

Motor: A Virtual Machine for High Performance Computing

Wojtek Goscinski and David Abramson
{wojtek.goscinski, david.abramson}@infotech.monash.edu.au
Faculty of Information Technology, Monash University

Abstract

High performance application development remains challenging, particularly for scientists making the transition to a Grid environment. In general areas of computing, virtual environments such as Java and .Net have proved successful in fostering application development. Unfortunately, these existing virtual environments do not provide the necessary high performance computing abstractions required by e-Scientists. In response, we propose and demonstrate a new approach to the development of a high performance virtual infrastructure: Motor is a virtual machine developed by integrating a high performance message passing library directly within a virtual infrastructure. Motor provides high performance application developers with a common runtime, garbage collection and system libraries, including high performance message passing, whilst retaining strong message passing performance.

1. Introduction

High performance computing (HPC) is adopting grid computing infrastructure, where users have access to a range of heterogeneous resources. Ideally, an e-Scientist, with access to a grid of resources, should be able to pick the resource which best suits his or her needs. However, realistically, e-Scientists are constrained by deployment issues such as platform and environmental dependence. Writing highly-tuned, complex scientific applications, for a range of heterogeneous resources remains difficult and time consuming.

To most effectively leverage a dynamic grid environment, the effort in porting and redeploying applications should be minimized. The current trend in software engineering has been to move toward a virtual machine architecture, such as Java or the Common Language Infrastructure (CLI) [1], and its commercial implementation, Microsoft .NET. These environments provide a common runtime environment, regardless of the underlying platform, enabling compile-once-run-anywhere

application deployment. Furthermore, they define a range of features and services, such as: type system, memory management, portable assembly format and security. In addition, they provide a common set of libraries which provide web services, user interface, security and networking functionality.

Taking a similar approach for HPC would provide e-Scientists with a *virtual high performance computing environment* over a range of heterogeneous resources, enabling compile-once-run-anywhere application deployment. However, existing virtual machines are inappropriate; neither Java nor the CLI provide necessary high performance functionality, particularly high performance message passing.

The Message Passing Interface (MPI) specification simplifies scientific application development because it defines a single message passing interface which is implemented over a wide range of platforms and interconnects. However, high performance application developers still face an assortment of non-MPI porting and deployment issues such as platform specific system calls, recompilation and library problems.

So far, implementations of high performance message passing libraries for virtual machine environments provide a managed¹ wrapper to an underlying native message passing implementation [2-7]. Using this approach, the virtual machine runtime and message passing library remain separate. The message passing library interfaces directly to the native operating system and is unable to access virtual machine services. In turn, the virtual machine does not trust the message passing library, treating it as an unsafe native code.

The aim of our research is to develop an object-oriented virtual environment for HPC, hiding underlying platform and interconnect heterogeneity.

Our approach integrates a high performance message passing library directly into a virtual machine environment, thus producing an overall virtual machine for high performance computing. In our architecture, the high performance message passing library exists along side other

¹ Code developed for a virtual machine is often referred to as *managed code*, which means it is managed by an underlying runtime environment. This is contrast to regular *native code* which interfaces directly to the operating system.

virtual machine services such as memory and process management, IO, security and networking (Figure 1).

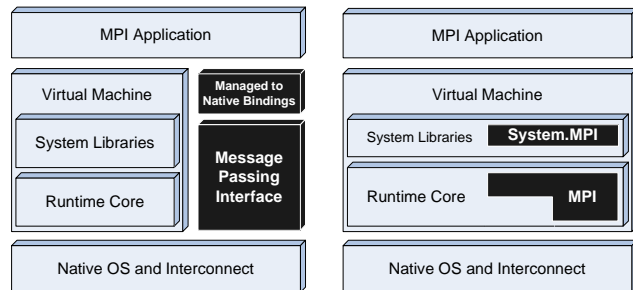


Figure 1. Existing architecture of managed wrapper implementations MPI (left), in comparison to our integration of MPI within a virtual machine (right).

To demonstrate our approach, we provide a proof-of-concept implementation in the form of Motor (MPI Rotor), a virtual machine that incorporates high performance message passing. Motor has been developed by integrating the MPICH2 [8] MPI code base with Microsoft’s Shared Source Common Language Infrastructure (SSCLI), also known as Rotor [9, 10].

In addition to implementing MPI, Motor implements a higher-level MPI-like object transport mechanism. This provides functionality not available in MPI; including the ability to transport arbitrary objects, trees of objects and arrays of objects. It also provides functionality not possible with other Java and .Net implementations of MPI, namely the ability to scatter / gather arrays of objects.

Motor is significant for a number of reasons. First, it allows users to write high performance message passing applications without regard for a particular operating system and interconnect. In contrast to native MPI applications, Motor applications are compile-once-run-anywhere. Therefore HPC developers have simple deployment and guaranteed instantiation regardless of underlying heterogeneity. Second, applications written for Motor are provided with runtime services and a guaranteed standard set of system libraries, including message passing. HPC developers can develop against a known and standard environment.

The outcome is significant. Both architecturally and practically, implementing high performance message passing directly within the virtual machine has a number of advantages over the existing wrapper approach:

- The message passing library is a *guaranteed* standard subset of the virtual infrastructure. This follows the same philosophy as Java and the CLI, which include essential business-oriented libraries as standard.
- The message passing library has intimate, efficient access to the virtual machine infrastructure: including memory management and the runtime object model. This means the message passing library is able to

achieve *strong performance*, while *protecting the integrity of the runtime object model*.

- Using a layered approach, Motor is portable. Higher level library functionality is completely portable with the virtual runtime. While lower level functionality can be re-implemented for specific platforms and interconnects.

This paper

- discusses the issues relating to implementing a high-performance message passing library within a virtual runtime (Section 2);
- introduces our architecture for a high performance virtual machine (Section 3 & 4);
- introduces relevant technology (Section 5 & 6);
- discusses the implementation of Motor (Section 7); and
- presents preliminary performance results (Section 8).

2. HPC over Virtual Machines

There are a number of approaches to developing a virtual environment for the heterogeneous grid. These include: Virtualizing the environment at the classical virtual machine level (e.g. VMWare or Xen) [11-13]; and at the subsystem component level [14]. None of these attempt to virtualize grid resources at the application runtime level (e.g. Java or .Net), which has proved a successful approach for business and desktop applications.

