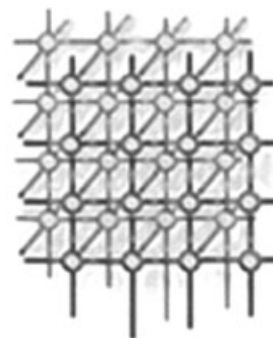


Parallel Programming on a High-Performance Application-Runtime



Wojtek James Goscinski^{*,†} and David Abramson

Faculty of Information Technology, Monash University, Vic. 3145, Australia

SUMMARY

High-performance application development remains challenging, particularly for scientists making the transition to a heterogeneous grid environment. In general areas of computing, virtual environments such as Java and .Net have proved to be successful in fostering application development, allowing users to target and compile to a single environment, rather than a range of platforms, instruction sets and libraries. However, existing runtime environments are focused on business and desktop computing and they do not support the necessary high-performance computing (HPC) abstractions required by e-Scientists. Our work is focused on developing an application-runtime that can support these services natively. The result is a new approach to the development of an application-runtime for HPC: the Motor system has been developed by integrating a high-performance communication library directly within a virtual machine. The Motor message passing library is integrated alongside and in cooperation with other runtime libraries and services while retaining a strong message passing performance. As a result, the application developer is provided with a common environment for HPC application development. This environment supports both procedural languages, such as C, and modern object-oriented languages, such as C#. This paper describes the unique Motor architecture, presents its implementation and demonstrates its performance and use. Copyright © 2008 John Wiley & Sons, Ltd.

Received 16 April 2007; Revised 18 December 2007; Accepted 10 February 2008

KEY WORDS: parallel programming; virtual machines; message passing

1. INTRODUCTION

Parallel programs are most commonly developed natively[‡]; they are written and compiled directly for a specific operating system and instruction set. This approach offers the highest level of

^{*}Correspondence to: Wojtek James Goscinski, Faculty of Information Technology, Monash University, Vic. 3145, Australia.

[†]E-mail: wojtek.goscinski@infotech.monash.edu.au

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



performance and allows the programmer full control of memory management. In contrast, the current approach in software engineering has been to move toward a ‘managed’ (see Footnote ‡) 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. They define a range of features and services, such as a common-type system, memory management, portable binary format, self-describing binaries and security. In addition, they provide a common set of libraries—a standard set of functionality, which is likely to be required by the user. In the case of Java and .Net, this includes web services, IO, database, user interface, security and networking functionality. Most significantly, runtime environments provide an *execution guarantee*. In essence, a runtime specification defines a contract between a user code and the runtime. Thereby, the code is guaranteed to be executed on a correctly implemented runtime, so long as both parties adhere to the contract.

Taking a similar approach for high-performance computing (HPC) would provide users with a *virtual HPC environment*, which would support the functionality and services that are important to high-performance application developers. For example, message passing, shared memory, process management and associated mathematics libraries, such as linear algebra libraries, are all important tools that should be considered for inclusion into a runtime for high-performance application development. Thereby, users would be provided with a common environment that would make high-performance applications easier to write and more accessible to scientists.

Such an environment is particularly valuable now that HPC is adopting grid computing infrastructure [2], 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 that best suits their needs. However, realistically, they are often constrained by deployment issues such as platform and environmental dependence. The Message Passing Interface (MPI) specification simplifies scientific application development because it defines a single interface that 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. Native code requires porting for different platforms, which is a major undertaking considering the potential complexity of e-Science software and the range of platforms that might constitute a grid. Furthermore, native code needs to be recompiled for each specific platform, which itself is a major effort. This limits users when making deployment and execution decisions. Writing highly tuned, complex scientific applications, for a range of heterogeneous resources remains difficult and time consuming [3–6].

An application-runtime[§] allows users to target and compile to a single environment, rather than a range of platforms, instruction sets and libraries. However, existing runtime environments alone are inappropriate for HPC; neither Java nor the CLI provides necessary high-performance functionality, such as the tools and interfaces mentioned above. Nevertheless, researchers are interested in using

[§]To avoid confusion, this paper avoids using the overloaded term ‘virtual machine’ when referring to application-runtime environments such as Java and the CLI. It is important to note that many virtual machines are not application-runtimes. For example, Parallel Virtual Machine (PVM) is called a virtual machine but it is not an application-runtime environment under a widely accepted definition because it does not provide a common and complete execution environment for application binaries (such as that provided by .Net and Java).



runtime for HPC and significant work has been carried out in this area [7–18]. In particular, a number of projects have proposed a high-performance communication extension to an existing runtime [7,9–15,17], including the Indiana University .Net bindings [17], MPJ [12], mpiJava [11] and work supported by the Java Grande Forum [18]. The most efficient of these systems have all proposed an external addition to a runtime, thereby supporting high-performance communication externally. Using this approach, the runtime and high-performance communication library remain separate. The 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.

In contrast, the goal of our research is to develop a *new* virtual environment for HPC by integrating the requirements of HPC developers *within* the runtime. The first steps toward this goal are to develop an environment that supports high-performance message passing. In our architecture, the high-performance message passing library exists alongside other virtual machine services such as memory and process management, IO, security and networking (Figure 1). To demonstrate our approach, we provide a proof-of-concept implementation in the form of Motor (**M**PI + **R**otor) [4]. Motor has been developed by integrating a high-performance message passing library, the MPICH2 [19] MPI library, within an existing runtime, Microsoft's Shared Source Common Language Infrastructure (SSCLI) [20,21] (also known as Rotor).

