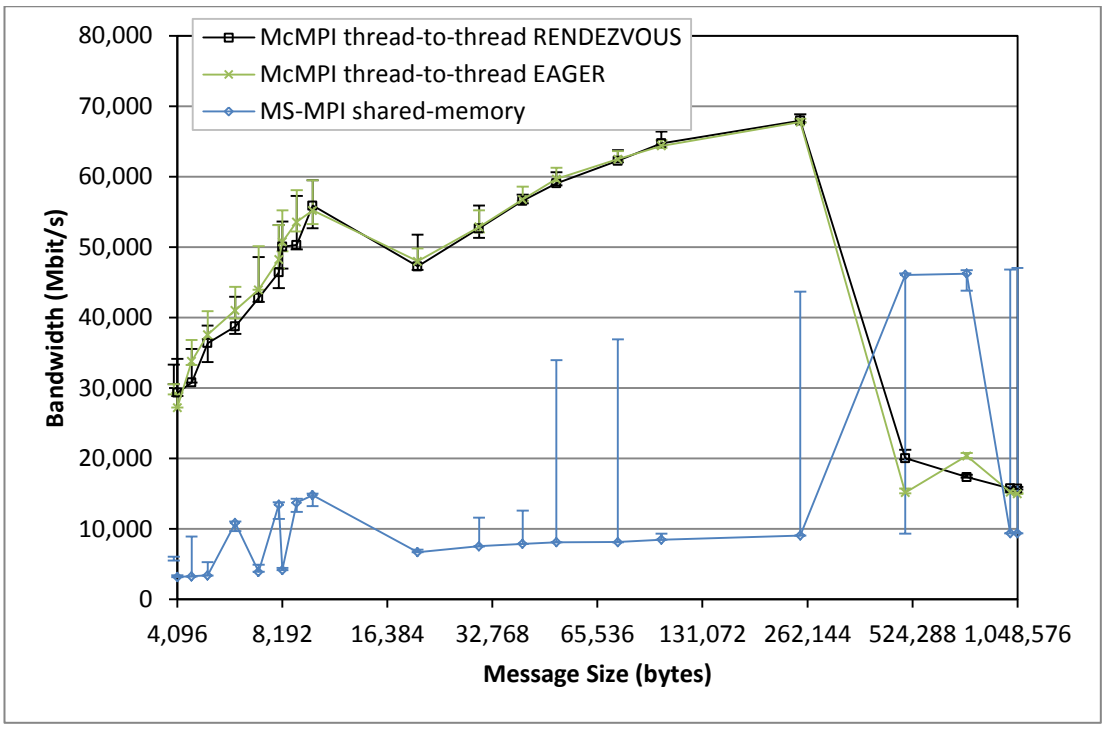


**Figure 18: Ping-pong shared-memory bandwidth on the Cluster machine.**

### 5.1.3 Comparison of the Performance of Ping-Ping and Ping-Pong

The ping-ping communication pattern times two simultaneous round-trips; data flows in both directions at the same time. If the hardware (plus the communication libraries and parallel program) supports bi-directional transmission, the bandwidth for ping-ping should be double the bandwidth of the ping-pong communication pattern. However, this theoretical assessment ignores factors that can dramatically affect bandwidth, such as, in the shared-memory case, cache usage and multiple memory channels.

Figure 19 (Server machine) and Figure 20 (Cluster machine) show the ratio of the fifth sextile bandwidth of 150 batches of two patterns (i.e. two bi-directional round-trips for ping-ping and two uni-directional round-trips for ping-pong) measured for ping-ping and ping-pong at message sizes between 4KB and 1MB. The error-bars were calculated as the ratio between the maximum bandwidth of ping-ping and the fourth sextile bandwidth of ping-pong (for the upward error-bar) and the ratio between the maximum bandwidth of ping-pong and the fourth sextile bandwidth of ping-ping (for the downward error-bar).

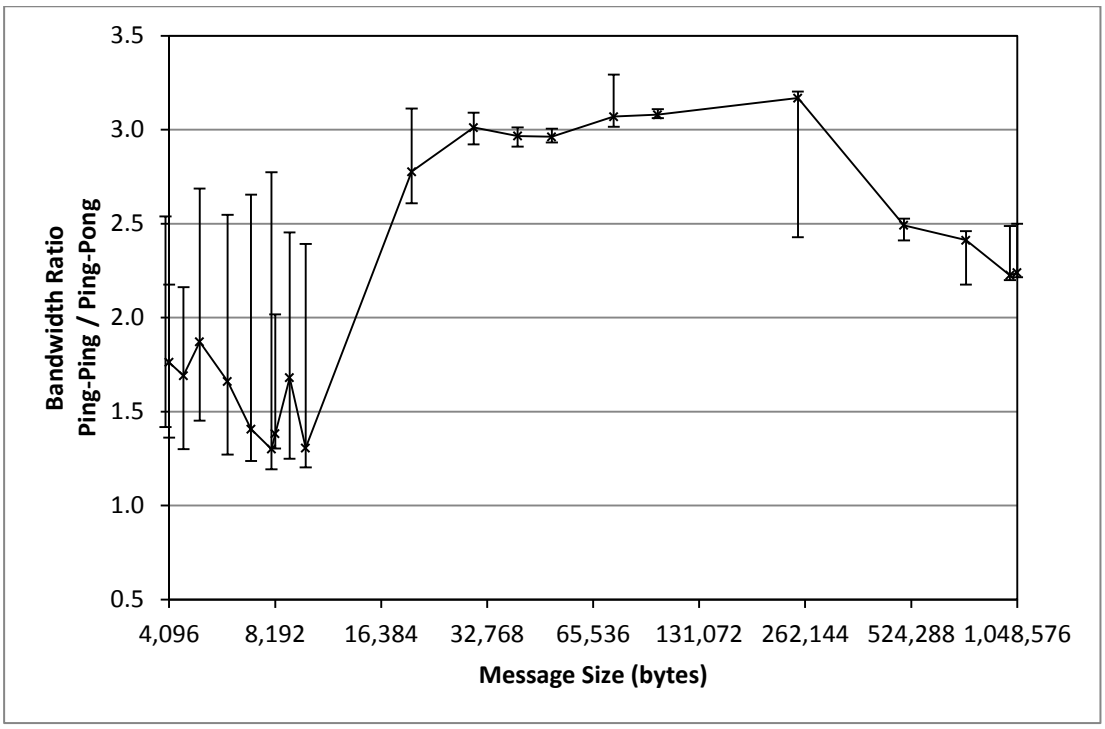


**Figure 19: comparison of ping-ping bandwidth with ping-pong bandwidth for McMPI using thread-to-thread delivery on the Server machine.**

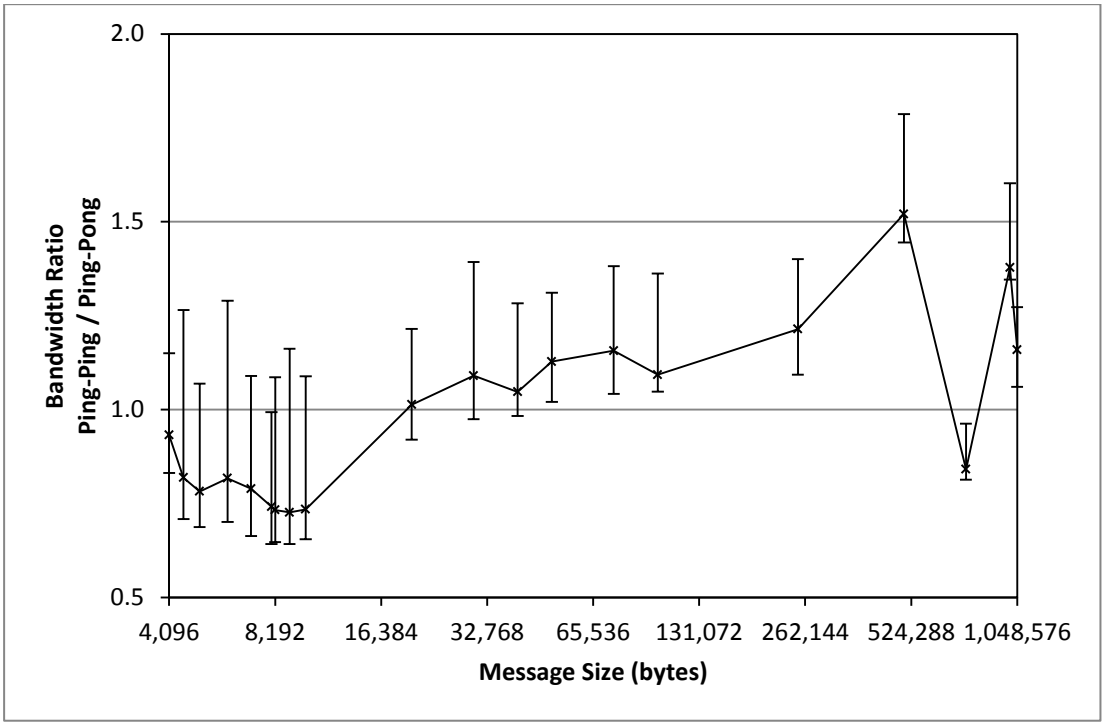


**Figure 20: comparison of ping-ping bandwidth with ping-pong bandwidth for McMPI using thread-to-thread delivery on the Cluster machine**

On the Server machine, the bandwidth ratio is between 1.5 and 2.5 for message sizes up to 20KB and above 256KB, i.e. spanning 2.0, the theoretical ideal value. However, for message

sizes between 32KB and 256KB, the bandwidth ratio increases to approximately 3.0, which is an unexpected, and unexplained, result. The experiment was repeated several times for confirmation.

In contrast, for the Cluster machine the bandwidth ratio is between 0.5 and 1.5 for most message sizes and only exceeds 1.5 for two messages sizes, suggesting that the hardware and operating system have a larger effect on the bi-directional capability than the message-passing library or the application code.

### 5.1.4 Comparison of the Performance of In-Order-Tags and Reverse-Order-Tags

Both the In-Order-Tags and the Reverse-Order-Tags communication patterns send and receive 150 batches of 45 messages from one MPI process to a second MPI process. The sending MPI process sends 45 messages (with tags of 10001 to 10045) and then a 46$^{th}$ message (with tag 0). It then receives a message (with tag 1). The receiving MPI process receives the 46th message (with tag 0) first and then receives 45 messages (with tags 10001 to 10045 for the In-Order-Tags pattern, or 10045 to 10001 for the Reverse-Order-Tags pattern). It then sends a final message (with tag 1). For both patterns, the timing is performed by the receiver MPI process and begins when the first receive operation (with tag 0) is complete and ends when all of the remaining 45 receive operations are complete but before the final message (with tag 1) is sent. The timed operation is local, i.e. no communication occurs between the start and end timestamps. The difference between the times recorded for the two patterns is due to the difference in matching send operations to receive operations. Matching 45 messages for the Reverse-Order-Tags pattern checks 990 unsuccessful combinations as well as the 45 successful combinations checked for the In-Order-Tags pattern.

Figure 21 and Figure 22 show the difference between the times recorded for In-Order-Tags and Reverse-Order-Tags on the server and Cluster machines, respectively. The average difference across all message sizes is also shown for each system.

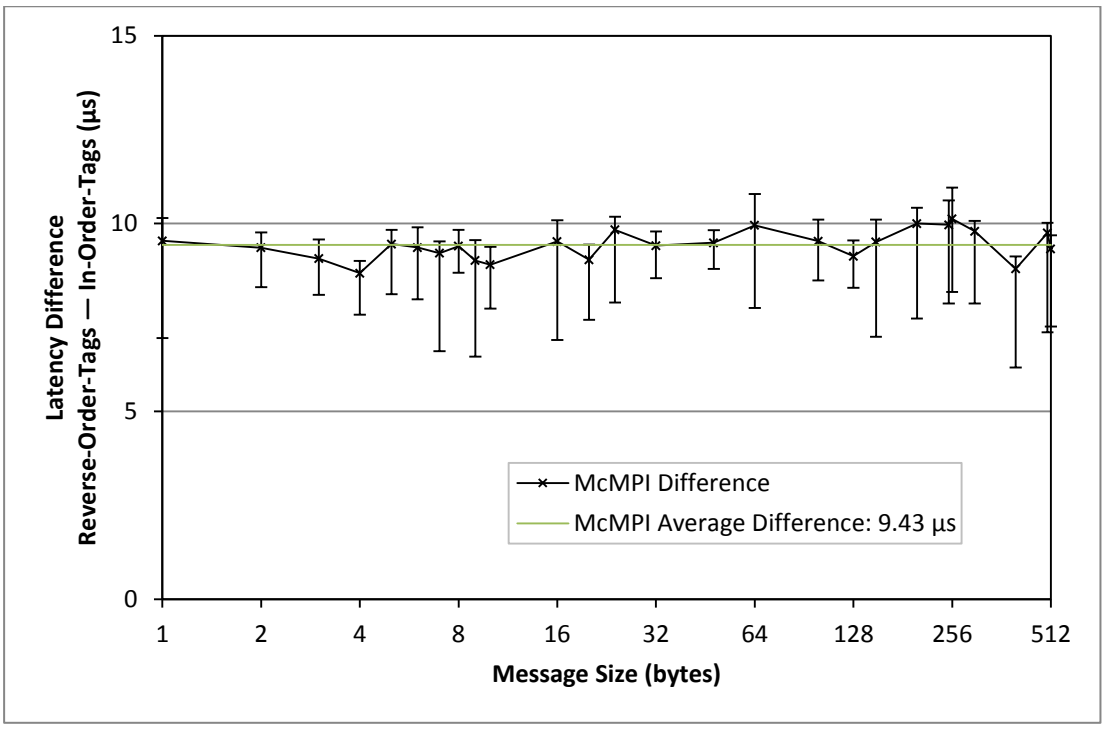


**Figure 21: comparison of In-Order-Tags latency and Reverse-Order-Tags latency for McMPI using thread-to-thread delivery on the Server machine.**

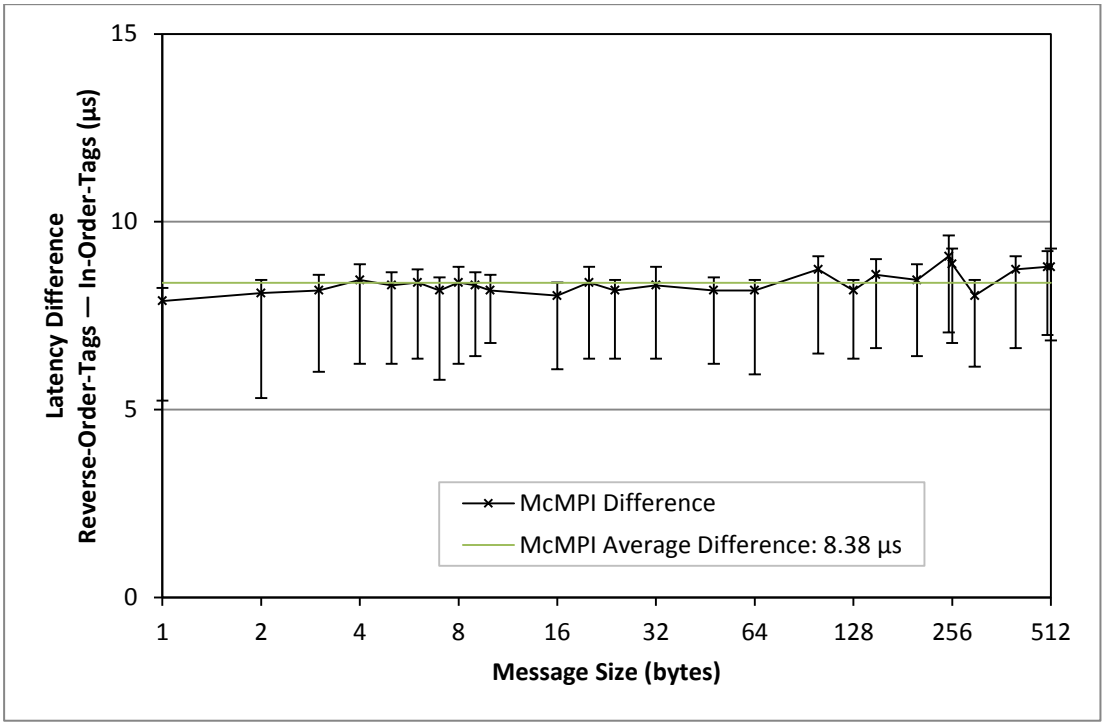


**Figure 22: comparison of In-Order-Tags latency and Reverse-Order-Tags latency for McMPI using thread-to-thread delivery on the Cluster machine**

Dividing the average difference in batch time (9.43µs on the Server machine and 8.38µs on the Cluster machine) by 990 gives an estimate for the time taken by an unsuccessful check,

i.e. 9.53ns for the Server machine and 8.46ns for the Cluster machine. The time per match for the Server machine measured here is greater than the time for the matching algorithm quoted in section 3.2.6 (5.2ns for the third version – measured on the Server machine) because here the unmatched queue elements are more likely to require an L2 cache read rather than be found in registers or L1 cache memory.

These new communication patterns were also used to test MPICH2 on the Server machine; the results are shown in Figure 23. Surprisingly, the difference between the time measured for the `InOrderTags` pattern and `ReverseOrderTags` pattern for MPICH2 is proportional to message size. This suggests that, for MPICH2, the algorithm for matching a send operation with a receive operation involves processing the entire message data, rather than just a fixed size structure such as the message envelope.

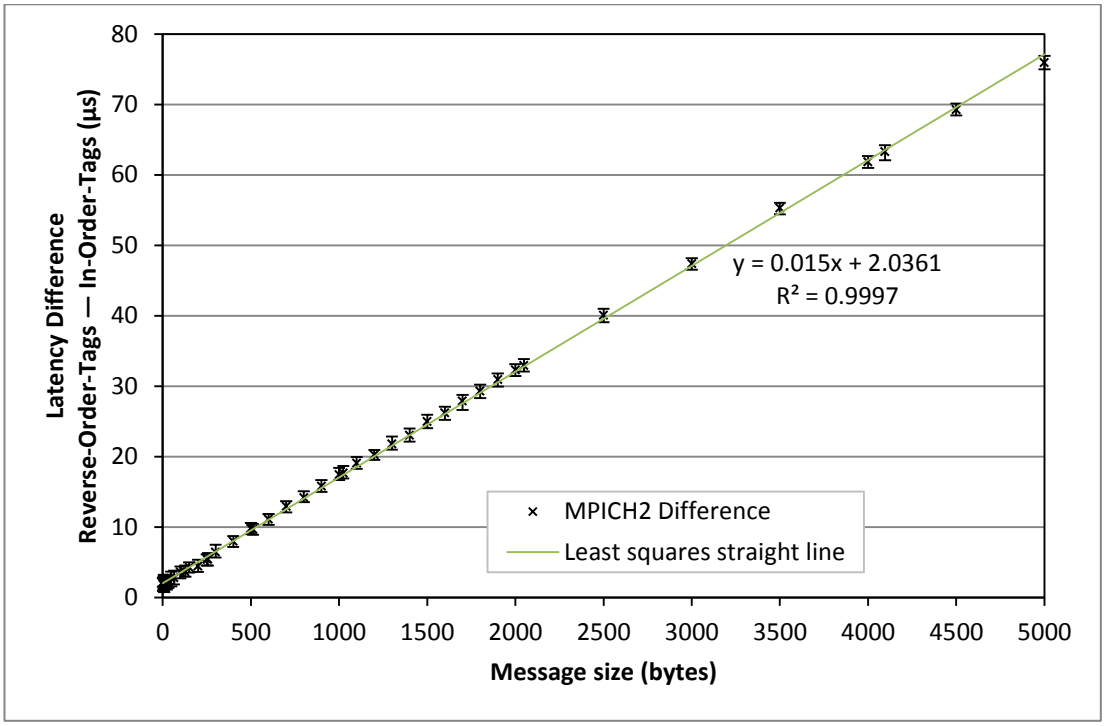


**Figure 23: comparison of In-Order-Tags latency and Reverse-Order-Tags latency for MPICH2 using shared-memory delivery on the Server machine.**

## 5.1.5 Investigation of the Effect of Processor Affinity

Figure 24 and Figure 25 show the ping-pong latency for McMPI (message size is 1 byte) with all possible 2-core processor affinity masks on the Server machine and the Cluster machine,

respectively. The core numbering system used here indicates which bits were set within the affinity mask for each test.