Motor is also unique as an MPI implementation. Although the managed-wrapper architecture has been implemented many times, we could find no major effort to integrate a high performance message passing library *directly within* a virtual runtime.

Virtual machines such as Java and the CLI are characterised by a number of features:

- A virtual runtime which Just-In-Time (JIT) compiles a processor-agnostic intermediary language.
- Memory management which compacts the memory space and collects abandoned memory.
- A strongly typed object oriented memory model.
- A standard set of system libraries.

These features provide an environment for easy prototyping and compile-once-run-anywhere application development. However, in exchange for platform independence and other runtime services, an overhead on performance is created. This paper does not benchmark virtual machines or test their suitability for HPC. One such study, comparing high performance benchmarks executed over different versions of the CLI and Java can be found in [15]. Whilst virtual machines introduce a performance overhead, their potential advantages in simplifying application development could help fostering grid computing within the general science community.

To date, both Java and the CLI focus on desktop and

business applications. Neither provides high performance computing abstractions such as high performance message passing. In response, a number of libraries have been developed to provide MPI for Java and the CLI. These are discussed in the following section.

2.1 MPI Implementations for Virtual Environments

Java has been the subject of efforts to provide an MPI library. mpiJava [5] is a Java wrapper to an underlying native MPI implementation. The mpiJava bindings are based on the MPJ API (discussed below). mpiJava supports transport of simple type arrays or objects. Object transport is implemented by serialization using a standard Java serialization mechanism. JavaMPI [6] provides an automatically generated wrapper to an underlying native MPI library. Both mpiJava and JavaMPI use the Java Native Interface (JNI), which provides a Java mechanism to call native code.

JMPI [2] is a pure Java implementation of a subset of MPI. Communication in JMPI is implemented over Java Remote Method Invocation (RMI). This results in a completely portable MPI library, but offers relatively low performance. jmpj [4] is another pure Java implementation, built over the Java sockets library. Unfortunately, [4] does not compare the performance of jmpj versus a native MPI implementation. MPIJ [3] is also a Java implementation of MPI. It uses a native marshaller to achieve results comparable against a native MPI.

The MPJ API [16] is an API specification for Java MPI bindings. Developed by the Message-Passing Working Group of the Java Grande Forum it was produced to create a standard Java MPI-like API specification. It is not an official binding to MPI, but does represent the most significant attempt to formalize such a binding. MPJ describes a Java-oriented adaptation of the official C++ object oriented bindings.

The CLI (and its commercial implementation .NET) is more recent than Java and has been the subject of one major effort to create an MPI library. The Indiana University .Net bindings [7], provide two bindings to the underlying native MPICH MPI library:

- C# bindings, based on the official MPI C++ bindings;
- MPI.NET, a higher level interface specifically focused towards the CLI environment.

The Indiana bindings use the CLI P/Invoke (Platform Invoke) interface to invoke the underlying MPI library. P/Invoke is similar to JNI; it allows managed CLI code to call native code. The Indiana bindings impose a slight overhead over the native MPICH, but suffer due to the overhead of object pinning [7].

The projects discussed can be grouped into two categories: pure managed implementations of MPI which

execute entirely over the virtual machine; and *managed-wrappers* which provide a managed interface to an underlying native MPI implementation. Pure managed implementations are portable but suffer from inefficiency.

Efficient MPI implementations require direct access to the underlying operating system or interconnect, which a pure Java or .NET implementation is unable to provide. Implementing MPI bindings as a wrapper to an underlying native MPI is efficient but introduces a number of issues which are also relevant to our runtime-internal implementation. These include: portability, managed-to-native interface, object model integrity and interaction with managed memory (some of these issues have been raised in [17]). These issues are discussed in the following subs-sections.

2.2 Architecture

To achieve the highest possible performance MPI implementations cooperate with underlying low level services provided by the operating system or transport services provided by the interconnect. In a virtual environment, the runtime provides an extra layer of services, for example, managed memory. However, the MPI implementations we have discussed are oblivious to the services provided by the virtual machine, instead interfacing directly with the operating system. Circumventing the virtual machine functionality is neither elegant and as our implementation demonstrates nor is it more efficient than integrating directly within the virtual machine. In particular, this paper discusses the interaction between the MPI library and managed memory.

Managed-wrappers lack portability; the managed-wrapper must be rebuilt and sometimes edited for each different underlying MPI implementation. This architecture conflicts with the compile-once-run-anywhere ethos of managed software development and has been raised in an earlier publication [17].

Finally, using a managed-to-native call mechanism such as JNI or P/Invoke imposes an overhead on each MPI call because both JNI and P/Invoke require marshalling and impose security mechanisms.

2.3 MPI over Managed Memory

Implementing an efficient zero-copy² MPI in a managed memory environment is problematic because memory might be moved or collected during an MPI operation. A

² Efficient message passing implementations interface directly to an underlying data transport mechanism, whether it is sockets, shared memory or a proprietary interconnect. The underlying transport mechanism reads and writes data directly to and from the supplied memory buffer. This is known as a zero-copy message transfer and is efficient because it avoids extraneous memory copies.

solution to this is to *pin*³ the object while transport is underway. The JNI interface automatically pins and unpins objects, while in the CLI it is the responsibility of the application. Pinning is necessary, but undesirable. It interferes with the garbage collectors standard behaviour and causes memory fragmentation. Furthermore, each pin / unpin operation imposes a slight performance overhead.

In [7] pinning imposes a significant overhead⁴. The regular C# MPI bindings require the user to manage pinning, while the higher level MPI.NET bindings provide automatic pinning of buffers for blocking operations. Relying on the user to pin and unpin memory buffers is dangerous because failing to unpin a memory buffer results in leaking memory. Automatically providing memory pinning is relatively simple for blocking MPI operations. However, non-blocking operations require unpinning later, once the underlying transport operation has actually finished. The MPI.NET description in [7] does not illustrate any mechanisms to automatically unpin buffers for non-blocking operations.

2.4 MPI in an Object Oriented Environment

Effectively implementing MPI within an object oriented environment is challenging because the MPI specification assumes a linear view of memory. In an object oriented model, memory is a graph of objects and the user has no safe access to the underlying linear view.