Motor supports both procedural and object-oriented languages. In addition to providing a standard MPI library for procedural languages, Motor provides two MPI-like libraries for efficient object-to-object transport and structured data transport. Because it has been designed to support object-oriented languages, Motor provides functionality not available in MPI, including the ability to transport arbitrary objects, trees of objects and arrays of objects, and the ability to scatter and gather arrays of objects.

Motor is significant for a number of reasons. First, it allows users to write 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 execution regardless of underlying heterogeneity. Second, applications written for Motor are provided with runtime services and a guaranteed standard set

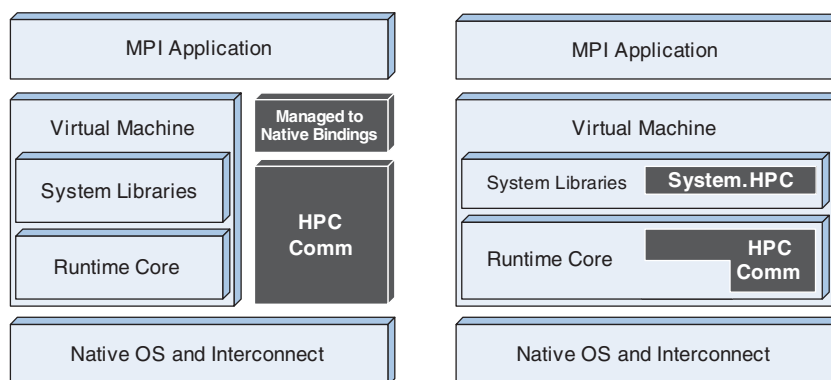


Figure 1. The external architecture (left) in contrast to the Motor-embedded architecture (right).



of System libraries, including message passing. HPC developers can develop against a known and standard environment. Third, Motor is language neutral. This means that it can be used for modern object-oriented languages and procedural languages. In particular, Motor is a modern runtime that can be used to execute legacy applications written in C or FORTRAN.

Both architecturally and practically, implementing high-performance message passing directly within the virtual machine has a number of advantages over the existing external 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. In the case of object-oriented languages, this means that it 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, whereas lower-level functionality can be re-implemented for specific platforms and interconnects.

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

This paper discusses previous work in the area of HPC using virtual machines (Section 2), introduces our architecture for a high-performance virtual machine (Sections 3 and 4), introduces the Object Passing Interface (OPI), an object-oriented MPI (Section 5), introduces relevant technology (Sections 6 and 7), discusses the implementation of Motor (Section 8) and presents performance results (Section 9).

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), which has been investigated by a number of projects [22–29] and at the subsystem component level (e.g. open runtime environment (OpenRTE) [30] and Harness [31]). None of these attempt to virtualize grid resources at the application-runtime level (e.g. Java or .Net), which has proved to be a successful approach for business and desktop applications. Our work is focused on application-runtime virtualization and therefore our review of earlier work is based around projects that have focused on this type of virtualization.

Application-runtimes such as Java and the CLI are characterized by a number of features, including a virtual runtime that just-in-time (JIT) compiles a processor-independent intermediary language (IL) and a standard set of System libraries. In addition, most common runtimes support memory management, which collects abandoned memory and an object-oriented memory model. The CLI is an example of a runtime specification that supports both procedural and object-oriented languages.

In a business and desktop computing environment these features provide an environment for easy prototyping and development. However, in exchange for platform independence and other



runtime services, an overhead on performance is created. This paper does not specifically benchmark application-runtimes, however, one study comparing high-performance benchmarks executed over different versions of the CLI and Java can be found in [16]. Although virtual machines do introduce a performance overhead, we believe that the potential advantages in simplifying application development could help foster grid computing within the general science community.

To date, both Java and the CLI focus on desktop, business and mobile applications. Neither explicitly supports HPC abstractions such as high-performance message passing.

2.1. MPI implementations for virtual environments

A number of environments have been developed, which provide a Java or .Net parallel communication mechanism. These projects have either made an argument for using application-runtime environments for HPC or evaluated the relevance of these environments for this task. In addition, this range of projects has demonstrated that the motivation to develop specialized application-runtime environments exists in the high-performance community. Our work continues this research though we have applied a significantly different approach.

mpiJava [11] is a Java wrapper to an underlying native MPI implementation. The mpiJava bindings are based on the MPJ application programming interface (API) (discussed below). mpiJava supports transport of simple-type arrays or objects. Object transport is implemented by serialization using a standard Java serialization. JavaMPI [15] 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 a native code. JMPI [10] is a pure Java implementation of a subset of MPI. Communication in JMPI is implemented over Java Remote Method Invocation. jmpi [14] is another pure Java implementation, built over the Java sockets library.

The MPJ API [12] is an API specification for Java MPI bindings. Developed by the Message Passing Working Group of the Java Grande Forum [18] it was produced to create a standard Java MPI-like API specification. It is not an official binding to MPI, but it is 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 [17] provide two bindings to the underlying native MPICH MPI library: C# bindings, based on the official C++ MPI bindings and MPI.NET, a higher-level interface specifically focused toward 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 [17].

These projects discussed can be grouped into two categories:

- Pure managed implementations of MPI that execute entirely over the virtual machine. These implementations are portable but suffer from inefficiency.
- Managed wrappers that provide a managed interface to an underlying and external native MPI implementation. These systems perform significantly better than pure managed systems because the high-performance communication mechanism has direct native access to transport services.