The results for the Server machine conform to the expected pattern, i.e. communication is fastest between processor cores that are in the same chip and share an L2 cache, slower for cores that are in the same processor chip but do not share an L2 cache and slowest for cores that are not in the same processor chip. The effects of some cores handling system interrupts or being given preferential access to system resources are not clear from these results.

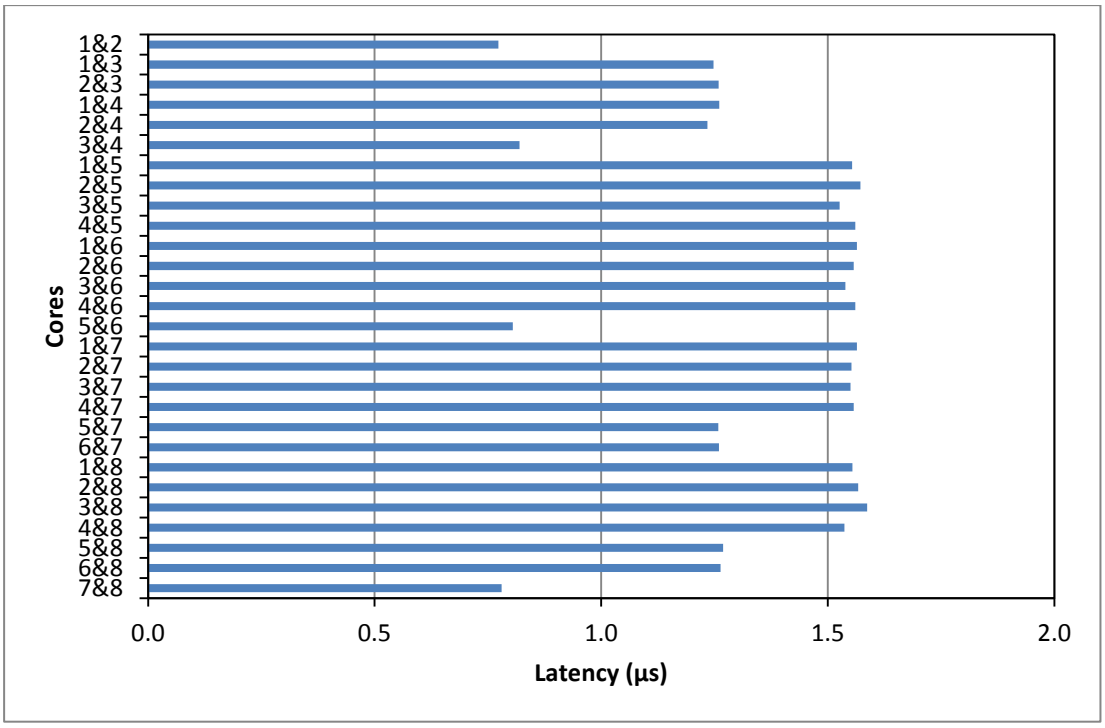


**Figure 24: the effect of affinity on McMPI ping-pong latency on the Server machine.**

While most combinations of processor cores conform to the expected pattern, the results for a single node of the Cluster machine show that combining core 3 and core 4 is much slower than anticipated. The expected pattern predicts that communication using cores 3&4 should be approximately the same performance as cores 1&2 (because these pairs of cores each share an L2 cache) and that communication using other combinations should be slower and approximately the same performance as each other (because they are all within a single processor chip but do not share an L2 cache). All combinations of cores follow this pattern except cores 3&4, with cores 1&2 having 1 byte message-passing latency of 0.82µs and other combinations having latency of 1.7µs, except for cores 3&4, which has a latency

of 6.6µs, over 8 times slower than expected and nearly 4 times slower than other combinations. This effect of processor affinity mask on this system is unexplained.

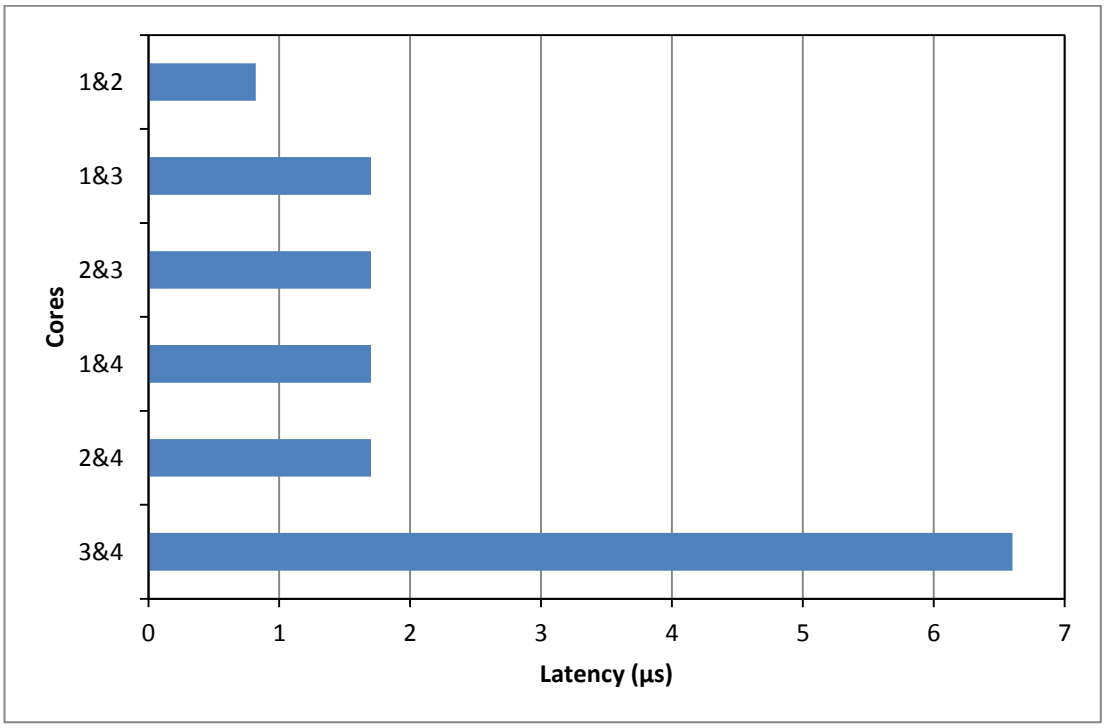


**Figure 25: the effect of affinity on McMPI ping-pong latency on the Cluster machine.**

# 5.2 Process-to-Process Delivery

No communication module that specifically handles process-to-process message delivery has been created for McMPI because it has been designed so that MPI processes running on the same node will be threads within a single operating system process rather than separate operating system processes.

However, for parallel programs where the user code is not thread-safe, different MPI processes must be separate operating system processes. It is possible to configure McMPI to use the TCP sockets communication module with local IP address and port combinations for local operating system processes. This configuration does not use the network hardware to transmit data: an internal loopback interface routes the data from the source socket to the destination socket in software. This configuration is also available to MPICH2 and MS-MPI, although it would not be used in practice because the loopback interface is much slower than using shared-memory. For MPICH2, the default channel setting is `sock`, which disables shared-memory and forces the use of the socket loopback interface for local

processes. For MS-MPI, the `MPICH_DISABLE_SHM` setting must be set to 1 to disable shared memory and force the use of the socket loopback interface.

Section 5.2.1 presents a baseline performance measurement for data transfer via TCP sockets using the loopback interface in C and C#.

Section 5.2.2 presents a summary of Ping-Pong performance results for all versions of McMPI using the process-to-process delivery mechanism, i.e. loopback TCP sockets.

Section 5.2.3 compares the performance of the best version of McMPI with MPICH2 and MS-MPI communicating process-to-process via loopback TCP sockets.

Section 5.2.4 compares the performance of Ping-Ping with that of the Ping-Pong using McMPI, to investigate the bi-directional efficiency of loopback socket communication.

Section 5.2.5 investigates the effect of socket buffer size on communication performance.

### 5.2.1 Performance of TCP Sockets (Loopback) in C and C#

Figure 26 (Server machine) and Figure 27 (Cluster machine) compare the latency of loopback TCP sockets when used in C and C#. Data sizes less than 16 bytes are not included because the minimum header size for MPI messages in McMPI is 17 bytes for an approval protocol message.

As can be seen from Figure 26, on the Server machine the C# socket can achieve the same latency as the C socket for most tested data buffer sizes but it does so less frequently. This variability in performance will mean that, on average over many round-trips, the C# socket will have higher latency than the C socket.

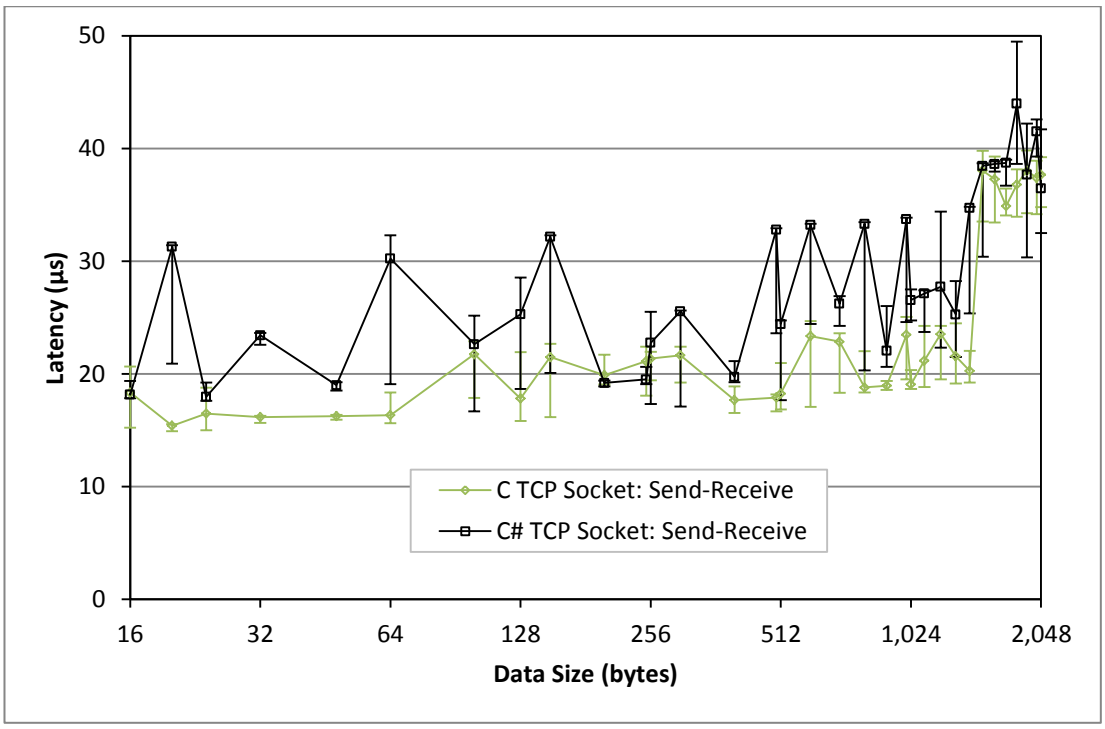


**Figure 26: comparison of latency for a TCP socket on the Server machine using the loopback interface in C (measured by Netpipe with the TCP module) and in C#.**

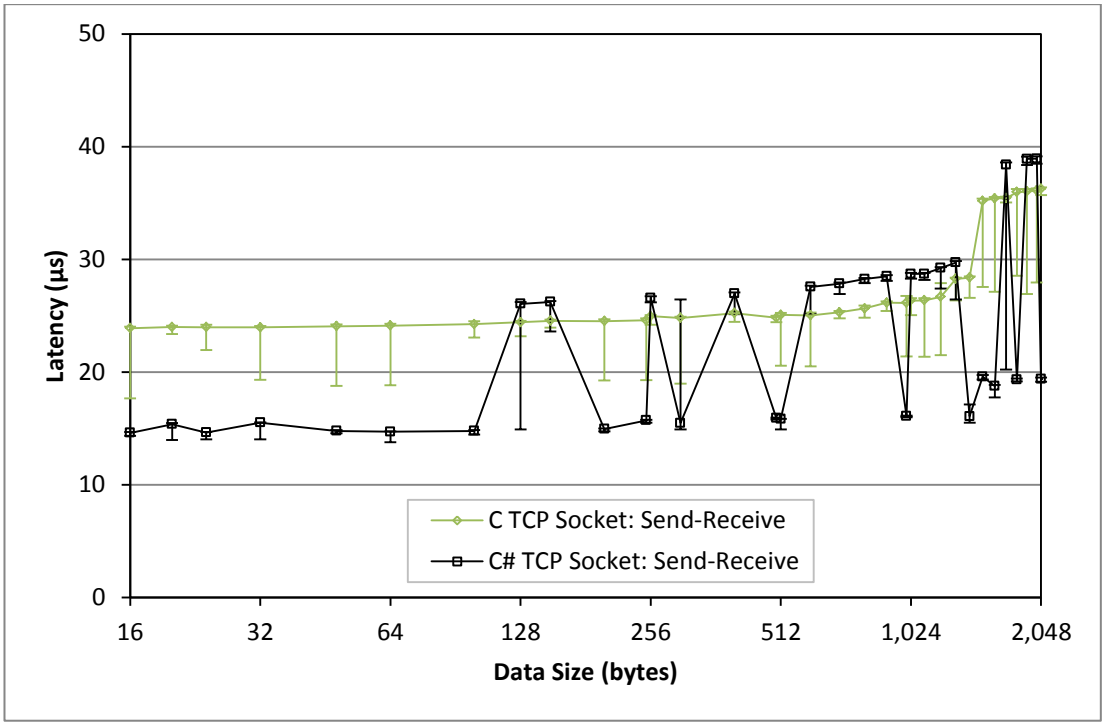


**Figure 27: comparison of latency for a TCP socket on the Cluster machine using the loopback interface in C (measured by Netpipe with the TCP module) and in C#**

Figure 27 shows that, on the Cluster machine, the C# socket performs better than the C socket for many message sizes. The MSDN documentation for the C# socket states that the socket object has been optimised differently on different editions of Microsoft Windows, which may explain the change in performance relative to the performance of the C socket.

The C# socket data presented in Figure 26 and Figure 27 displays a "stepped" distribution, with clusters of data points at particular values and gaps in between. This pattern occurs to varying degrees in many of the subsequent C# plots. Section 5.3.6 examines this in detail.

Table 1 (Server machine) and Table 2 (Cluster machine) summarise the performance of other combinations of communication methods for a C# TCP socket by presenting the lowest first sextile latency measured for data sizes above 16 bytes. Data is sent in one of three ways and received in one of five ways, as follows:

- "Send" uses the blocking `Send` method in the main thread (this cannot be used for an MPI library but is included for comparison to NetPIPE in C, i.e. Figure 26)
- "Send Write Thread" uses the blocking `Send` method in a write-thread (the main thread enqueues a data buffer that the write-thread dequeues and sends)
- "SendAsync" uses the non-blocking `SendAsync` method in the main thread
- "Receive" uses the blocking `Receive` method in the main thread(this cannot be used for an MPI library but is included for comparison to NetPIPE in C, i.e. Figure 26)
- "Receive Read Thread" uses the blocking `Receive` method in a read-thread (the main thread enqueues a data buffer that the read-thread dequeues and fills)
- "Select Receive" uses the `Select` method to wait for data and then uses the `Receive` method in the main thread to receive that data
- "Select Receive Read Thread" uses the `Select` and `Receive` methods in a read-thread (the main thread enqueues a data buffer that the read-thread fills)
- "ReceiveAsync" uses the non-blocking `ReceiveAsync` method in the main thread

| Min for sizes at least 16B | *Receive* | **Receive ReadThread** | **SelectReceive** | **SelectReceive ReadThread** | **ReceiveAsync** |
|---|---|---|---|---|---|
| *Send* | *18.0 µs* | *30.8 µs* | *23.9 µs* | *29.4 µs* | *32.6 µs* |
| Send WriteThread | *35.6 µs* | 42.8 µs | 42.1 µs | 45.2 µs | 47.1 µs |
| **SendAsync** | *26.8 µs* | **33.2 µs** | **30.3 µs** | **36.9 µs** | **37.5 µs** |

**Table 1: lowest latency for each combination of communication methods of a C# TCP socket on the Server machine for data sizes between 16 bytes and 1MB. Combinations that cannot be used for MPI are shown in italics and those used to create versions of the TCP sockets communication module for McMPI are in bold.**

| Min for sizes at least 16B | *Receive* | **Receive ReadThread** | **SelectReceive** | **SelectReceive ReadThread** | **ReceiveAsync** |
|---|---|---|---|---|---|
| *Send* | *14.6 µs* | *27.9 µs* | *18.0 µs* | *29.2 µs* | *27.0 µs* |
| Send WriteThread | *28.6 µs* | 29.7 µs | 34.6 µs | 31.1 µs | 37.0 µs |
| **SendAsync** | *15.9 µs* | **19.9 µs** | **30.7 µs** | **21.2 µs** | **28.1 µs** |

**Table 2: lowest latency for each combination of communication methods of a C# TCP socket on the Server machine for data sizes between 16 bytes and 1MB.**

The data presented in Table 1 and Table 2 indicates there is no justification for creating a TCP communication module using the `SendWriteThread` algorithm because the `SendAsync` algorithm can always achieve a lower latency. The choice of receiving algorithm is less clear: the data from the Server machine indicates that the `SelectReceive` algorithm is the best but the Cluster machine data contradicts this and indicates that algorithm is the worst (when combined with `SendAsync`).

Figure 28 and Figure 29 show the first sextile latency from 1500 timed round-trips for each tested data size between 16 bytes and 1500 bytes using a C# TCP socket with the four communication method combinations. The variability seen for the send-receive command combination in C# (see Figure 26) affects all other command combinations for a C# TCP socket, as can be seen by the error-bars, and it is not eliminated by using processor affinity to restrict the two processes to two separate cores. Monitoring system activity during the test on the Server machine reveals that some work is spread across all processor cores, suggesting that kernel processes are not restricted by the processor affinity mask, which might cause this variability.

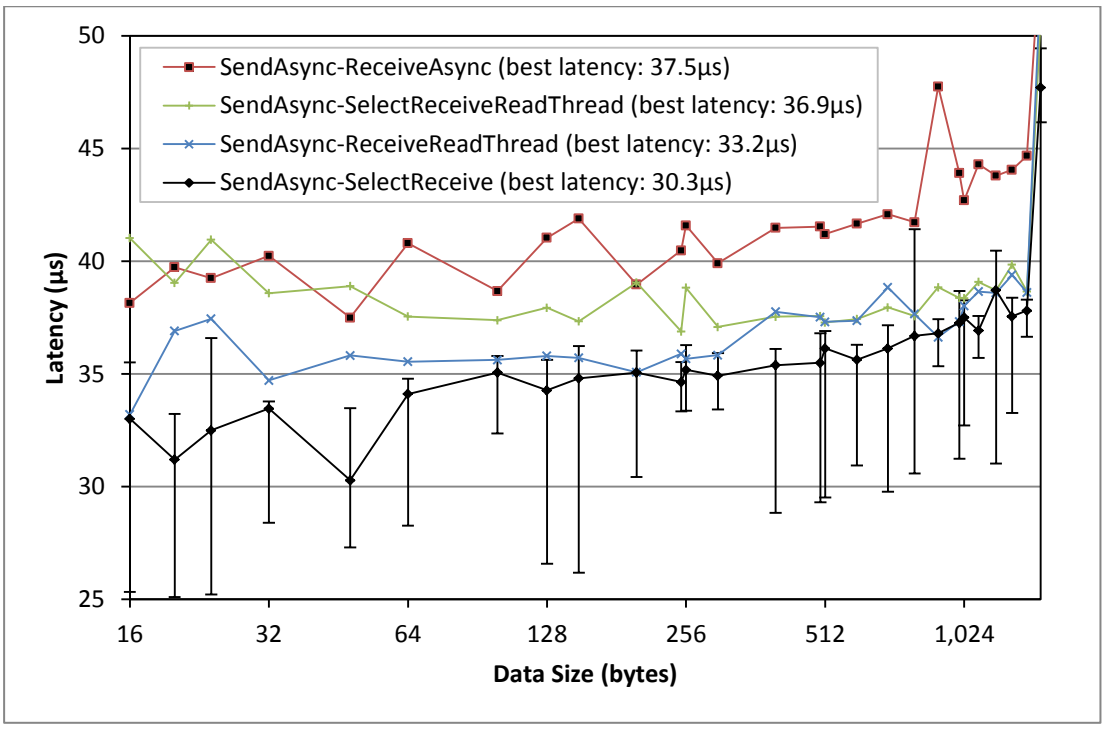


**Figure 28: comparison of C# TCP socket latency on the Server machine using the loopback interface for the four combinations of commands used to create versions of the TCP communication module for McMPI. For visual clarity, error-bars have only been included for the lowest latency combination. Best latency refers to lowest first sexile value.**

The extreme volatility seen on the Cluster machine (Figure 29) is unexplained. The upward error-bar for `SendAsync-ReceiveReadThread` message size 200 bytes indicates that at least two-thirds of the timed round-trips took over 100µs and the downward error-bar for message size 1400 bytes indicates that none of the timed round-trips took less than 100µs. All other upward error-bars in Figure 29 indicate that at least one third of the round trips were completed in less than 40µs. Further analysis (not shown here) exposes that this effect is not limited to the two data points discussed here but is not consistent enough to suggest a pattern or a specific cause.