MPI defines buffer-to-buffer message transport; the user provides a buffer, which is read or written to, by the underlying data transport mechanism. With an object oriented view of memory, a direct buffer-to-buffer transport is only possible when transporting single objects (including single structures and arrays of simple types). This can be called an object-to-object transport. Arrays of objects and trees of objects are not located at a single physical memory location and a buffer-to-buffer transport is impossible.

Object-to-object transport can destroy object model integrity. Implementing the MPI interface directly, as specified and without any buffer boundary checks, allows the user to inadvertently or purposefully destroy the integrity of the object model. Compromising the integrity of the object model can easily occur in two ways:

- Overwrite the end of an object, corrupting the object header and object data of the next object in memory,
- Overwrite an object reference with data. Within a strongly typed object model, object references are

³ Pinning is a request to the garbage collector to temporarily not move or unallocate the requested object, until it is unpinned.

⁴ In our own tests with SSCLI, pinning imposed a smaller overhead than presented in [7]. This is a result of the SSCLI build type used. *Fastchecked* builds, used in [7], impose a greater pinning overhead than the *Free* build we used.

guaranteed to be either null or reference an object of the correct type.

In both cases, the result would be an environment crash at the next garbage collection. This is an unacceptable outcome considering the purpose of the virtual machine is to provide a *safe* environment.

Neither the C# MPI bindings presented in [7], mpiJava nor the MPJ API [16] consider object-model integrity. However, the higher level MPI.NET bindings automatically extract object length information, ensuring the transport does not overwrite the next object.

In addition to regular MPI buffer-to-buffer transport, we consider structured data transport, which involves transporting an entire object, including other referenced objects and required type information. Object transport requires a serialization and deserialization mechanism to transform the typed object tree view of memory to a flat linear transportable buffer, and back again. This provides automatic transport of structured data, without the need to write code to pack and unpack memory. However, performance suffers due to the overhead of serialization to a temporary buffer.

Many of the projects discussed suggest the standard runtime serialization mechanism for object transport. For example, mpiJava provides the OBJECT datatype which uses the standard Java serialization mechanism to transport objects. However, the standard serialization mechanism produces a single atomic flat representation, which cannot be split or offset like standard memory. Implementing more advanced MPI functionality such as scatter and gather operations is difficult. For example, to scatter an array of objects over N hosts, the MPI library would need to create N new sub-arrays and serialize them individually. This is inefficient considering a custom serialization mechanism could be implemented to automatically create a split representation.

3. Introduction to Motor

Motor is a synthesis of the Microsoft SSCLI and the MPICH2 [8] MPI library. Motor is based on the CLI, rather than Java, for two reasons:

- The CLI is a language neutral environment. .Net compilers exist for C#, C++, JavaScript, Java, VB.NET, Fortran and others. This means that the CLI can provide a modern runtime environment for legacy HPC applications written in C++ and FORTRAN.
- The CLI has true multidimensional arrays such as those in C and FORTRAN, and these are important for scientific codes. In contrast, Java uses the arrays-of-arrays⁵ model. Transporting a multidimensional array

⁵ Java n dimensional arrays are implemented as one dimensional arrays, with n-1 dimensional array elements.

constructed using the arrays-of-arrays model is difficult because the array is constructed of multiple objects.

The open source CLI implementation, Mono [18], was also considered for this work. However, we specifically chose the SSCLI because it provides a comprehensive *shared source* implementation of the CLI, which by all accounts is very similar to the commercial .Net. The SSCLI is detailed in Section 5.

In contrast to the managed-wrapper MPI implementations we discussed, the Motor MPI library has been implemented directly within the virtual machine. The message passing library has been implemented in the same way as the System libraries in SSCLI and commercial .NET. It consists of a managed library located in the System.MP namespace, which interfaces to the Message Passing Core, within the virtual runtime. The Message Passing Core is based on the Windows MPICH2 code-base, ported to the SSCLI Platform Adaptation Layer, which defines a virtual subset-Windows API.

The choice of the MPICH2 library was influenced primarily by its layered design, allowing it to support different platforms and interconnects. MPICH2 is described in Section 6.

4. Logical Design

This section discusses the design issues involved in developing Motor, including the overall architecture, bindings and interaction with managed memory.

4.1. Overall Architecture of Motor

Figure 2 illustrates the architecture of the Motor virtual machine. Managed MPI applications are written in a managed language, such as C#, against the System library, which includes an MPI library. The System MPI library is similar to the earlier mentioned managed-wrapper MPI implementations. However, this library interfaces directly to the Message Passing Core, an MPI library implemented within the underlying runtime.

The Message Passing Core implements the non-transport specific parts of the message passing library. It is implemented over a virtualization layer which provides transport and platform virtualization. The virtualization layer provides portability; the Motor runtime can be redeployed with different transport or platform interfaces. For example, implementing a shared memory implementation of MPI would require swapping in a shared memory transport mechanism.

Placing the message passing library within the runtime gives it access to internal runtime resources. Particularly, Motor interacts with the garbage collector and the runtime object / class model. The way Motor

interacts with these services is implementation specific and presented in Section 7.

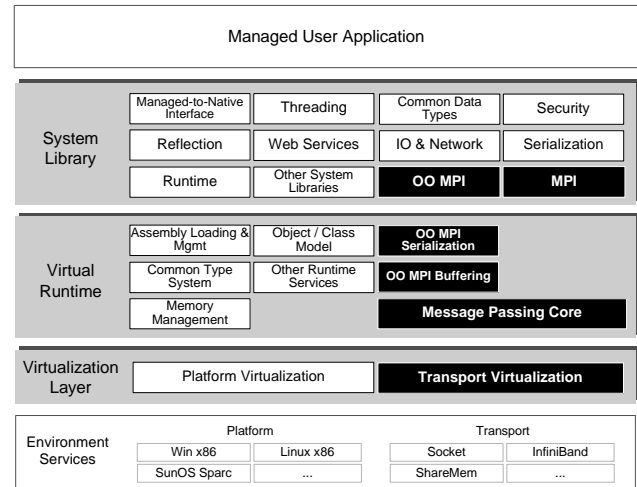


Figure 2. The Motor Architecture. Extensions to the architecture of the regular CLI virtual environment are accentuated in black

4.2 Bindings

We aim to support both an efficient buffer-to-buffer transport and a higher-level object transport. Therefore, we propose two sets of message passing operations:

- A set of MPI operations (described in Section 4.2.1): Based on the standard MPI operations they provide efficient object-to-object transport.
- An Extended Object Oriented set of operations for object transport (described in Section 4.2.2): These operations extend the MPI library to transport objects, array of objects and entire object trees. They provide automatic serialization, deserialization and buffer management. This provides the user with automatic transport of structured data.