The common characteristic of these two classifications is that the runtime environment is unaware of the HPC communication library. In contrast, the Motor architecture is *embedded*, whereby the runtime environment provides a shared memory or message passing abstraction.

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 architectural issues that are also relevant to our embedded implementation. These include portability, managed-to-native interface, object model integrity and interaction with managed memory (some of these issues have been raised earlier [32]).

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 [32].

Using a managed-to-native call mechanism such as JNI [33] or P/Invoke [34] imposes an overhead on each MPI call because both JNI and P/Invoke require marshaling and impose security mechanisms.

Object model integrity and interaction with managed memory are discussed in Section 5.

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 that 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 addition to these architectural problems, all the described systems target a specific language. A number of runtimes have been developed, which provide a common environment for a range of languages, including the CLI and Parrot [35].

3. INTRODUCTION TO MOTOR

Motor is a synthesis of the Microsoft SSCLI and the MPICH2 [19] MPI library.

A number of runtime environments—including various versions of the CLI, Java and Python—were considered for our implementation. We decided to use an implementation of the CLI primarily because it is the predominant language-neutral environment. CLI compilers exist for procedural languages such as C, FORTRAN and Pascal, and object-oriented languages such as C#, Java and Python.

In addition, the CLI is considered a better choice over Java for scientific computing because it supports true multidimensional arrays such as those in C and FORTRAN. In contrast, Java uses the arrays-of-arrays model, which means that in Java n -dimensional arrays are implemented as one-dimensional arrays, with $n - 1$ -dimensional array elements. Transporting a multidimensional array constructed using the arrays-of-arrays model is difficult because the array is constructed of multiple objects. Each of these objects could be placed at a different location in memory. Therefore, the array needs to be transported in multiple messages, or it needs to be packed or serialized before



transport. In contrast, a true multidimensional array can be transported in one message because it exists at one location.

Both the open source Mono and the SSCLI were considered for this work. The SSCLI performs the worst of all benchmarked CLI implementations [16]. However, it was chosen as the basis of Motor because it is a comprehensive CLI implementation, which by all accounts is very similar to the optimized commercial .Net [21]. Using the SSCLI as the basis of our prototype provides the best insight into developing Motor using a highly tuned commercial runtime such as Java or .Net. The SSCLI is described in Section 6.

The choice of the MPICH2 library was influenced primarily by its layered design, allowing it to support different platforms and interconnects. In addition, MPICH2 is ported to the Windows operating system, providing a starting point for our integration into the SSCLI. MPICH2 is described in Section 7.

In contrast to the managed-wrapper MPI implementations that we discussed, the Motor message passing library has been implemented directly within the virtual machine. The message passing library has been implemented in the same manner 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 (MPC), within the virtual runtime. The MPC is based on the Windows MPICH2 code-base, ported to the SSCLI Platform Adaptation Layer, which defines a virtual subset-Windows API.

4. LOGICAL DESIGN

Motor provides two sets of operations for parallel communication, supporting both procedural and object-oriented languages:

- (1) A standard MPI implementation for procedural languages. This is a language-neutral interface for parallel programming on a distributed memory system.
- (2) A set of MPI-like operations for object transport in object-oriented languages. These operations, which are called the Motor Object Passing Interface (OPI), do not directly implement the MPI specification. Rather, they are a set of MPI-derived operations, which are based on the MPI-2 C++ bindings and object model. However, the OPI is specifically designed for an object-oriented environment and supports two types of operations:
 - A set of regular object-to-object transport operations, which are adapted to preserve object model integrity in a strongly typed object-oriented environment.
 - A set of higher-level operations for structured data transport in object-oriented languages. These operations are called the Motor Structured Data Transport Operations[¶] (SDTO) and they provide automatic serialization and buffer management.

The Motor MPI library is an embedded implementation of the MPI standard. Therefore, we do not discuss the design of this interface in detail. However, the OPI is a new interface design, which

[¶]These operations are called the Extended Object-Oriented Operations in an earlier publication [4].

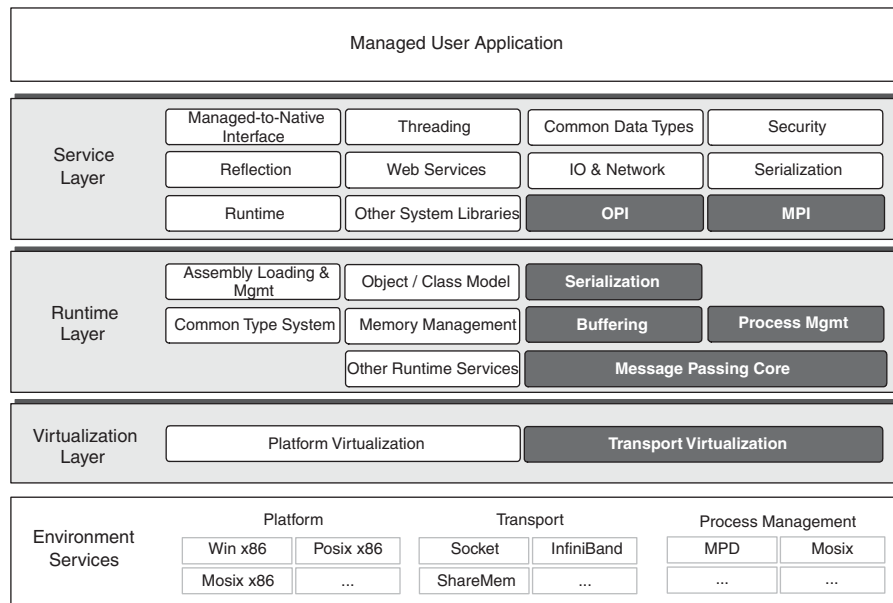


Figure 2. The Motor architecture. Extensions to the architecture of the regular CLI virtual environment are accentuated in black.

we have developed specifically for an object-oriented environment. Therefore, the OPI design is discussed in detail in Section 5.