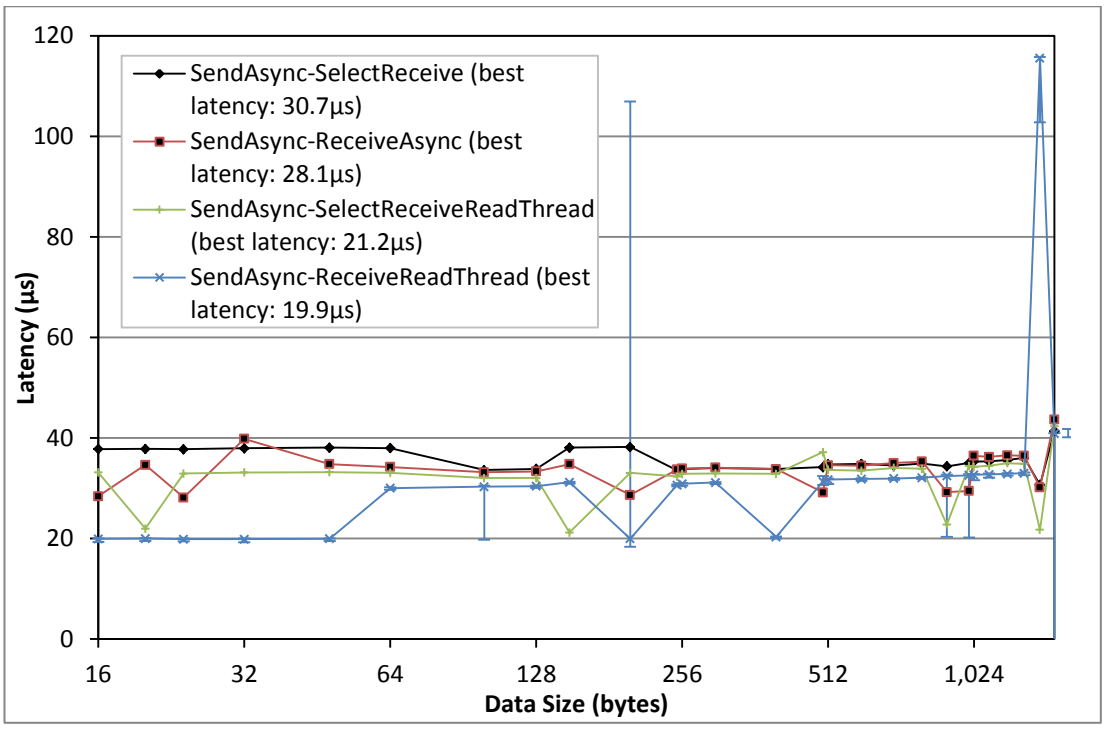


**Figure 29: comparison of C# TCP socket latency on the Cluster machine using the loopback interface for the four combinations of commands used to create versions of the TCP communication module for McMPI. For visual clarity, error-bars have only been included for the lowest latency combination. Best latency refers to lowest first sexile value.**

## 5.2.2 Performance of All Versions of McMPI

Six TCP socket communication modules were created for McMPI – four of them directly relate to the command combinations highlighted in Table 1. All six use the `SendAsync` method to send data, so the naming convention in Figure 30 and Figure 31 concatenates the socket method(s) used to receive data and the number of read-threads employed. Thus, for example, "select receive read thread" from section 5.2.1 becomes "Select-Receive, Read Threads: One" in this section. "Read Threads: One" indicates a single read-thread per operating system process that handles all socket connections to other operating system processes. Having a single read-thread per socket in each operating system process is indicated by "Read Threads: Many". The two new combinations call the `ReceiveAsync` from a read-thread, although it is still the main thread that waits for completion.

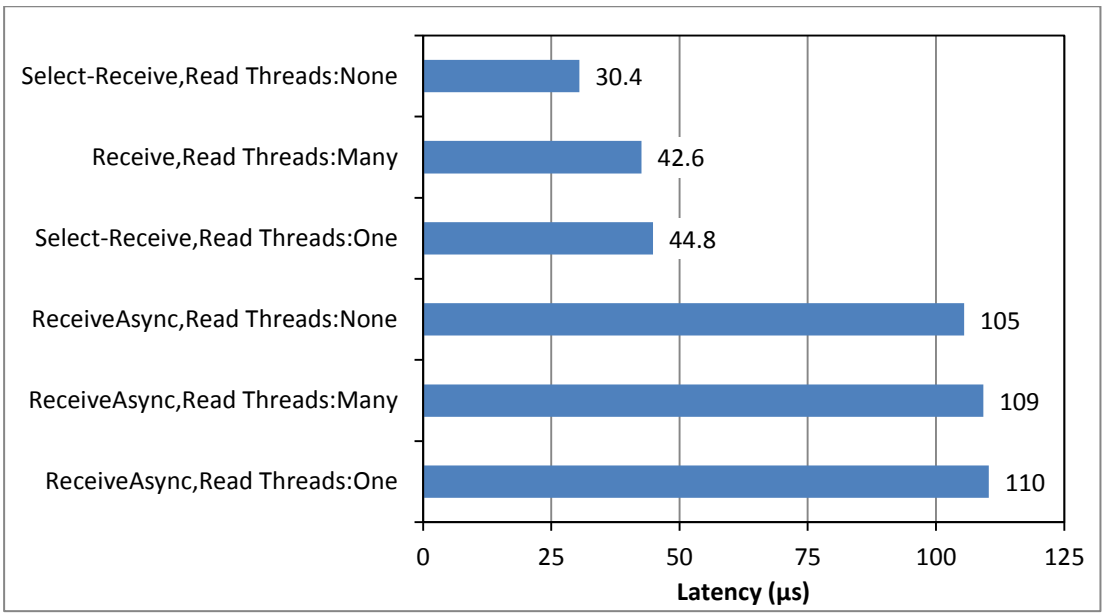


**Figure 30: summary of the lowest process-to-process ping-pong latency on the Server machine (using the loopback interface of TCP sockets in C#) for all tested message sizes (between 1 byte and 1MB) for each version of McMPI.**

Figure 30 shows that the communication module using the `Select` method in the main thread to wait for data to arrive and then the blocking `Receive` method in the main thread to receive that data has significantly lower latency than all the other communication modules tested. The latency of this communication module is similar to the latency of the related command combination for the TCP socket in isolation, i.e. 30.3µs (see Table 1). In addition, it shows the unexpected result that the communication modules involving the `ReceiveAsync` method are between two and four times slower than those that do not. The implementation of the communication modules may be the cause: when using the `Receive` method, the exact size of the data that is known to be present is used (either the size of a protocol header or the data size specified in a protocol header), whereas, when using the `ReceiveAsync` method, the size of the internal receive buffer is always used, which relies on the fact that the `ReceiveAsync` will return control when all available data has been received, even if that is less than the amount requested. The socket may wait for a short time in case more data arrives before returning control, causing the unexpected delay.

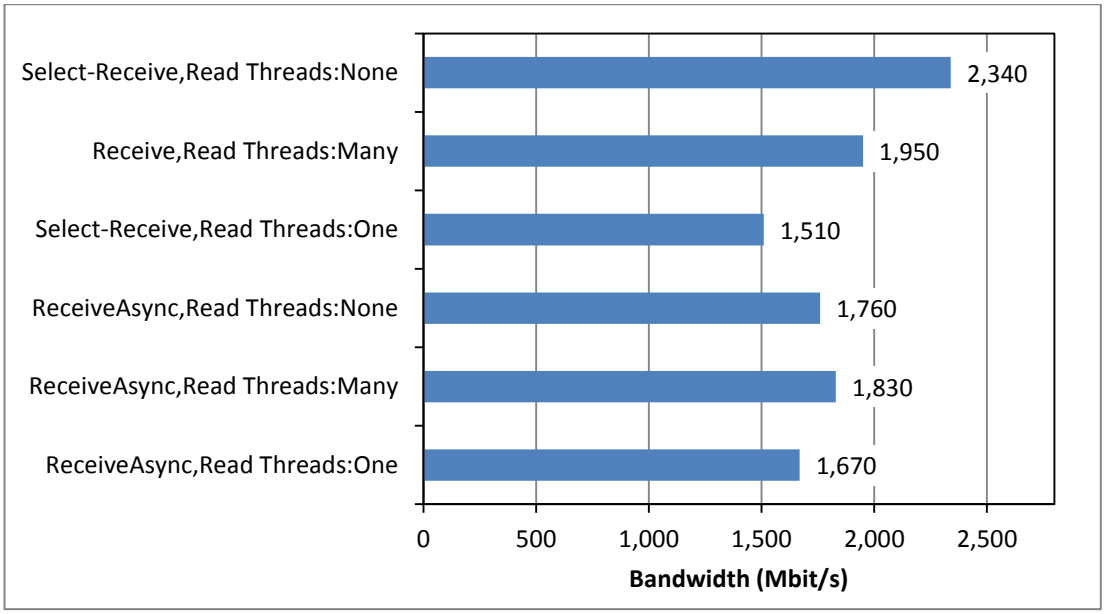


**Figure 31: a summary of the highest process-to-process ping-pong bandwidth on the Server machine (using the loopback interface of TCP sockets in C#) for all tested message sizes (between 1 byte and 1MB) for each version of McMPI.**

As shown in Figure 31, the communication module that achieves the lowest latency also achieves the highest bandwidth. The clear performance difference between the `ReceiveAsync` method and the `Receive` method for latency (seen in Figure 30) is not present here for bandwidth. This suggests that the overhead is a one-off delay (i.e. once per transfer) rather than related to the size of the message data, which supports the hypothesis that, when there is less data available than the amount requested, the socket waits for a short time in case more data arrives before returning control with a part-filled buffer. The `ReceiveAsync` modules could be modified to request only data known to be present but this was not done. The results presented in Figure 28 and Figure 29 for data transfer without message-passing (for which the exact sizes of receive buffers were known in advance and used for all transfers) indicate that, although this approach is much faster, it is still slower than using an entirely different method for receiving socket data, i.e. the `Receive` method with either the `Select` method or a read thread.

### 5.2.3 Comparison of the Performance of McMPI with MPICH2 and MS-MPI

The best performing version of McMPI on the Server machine, i.e. with the "Select-Receive, Read Threads: None" TCP communication module, is compared to MPICH2 and MS-MPI in Figure 32. For both MPICH2 and MS-MPI the shared-memory capability was disabled so

that communication must use TCP sockets via the loopback interface for all three MPI libraries. Processor affinity was used in all of these tests to restrict the two processes to two cores that share an L2 cache and do not handle system interrupts.

The latency of the eager protocol for McMPI matches the latency of MPICH2 for message sizes between 20 bytes and 1400 bytes but is approximately 5µs slower than both MPICH2 and MS-MPI for messages smaller than 20 bytes. The step in latency for message sizes of 1500 bytes or more occurs because the socket must split the message into multiple TCP packets when the amount of data exceeds the maximum segment size, which is 1460 bytes for the Server machine. As expected, the rendezvous protocol for McMPI has approximately three times the latency of the eager protocol for small messages.

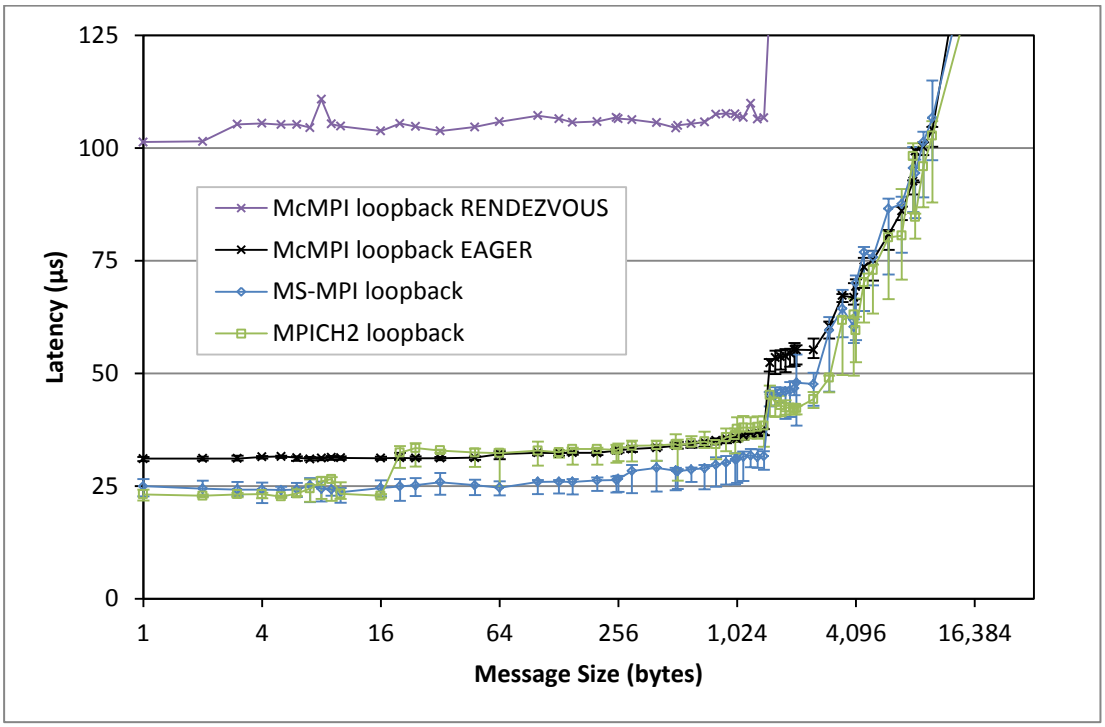


**Figure 32: ping-pong latency for process-to-process message-passing via TCP sockets using the loopback interface on the Server machine. Data-points represent one quarter of the first sextile batch time, from 1500 batches; each batch times two round-trips. Error bars represent the minimum and second sextile latency measurements.**

Figure 33 compares the latency of McMPI with the latency of MS-MPI on the Cluster machine. The version of McMPI used was the same version used for the Server machine test, i.e. the "Select-Receive, Read Threads: None" TCP communication module. As shown in Figure 33, the eager protocol of McMPI can achieve the same minimum latency as MS-MPI for message sizes above 3000 bytes. For message sizes below 3000 bytes, McMPI

generally exhibits poor latency relative to Ms-MPI. However, there are three anomalous data points at message sizes 7 bytes, 32 bytes and 1800 bytes. For message size 7 bytes, the difference between the first sextile latency of MS-MPI (25.1µs) and of McMPI (26.0µs) is only 0.9µs compared with typically 15µs for message sizes between 1 byte and 24 bytes. For message size 32 bytes, the downward error-bar for McMPI (representing the minimum latency) extends to 25.7µs, which is between the first sextile (25.9µs) and the minimum (23.0µs) latency for Ms-MPI, whereas, for message sizes between 48 bytes and 600 bytes, the minimum latency for McMPI is typically 10µs greater than the first sextile latency for MS-MPI. For message size 1800 bytes, the minimum and first sextile latency for McMPI are more commensurate with the trend for MS-MPI for message sizes up to 1300 than the trend for McMPI. These three data points indicate that McMPI can achieve latency for small messages similar to that for MS-MPI but that it does so extremely infrequently.

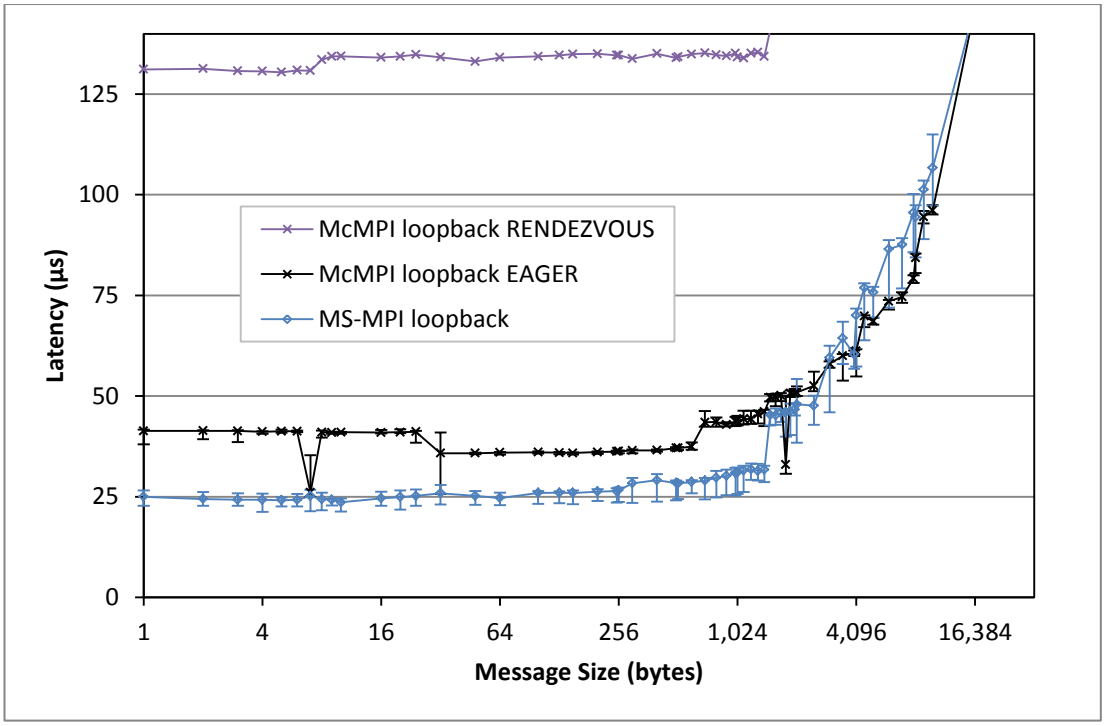


**Figure 33: ping-pong latency for process-to-process message-passing via TCP sockets using the loopback interface on the Cluster machine. Data-points represent one quarter of the first sextile batch time, from 1500 batches; each batch times two round-trips. Error bars represent the minimum and second sextile latency measurements.**

As shown in Figure 34 and Figure 35, the rendezvous protocol for McMPI has the highest bandwidth of all the tested MPI libraries on both the Server machine and the Cluster machine for message sizes between 100,000 bytes and 1MB.