Motor keeps these operations separate because they perform two different types of transport: regular MPI operations are efficient; whereas the Extended Object Oriented Operations are slower but provide the ability to automatically transport structured data.

4.2.1 MPI Bindings

The Motor MPI bindings are based on the official C++ MPI bindings, providing the same functionality and class model. However, individual MPI operations have been simplified to protect the integrity of the underlying object model (as discussed in Section 2.4). Figure 3 shows an example of send and receive routines. The changes can be summarised as follows:

- The provided buffer must be a single object (arrays of simple types are allowed because they are primarily an

object). Whilst raw memory addresses are available in the CLI using them does not constitute *safe* programming. The count parameter will always equal one and thus has been removed.

- Only object types with no object references or arrays of simple types can be used as send or receive objects. This prevents overwriting references and protects the integrity of the object model. Passing objects with references is only allowed through the OO operations.
- Object type is easy to determine and therefore the data type, `MPI_Datatype`, parameter has been removed.
- Transporting portions of objects or offsetting into an object is not supported because there is no safe way to refer to a subset of an object. However, transporting portions of an array is supported. An overloaded set of operations cater for array transport and include an offset and count parameter.

These changes protect the integrity of the underlying object model, while retaining a high performance zero-copy transfer. In contrast, the extended object oriented operations, described in the following section, provide a far more flexible mechanism to transport structured data.

The MPI pack and unpack operations have been abandoned. This functionality has been replaced by the ability to transport structured data using the extended object oriented operations.

```
void Recv (object obj, int source, int tag,
          Status status)
void Recv (object obj, int offset,
          int numcomponents, int source, int tag,
          Status status)
void Send(object obj, int dest, int tag)
void Send(object obj, int offset, int length,
          int dest, int tag)
```

Figure 3. Example MPI operations.

4.2.2 Extended Object Oriented Operations

To allow object transport, we have defined an extended set of object oriented operations which are distinguished by an “O” prefix. Figure 4 provides an example of `OSend` and `ORecv` routines. These operations are not direct object-to-object operations. Rather, they use a custom serialization mechanism to flatten the object tree and deserialization to recreate the object tree. These functions can transport:

- Single objects: transporting a single object is the default behaviour of the OO operations. Any object can be transported. The default behaviour of the system is to transport an object’s simple data and avoid transporting referenced objects. References are replaced with null.
- Arrays of objects: an array is treated differently from a normal object. By default an array is transported together with the array-entry objects it references. An overloaded set of operations include an offset and count

parameter for array subset transfer.

- Trees of objects: to transport a tree of objects a custom attribute, *Transportable*, is used to define which object references should be propagated during the operation.

```
object ORecv (int source, int tag, Status status)
object ORecv (int source, int offset, int
              numcomponents, int tag, Status status)
void OSend(object obj, int dest, int tag)
void OSend(object obj, int offset, int
           numcomponents, int dest, int tag)
```

Figure 4. Example extended object oriented operations.

The *Transportable* attribute is implemented using a CLI declarative metadata mechanism called a ‘custom attribute’ (Java has a similar mechanism called an ‘annotation’). Custom attributes are declarations added to code, which can be queried at runtime, allowing the developer to describe and query type specific information. In this case, we use the ‘*Transportable*’ attribute to decorate object references which should be propagated when transporting an object (Figure 5). Our *Transportable* attribute is similar to the CLI *Serializable* attribute which is used to decorate classes for serialization. The main difference between the *Serializable* and *Transportable* attributes is that *Transportable* is an opt-in mechanism. Users specify the references which they *want* propagated. On the other hand, the *Serializable* attribute requires users to specifically opt fields *out* of propagation. We prefer an opt-in mechanism to limit the amount of data users are unwittingly sending and receiving.

```
[Transportable] class LinkedList
{
    [Transportable] public int[] array;
    [Transportable] public LinkedList next;
    public LinkedList next2;
}
```

Figure 5. `LinkedList`, a linked list element which uses the *Transportable* attribute to define which references should be propagated during transport. In this case, the array object and next will be propagated. next2 will not.

The Motor extended object oriented operations cover a range of operations including: the range of regular send / receive operations; broadcast operations and scatter / gather operations for arrays of objects. To enable scatter / gather operations our custom serialization mechanism is able to produce a split serialized array representation. The implementation of these operations, including description of our serialization mechanism is described in Section 7.5.

4.3 Memory management

Implementing MPI efficiently in a managed memory environment is somewhat awkward due to the need to pin memory being accessed by the underlying transport.

Unfortunately, pinning is unavoidable because the low level transport mechanisms do not understand managed memory. We believe the programmer should not be expected to manage memory pinning because failing to unpin memory would cause memory to leak.

However, pinning is not necessary for *every* MPI operation, and is only required if garbage collection might occur and if the object has the potential to be moved during that collection. By implementing MPI alongside the garbage collector, we apply a *pinning policy* which determines if these conditions are satisfied and the object actually requires pinning. This policy relies on access to the object internal memory location and knowledge of the internal mechanics of the garbage collector. Section 7.4 provides a discussion of how Motor minimizes the overhead imposed by pinning and unpinning objects.

Implementing pinning for non-blocking operations is more problematic because it is not clear when the memory block can be released and thus unpinned. There are a number of inelegant solutions to this problem such as:

- Test and release the pinned memory when the user calls a status checking operation such as `MPI_Test`. However, if the user never calls another MPI operation then the memory buffer will never be released.
- Test non-blocking transport operations and unpin buffers in a separate thread. This solution imposes an unnecessary overhead.