Figure 2 illustrates the architecture of the Motor virtual machine. Applications are written in a procedural language such as C or FORTRAN or an object-oriented language such as C#. These applications are compiled to the CLI IL. They interface to the runtime through the System library, which includes the above-mentioned message passing libraries. The System MPI library is similar to the earlier mentioned managed-wrapper MPI implementations. However, this library interfaces directly to the MPC, an MPI library implemented within the underlying runtime.

The MPC implements the non-transport-specific parts of the message passing library. It is implemented over a virtualization layer that provides transport and platform virtualization. This 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 the addition of 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 and class model. The manner in which Motor interacts with these services is presented in Section 8.

5. THE OPI DESIGN

The Motor OPI operations are based on the MPI specification and provide efficient object-to-object transport (Section 5.1) and structured data transport (Section 5.2).



5.1. Object-to-object transport

Object-to-object transport means that the developer provides a send object and a receive object. The operations transport object data, whereas internal object headers are not transported. Figure 3 illustrates Motor MPI operations, including selected communicator operations (i) blocking and immediate point-to-point operations (ii) selected collective operations such as broadcast and scatter/gather (iv) and selected MPI-2 functionality, such as dynamic process management (v).

The OPI deviates from the MPI specification to protect memory model integrity. Sections 5.1.1 and 5.1.2 introduce the importance of protecting object model integrity and discuss object-to-object

```
public class Comm
{
    static public Comm World ;
    static public Comm Self ;
    static public Comm Null ;
    static public Intercomm GetParent ()
    ...
    public int GetRank ()
    public int GetSize ()
    ...

    public void Recv (object obj, int source, int tag, Status status)
    public void Recv (object obj, int offset, int numcomponents, int source,
        int tag, Status status)
    public void Send (object obj, int dest, int tag)
    public void Send (object obj, int offset, int length, int dest, int tag)
    public void IRecv (object obj, int source, int tag, Request request)
    public void ISend (object obj, int dest, int tag, Request request)
    ...

    public object ORecv (int source, int tag, Status status)
    public void OSend (object obj, int dest, int tag)
    ...
}

public class Intracomm : Comm
{
    public void Barrier ()
    public void Bcast (object buffer, int root)
    public void Gather (object sendbuf, object recvbuf, int root)
    public void Scatter (object sendbuf, object recvbuf, int root)
    ...

    public Intracomm Spawn (string command, string[] argv,
        int maxprocs, Info info, int root)
    ...

    public object OBcast (object buffer, int root)
    public object OScatter (object sendbuf, int root)
    ...
}
```

Figure 3. A subset of the Motor OPI System.MP object model. Structured data transport operations can be distinguished by an O prefix.



transport in a memory-managed environment. On the basis of this discussion, Section 5.1.3 presents a detailed description of these operations.

5.1.1. Object model integrity

Message passing specifications, such as MPI, assume a linear view of memory. For example, MPI specifically defines *buffer-to-buffer* message transport. The user provides three major parameters: a buffer that is read or written to by the underlying data transport mechanism, a count that defines the number of elements transported and a data type that defines the size of each element. This means that the developer defines the memory starting point and length.

However, in an object-oriented model, memory is a managed graph of objects and the managed code has no regular safe access to the underlying linear view. Therefore, buffer-to-buffer transport is not possible because the developer cannot explicitly provide a memory location. Furthermore, as we will discuss, allowing the developer to explicitly define a memory length would be dangerous to the integrity of the memory model and the overarching environment. Owing to these differences, we refer to buffer-to-buffer transport in an object-oriented environment as *object-to-object* transport, where the user provides both a send object and a receive object. The remainder of this section discusses problems with implementing an MPI object-to-object transport in a runtime environment. These are introduced so that we are able to define a Motor MPI-like interface that provides an efficient zero-copy transport while functioning safely within the rules of the runtime environment.

Object-to-object transport is only possible when transporting single objects (including single structures and arrays of simple types because these will be typically located at one location in the memory). Transporting multiple objects using a single object-to-object operation is impossible because multiple objects will most likely be located at different locations in the memory. Likewise, arrays of objects and trees of objects are not located at a single physical memory location and therefore transporting them as a single transport is impossible. Even if they were located at a single location in the memory, objects would be separated by an object header, which should not be transported to another runtime instance.

Implementing the MPI interface as specified, and without any buffer boundary checks or restrictions, allows the user code to inadvertently (or purposefully) destroy the integrity of the object model. This is because in a strongly typed object-oriented memory model, object references are guaranteed to be either null or reference an object of the correct type. Therefore, compromising the integrity of the object model can occur in two ways:

- By overwriting the end of an object, corrupting the object header and object data of the next object in the memory. This would occur if the developer provided a memory length greater than the receiving object.
- By overwriting an object reference with data. This would occur if a receiving object referenced other objects and those references were overwritten by data during the receive operation.