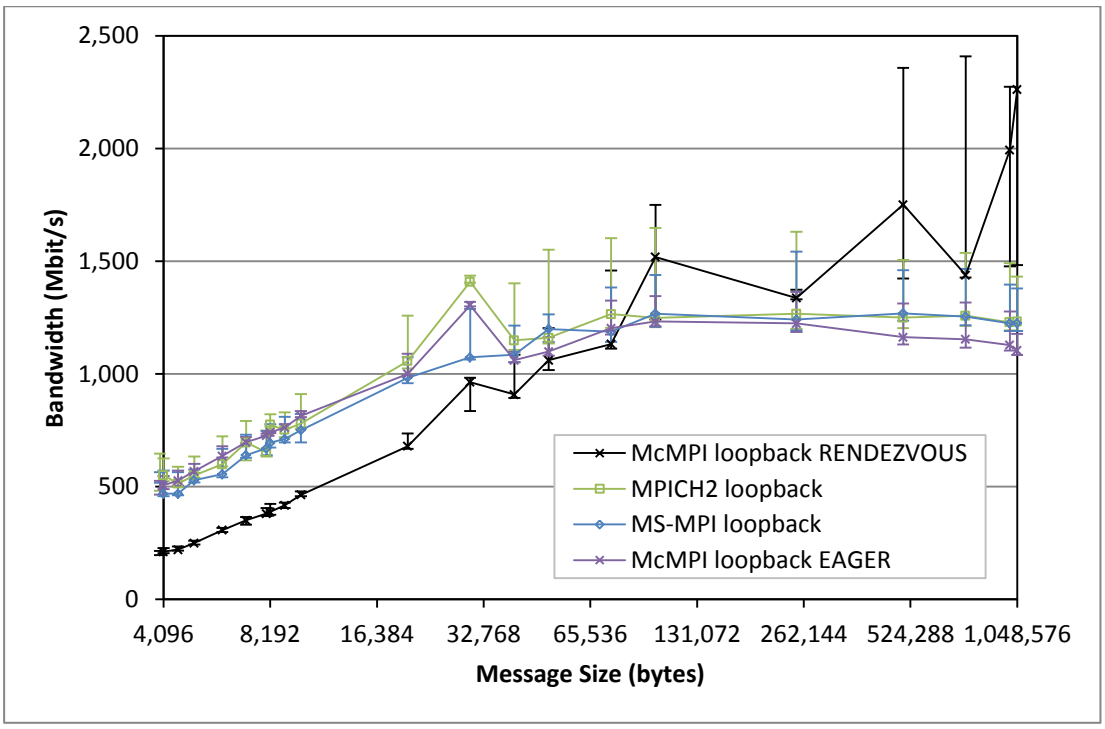


**Figure 34: ping-pong bandwidth for process-to-process message-passing via TCP sockets using the loopback interface on the Server machine. Data-points represent the fifth sextile bandwidth (message size divided by batch time), from 1500 batches; each batch times two round-trips. Error bars represent the maximum and fourth sextile bandwidth.**

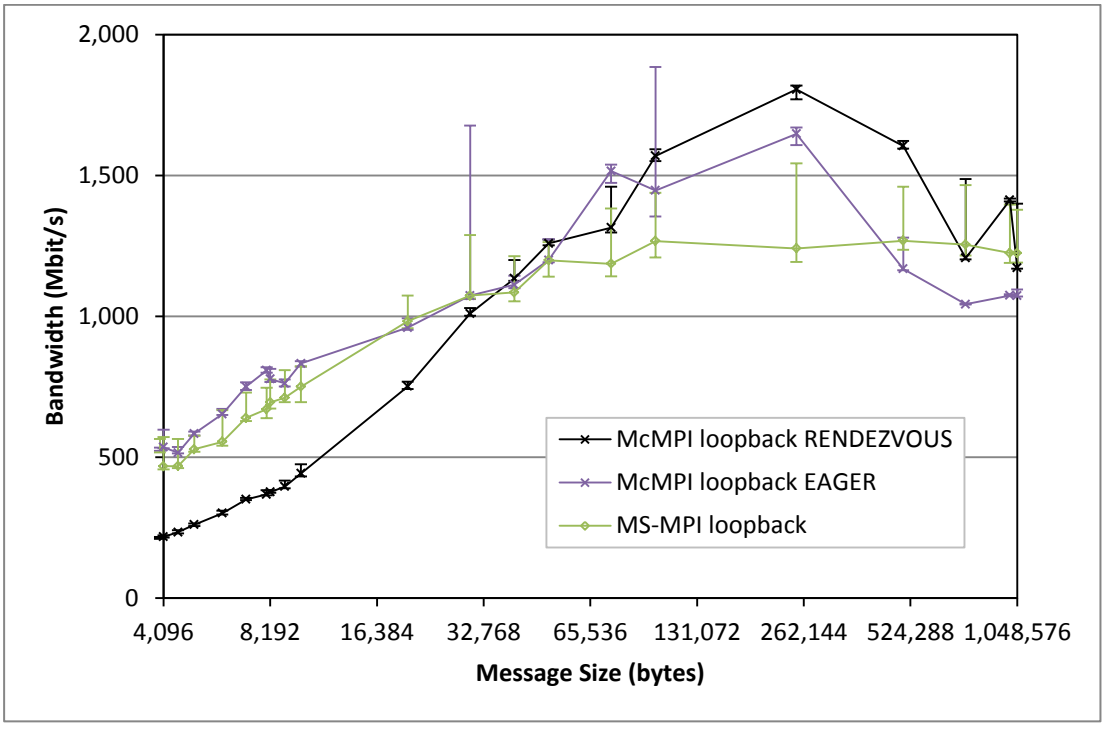


**Figure 35: ping-pong bandwidth for process-to-process message-passing via TCP sockets using the loopback interface on the Cluster machine. Data-points represent the fifth sextile bandwidth (message size divided by batch time), from 1500 batches; each batch times two round-trips. Error bars represent the maximum and fourth sextile bandwidth.**

Below 100,000 bytes the eager protocol for McMPI has higher bandwidth than the rendezvous protocol, which suggests the eager limit for standard mode send (the point at which the MPI library switches between the eager and rendezvous protocol) should be set to 100,000 bytes.

For the Server machine, the bandwidths of MPICH2 and MS-MPI are almost identical to each other, and to the bandwidth of the McMPI eager protocol, for message sizes up to 256KB. For the Cluster machine the bandwidths of McMPI and MS-MPI are very similar for message sizes up to 50,000 bytes but, whereas the bandwidth for MS-MPI remains constant for large message sizes, both the eager protocol and the rendezvous protocol of McMPI show an increase in bandwidth for some larger message sizes.

The rendezvous protocol in McMPI guarantees that the user receive buffer is available before the message data is transferred. This is better than the McMPI eager protocol for large messages because it avoids the creation of a temporary buffer big enough to hold the entire message as well as the memory copy of the message data into this temporary buffer.

The socket buffer size for McMPI is set to 32KB to avoid the drop in performance when using the default buffer size of 8KB. Section 5.2.5 investigates this effect in detail.

### 5.2.4 Comparison of the Performance of Ping-Ping and Ping-Pong

Figure 36 (Server machine) and Figure 37 (Cluster machine) show the ratio of the fifth sextile bandwidth of 150 batches of two patterns (i.e. two bi-directional round-trips for ping-ping and two uni-directional round-trips for ping-pong) measured for ping-ping and ping-pong at message sizes between 4KB and 1MB. The error-bars were calculated as the ratio between the maximum bandwidth of ping-ping and the fourth sextile bandwidth of ping-pong (for the upward error-bar) and the ratio between the maximum bandwidth of ping-pong and the fourth sextile bandwidth of ping-ping (for the downward error-bar).

The bandwidth ratio of ping-ping and ping-pong for McMPI rendezvous on the Server machine rises towards 2.0 for large message sizes but, in general, these results show that the socket loopback interface does not support bi-directional communication at the same bandwidth as uni-directional communication.

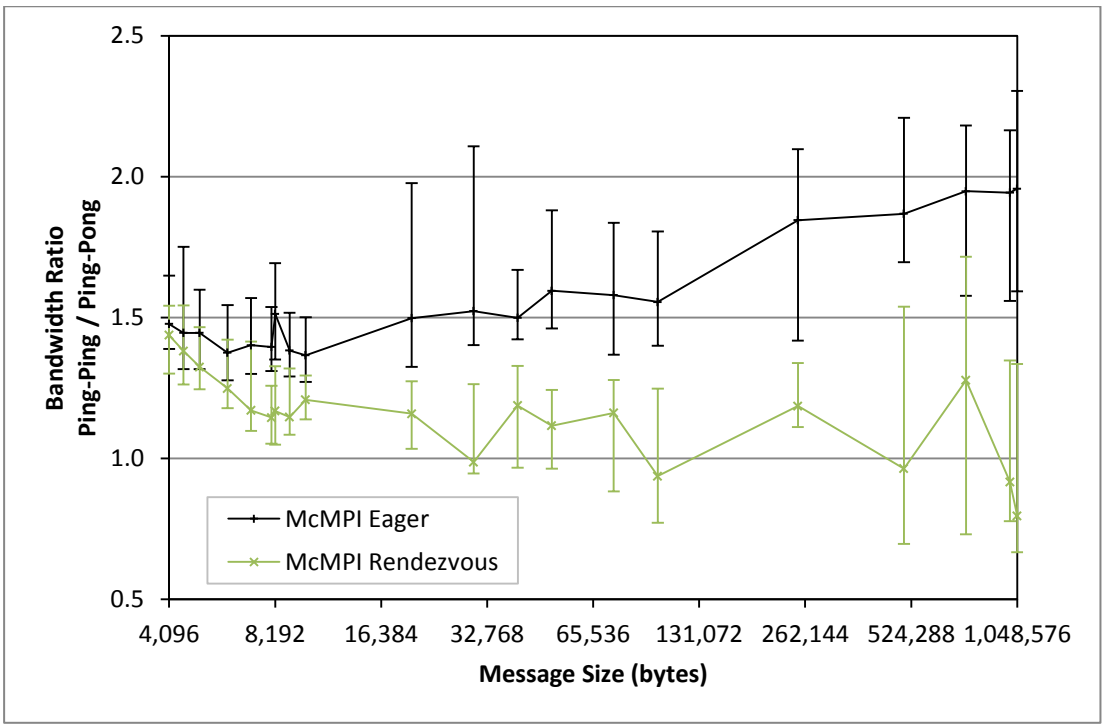


**Figure 36: comparison of ping-ping bandwidth with ping-pong bandwidth for McMPI using loopback sockets for process-to-process message delivery on the Server machine.**

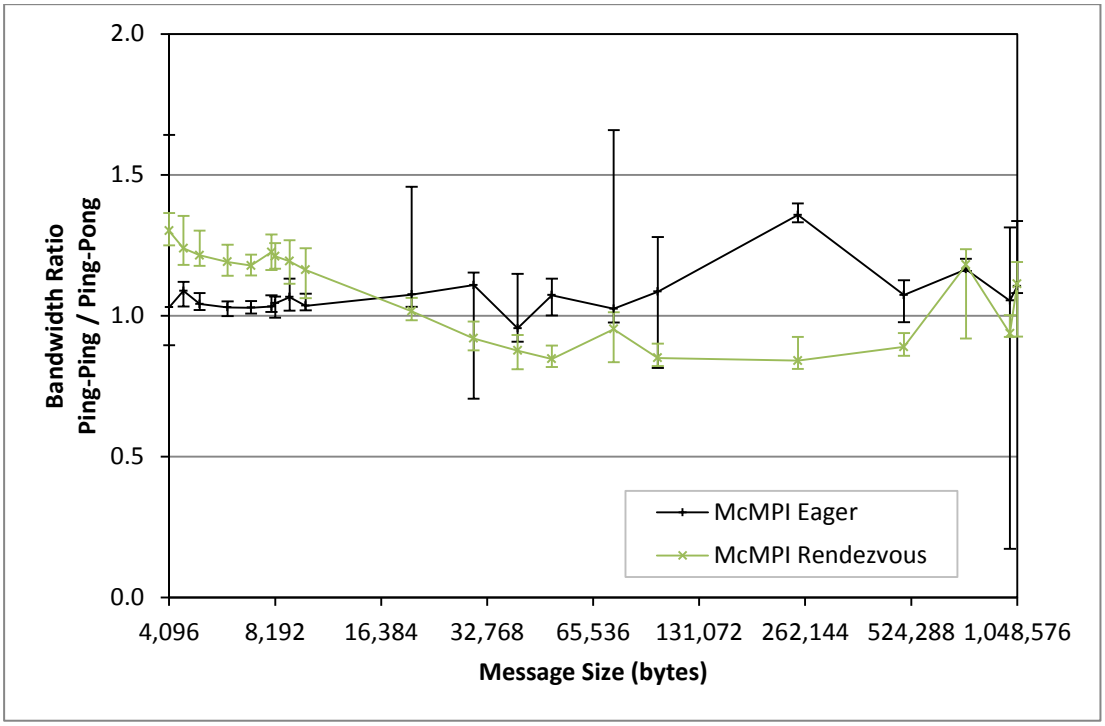


**Figure 37: comparison of ping-ping bandwidth with ping-pong bandwidth for McMPI using loopback sockets for process-to-process message delivery on the Cluster machine.**

### 5.2.5 Investigation of the Effect of Socket Buffer Size

The default buffer size for the socket object in C# is 8KB. The MS-MPI library increases this default buffer size to 32KB. Both the socket object and MS-MPI allow the socket buffer size to be set by the user because different buffer sizes will perform better on different machines. To investigate the effect of this setting for McMPI, four different buffer sizes were tested on the Server machine for message sizes between 4KB and 1MB.

As shown in Figure 38, the size chosen as the default for MS-MPI (i.e. 32KB) causes McMPI to perform better than the other three buffer sizes, especially for message sizes in the region of 32KB and for message sizes above 64KB.

Interestingly, the peak in bandwidth seen at message size 30,000 bytes for a buffer size of 32KB is not seen for any other buffer size, neither at message size 30,000 bytes nor at any message size similar to those buffer sizes. This peak might occur because the L1 cache for the Server machine processors is 32KB, although this is shared between data and instruction caches.

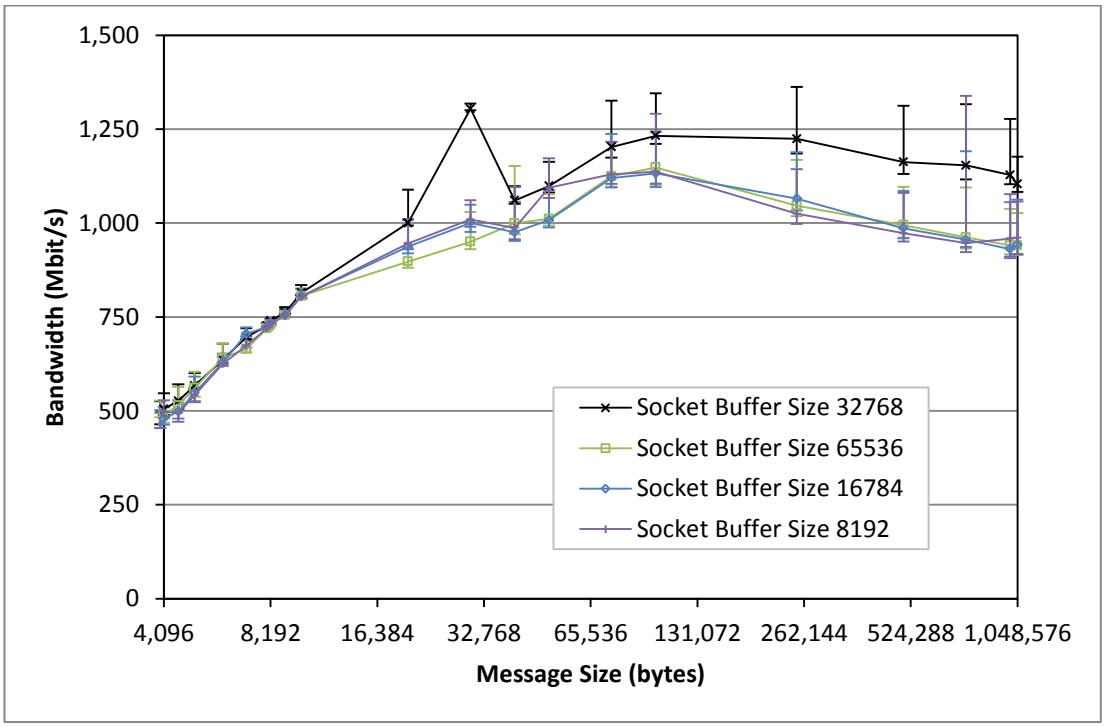


**Figure 38: the effect of socket buffer size on process-to-process bandwidth for the Server machine.**

## 5.3 Machine-to-Machine Delivery

For all three MPI libraries the default settings are sufficient to enable communication via TCP sockets for MPI processes that cannot communicate via shared memory, e.g. when they are executing on different machines. For McMPI, the configuration settings include the IP address and port combination to use for each MPI process. For MPICH2, the default channel setting is `sock`, which disables shared-memory and forces the use of TCP sockets. For MS-MPI, the `MPICH_DISABLE_ND` setting can be set to 1 to disable "network direct" if it is installed and enabled on the machine (it is not installed on any of the machines used here) and force the use of TCP sockets.

Section 5.3.1 presents a baseline performance measurement for data transfer via TCP sockets using Gigabit Ethernet network hardware in C and C#.

Section 5.3.2 presents a summary of Ping-Pong performance results for all versions of McMPI using the machine-to-machine delivery mechanism, i.e. Ethernet TCP sockets.

Section 5.3.3 compares the performance of the best version of McMPI with MPICH2 and MS-MPI communicating machine-to-machine via Ethernet TCP sockets.

Section 5.3.4 compares the performance of Ping-Ping with that of Ping-Pong using McMPI, to investigate the bi-directional efficiency of Ethernet socket communication.

Section 5.3.6 investigates the stepped distribution of data points for C# result-sets.

### 5.3.1 Performance of TCP Sockets (Ethernet) in C and C#

Figure 39 (Desktops) and Figure 40 (Cluster machine) compare the latency of TCP sockets over Gigabit Ethernet when used in C and C#. The same programs that were used to measure loopback latency (see section 5.2.1) were used here but this test was performed with one process on each of the two Desktop machines or with one process on each of two separate nodes of the Cluster machine (the hardware and software configuration is described in section 4.3.3). The data values represent the first sextile value and the error-bars represent the minimum (downward) and second sextile (upward) values from 1500 timed round-trips (with each round-trip time halved to get the latency).

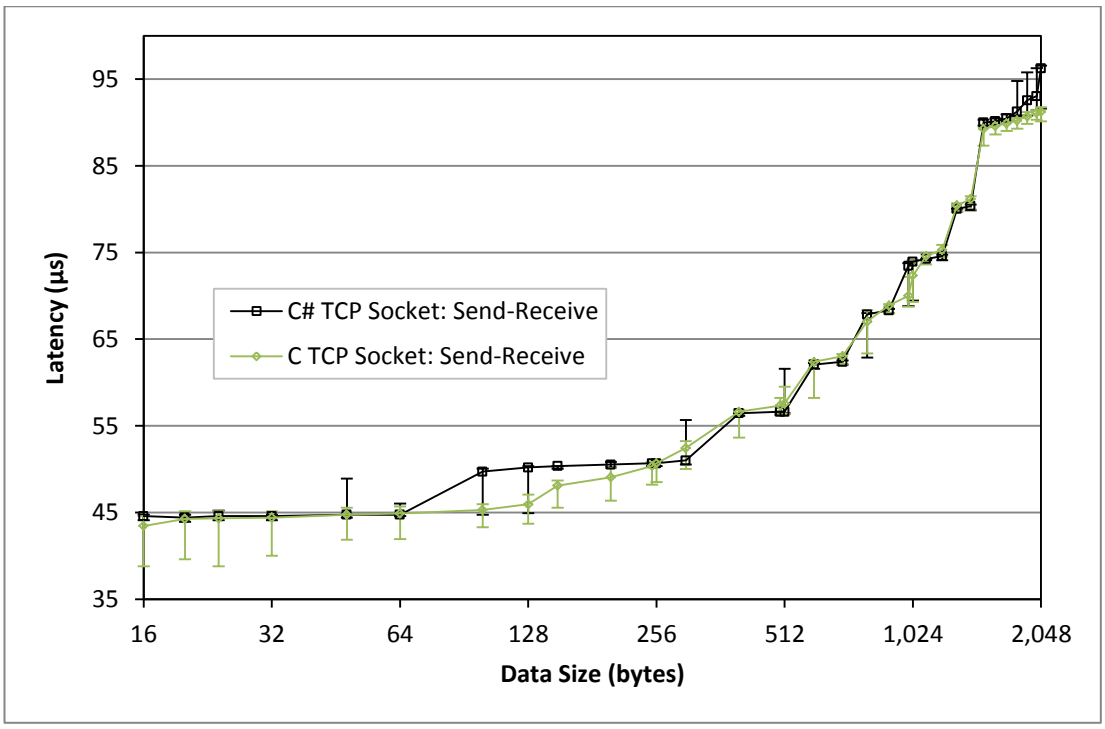


**Figure 39: comparison of latency for a TCP socket on the Desktops using Gigabit Ethernet in C (measured by NetPIPE with the TCP module) and in C#.**

The timing data for Figure 39 clearly shows the "stepped" distribution mentioned in earlier sections for other C# result-sets. Section 5.3.6 examines this in detail.