The Motor solution to unpinning objects after non-blocking operations is to augment the garbage collector so that it understands pinning operations which are dependant on the status of an operation. During the mark phase of collection, the garbage collector iterates through a list of pinning requests. This provides an opportunity to check the status of an operation and selectively mark the object as pinned, depending on that status. Otherwise, the pinning request can be safely ignored and the object released from further pinning.

5. The Shared Source CLI

Microsoft's SSCLI includes the CLI runtime, base libraries, a JIT compiler, a C# compiler and a Jscript compiler [9, 10]. The following section provides an overview of relevant parts of the SSCLI. Figure 2 provides a diagram of the original SSCLI, with our additions in bold.

5.1 System Library and the CLI Runtime

Applications written for the CLI interface with the System library which provides a high level managed library (written in C#). The System library interfaces to the underlying CLI runtime.

The CLI runtime provides memory management, JIT compilation, the common type system and library loaders. In addition, much of the functionality offered by the System library is actually implemented by the CLI runtime. This is because the functionality offered by the System library requires intimate interaction with the underlying object model and runtime services, and this is only available from within the runtime. Likewise, performance critical functionality of the System library is implemented in the underlying runtime because the C++ runtime can achieve higher performance than the managed System library.

The interface between the System library and the underlying virtual runtime is implemented using an internal call mechanism. This mechanism is declared in the System library using the `InternalCall` attribute. Within the runtime, this call mechanism is called an FCall (Fast Call). FCalls are internally trusted. Therefore, they are more efficient than `P/Invoke` calls because they do not have parameter marshalling and security checks.

FCalls have a number of peculiarities which make them difficult to implement. First, their parameter ordering must match the call ordering of the JIT calling convention. Therefore, they are declared and implemented using a set of macros which define the correct parameter ordering. Second, they must behave like managed code. This means they must periodically yield to the garbage collector, in case garbage collection is necessary. If yielding is not performed and a garbage collection is required, the FCall would make all other threads wait until it polls for collection. Only when *all* threads enter the safe state does collection commence. Third, unlike in managed code, the runtime cannot and does not keep track of object pointers in an FCall. Therefore, it is the programmer's responsibility to protect object pointers by declaring them using a set of provided macros. Programmer-declared object pointers within FCalls are updated during garbage collection.

When writing FCalls it is essential to understand under which circumstances garbage collection can occur and which object pointers need to be protected. Otherwise, object pointers will be inadvertently trashed when the objects are moved or collected during garbage collection.

5.2 The Garbage Collector

The SSCLI virtual runtime provides a two-generational garbage collector. Objects are originally allocated in the younger generation and if they pass a garbage collection, they are promoted to the elder generation. Generational garbage collection exploits the fact that different objects have different lifetimes. Objects in the youngest generation have a low survival rate, while objects in the elder generation have a higher survival rate. Therefore, the

younger generation is collected often, while the elder generation is collected less frequently. When a set of objects are promoted to the elder generation, they are copied to the elder generation, with compaction to reduce fragmentation. Once in the elder generation, objects are collected if abandoned, but are no longer compacted.

The garbage collector maintains a list of objects which require pinning and these objects are not moved. Rather, the entire block of younger generational memory is assigned to the elder generation thereby promoting pinned objects. A new younger generation is allocated. Non-pinned objects are copied and compacted as before.

Garbage collection occurs when remaining memory on the heap runs low. Specifically, it is triggered by a request for a new object. To perform a garbage collection, all threads must be frozen in a safe point. To facilitate this, the jitted code periodically polls to yield itself to garbage collection, in case it is necessary.

5.3 The Runtime Object and Class Model

Every SSCLI object is an instance of the most basic type, `System.Object`, and exists on the garbage collected heap. Within the SSCLI runtime, `System.Object` is defined by the internal type, `Object`. `Object` contains just one field, a reference to the object's `MethodTable`. All of an object's instance data exists directly after the `MethodTable` reference.

The `MethodTable` is the gateway to commonly accessed type information. This includes a reference to an array of `FieldDesc` entries. Each field of every class type is described with a `FieldDesc`; a highly optimized structure, using a bit field to describe field information.

In addition to the runtime structures described, the SSCLI also provides access to type metadata, a far less efficient repository of *all* class information. The reflection library uses metadata to provide dynamic runtime access to type information.

5.4 The Platform Adaptation Layer

The SSCLI runtime is implemented over the Platform Adaptation Layer (PAL), a virtual subset of the Windows API. This layer provides portability. Much of the work involved in porting the SSCLI to another platform is in implementing the PAL for the target platform. Because the PAL is essentially a subset of the Windows API, the Windows implementation is thin, while the other major implementation, the UNIX PAL, is thicker. We have not tested whether the PAL implementation has a significant effect on the performance of the SSCLI. However, a difference in performance should be expected.

6. MPICH2

MPICH2 implements the MPI-1 and MPI-2 specifications using a layered approach for portability (Figure 6). The top layer defines a platform and interconnect generic MPI interface. The next layer is the Abstract Device Interface (ADI), or *device*, layer which defines operations such as message queuing, packetizing, handling heterogeneous communication and data transfer. CH3 is the most common device implementation. CH3 also defines a lower level *channel* layer, which is specifically responsible for data transfer.

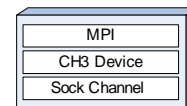


Figure 6. The architecture of MPICH2 over Windows Sockets.

Implementing MPICH2 with a new transport requires developing a new channel [19]. For the simplest port, this requires implementation of five functions which define the simplest functionality required to move a message from one address space to another [20]. A more comprehensive port might re-implement the entire device.

The most common MPICH2 configuration is the sock channel with the CH3 device, providing communication over TCP sockets. There are two versions of the sock channel: Windows and Posix. Other channels include: Shm, communication over shared memory; and ssm, communication over shared memory and sockets.

7. Motor Implementation

Implementing MPICH2 within the SSCLI involved a number of steps, including: porting the MPICH library to the PAL; implementing an FCall MPI interface; and implementing a System library in the System.MP namespace.

The MPICH2 layered model and SSCLI Platform Adaptation Layer provided us with existing layers of virtualization to realize an easily portable infrastructure, as discussed in Section 4.1.

The current Motor runtime is implemented for Windows using the MPICH2 Windows sock channel within the CH3 device (Figure 7). Porting Motor to a Unix platform would involve swapping in the MPICH Posix Sock channel and Unix PAL.