Corruption of the underlying object tree *will* crash the application environment. This is because the garbage collector requires object model integrity to successfully perform its service. These two occurrences are also a problem in a linear static memory model where it is the responsibility of the developer to maintain data integrity. However, in a managed environment, it is the responsibility of



the *runtime* to guarantee the integrity of the memory model. It would be counter-intuitive to allow the embedded message passing library to compromise object model integrity.

5.1.2. *Object-to-object transport with managed memory*

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 it is efficient because it avoids extraneous memory copies. However, this principle interferes with garbage collection.

Several approaches and algorithms for garbage collection have been developed [36]. Environments such as Java and .Net employ generational collection where new objects are allocated in the younger generation and then if they pass a garbage collection, they are promoted to the elder generation. In essence, generational garbage collection applies a heuristic that exploits the fact that objects have varying lifetimes. Objects in the youngest generation have a low survival rate, whereas objects in the elder generation have a higher survival rate. Therefore, the younger generation is collected often, whereas 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 and compacted to reduce fragmentation.

Implementing an efficient zero-copy MPI in any managed memory environment that has the power to move memory is problematic because the memory might be moved or collected during an MPI operation. Therefore, the solution is to pin the object while transport is underway. Pinning is necessary, but undesirable. It interferes with the garbage collector's standard behavior and causes memory fragmentation. Furthermore, each pin/unpin operation imposes a small performance overhead. For example, in [17] the Indiana CLI MPI bindings suffer from memory pinning.

We believe that relying on the user to pin and unpin memory buffers is dangerous because failing to unpin a memory buffer results in leaking memory. Therefore, we suggest that the runtime should provide automatic pinning and unpinning. Automatically providing memory pinning is relatively simple for blocking MPI operations: the object is pinned at the start of the operation and unpinned once the operation is completed. However, non-blocking operations require unpinning later, once the underlying transport operation has actually been completed. The MPI.NET description in [17] does not illustrate any mechanisms to automatically unpin buffers for non-blocking operations.

Motor provides automatic pinning and unpinning of managed send and receive objects. The manner in which this is achieved is presented in Section 8.3.2.

5.1.3. *Motor OPI object-to-object interface*

The Motor OPI object-to-object interface is based on the official C++ MPI interface, providing the same basic underlying functionality and class model (Figure 3). However, individual OPI operations have been modified to protect the integrity of the memory model. The changes are as follows:

- (1) The provided buffer is a single object (arrays of simple types are allowed because they are a single object). The *count* parameter will always be 1 and has been removed.



- (2) Only object types with no object references may be used as send or receive objects. This prevents overwriting references. Passing objects with references is only allowed through the SDTO.
- (3) Object type is easily determined in a strongly typed object-oriented environment and therefore the data type (MPI_Datatype) parameter has been removed.
- (4) Transporting portions of objects or offsetting into an object is not supported because in most object-oriented environments there is no safe way to refer to a subset of an object.
- (5) However, transporting array subsets 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. The effect of these changes is discussed in Section 9.2.1, which describes porting an existing scientific application to the Motor runtime.

5.2. The Motor SDTO

The Motor SDTO are a set of higher-level operations that extend the OPI library to transport structured data, including objects, array of objects and entire object trees. They provide automatic serialization, de-serialization and buffer management. These operations do *not* provide a zero-copy object-to-object transport. Rather, the sender provides a send object that is serialized to a buffer and transported. The recipient receives to a buffer, deserializes objects from the buffer and returns these to the user.

The Motor OPI design separates structured data transport operations because they perform fundamentally different types of transport: OPI operations are efficient zero-copy operations; whereas the SDTO are slower, using serialization. However, the SDTO provide the ability to transport a wider range of structured data. Figure 3 shows a range of SDTO, including regular send/receive operations (iii); and collective communications, including broadcast operations and scatter/gather operations for arrays of objects (vi).

Section 5.2.1 introduces structured data transport, whereas Section 5.2.2 describes structured data transport in detail, focusing on the interface and the serialization and buffering mechanisms.

5.2.1. Structured data transport

Because the MPI specification assumes a linear view of the memory, it does not take advantage of the structural and self-describing nature of an object-oriented memory model. For example, MPI provides pack and unpack routines for the transport of structural data, which must be used by the developer explicitly. In an object-oriented environment, structural data can be transported automatically by serializing a subset of the object model and then transporting the serialized representation. At the receiving end, the flat representation is deserialized. Because object transport requires a serialization mechanism, it does not achieve the same level of performance. However, it is a significant feature in terms of usability; developers can *automatically* transport structured data and avoid data packing and unpacking operations.

A number of MPI implementations (including mpiJava and the Indiana MPI .NET bindings) have discussed using the standard runtime-provided object serialization mechanism for object transport



(Section 2.1). However, standard serialization mechanisms produce a single atomic flat representation, which cannot be split or offset similar to the standard memory. Therefore, implementing 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 subarrays and serialize them individually. An embedded passing implementation is at a significant advantage to provide a fast object transport mechanism because it can support a fast custom serialization mechanism. In addition, a custom serialization mechanism can be designed to create a split array representation, which can be used for scatter and gather routines.