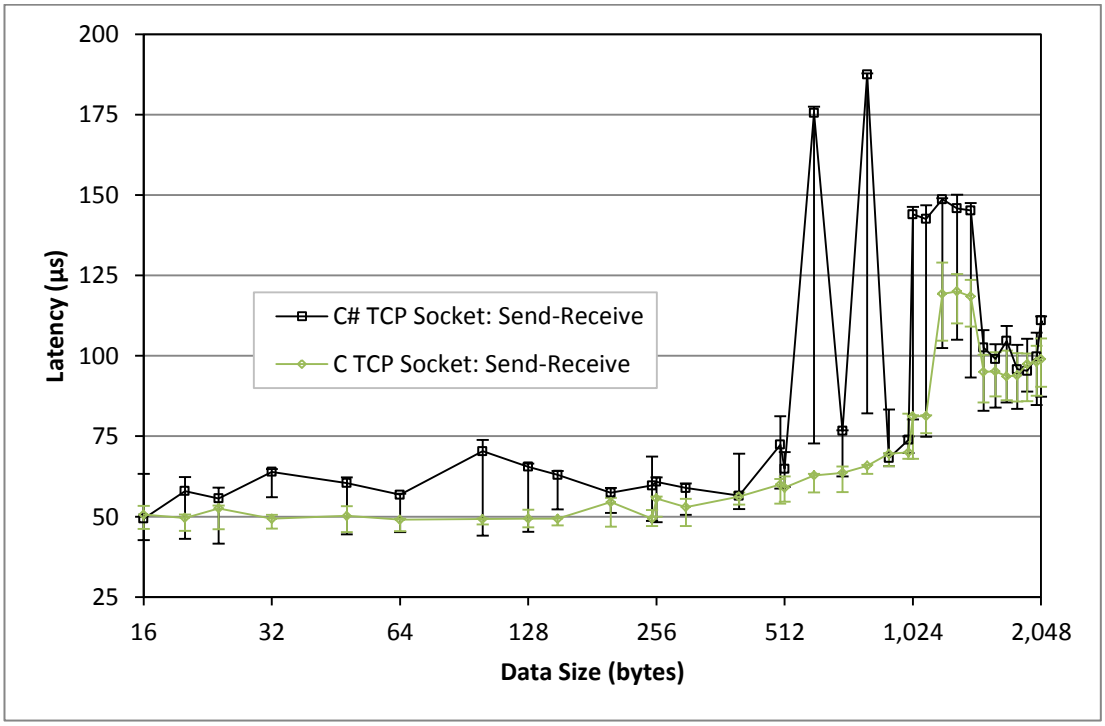


**Figure 40: comparison of latency for a TCP socket on the Cluster machine using Gigabit Ethernet in C (measured by NetPIPE with the TCP module) and in C#.**

As can be seen from Figure 39, the minimum latency achievable by the C socket cannot be achieved by the C# socket, which is up to 5µs slower, for most tested data buffer sizes. However, the most frequent minimum (the first sextile value) is similar for data sizes less than or equal to 64 bytes as well as for data sizes between 256 bytes and 1,400 bytes.

In contrast, Figure 40 shows that, on the Cluster machine, the C# socket can achieve a minimum latency less than that of the C socket for more than half of the tested data sizes. In addition, the first sextile latency for the C# socket is greater than that for the C socket for most data sizes (for data sizes 600 and 800 in particular it is over 100µs higher). This high variability may be due to interference from other jobs running at the same time.

Table 3 (Desktops) and Table 4 (Cluster machine) summarise the performance of communication method combinations for a Gigabit Ethernet TCP socket in C# by presenting the lowest first sextile latency measured for data sizes above 16 bytes. Here, as with loopback sockets in Table 1 and Table 2, there is no justification for creating a TCP communication module using `SendWriteThread` (because `SendAsync` is always better) but it is less clear which receive method to use. On the Desktops, the `SelectReceive` algorithm is best but on the Cluster machine it is the worst algorithm.

| Min for sizes at least 16B | *Receive* | **Receive ReadThread** | **SelectReceive** | **SelectReceive ReadThread** | **ReceiveAsync** |
|---|---|---|---|---|---|
| *Send* | *44.6 µs* | *56.1 µs* | *52.9 µs* | *56.5 µs* | *56.6 µs* |
| Send WriteThread | *64.8 µs* | 74.4 µs | 73.5 µs | 75.1 µs | 76.8 µs |
| **SendAsync** | *53.9 µs* | **64.5 µs** | **62.2 µs** | **64.6 µs** | **64.5 µs** |

**Table 3: lowest latency of all TCP socket communication method combinations measured for data buffer sizes between 16 bytes and 1MB on the Desktops. Combinations that cannot be used for MPI are shown in italics. Combinations used to create TCP sockets communication modules for McMPI are shown in bold.**

| Min for sizes at least 16B | *Receive* | **Receive ReadThread** | **SelectReceive** | **SelectReceive ReadThread** | **ReceiveAsync** |
|---|---|---|---|---|---|
| *Send* | *49.4 µs* | *55.9 µs* | *55.9 µs* | *58.0 µs* | *58.0 µs* |
| Send WriteThread | *58.7 µs* | 61.3 µs | 64.8 µs | 62.4 µs | 67.5 µs |
| **SendAsync** | *53.9 µs* | **59.2 µs** | **60.3 µs** | **59.0 µs** | **60.0 µs** |

**Table 4: lowest latency of all TCP socket communication method combinations measured for data buffer sizes between 16 bytes and 1MB on the Cluster machine.**

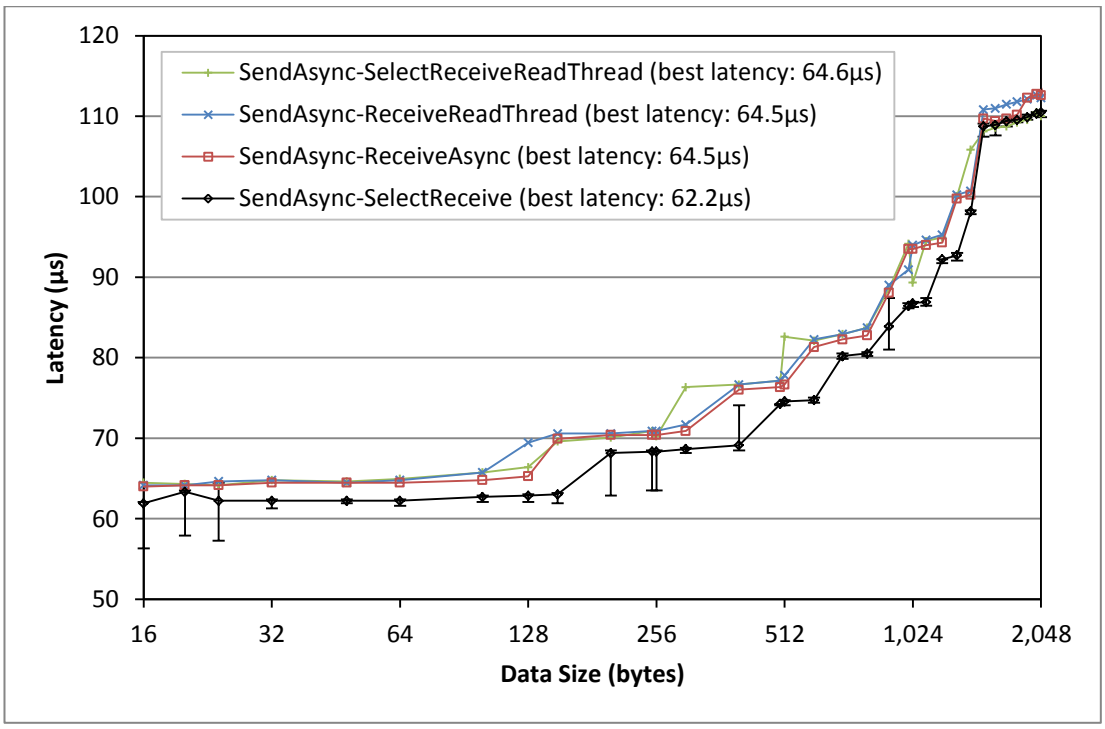


**Figure 41: comparison of C# TCP socket latency using Gigabit Ethernet on the Desktops for four combinations of commands used to create versions of the TCP communication module for McMPI. Error-bars have only been included for the SendAsync-SelectReceive line for visual clarity.**

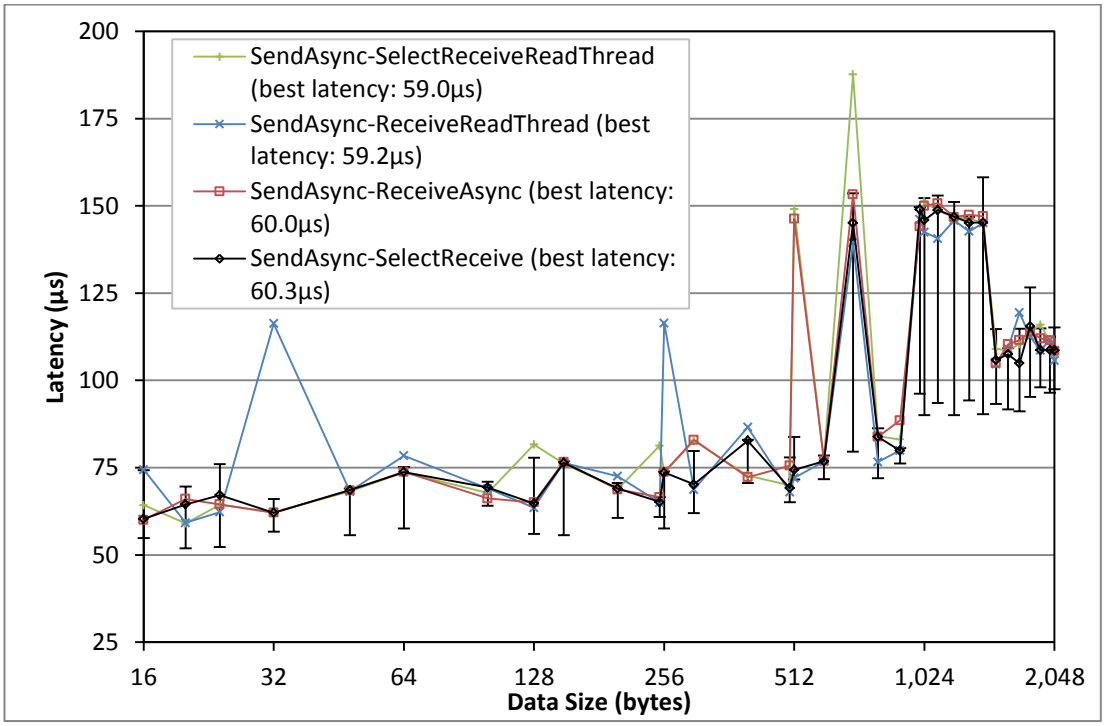


**Figure 42: comparison of C# TCP socket latency using Gigabit Ethernet on the Cluster machine for four combinations of commands used to create versions of the TCP communication module for McMPI. Error-bars have only been included for the SendAsync-SelectReceive line for visual clarity.**

Figure 41 (Desktops) and Figure 42 (Cluster machine) show the first sextile latency from 1500 timed round-trips for each tested data size between 16 bytes and 2KB using the four method combinations used to create communication modules for McMPI.

In Figure 41, the `SelectReceive` algorithm has lowest latency at all data sizes. However, on the Cluster machine (Figure 42), none of the receiving methods is clearly better than the others, although the `SelectReceive` algorithm is the best compromise because it exhibits less volatility over all data sizes.

### 5.3.2 Performance of All Versions of McMPI

Figure 43 shows the lowest first sextile latency on the Desktops of 150 batches of two round-trips (for tested message sizes between 1 byte and 1MB) for each version of McMPI, i.e. for each version of the TCP communication module. The unexpected result that the communication modules that use the `ReceiveAsync` method are much slower than those that do not (see section 5.2.2) is replicated here. This further supports the hypothesis that requesting more data than is available and relying on the socket to return control before filling the receive buffer may cause the socket to wait for more data to arrive and delay the return of control.

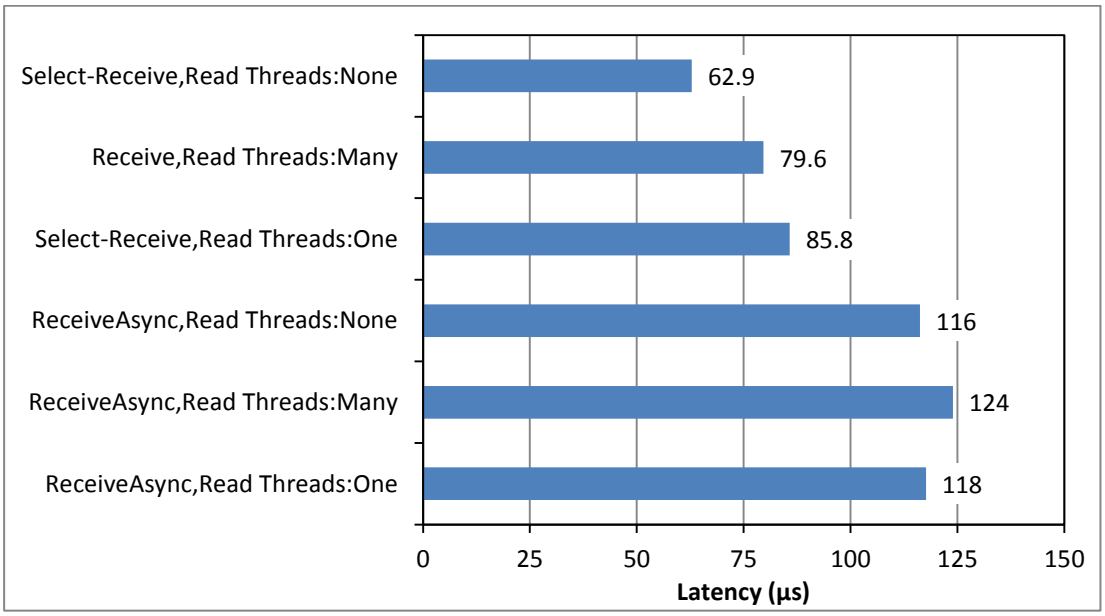


**Figure 43: summary of the lowest first sextile latency from 1500 batches of 2 round-trips for machine-to-machine ping-pong (using Gigabit Ethernet TCP sockets in C# on the Desktops) for all tested message sizes (between 1 byte and 1MB) for each version of McMPI.**

The latency measured for “Select-Receive, Read Threads: None” module is similar to the latency of the related command combination for the TCP socket in isolation, i.e. 62.2µs from Table 3. However, the latencies measured for “Receive, Read Threads: Many” and “Select-Receive: Read Threads: One” are surprisingly high compared with the corresponding latencies measured for data transfer using the TCP socket without message-passing (64.5µs and 64.6µs from Table 3).

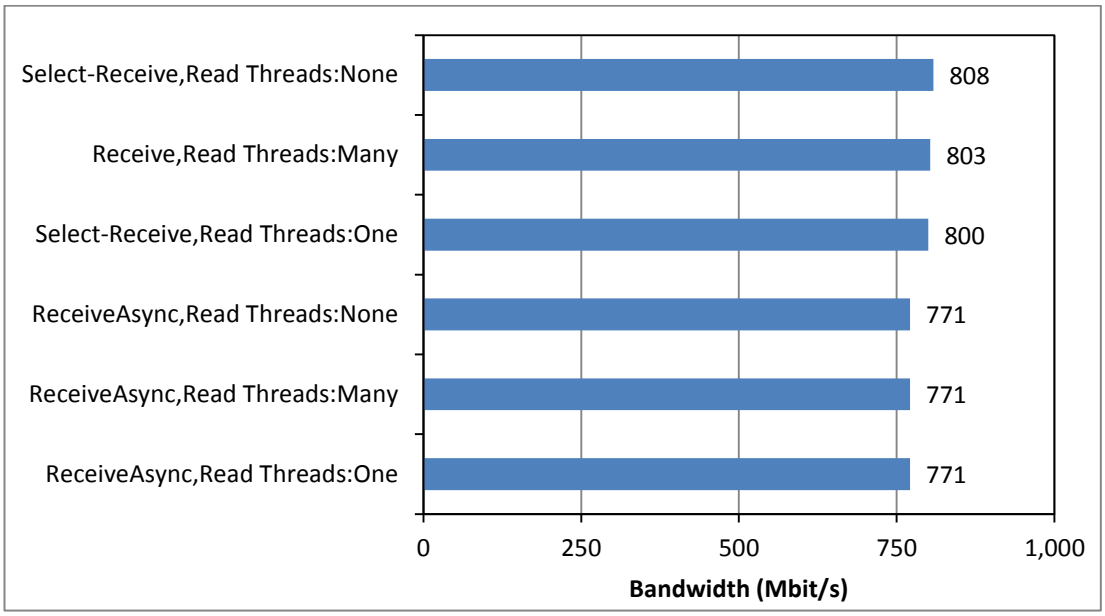


**Figure 44: summary of the highest fifth sextile bandwidth from 150 batches of 2 round-trips for machine-to-machine ping-pong (using Gigabit Ethernet TCP sockets in C# on the Desktops) for all tested message sizes (between 1 byte and 1MB) for each version of McMPI.**

Figure 44 shows the highest fifth sextile bandwidth of 150 batches of two round-trips (for tested message sizes between 1 byte and 1MB) for each version of McMPI on the Desktops. The “Select-Receive, Read Threads: None” communication module, which achieves highest bandwidth for loopback sockets, also achieves highest bandwidth for Ethernet sockets.

### 5.3.3 Comparison of the Performance of McMPI with MPICH2 and MS-MPI

Figure 45 and Figure 46 compare the latency of the eager protocol for McMPI with the latency of MPICH2 on the Desktops and the latency of MS-MPI on the Cluster machine, respectively.