At the time of writing, Motor implements a representative range of MPI-1 functionality, but does not implement the entire function set. Implemented functions include: blocking, synchronous and immediate point-to-point routines; selected communicator routines; and selected collective routines such as broadcast and scatter /

gather. In addition, we have implemented selected MPI-2 functionality such as dynamic process management and dynamic intercommunication routines. The object model is based on the official MPI-2 C++ bindings.

In addition Motor implements the extended Object Oriented operations.

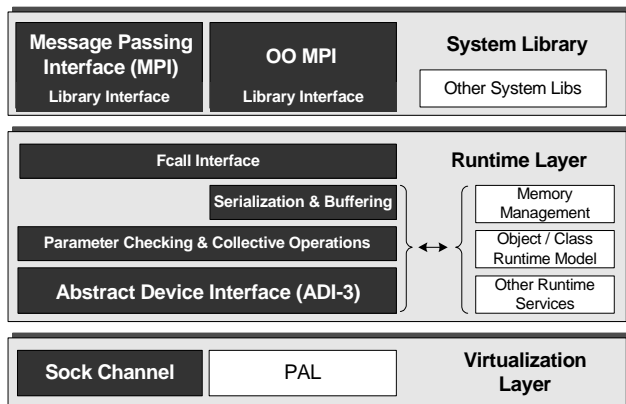


Figure 7. Motor implementation over Windows, using Windows Sockets. Windows specific functionality is contained entirely within the virtualization layer. Motor extensions to Rotor are accentuated in black.

7.1 Porting MPICH2 to the SSCLI PAL

MPICH2 provides a Windows port which formed the basis of our port to the PAL. In our case, a handful of Windows API operations, used by MPICH2, were unsupported by the PAL. Many of these functions were mapped to PAL-supported functions, while the PAL was extended by a small handful of functions. Only the lowest layer of MPICH, the Sock channel was not ported to the PAL. It is part of the virtualization layer and interfaces directly to the operating system, including the Windows specific I/O completion ports (IOCP) mechanism, which is unsupported by the PAL.

A number of changes were made to the MPICH2 library, including:

- Blocking system calls were replaced with a *polling-wait*, which periodically releases and polls the garbage collector. This is done to ensure that the thread performing the FCall does not block the entire runtime when a garbage collection is required.
- The MPICH2 library has been changed to expect a SSCLI `Object*` as the provided buffer, rather than `void*`. The library resolves the `Object` to the offset location of its instance data, to pass to the underlying transport. In addition, the MPICH2 library is responsible for pinning the `Object`, which is discussed in more detail in Section 7.4.

7.2 System.MP Managed Library

The Motor managed message passing library was implemented in the `System.MP` namespace. It is written in C# and provides a managed object oriented interface to the underlying runtime-internal MPI Core. The library includes a private `MPDirect` class which contains `InternalCall` definitions into the underlying runtime. Each MPI function in `System.MP` interfaces to an `MPDirect` `InternalCall` function, which is matched by an underlying `FCall` in the runtime (see Figure 8 for a `Recv` operation example). The `System.MP` library has been implemented using the design considerations raised in Sections 4.2.1 and 4.2.2.

```
public void Recv (object obj, int source, int tag,
    Status status)
    [MethodImplAttribute(MethodImplOptions
        .InternalCall)]
    public static extern void Recv (object obj,
        int source, int tag, int comm,
        ref MPI_Status status,
        ref MyStackCrawlMark stackMark)
    FCIMPL6(void, MP_Recv, Object *object, int source,
        int tag, MPI_Comm comm, MPI_Status *status,
        StackCrawlMark* stackMark)
```

Figure 8. The Motor `Recv` call, `InternalCall` `Recv` function and its matching `MP_Recv` `FCall` implementation. Notice that the `MP_Recv` is defined within a macro to ensure correct parameter ordering.

7.3 Message Passing FCall Interface

The `FCall` interface exposes MPI functionality from within the runtime and interfaces between the object oriented runtime and the ported MPI Core.

Regular MPI `FCalls` perform a number of tasks, including: check parameters; evaluate object size; ensure the send or receive object does not contain object references and perform the operation.

`FCalls` for the extended OO operations perform the following tasks: check parameters; perform the operation; and perform serialization or deserialization.

Furthermore, each `FCall` function protects object pointers and yields to garbage collection.

7.4 MPI with Managed Memory

This section describes how the Motor MPI operations interact with managed memory and describes the implementation of the Motor pinning policy.

The most obvious interaction between the MPI library and the garbage collector is garbage collection polling. A typical blocking MPI operation has polling implemented in three places:

- Upon entry to the `FCall`, before the operation has

commenced;

- Immediately prior to exiting the FCall, after the operation has been completed; and
- While in a polling-wait state, waiting for the operation to complete, after it has commenced.

Polling ensures the FCall thread does not block other threads which might require garbage collection.

For regular MPI operations, Motor applies a pinning policy, as introduced in Section 4.3. This policy ensures that the library only requests pinning when it is apparent that the object is at risk of being moved during an impending garbage collection. The pinning policy does not reduce the chances that a garbage collection will occur or that objects will require pinning *during* that collection. However, it does minimise the performance overhead imposed by pinning unnecessarily for each operation.

The pinning policy is different for blocking operations and non-blocking operations.

For *blocking operations* Motor checks the objects internal memory address against the boundaries of the younger generation. If the object is outside this boundary, then it has already been promoted to the elder generation and is not at risk of being moved during collection. Motor does not pin the object.

On the other hand, if the object has been established as a resident of the younger generation, it might be moved during collection. In this case, pinning is not performed automatically, but is deferred until the operation enters a polling-wait state for the transport to complete. This is done because many blocking MPI operations complete quickly and never need to enter the polling-wait. These operations do not need to pin because without entering the polling-wait there is no opportunity for garbage collection *before* the operation has completed.

For *non-blocking operations*, Motor also checks whether the object belongs to the younger generation. If so, the object is pinned immediately. As discussed in Section 4.3, Motor releases the pinned object after the underlying operation has finished.