5.2.2. Motor OPI structure data transport interface

The SDTO (shown in Figure 3, distinguished by an ‘O’ prefix) are designed to transport:

- Single objects: transporting a single object is the default behavior and any object can be transported. This operation transports an object’s data and avoids transporting referenced objects, which 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 provide an offset and count parameter to transport subsets of arrays.
- Trees of objects: to transport a tree of objects this interface defines a declarative attribute^{||}, *Transportable*. This attribute is used to decorate object references that should be propagated during the operation.

Figure 4 shows an example of a linked list element that uses the *Transportable* attribute to denote references to be propagated during transport. In this case, the buffer object and next will be propagated, next2 will not.

Runtime serialization mechanisms also support a declarative attribute that defines reference propagation. A key distinguishing feature of our *Transportable* attribute is that it is an *opt-in*

```
class LinkedTree
{
    [Transportable] public int[] buffer;
    [Transportable] public LinkArray next;
    public LinkArray next2;
}
```

Figure 4. An example of the use of the *Transportable* attribute to annotate references for serialization. This example uses C# syntax.

^{||}Declarative attributes are declarations added to the code, which can be queried at runtime, allowing the developer to describe and query type information. ‘Custom attribute’ is the CLI name for these declarative metadata mechanisms, whereas in Java they are termed as ‘annotations’.



mechanism. Users specify the references that they want to be propagated. On the other hand, reference propagation attributes in Java and the CLI require 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.

Motor implements a custom internal serialization mechanism to provide serialization for structured data transport. Existing runtime environments already provide a serialization mechanism. However, these mechanisms are comprehensive. For performance, the Motor serialization mechanism can be built to include the minimum information necessary to transport an object. In addition, implementing scatter and gather operations requires a mechanism to split the serialized representation. Standard serialization mechanisms produce a single atomic flat representation, which cannot be split or offset similar to the standard memory. The Motor serialization mechanism automatically creates a split representation.

Structured data transport requires a buffer for serialization, which introduces two issues. First, continually allocating buffers for serialization could result in a large overhead on performance. Second, memory-bound applications require diligent user management of memory, including memory buffers. However, some users would prefer automatic buffer management.

In response, the Motor design proposes two approaches to buffering for object transport serialization:

- The user can manage their buffering using regular OPI routines. The Motor custom serialization mechanism is exposed to the user, allowing them to explicitly prepare a buffer, serialize the object tree and perform transport, using regular OPI routines.
- The SDTO provide *automatic* buffer management. Buffers are automatically created when required and stored in a stack for later use. At garbage collection, the stack is checked for buffers that have been unused since the last garbage collection and these are unallocated, freeing unused memory.

6. THE SSCLI

Microsoft's SSCLI includes the CLI runtime, base libraries, a JIT compiler, a C# compiler and a Jscript compiler [20,21]. It is the reference implementation of the CLI and supports both procedural and object-oriented languages. The next section provides an overview of relevant parts of the SSCLI. Figure 2 provides a diagram of the original SSCLI, with our additions in bold. The object-oriented and managed memory aspects of the SSCLI are described in detail because they are relevant to our OPI implementation.

6.1. System library and the CLI runtime

Applications are written for the CLI interface with the System library, which is 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, which 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), which is a trusted call. FCalls are more efficient than the regular managed-to-native P/Invoke calls because they do not have parameter marshaling and security checks.

FCalls have a number of peculiarities that 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 that define the correct parameter ordering. Second, they must behave similar to a managed code. This means that 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 the 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. 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.

6.2. 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, whereas objects in the elder generation have a higher survival rate. Therefore, the younger generation is collected often, whereas 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 that require pinning and these objects are not moved. However, automatic pinning has a two-fold negative effect and is undesirable because:

- (1) Continually requesting memory pinning for each message passing operation adds a small performance overhead.
- (2) Pinned objects disrupt the *normal* ideal operation of the garbage collector and cause fragmented memory.

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 JIT-compiled code periodically polls, to yield itself to garbage collection, in case it is necessary.



6.3. 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`, which contains just one field, a reference to the object's `MethodTable`. All of an object's instance data exist 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`; an 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 less efficient repository of *all* class information. The reflection library uses metadata to provide dynamic runtime access to type information.

6.4. 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 (for the most part, it remaps PAL functions to their underlying Windows version), whereas the other major implementation, the UNIX PAL, is thicker (the UNIX PAL consists of approximately 4.5 times more lines of code than the Windows PAL). 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.

7. MPICH2

MPICH2 [19] is an implementation of the MPI-1 and MPI-2 specifications, which uses a layered approach for portability (Figure 5). 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.

Implementing MPICH2 with a new transport requires developing a new channel [37]. 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 [38]. A more comprehensive port might re-implement the entire device.

The default MPICH2 configuration is the sock channel with the CH3 device, providing communication over transmission control protocol sockets. There are two versions of the sock

^{**}To port the SSCLI to another platform, the other major necessary change would be a new JIT implementation for the specific instruction set.

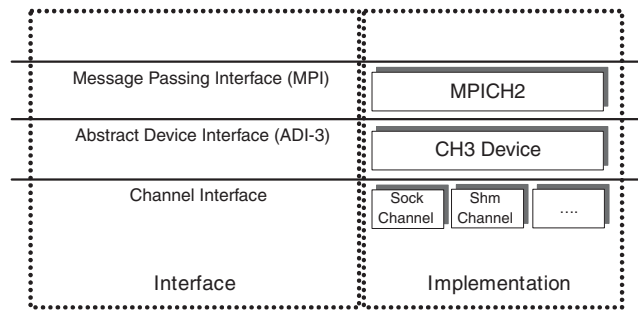


Figure 5. The MPICH2 layered architecture (right) with an example implementation, the CH3 device with the socket channel (left).

channel: Windows and Posix. Other channels include Shm, communication over shared memory; and ssm, communication over shared memory and sockets.