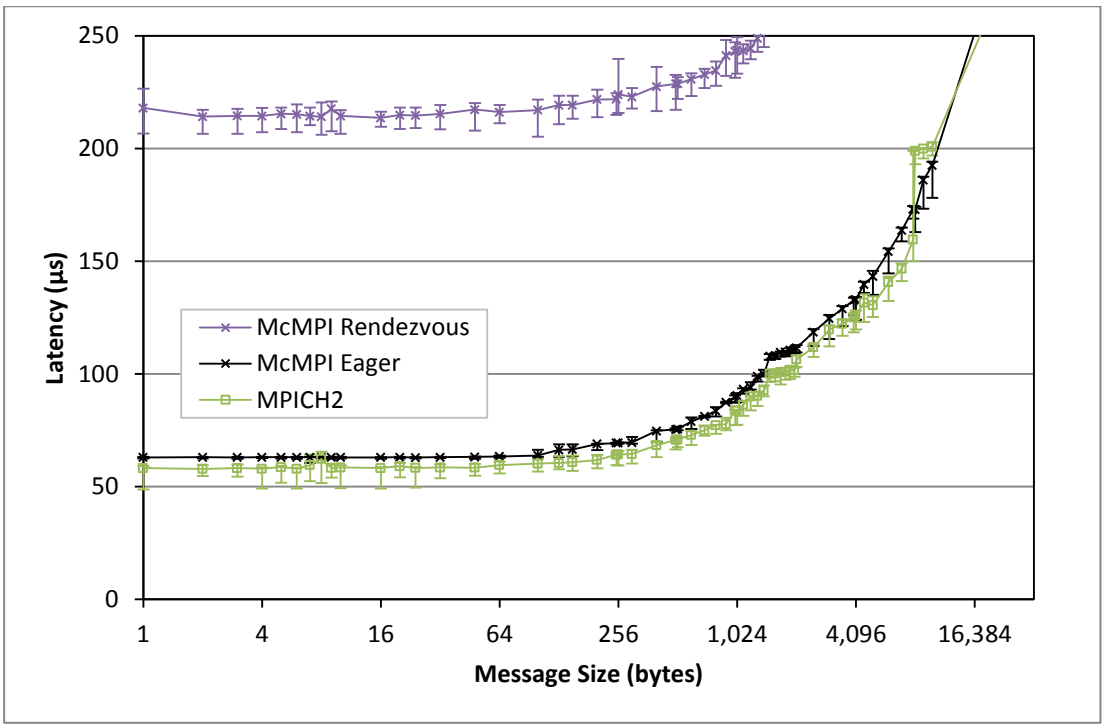


**Figure 45: ping-pong latency for machine-to-machine message-passing via TCP sockets using Gigabit Ethernet on the Desktops. Data-points represent one quarter of the first sextile batch time, from 1500 batches; each batch times two round-trips. Error bars represent the minimum and second sextile latency measurements.**

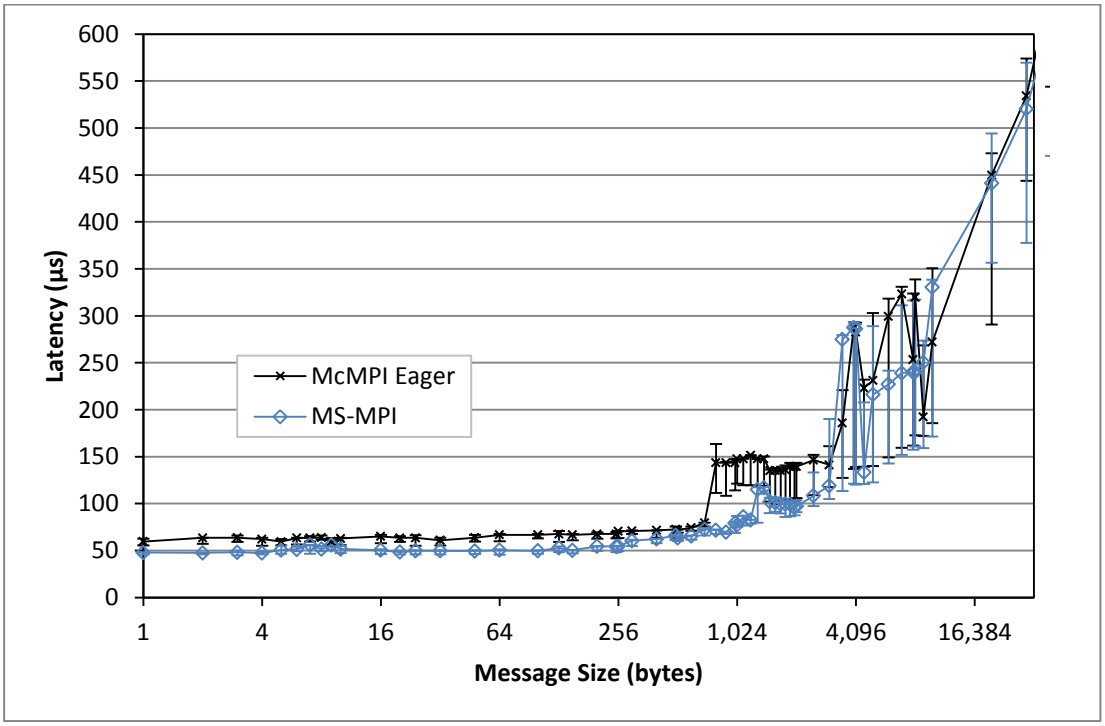


**Figure 46: ping-pong latency for machine-to-machine message-passing via TCP sockets using Gigabit Ethernet on the Cluster machine.**

In both cases, the eager protocol for McMPI has higher latency than the comparison MPI library for all tested message sizes up to 3,000 bytes. For MPICH2, the difference is relatively small: the average difference between McMPI and MPICH2 for message sizes up to 700 bytes is 4.8µs, i.e. 7.8%. For MS-MPI, difference is larger: the average difference for message sizes up to 700 bytes is 12.1µs, i.e. 23.2%.

Figure 47 and Figure 48 compare the bandwidth of the eager and rendezvous protocols for McMPI with the bandwidth of MPICH2 on the Desktops and the bandwidth of MS-MPI on the Cluster machine, respectively.

The rendezvous protocol for McMPI performs less well than the eager protocol on the Desktops for all tested message sizes except 1MB. On the Cluster machine, the rendezvous protocol performs better than the eager protocol for almost all tested message sizes greater than 16KB.

On both machines, the bandwidth of the eager protocol for McMPI exceeds that of the comparison library for some tested message sizes. Relative to MPICH2, the eager protocol of McMPI has better bandwidth for message sizes between 8KB and 10KB only. Relative to MS-MPI, the eager protocol of McMPI has better bandwidth for message sizes 9,000 and 10,000 bytes and for all tested message sizes above 64KB.

The rendezvous protocol of McMPI also achieves higher bandwidth than MS-MPI on the Cluster machine for message sizes 9,000 and 10,000 bytes and for all tested message sizes above 64KB.

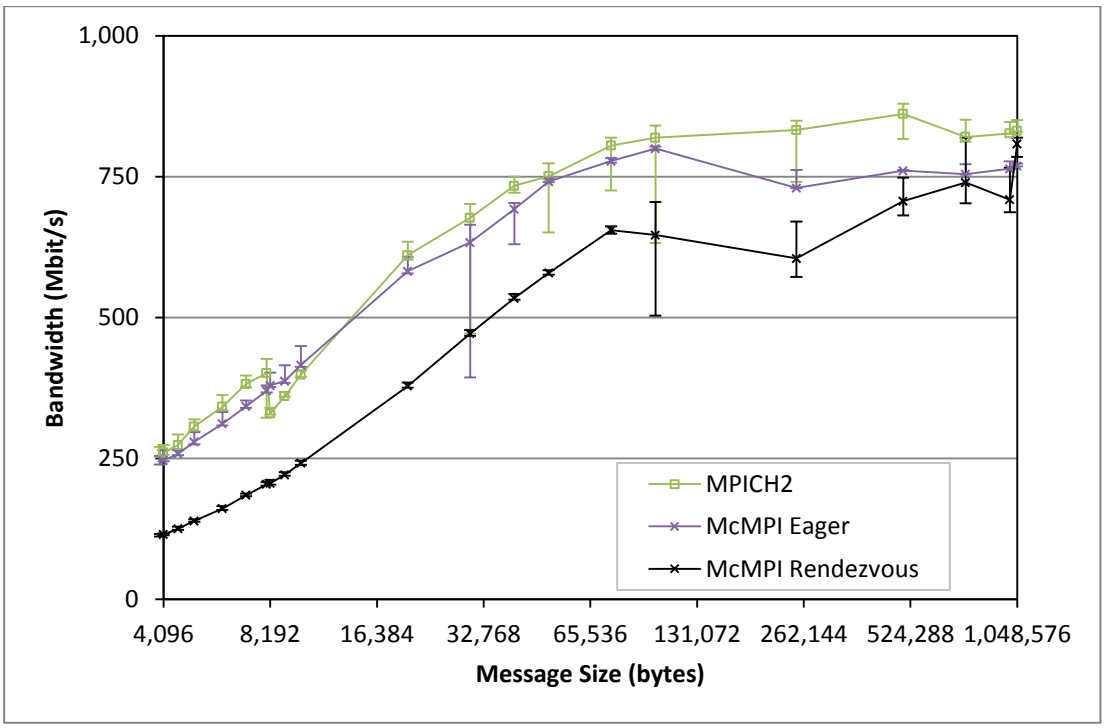


**Figure 47: ping-pong bandwidth for machine-to-machine message-passing via TCP sockets using Gigabit Ethernet on the Desktops. Data-points represent the fifth sextile bandwidth, from 1500 batches; each batch times two round-trips. Error bars represent the fourth sextile and maximum bandwidth measurements.**

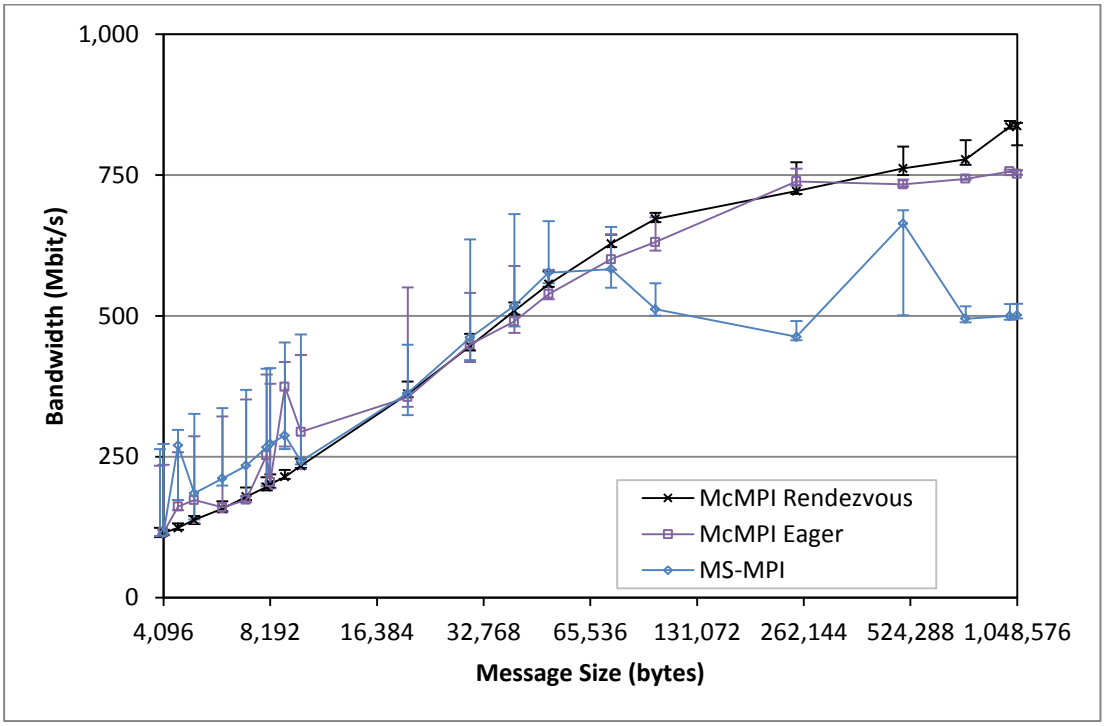


**Figure 48: ping-pong bandwidth for machine-to-machine message-passing via TCP sockets using Gigabit Ethernet on the Cluster machine.**

### 5.3.4 Comparison of the Performance of Ping-Ping and Ping-Pong

Figure 49 (Desktops) and Figure 50 (Cluster machine) show the ratio of the fifth sextile bandwidth of 150 batches of two patterns (i.e. two bi-directional round-trips for ping-ping and two uni-directional round-trips for ping-pong) measured for ping-ping and ping-pong at message sizes between 4KB and 1MB. The error-bars were calculated as the ratio between the maximum bandwidth of ping-ping and the fourth sextile bandwidth of ping-pong (for the upward error-bar) and the ratio between the maximum bandwidth of ping-pong and the fourth sextile bandwidth of ping-ping (for the downward error-bar).

The bandwidth ratio of ping-ping and ping-pong using the eager protocol of McMPI on the Cluster machine is approximately 2.0 for all tested message sizes, which shows that the Gigabit Ethernet sockets can support bi-directional communication at the same bandwidth as uni-directional communication on that machine. The bandwidth ratio of ping-ping and ping-pong using the rendezvous protocol of McMPI approaches 2.0 for large message sizes.

The results for the Desktops are more variable but, in general, show that bi-directional communication achieves lower bandwidth than uni-directional communication.

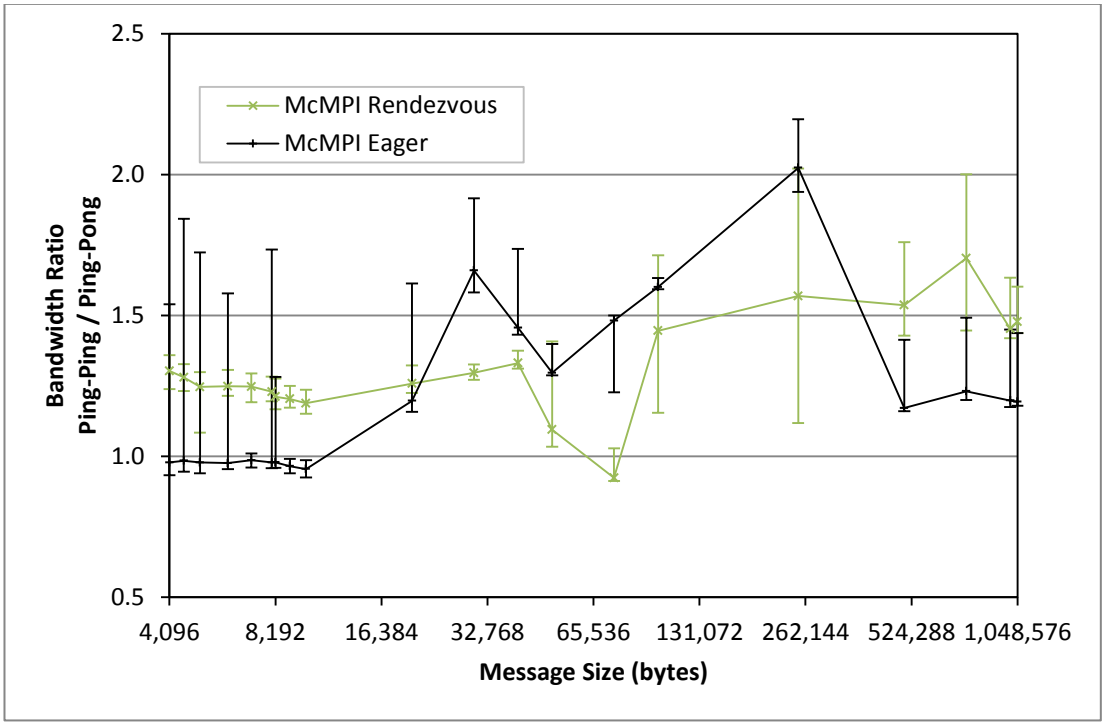


**Figure 49: comparison of ping-ping bandwidth with ping-pong bandwidth for McMPI using Gigabit Ethernet TCP sockets for machine-to-machine message delivery on the Desktops.**

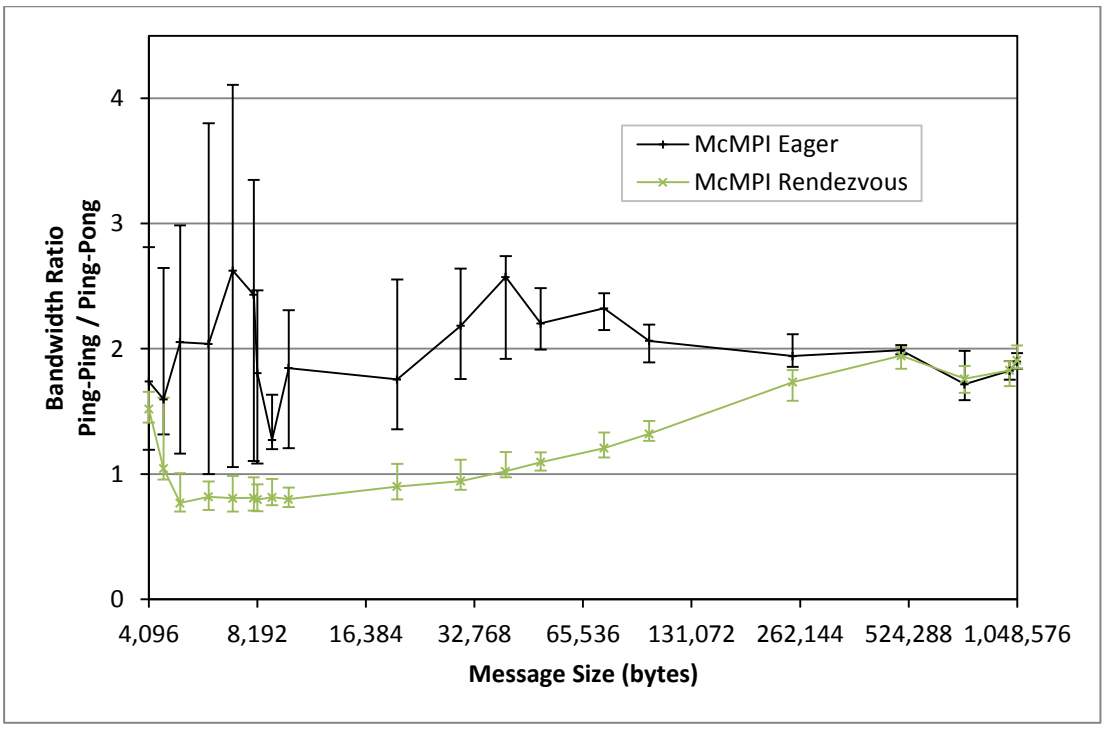


**Figure 50: comparison of ping-ping bandwidth with ping-pong bandwidth for McMPI using Gigabit Ethernet TCP sockets for machine-to-machine message delivery on the Cluster machine.**

### 5.3.5 Investigation of Transfer Protocol Message Implementations

Three methods of implementing the transfer protocol message of the rendezvous protocol were tested. The header for the transfer protocol message was always created in memory allocated by the McMPI library (the write-buffer). The data for the message is supplied to the McMPI library in a buffer allocated by the user (the send buffer).

Transfer option 1 sends the used portion of the write-buffer (which contains the message header) and the used portion of the send-buffer (which contains the message data). This implementation forces the socket object to pin the memory associated with the send-buffer for the duration of the send command and to unpin it before returning control back to the user. Note that the write-buffer remains pinned throughout the lifetime of McMPI.

Transfer option 2 fills the unused portion of the write-buffer by copying message data from the send-buffer. It then sends the whole write-buffer and the rest of the used portion of the send-buffer. This allows the socket object to pin a small portion of the send-buffer, which should be more efficient, but incurs the overhead of a memory-copy for part of the message data.

Transfer option 3 repeatedly (re-)fills and sends the write-buffer by copying data in chunks from the send-buffer. This removes the needs for the socket object to pin the send-buffer but incurs the overhead of a memory copy for all the message data.

As shown in Figure 51, the bandwidth for large message transfers is highest for transfer option 1, i.e. memory pinning is more efficient than memory copying for large buffers. However, for smaller messages, transfer option 2 has highest bandwidth, which means a hybrid approach, where some memory is copied into an already pinned buffer and some memory is pinned when needed, can be more efficient.

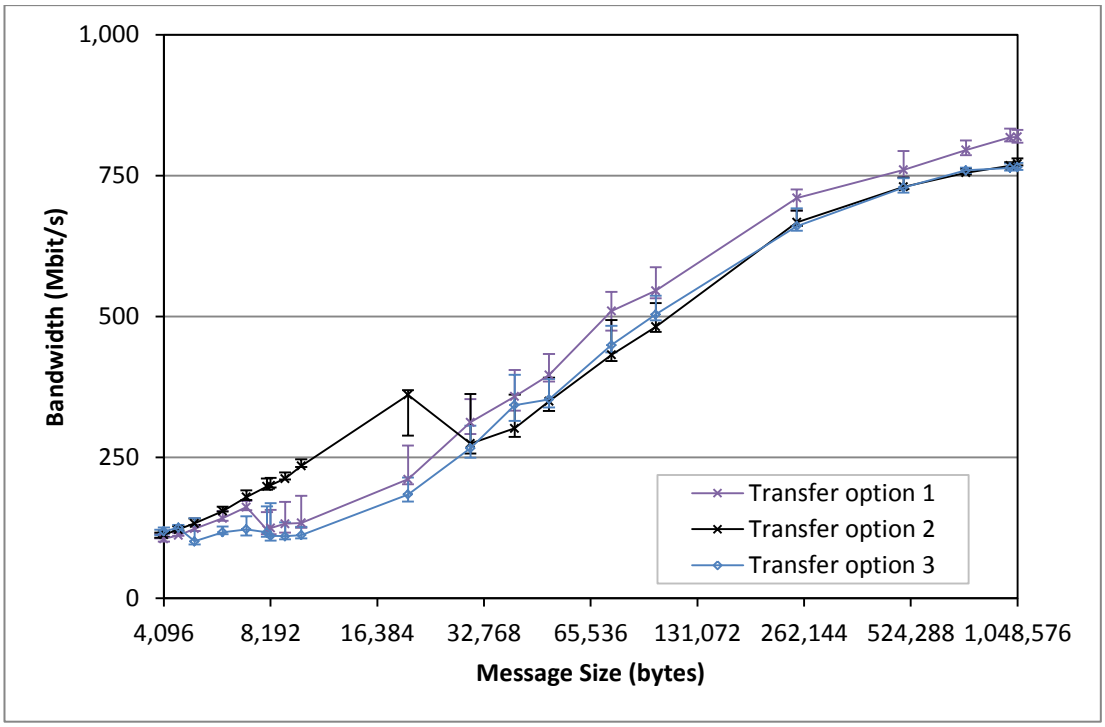


**Figure 51: comparison of three implementations of the transfer protocol message.**
**transfer option 1: sends the header from the McMPI write-buffer and then the data from the send-buffer**
**transfer option 2: sends header and some data from the McMPI write-buffer, then the rest of the send-buffer**
**transfer option 3: repeatedly sends the McMPI write-buffer, re-filling it by copying from the send-buffer**

## 5.3.6 Investigation of the Stepped Data Distribution

It has been noted that various results-sets gathered during this work have demonstrated a "stepped" distribution of data values, i.e. there appear to be distinct clusters of values in particular ranges with few data values in between these ranges and with the ranges occurring at regular intervals.