We found the simplest and most efficient way to achieve this was to leave the pinning request until the mark phase of the next garbage collection. During the mark phase the garbage collector checks the status of the underlying non-blocking transport operations. If the operation is ongoing, the object is marked as pinned and therefore remains untouched during the impending sweep phase. Otherwise, the pinning request is no longer necessary and is disregarded. Therefore, during the sweep phase, only objects which are currently being accessed by an underlying transport operation are pinned. Checking the status of an operation causes the garbage collector minimal extra work during the mark phase of collection.

The Motor extended object oriented operations do not need to pin memory because the Motor custom

serialization mechanism provides a static memory buffer.

7.5 Object Oriented Message Passing Operations

The Motor extended object oriented operations have been implemented with a custom serialization mechanism. It produces a flat object-tree representation with two parts:

- A type table, which details class information; and
- Object data, which consists of the objects laid out side-by-side, prefixed with an internal type reference. Object references are exchanged for their local internal equivalent. References to objects not included in the serialization are swapped to null.

The custom Motor serialization mechanism is recursive and follows every ‘Transportable’ object reference. Introspecting type fields for a Transportable attribute is possible using the reflection library. However, this is a relatively slow operation because it accesses type metadata. Instead, we implemented a Transportable bit on the `FieldDesc` structure.

For each object, the serialization mechanism iterates through all the `FieldDesc` entries for that type. Entries which are Transportable and not null, are also serialized.

To support collective communication operations such as scatter or gather, the serialization mechanism produces a split representation. A single split representation is constructed of many regular representations, each with an individual type table and each individually deserialisable at the receiving end. For scatter operations the serialization mechanism automatically splits the array and flattens referenced objects. Conversely, for gather operations the deserialization mechanism takes many split representations and reconstructs them into a single array.

Motor provides buffers for object oriented message passing operations, which are allocated from static runtime memory. They are created on demand and stored in a stack for later use. At garbage collection the stack is checked for buffers which are unused since the last garbage collection and these are unallocated.

Before sending the serialized buffer, Motor sends the size of the buffer. This ensures the receiver can prepare a sufficient buffer and is also used by `mpiJava`.

8. Performance

This paper provides *preliminary* performance results to show that Motor retains the high performance of its synthesized components. To evaluate performance of the Motor MPI library we compared a number of applications implementing an MPI Ping-Pong algorithm, where two processes take turns to send and receive a piece of data. A *single iteration is the time for a round trip*. Each experiment performed 200 iterations, the last 100 of which

were timed. A range of buffer sizes were tested. Each buffer size was tested three times. The average time in microseconds per iteration was calculated for all three experiments. The following implementations of the Ping-Pong code were compared:

- A C# Motor application;
- A native C++ application using MPICH2;
- A C# application using the Indiana C# .Net bindings [7], hosted by both the SSCLI and .NET v1.1. Pinning is performed for each MPI operation; and
- A Java application using mpiJava, version 1.2.5 using the Sun JDK 1.5.0_04.

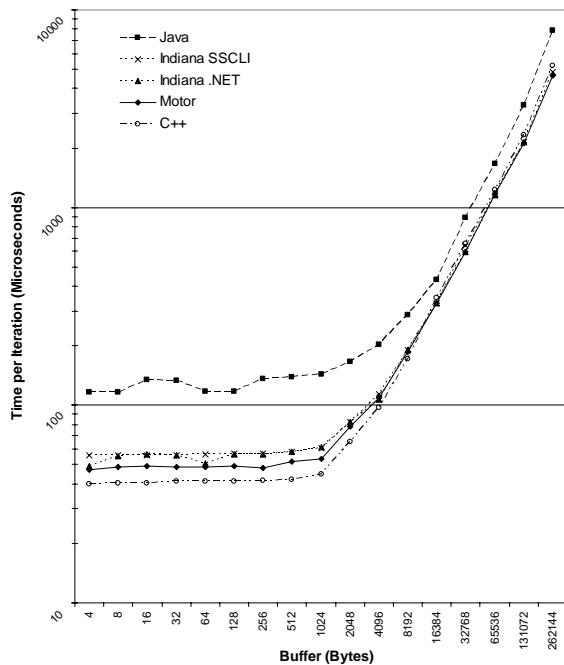


Figure 9. Ping-pong comparison of regular MPI operations, showing time per iteration.

The Indiana C# bindings and mpiJava were all originally implemented over MPICH, which does not perform as well as MPICH2. To provide a fair comparison, they were reimplemented over MPICH2 v1.0.2. All tests were performed on a Pentium M 1.7GHz, with 1GB RAM, running Windows XP SP2. A single node was used because we are only interested in the performance of the MPI implementation, rather than the underlying transport.

Figure 9 graphs these results. As expected, the C++ code shows the strongest performance. Significantly, Motor shows the next best performance. We can attribute these positive results to two reasons:

- The Motor MPI library uses the fast InternalCall mechanism, imposing a smaller overhead than

managed-to-native call mechanisms such as P/Invoke.

- Due to the pinning policy discussed, pinning is performed only when necessary, reducing overhead.

Motor performs noticeably better than the Indiana bindings hosted by the SSCLI using MPICH2; 16% at a peak; 8% on average over all buffer sizes; and 3% on average over buffer sizes greater than 65,536 bytes. These are encouraging results considering Motor is a synthesis of the major components: SSCLI and MPICH2.

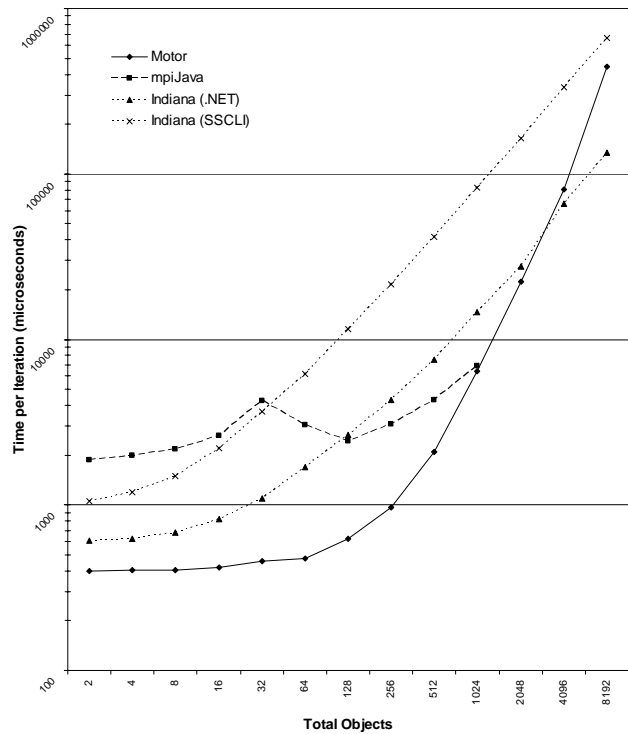


Figure 10. Ping-pong comparison of transport of a linked-list of objects. The bump in mpiJava is consistent and might suggest Java employs different serialization algorithms or data structures to serialize small or large numbers of objects. mpiJava results stop at 1024 objects because longer linked lists caused a stack overflow exception in the Java serialization mechanism. It is interesting to note the difference in performance of the .Net and SSCLI serialization mechanisms.