8. MOTOR IMPLEMENTATION

The Motor implementation consists of three parts (Figure 6):

- (1) The Microsoft SSCLI, which provides an existing runtime (shown in white).
- (2) MPICH2, which provides a portable MPI implementation (shown in gray).
- (3) An additional set of new Motor components (shown in black), which implement the OPI libraries and interface to the integrated MPICH2 library.

In addition, Figure 6 illustrates the integration of the layered MPICH2 architecture (shown on the left) within the layered SSCLI runtime environment (shown on the right).

The Motor message passing library has been implemented in a similar manner to the System libraries in the SSCLI and commercial .Net. It consists of a managed library located in the System namespace, which interfaces to the message passing library, within the virtual runtime. The MPC is based on the Windows MPICH2 code-base, ported to the SSCLI PAL, which defines a virtual subset-Windows API. This composition, its individual components and the integration of MPICH are discussed further in the following sections.

System Library: The SSCLI System Library is extended with a managed message passing library, located in the System.MP namespace. This is an interface to the underlying embedded message passing library, including both the MPI and the OPI message passing operations. Sections 8.2 and 8.3 discuss the specific MPI and OPI in detail.

The Motor runtime layer: The SSCLI runtime layer is extended with three major components:

- (1) The MPC, an embedded message passing library, which is based on the MPICH2 code-base. This component is the basis of the MPI and the OPI operations. The MPC consists of the parameter checking and collective communication functionality from the MPICH2 MPI interface, and the CH3 code-base, which is an implementation of the ADI-3.

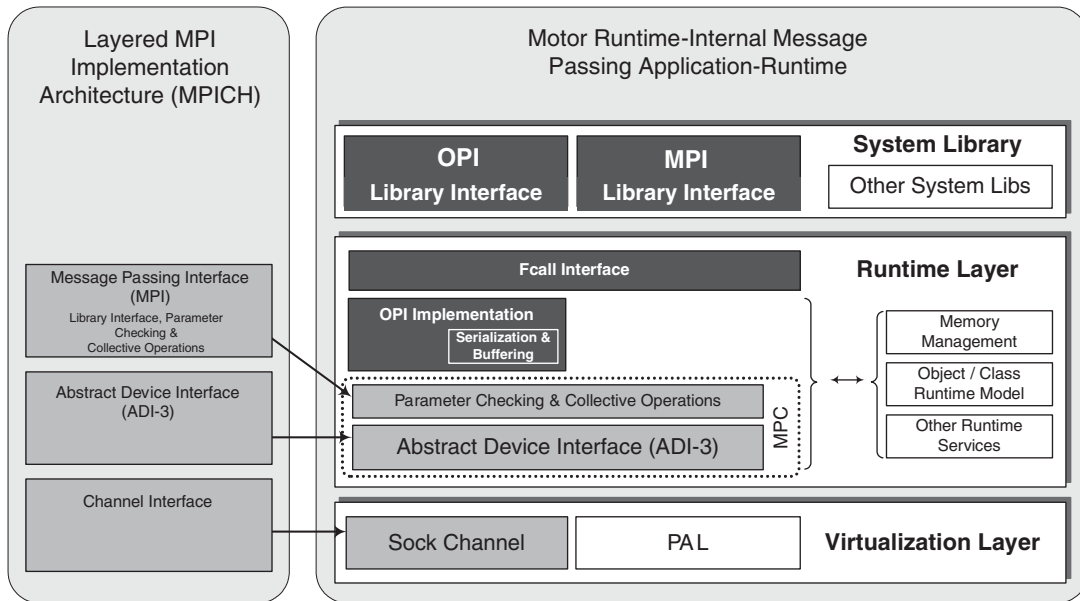


Figure 6. The Motor runtime architecture and its synthesized components. Original SSCLI components are in white. Ported MPICH2 components are in gray. New Motor components are highlighted in black.

- (2) An FCall interface that exposes the MPI and OPI operations to the System library. The implementation of this interface is discussed in Section 8.1.2.
- (3) The OPI Implementation, a library that maps between the object-oriented library interface and the MPC. This includes structured data transport implementation, which provides serialization and buffering.

Both the MPC and the OPI library interact with the runtime object/class model and managed memory (as illustrated in Figure 6). This interaction is described in further detail in Sections 8.2 and 8.3.

The virtualization layer: The SSCLI PAL has been augmented with a low-level message transport interface, the MPICH Channel Interface. Together these form an overall virtualization layer, over which the Motor Runtime is implemented. The current Motor runtime has been implemented on Windows using the MPICH2 Windows sock channel and the Windows PAL.

8.1. Integration of MPICH2 within the SSCLI

MPICH2 was integrated within the SSCLI in a number of steps:

- (1) Porting the MPICH library to the PAL.
- (2) Implementing an FCall MPI interface that exposes the embedded message passing library to the System library.
- (3) Augmenting MPICH2 to poll for garbage collection for object-oriented languages.



Sections 8.2 and 8.3 further expand on this section by presenting the embedded implementation of the MPI and OPI libraries.