Initially, it was thought that this could be an artefact of the clock used to obtain the timing values, such as the resolution of the clock being approximately the size of the intervals between data clusters. However, this possibility was investigated and eliminated. The clock does measure times between the clusters but, in each data set, there are very few data values in the gaps. For other codes that use the same clock, both in C and in C#, the times measured form a continuous distribution.

In order to investigate this stepped distribution, the data values from four result-sets (all obtained using the Desktops) were re-analysed using a simple "binning" technique to produce the six histograms that are presented in Figure 52, Figure 53, Figure 54, Figure 55, Figure 56 and Figure 57.

Figure 52 presents timing data from the C# TCP socket using the `Send` and `Receive` methods for message sizes between 512 and 900 bytes (this data was also used to produce five of the data points and error-bars in Figure 39). This data subset was chosen because of the clear stepped arrangement in the C# TCP Socket line and because the C TCP Socket line appears to follow the same stepped arrangement. The same subset of data from the C TCP socket, i.e. using the `Send` and `Receive` methods for message sizes between 512 and 900 bytes, is presented in Figure 53 so that the two distributions can be compared. The C# TCP socket data is clearly clustered into three groups centred at 57µs, 63µs and 69µs, i.e. occurring at 6µs intervals. There are some timing values in between these clusters, proving that the clock can record times at a high enough resolution to capture this distribution accurately. The C TCP socket data shows some clustering of values but the distribution is smoother, with many more timing values recorded in the intervals between clusters.

Figure 54 also presents timing data from the C# TCP socket using the `Send` and `Receive` methods but for message sizes between 48 and 150 bytes. Figure 55 presents the equivalent timing data from the C TCP socket for comparison. For these message sizes, the difference in the distribution is striking, with the C# TCP socket data displaying distinct clusters and the C TCP socket data displaying a normal distribution for each message size.

Figure 56 presents timing data from the C# TCP socket using the `SendAsync` and `Select-Receive` methods, i.e. the methods used within McMPI. The message sizes are between 512 and 900 bytes, as for Figure 52. This data was previously presented as five of the data points from the `SendAsync-SelectReceive` line in Figure 41, which appears

to show the same stepped pattern. As can be seen from Figure 56, the data values are clustered into groups centred at 45µs, 51µs and 56µs, i.e. at 5-6µs intervals. Again, a small number of data values are recorded in the intervals. This shows that the stepped pattern is present in other modes of operation of the C# TCP socket.

All tests of the C# TCP socket were performed by timing 1,500 batches of one round-trip in each batch. The times quoted here are the latency of the communication, i.e. half the batch time. Therefore, an interval of 5-6µs between clusters implies that each batch includes zero or more 10-12µs delays. In addition, if this delay is caused by the C# TCP socket, then each round-trip incurs zero or more 10-12µs delays.

All of the tests of the McMPI library were performed by timing 1,500 batches of two round-trips in each batch. The times quoted here are the latency of the communication, i.e. one quarter of the batch time. Therefore, a round-trip delay of 10-12µs (caused by the C# TCP socket used by McMPI) would produce a 2.5-3.0µs interval between clusters of data values.

Figure 57 presents timing data from the eager protocol of McMPI with the TCP communication module that uses the `SendAsync` and `Select-Receive` methods. The message sizes are between 512 and 900 bytes, as for Figure 56. This data was previously presented as five data points from the McMPI Eager line in Figure 45. The data in Figure 57 shows some signs of the clustering seen in Figure 56.

Figure 58 presents timing data from the eager protocol of McMPI with the TCP communication module that uses the `SendAsync` and `Select-Receive` methods. The message sizes are between 512 and 900 bytes, as for Figure 57, but data was binned into groups of 0.5µs to increase the resolution of this plot. The evidence of clustering is stronger in this plot – the peaks and troughs are periodic with an interval of approximately 2.5µs. Each cluster begins with a peak and then declines nearly linearly until the trough at the end.

One possible explanation of this is that the McMPI library and the program used to test it might hide the clustering effect by performing some computation after the communication delay. To complete communications, the McMPI library must create header buffers, search message queues, add items and remove items in messages queues, and set status information. None of these tasks are performed by the program that tests the C# TCP socket. This difference could mean that the clustering effect exists for the C# TCP socket

used by McMPI but is obscured because the computation after the delay produces a broader distribution of timing values.

Figure 59 presents timing data from the eager protocol of McMPI with the TCP communication module that uses the `SendAsync` and `Select-Receive` methods, as for Figure 58. The message sizes are between 48 and 150 bytes, as for Figure 54, but data was binned into groups of 0.5µs to increase the resolution of this plot. The clustering effect of the C# TCP socket is clearly visible for all message sizes on Figure 59, with clusters centred at 63.5µs, 66µs and 68.5µs (for message sizes 48 and 64 bytes) and at 64µs, 66.5µs and 69µs (for message sizes 100, 128 and 150 bytes), i.e. at 2.5µs intervals.

The presence of clustering at 2.5µs intervals for McMPI latencies (with timed batches of two round-trips) and at 5µs intervals for C# TCP socket latencies (with timed batches of one round-trip) strongly suggests that the C# TCP socket incurs zero or more 10µs delays during each round-trip communication. The absence of clustering in the C TCP socket indicates that this delay is caused by the C# TCP socket itself, i.e. by the `Socket` object code (a class within the Microsoft .Net Framework), which is a wrapper for the C TCP socket.

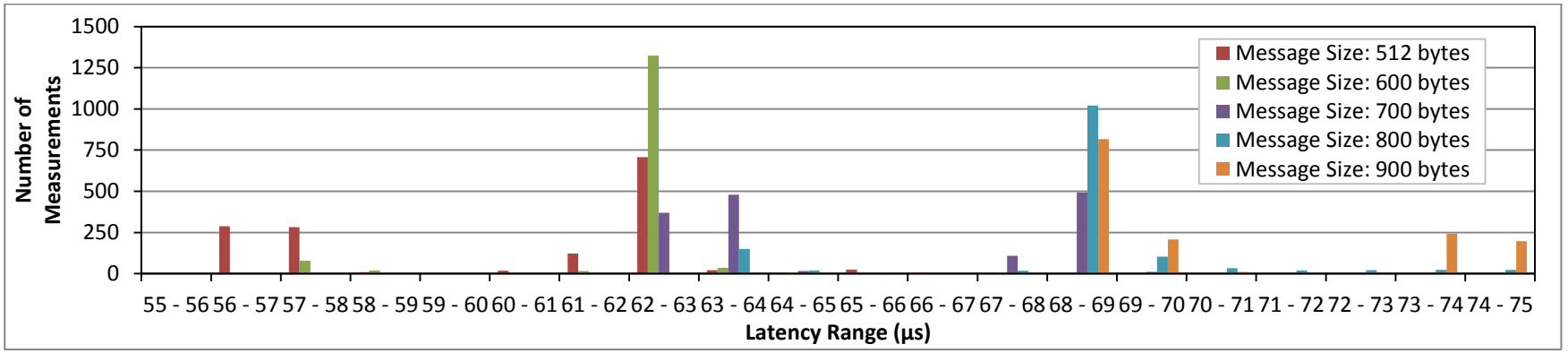


**Figure 52: measurements of ping-pong latency on the Desktops for Gigabit Ethernet TCP sockets in C# (with no message-passing).**
**This is the same data as `C# TCP Socket: Send-Receive` in Figure 39 but with data values "binned" into 1µs groups and the group counts shown as a histogram.**

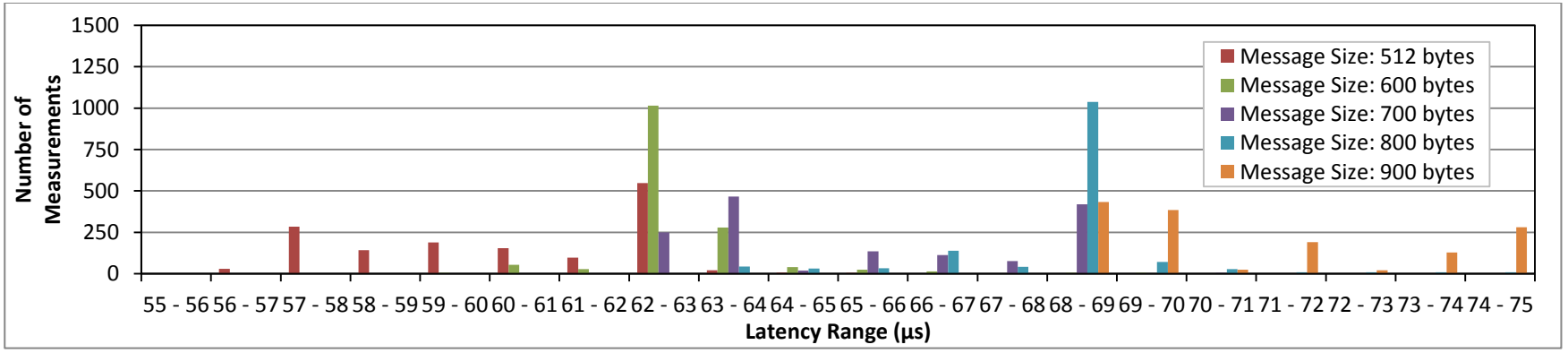


**Figure 53: measurements of ping-pong latency on the Desktops for Gigabit Ethernet TCP sockets in C (with no message-passing).**
**This is the same data as `C TCP Socket: Send-Receive` in Figure 39 but with data values "binned" into 1µs groups and the group counts shown as a histogram.**

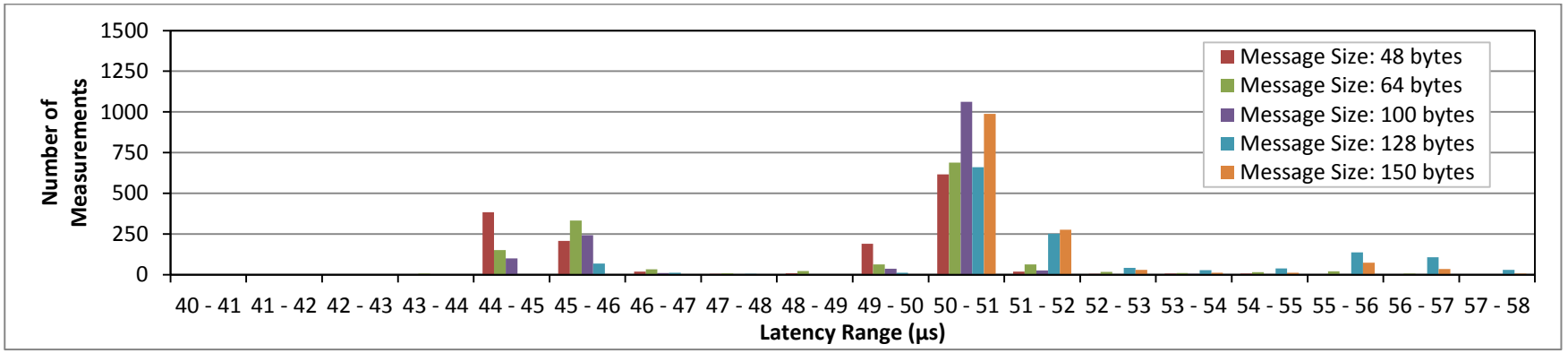


**Figure 54: measurements of ping-pong latency on the Desktops for Gigabit Ethernet TCP sockets in C# (with no message-passing).**
**This is the same data as `C# TCP Socket: Send-Receive` in Figure 39 but with data values "binned" into 1µs groups and the group counts shown as a histogram.**

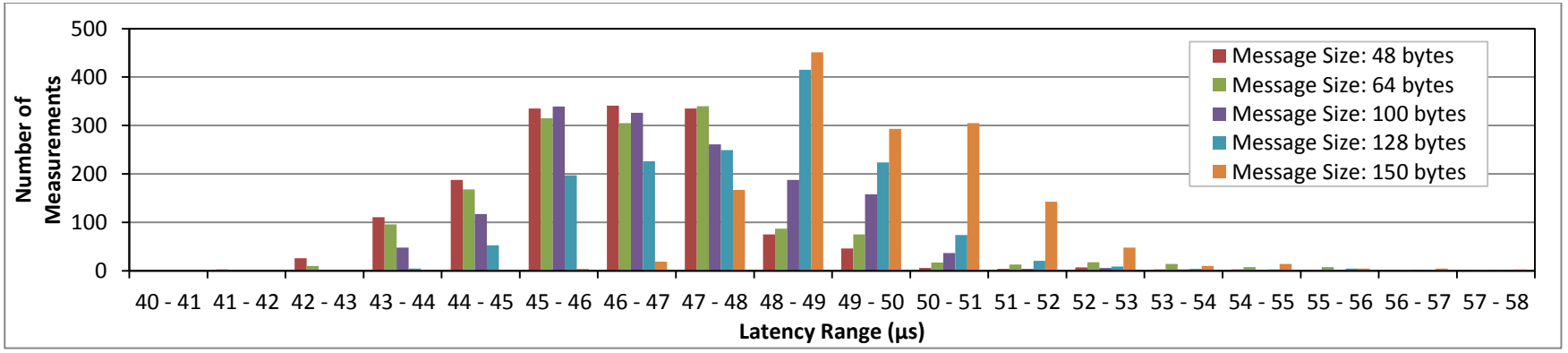


**Figure 55: measurements of ping-pong latency on the Desktops for Gigabit Ethernet TCP sockets in C (with no message-passing).**
**This is the same data as `C TCP Socket: Send-Receive` in Figure 39 but with data values "binned" into 1µs groups and the group counts shown as a histogram.**

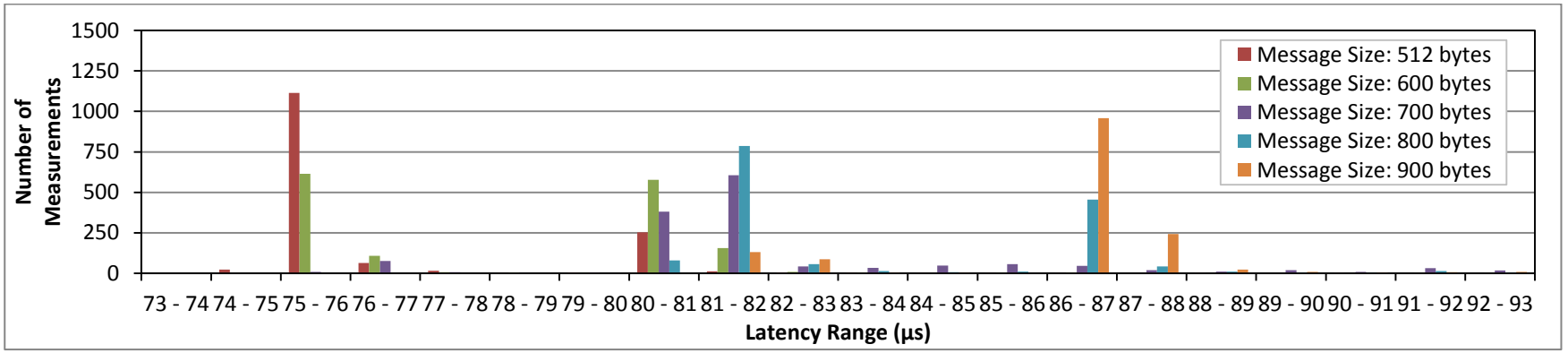


**Figure 56: measurements of ping-pong latency on the Desktops for Gigabit Ethernet TCP sockets in C# (with no message-passing).**
**This is the same data as `SendAsync-SelectReceive` in Figure 41 but with values "binned" into 1µs groups and the group counts and shown as a histogram.**

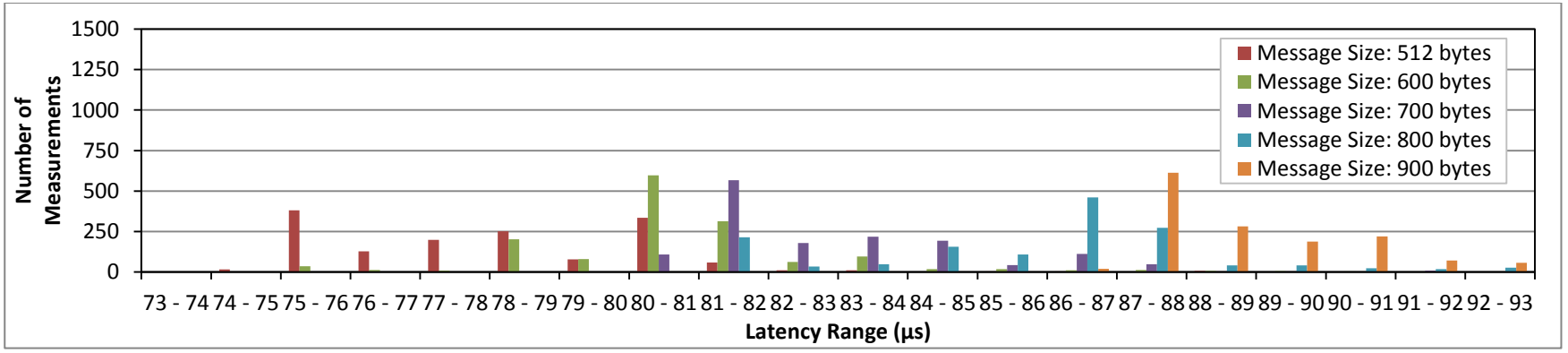


**Figure 57: measurements of ping-pong latency on the Desktops for Gigabit Ethernet TCP sockets in C# (with message-passing via McMPI).**
**This is the same data as `McMPI Eager` in Figure 45 but with values "binned" into 1µs groups and the group counts and shown as a histogram.**

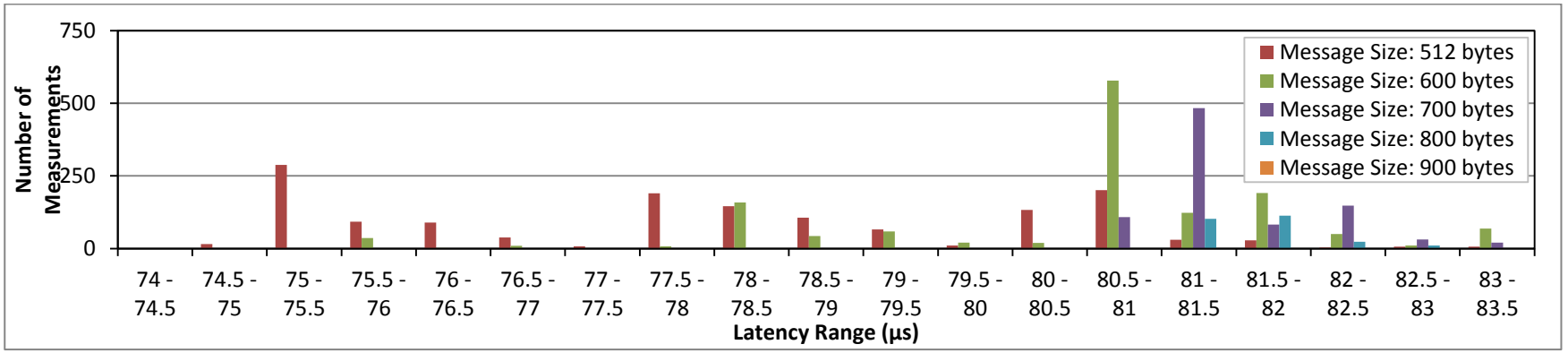


**Figure 58: measurements of ping-pong latency on the Desktops for Gigabit Ethernet TCP sockets in C# (with message-passing via McMPI). This is the same data as `McMPI Eager` in Figure 45 but with values "binned" into 0.5µs groups and the group counts and shown as a histogram.**

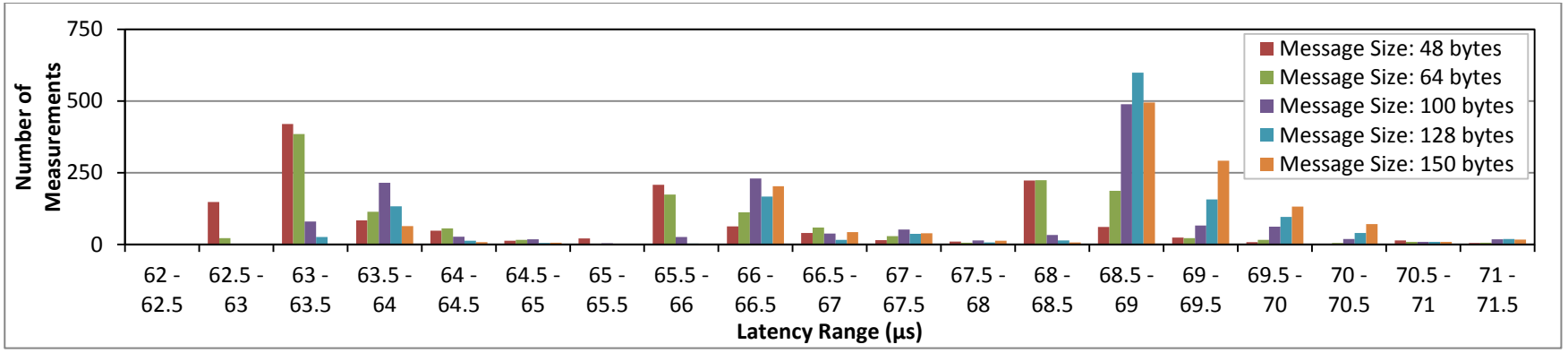


**Figure 59: measurements of ping-pong latency on the Desktops for Gigabit Ethernet TCP sockets in C# (with message-passing via McMPI). This is the same data as `McMPI Eager` in Figure 45 but with values "binned" into 0.5µs groups and the group counts and shown as a histogram.**

# Chapter 6

# Conclusions

## 6.1 Summary Conclusions

This work demonstrates the feasibility of high-performance message-passing communication in a modern high-productivity language.