To test the performance of the Motor extended object oriented operations we also used a Ping-Pong code similar to the previous test. However, this experiment sent and received structured data, serialized and deserialized using the particular mechanism of each system. The cost of serialization was intentionally included. We compared:

- A C# Motor application, using the Motor extended object-oriented operations;
- A Java mpiJava application, using the mpiJava MPI.Object datatype, which uses the standard Java serialization mechanism to transport objects; and
- An Indiana C# .NET application, hosted by both the SSCLI and commercial .NET. To provide object tree

transport, we used the standard CLI binary serialization mechanism to produce a buffer to be transported using the standard MPI routines.

The structured data was in the form of a linked list, with each list element containing a buffer (Figure 5 shows a similar structure). The *total* data buffer was 4096 bytes, evenly distributed over the *entire* linked list. The total number of objects transported is twice the number of linked list elements because the data array referenced by each linked list element is itself an object.

Figure 11 graphs these results. The Motor serialization mechanism performs best for numbers of objects less than 2048. Poorer results for large numbers of objects can be attributed to the fact that at the time of writing we employ a linear structure to record objects visited during serialization. This causes excessive search times with large numbers of objects and will be improved when we implement an efficient structure to record objects visited.

9. Conclusion

Motor provides a vehicle for research into virtual infrastructures for high performance computing. For the e-Scientist, an environment such as Motor is significant because it provides compile-once-run-anywhere application deployment, a *guaranteed* message passing library and common runtime services. This is of particular significance to e-Scientists working in a dynamic grid environment where fast application development and deployment is particularly important.

Architecturally, Motor places message passing alongside other essential CLI runtime services. This gives the message passing library access to the entire internal runtime allowing us to implement a pinning policy, fast serialization and guarantee object model integrity.

Finally, we have demonstrated Motor's strong performance for both regular buffer-to-buffer MPI and the higher level extended object oriented operations.

In the future, we plan to integrate the Motor MPI library more closely with other runtime services to provide transparent process management. The layered Motor architecture will allow us to port Motor to other platforms and interconnects.

References

- [1] "Standard ECMA-335: Common Language Infrastructure, 3rd edition (June 2005)", 2005, <http://www.ecma-international.org/publications/standards/Ecma-335.htm>.
- [2] S. Morin, I. Koren, and C. Krishna, "JMPI: Implementing the message passing standard in Java" *International Parallel and Distributed Processing Symposium: IPDPS 2002 Workshops.*, 2002.
- [3] G. Judd, M. Clement, Q. Snell, et al., "Design issues for efficient implementation of MPI in Java" *Proceedings of the ACM 1999 Java Grande Conference*, 1999.
- [4] K. Dincer, "Ubiquitous message passing interface implementation in Java: jmpii" *Proceedings of the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, 1999.
- [5] M. Baker, B. Carpenter, G. Fox, et al., "mpiJava: An Object-Oriented Java interface to MPI" *International Workshop on Java for Parallel and Distributed Computing*, 1999.
- [6] S. Mintchev, "Writing programs in JavaMPI" *Technical Report MAN-CSPE-02*, 1997.
- [7] J. Willcock, A. Lumsdaine, and A. Robison, "Using MPI with C# and the Common Language Infrastructure" *Concurrency and Computation: Prac. and Experience*, 2005.
- [8] "MPICH2 Homepage" <http://www-unix.mcs.anl.gov/mpi/mpich2/>.
- [9] "Shared Source Common Language Infrastructure 1.0 Release" <http://msdn.microsoft.com/net/sscli/>.
- [10] D. Stutz, T. Neward, and G. Shilling, *Shared source CLI essentials*, 1st ed. Sebastopol, CA Farnham: O'Reilly, 2003.
- [11] R. Figueiredo, P. Dinda, and J. Fortes, "A Case for Grid Computing on Virtual Machines" *Proc. International Conference on Distributed Computing Systems*, 2003.
- [12] S. Adabala, V. Chadha, P. Chawla, et al., "From virtualized resources to virtual computing grids: the In-VIGO system" *Future Generation Computer Systems*, 2005.
- [13] X. Zhao, K. Borders, and A. Prakash, "SVGrid: A Secure Virtual Environment for Untrusted Grid Applications" *Proceedings of the 3rd international Workshop on Middleware For Grid Computing*, 2005.
- [14] R. Castain, T. Woodall, D. Daniel, et al., "The Open Run-Time Environment (OpenRTE): A Transparent Multi-Cluster environment for High-Performance Computing" *Euro PVM/MPI 2005*, 2005.
- [15] W. Vogels, "HPC.NET - are CLI-based Virtual Machines Suitable for High Performance Computing?" *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, 2003.
- [16] B. Carpenter, V. Getov, G. Judd, et al., "MPJ: MPI-like message passing for Java" *Concurrency: Practice and Experience 2000*, 2000.
- [17] M. Baker and B. Carpenter, "Thoughts on the structure of an MPJ reference implementation" *Technical Report CRPC-TR99803*, 1999.
- [18] "Mono" <http://www.mono-project.com/>.
- [19] W. Gropp and E. Lusk, "Creating a new MPICH device using the channel interface" *Technical Report ANL/MCS-TM-213*, 1995.
- [20] W. Gropp, E. Lusk, N. Doss, et al., "A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard" *Parallel Computing*, 1996, <http://citeseer.ist.psu.edu/gropp96highperformance.html>.