The design ideas explored, and the implementation issues overcome, have been combined to produce McMPI, a new message-passing library written in pure C#, which implements point-to-point communication, as defined by the MPI Standard. The layered and modular design of McMPI incorporates well-known object-oriented design patterns and uses traditional software engineering techniques, such as data-flow diagrams, where appropriate.

The thread-as-rank threading model is mentioned in the MPI Standard but not explicitly specified or endorsed. However, this threading model was chosen for McMPI because C# implicitly supports multi-threaded code. The resulting thread-to-thread delivery mechanism in McMPI sustains better shared-memory communication performance (both lower latency for small messages and higher bandwidth for larger messages) than that achieved by either MPICH2 or MS-MPI on two different HPC test systems.

Unexpectedly, different versions of the TCP communication module affect the performance of the thread-to-thread message delivery mechanism, even though the entire communication layer is bypassed for local message delivery. The performance data gathered for the six TCP communication modules in this work support the hypothesis that increasing the number of background threads in the process decreases the number of useful time-slices allocated to active foreground threads.

In general, performance testing treats each MPI library as a black-box, i.e. it makes no assumptions about, and takes no account of, its internal implementation. However, it is possible to devise tests that isolate parts of the internal implementation and reveal their performance characteristics. For this work, a test was devised (comparing the performance of a series of messages tagged in the same order as receive operations with a series of

messages tagged in reverse order) to isolate the matching algorithm in order to measure the time taken by an unsuccessful match.

Setting process affinity, i.e. limiting which core(s) a process can use, is a useful tool to increase and stabilise the performance of inter-process communication. Communication between cores in the same multi-core CPU should be faster than communication between CPUs on the same motherboard and avoiding the core that handles system interrupts should yield a more predictable response time. However, the hardware in the Cluster machine used for testing in this work exhibits an unexpected anomaly whereby communication between the two cores of one dual-core CPU in a node is 8 times slower than between the two cores of the other (identical) dual-core CPU and 4 times slower than communication between the two CPUs.

Process-to-process message delivery is usually achieved by taking advantage of shared memory, e.g. by mapping a region of memory into the address space of both processes. This was not done as part of this work but is suggested as an area for potential further work in section 6.2. In McMPI, messages can be delivered between processes using the TCP communication module. When the two processes share a network card, the TCP stack will use the loopback socket interface instead of transmitting the data across the network hardware. It is possible to force other MPI libraries (in particular MPICH2 and MS-MPI) to use loopback sockets and thereby obtain comparative performance data. For small message sizes, the latency for McMPI is similar to MPICH2 but worse than MS-MPI. For larger message sizes, the bandwidth for McMPI is better than both MPICH2 and MS-MPI.

Both MPICH2 and MS-MPI allow the socket buffer size to be changed because it is likely that different buffer sizes will perform better on different machines. The loopback socket interface is particularly useful when investigating which buffer size is most appropriate because bypassing the network hardware decreases the overall transmission time, which makes small performance changes more noticeable. The default buffer size for sockets in Windows operating systems is 8KB. However, the best socket buffer size for McMPI on the Server machine used for testing in this work is 32KB, which is the size of the L1 cache on that machine and is the default buffer size for both MPICH2 and MS-MPI.

The distributed-memory communication performance of McMPI using C# TCP sockets is comparable to the performance of MPICH2 and MS-MPI. The latency for small messages in

McMPI is approximately 5µs (8%) higher than that of MPICH2 and 12µs (23%) higher than that for MS-MPI. The bandwidth for larger messages in McMPI is approximately 6% lower than that for MPICH2 but it is approximately 50% higher than that for MS-MPI.

## 6.2 Further Work

Application-level benchmark testing would ascertain whether the performance achieved by the communication patterns tested as part of this work could be maintained for codes that include significant computation as well as communication.

The thread-to-thread delivery mechanism could use the techniques found in common implementations of lock-free queues, specifically the compare-and-swap operation, to reduce the contention for the request and unmatched queues. Currently, these queues are protected from multiple simultaneous accesses using a single lock. A lock-free algorithm could be developed that allows multiple threads to search these queues simultaneously whilst preventing a race-condition when multiple threads attempt to modify these queues. Reducing contention between threads might improve the latency of message delivery and may help with scaling, i.e. when many threads in each OS process are all contending for the single joint queue lock. In addition, for large messages, the sending and receiving thread could each copy half the message data from the send-buffer to the receive-buffer to maximise use of available memory bandwidth, i.e. multiple physical memory channels.

A process-to-process delivery mechanism could be developed using the memory-mapped-file (MMF) functionality from .Net Framework 4.0, which should give much better performance than loopback sockets and may be able to match the shared-memory performance of MPICH2 and MS-MPI.

The machine-to-machine communication modules that use the `ReceiveAsync` socket operation should be re-coded to request the exact amount of data that is known to be present in the socket (in the same manner as the communication modules that use the `Receive` socket operation). Comparing the performance with existing communication modules would resolve whether the `Socket` object waits for more data to arrive before returning control with a part-filled receive buffer.

The `Socket` object in C# is a wrapper for a socket in C (from the Winsock libraries in the Windows operating system), which means the translation overhead still exists, just at a

lower level. If a new C# `Socket` object were created, for example by modifying the code in the existing C# `Socket` object or by building a new `Socket` object in pure C#, perhaps the overhead could be improved or even eliminated altogether.

Currently, transmitted data is read from the network hardware into an unmanaged buffer by the network driver. The C# `Socket` object instructs the C socket to copy that data into a pinned managed-memory buffer. The C socket is only allowed to write to the managed-memory buffer because it is pinned. Currently in McMPI, this pinned buffer is an internal buffer allocated by the library rather than the user receive buffer in order to avoid repeatedly pinning different bits of memory. Thus, a second memory copy is required, from the pinned internal buffer into the unpinned user receive buffer. If the C# `Socket` object code were modified so that it obtained a reference to the unmanaged buffer used by the network driver, it could do the memory copy itself. A C# object is allowed to copy from an unmanaged buffer into a managed buffer without needing to pin it. This modification would eliminate the need for pinning an intermediary buffer and performing a second memory copy; the data would be copied directly from the buffer used by the network driver into the user receive buffer.

Further changes can be made in the notification mechanism. Currently, the C socket allocates a completion port to the socket and signals it when an operation completes. The C socket then calls a call-back method in the C# `Socket` object, which waits for a thread to become alertable, marshals the call into the thread context of the thread that initiated the operation and starts to raise an event on the status object passed when the operation was started. At this point, the normal sequence is overridden by McMPI code. This is currently the earliest intervention point possible by C# code. However, the context-switching step is not needed in McMPI because all threads are equally capable of handling the completion call-back, so a simpler notification mechanism is possible. If the C# `Socket` object provided a way to pass the unmanaged reference to a wait handle to the C socket then that wait handle could be signalled instead of the completion port. As the managed application is waiting for this wait handle to be signalled, the whole mechanism of call-backs can be removed, including the need for an alertable thread and thread context-switching.

## 6.3 Concluding Remarks

This work examines the feasibility of a high performance message passing communication layer written in C#. It aims to provide a tool that encourages commercial businesses to use high performance computing and it extends the previous work on message passing in C# by implementing the communication library itself in C#.

# Bibliography


[1] N. Carriero and D. Gelernter, “Linda in context,” *Commun. ACM,* vol. 32, no. 4, pp. 444-458, 1989.

[2] openmp.org, “The OpenMP API specification for parallel programming,” [Online]. Available: http://openmp.org/wp/. [Accessed 31 May 2012].

[3] J. K. Bennet, J. B. Carter and W. Zwaenepoel, “Munin: distributed shared memory based on type-specific memory coherence,” in *PPOPP '90: Proceedings of the second ACM SIGPLAN symposium on Principles \& practice of parallel programming*, New York, 1990.

[4] J. B. Carter, J. K. Bennett and W. Zwaenepoel, “Implementation and performance of Munin,” in *SOSP '91: Proceedings of the thirteenth ACM symposium on Operating systems principles*, New York, 1991.

[5] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun and V. Sarkar, “X10: an object-oriented approach to non-uniform cluster computing,” *SIGPLAN Not.,* vol. 40, no. 10, pp. 519-538, 2005.

[6] K. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger, P. a. I. C. Husbands, A. Kamil, R. Nishtala, J. Su, M. Welcome and T. Wen, “Productivity and performance using partitioned global address space languages,” in *PASCO '07: Proceedings of the 2007 international workshop on Parallel symbolic computation*, New York, 2007.

[7] W.-Y. Chen, D. Bonachea, C. Iancu and K. Yelick, “Automatic nonblocking communication for partitioned global address space programs,” in *CS '07: Proceedings of the 21st annual international conference on Supercomputing*, New York, 2007.

[8] S. Yalamanchili, J. Young, J. Duato and F. Silla, “A Dynamic, Partitioned Global Address Space Model for High Performance Clusters,” Georgia Institute of Technology, 2008.

[9] R. W. Numrich and J. Reid, “Co-array Fortran for parallel programming,” *SIGPLAN Fortran Forum,* vol. 17, no. 2, pp. 1-31, 1998.

[10] R. Chandra, A. Gupta and J. L. Hennessy, “COOL: An Object-Based Language for Parallel Programming,” *Computer,* vol. 27, no. 8, pp. 13-26, 1994.

[11] S. Gregory, Parallel logic programming in PARLOG: the language and its implementation, Boston, MA: Addison-Wesley Longman Publishing Co., Inc, 1987.

[12] D. Callahant, B. L. Chamberlaint and H. P. Zimaj, “The Cascade High Productivity Language,” in *Ninth International Workshop on High-level Parallel Programming Models and Supportive Environments, Proceedings*, 2004.

[13] Y. Aridor and D. B. Lange, “Agent design patterns: elements of agent application design,” in *AGENTS '98: Proceedings of the second international conference on Autonomous agents*, New York, 1998.

[14] M. Cole, “Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming,” *Parallel Computing,* vol. 30, no. 3, pp. 389-406, 2004.

[15] M. Danelutto and M. Aldinucci, “Algorithmic skeletons meeting grids,” *Parallel Computing,* vol. 32, no. 7-8, pp. 449-462, 2006.

[16] S. Siu, M. De Simone, D. Goswami and D. Singh, “Design patterns for parallel programming,” in *Proceedings of the 1996 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'96)*, 1996.

[17] D. Goswami, A. Singh and B. R. Preiss, “From Design Patterns to Parallel Architectural Skeletons,” *Journal of Parallel and Distributed Computing,* vol. 62, no. 4, pp. 669-695.

[18] S. MacDonald, J. Schaeffer and D. Szafron, “Pattern-Based Object-Oriented Parallel Programming,” in *ISCOPE '97: Proceedings of the Scientific Computing in Object-Oriented Parallel Environments*, London, 1997.

[19] S. MacDonald, D. Szafron and J. Schaeffer, “Object-Oriented Pattern-Based Parallel

Programming with Automatically Generated Frameworks," in *COOTS*, 1999.

[20] K. Tan, D. Szafron, J. Schaeffer, J. Anvik and S. MacDonald, "Using generative design patterns to generate parallel code for a distributed memory environment," in *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, New York, 2003.

[21] A. Singh, J. Schaeffer and M. Green, "A Template-Based Approach to the Generation of Distributed Applications Using a Network of Workstations," *IEEE Trans. Parallel Distrib. Syst.,* vol. 2, no. 1, pp. 1045-9219, 1991.

[22] J. Schaeffer, D. Szafron, G. Lobe and I. Parsons, "The Enterprise Model for Developing Distributed Applications," *IEEE Parallel Distrib. Technol.,* vol. 1, no. 3, pp. 85-96, 1993.

[23] J. Anvik, J. Schaeffer, D. Szafron and K. Tan, "Asserting the utility of CO2P3S using the Cowichan problem set," *J. Parallel Distrib. Comput.,* vol. 65, no. 12, pp. 1542-1557, 2005.

[24] D. J. Holmes, *Investigating the Feasibility of an MPI-like Library Implemented in .Net Using Only Fully Managed Code,* Edinburgh: University of Edinburgh, 2007.

[25] IEEE, "MPI - A Message-Passing Interface," in *Supercomputing 93 Conference*, 1993.

[26] B. Carpenter, G. Fox, S.-H. Ko and S. Lim, "mpiJava 1.2: API Specification," *Northeast Parallel Architecture Center,* vol. 66, 1999.

[27] B. Carpenter, V. Getov, S. A. Judd G and F. G, "MPJ: MPI-like message passing for Java," *Concurrency-Practice and experience,* vol. 12, no. 11, pp. 1019-1038, 9 2000.

[28] E. Ong, "MPI Ruby: Scripting in a Parallel Environment," *Computing in Science and Engineering,* vol. 4, no. 4, pp. 78-82, 2002.

[29] S. Mitchev, "Writing programs in JavaMPI," School of Computer Science, University of Westminster, 1997.

[30] M. Baker, B. Carpenter, G. Fox and S.-H. Ko, “mpiJava: A Java MPI Interface,” in *Proceedings of First UK Workshop on Java for High Performance Network Computing*, 1998.

[31] A. Nelisse, J. Maassen, T. Kielmann and H. E. Bal, “CCJ: Object-based Message Passing and Collective Communication in Java,” in *In Concurrency and Computation: Practice and Experience*, 2003.

[32] S. Morin, I. Koren and C. M. Krishna, “JMPI: Implementing the Message Passing Standard in Java,” in *International Proceedings of Parallel and Distributed Processing Symposium*, 2002.

[33] K. Dincer, “A ubiquitous message passing interface implementation in Java: jmpi,” in *14th International Parallel Processing Symposium & 10th Symposium on Parallel and Distributed Processing Proceedings*, 1999.

[34] N. Mohamed, J. Al-Jaroodi, H. Jiang and D. Swanson, “JOPI: a Java object-passing interface,” in *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, Seattle, Washington, USA, 2002.

[35] M. Bornemann, R. van Nieuwpoort and T. Kielmann, “MPJ/Ibis: A flexible and efficient message passing platform for Java,” in *Recent advances in parallel virtual machine and message passing interface, proceedings*, Berlin, 2005.

[36] M. Baker, B. Carpenter and A. Shafi, “MPJ Express: Towards thread safe Java HPC,” in *IEEE International Conference on Cluster Computing*, New York, 2006.

[37] G. L. Taboada, J. Touriño and R. Doallo, “F-MPJ: scalable Java message-passing communications on parallel systems,” *Journal of Supercomputing,* vol. 60, no. 1, pp. 117-140, April 2012.

[38] G. L. Taboada, S. Ramos, R. R. Expósito, J. Touriño and R. Doallo, “Java in the High Performance Computing arena: Research, practice and experience,” *Science of Computer Programming,* p. 10.1016, 2011.

[39] G. L. Taboada, J. Touriño and R. Doallo, “Java Fast Sockets: Enabling high-speed Java communications on high performance clusters,” *Computer Communications,* vol. 31, no. 17, p. 4049–4059, 20 November 2008.

[40] J. Willcock, A. Lumsdaine and A. Robison, “Using MPI with C# and the common language infrastructure,” *Concurrency and Computation-Practice & Experience,* vol. 17, no. 7-8, pp. 895-917, Jun 2005.

[41] D. Gregor and A. Lumsdaine, “Design and Implementation of a High-Performance MPI for C# and the Common Language Infrastructure,” in *Proceedings of the 2008 ACM SIGNPLAN Symposium on Principles and Practice of Parallel Programming*, 2008.

[42] Microsoft Corporation, “Microsoft MPI,” 2009. [Online]. Available: http://msdn.microsoft.com/en-us/library/bb524831(VS.85).aspx. [Accessed 2009].

[43] ECMA, “ECMA C# Language Specification,” June 2006. [Online]. Available: www.ecma-international.org/publications/standards/ecma-334.htm. [Accessed 31 5 2012].

[44] ECMA, “Standard ECMA-335 Common Language Infrastructure (CLI),” Dec 2010. [Online]. Available: http://www.ecma-international.org/publications/standards/ecma-335.htm. [Accessed 31 5 2012].

[45] W. Gropp, E. Lusk, N. Doss and A. Skjellum, “A high-performance, portable implementation of the MPI message passing interface standard,” *Parallel Comput.,* vol. 22, no. 6, pp. 789-828, 1996.

[46] MPI-Forum, “MPI Documents,” University of Tennessee, September 2009. [Online]. Available: http://www.mpi-forum.org/docs/docs.html. [Accessed 25 05 2012].

[47] J. Dongarra, A. Geist, W. Jiang, J. Kohl, R. Manchek, P. Papadopoulos, V. Sunderam and M. Fischer, “An Introduction to PVM Programming,” 5 December 2011. [Online]. Available: http://www.csm.ornl.gov/pvm/intro.html#communication. [Accessed 15 September 2012].

[48] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.

[49] top500.org, "Top 500," Nov 2011. [Online]. Available: http://i.top500.org/stats/list/38/connfam. [Accessed 31 May 2012].

[50] Intel, "Intel MPI Benchmarks," [Online]. Available: http://software.intel.com/en-us/articles/intel-mpi-benchmarks/. [Accessed 2012 May 2012].

[51] Q. O. Snell, A. R. Mikler and J. L. Gustafson, "NetPIPE - A Network Protocol Independent Performance Evaluator," 5 4 1996. [Online]. Available: http://www.bitspjoule.org/NetPIPE/paper/full.html. [Accessed 25 5 2012].

[52] T. Benjegerdes, "NetPIPE - A Network Protocol Independent Performance Evaluator," 8 9 2009. [Online]. Available: http://bitspjoule.org/netpipe/. [Accessed 25 5 2012].

[53] Argonne National Laboratories, "MPICH2," Argonne National Laboratories, [Online]. Available: http://www.mcs.anl.gov/research/projects/mpich2/. [Accessed 31 May 2012].

[54] G. L. Taboada, J. Tourino and R. Doallo, "Performance analysis of Java message-passing libraries on fast Ethernet, Myrinet and SCI clusters," in *Cluster Computing, 2003. Proceedings*, 2003.