8.1.1. Porting MPICH to the SSCLI PAL

To provide portability, the SSCLI runtime is implemented entirely over the SSCLI virtualization layer, the PAL. Therefore, to integrate MPICH2 within the SSCLI runtime, it also had to be implemented over the PAL.

Because the PAL is a subset of the Windows API, it was decided to use the Windows MPICH2 port [19] as the starting point for this implementation. However, 32 Windows API operations and a much larger number of constants and macros used by the MPICH2 library were unsupported by the PAL. These previously unsupported operations were implemented in two ways:

- Some operations and *all* constants and macros were completely implemented *within* the PAL. For example, simple operations (e.g. `htonl`, which converts a 32-bit long number to network byte order) were implemented. To implement a new extended version of the PAL for a particular platform, these operations will *not* need to be implemented again.
- The remaining 27 previously unsupported operations were implemented in the PAL and mapped to their underlying Windows implementation (for example, the Windows `GetComputerName` and `GetModuleHandle` operations).

As illustrated in Figure 6, only the lowest layer of MPICH, the channel layer, was not ported to the PAL, because this layer forms part of the Motor virtualization layer.

8.1.2. Exposing the embedded communication library

This section details the manner in which the Motor message passing library is exposed from within the Motor runtime using the SSCLI FCall mechanism. A message passing call will pass through the following three layers, which are illustrated in Figure 7 using an example OPI Recv:

- (i) The user code calls the managed MPI or the OPI operation. OPI operations are implemented directly within the `System.MP` System library, whereas MPI operations are implemented in an external library written in C, which is described in the next section.

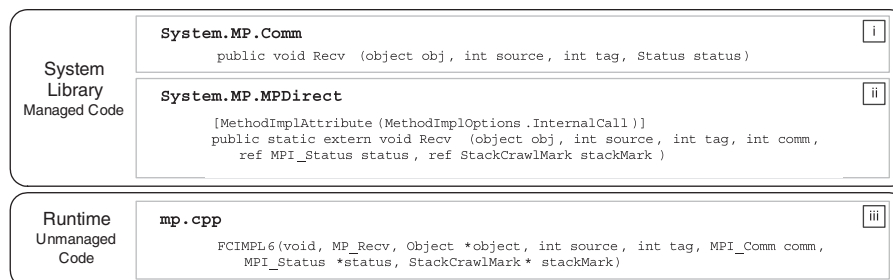


Figure 7. The System library to internal runtime call stack.



- (ii) The System.MP library includes an MPDirect class that contains InternalCall definitions into the underlying runtime. The MPI or the OPI operation calls the matching MPDirect InternalCall function.
- (iii) Each MPDirect InternalCall is matched by an embedded FCall.

In the FCall parameters, types are mapped to their embedded C++ equivalent. In the case of the OPI, the provided object is a pointer to the internal representation of a CLI object type: Object. Note that the MP_Recv is defined within a macro (FCIMPL6), which enforces correct parameter ordering.

8.2. Motor MPI implementation

The Motor MPI interface is written in C and compiled to IL using the lcc.NET [39] compiler.

The lcc.NET compiler is an addition to lcc [40], which is a retargetable compiler, meaning that it can be augmented to support a variety of instruction sets and architectures. In the case of lcc.NET, it has been augmented to support the .Net runtime and IL instruction set. The Microsoft C++ Managed compiler was also considered for our work. However, the C++ Managed compiler produces a hybrid managed and native output, which is not supported by the SSCLI.

The Motor MPI library calls the System.MP interface routines into the runtime. However, lcc.NET is an ANSI C compiler and therefore it does not support any object-oriented syntax. Therefore, it does not support any mechanisms to call into the object-oriented System library. For this reason, the Motor MPI interface defines stub MPI functions. A user writes an application in C, or another procedural language, which can be compiled to IL. Once the user application is compiled to IL, these stubs are discarded and the application is re-linked against the operations exposed by the Motor System library.

8.3. Motor OPI implementation

The OPI library is implemented directly within the System.MP library and the object model is based on the official MPI-2 C++ bindings. Figure 3 shows the most significant classes: Comm and Intracomm. Similar to all SSCLI System libraries, the System.MP library is written in C#.

Using access to the runtime object/class model, the MPI implementation checks the provided object attributes and then performs the operations by interfacing to the underlying MPC. For each transport operation, the MPI implementation checks:

- The size of the supplied sending or receiving object, which is used in the underlying data transport.
- Whether the supplied sending or receiving object references other objects, which is not allowed in order to protect object model integrity^{††}. If it does, the operation throws an exception.

For OPI operations, the ported MPICH2 library has been augmented to allow an SSCLI Object* as the provided buffer. The MPC has been changed to allow sending and receiving data in terms

^{††}The SSCLI object/class model keeps track of whether particular object types contain references to other objects, and therefore performing this check is very efficient. This information is kept by the SSCLI to help the garbage collector efficiently perform the mark phase of collection.



of the internal object representation: `Object`. When required, the MPC resolves the `Object` to the offset location of its instance data, to pass to the underlying transport.

The following sections describe the interaction between the Motor-embedded message passing library and the SSCLI memory management system. These sections demonstrate the closely coupled interaction between the Motor message passing library and runtime services.

8.3.1. Garbage collection polling

To implement the OPI library, it was essential to ensure that no existing runtime services, such as garbage collection, were disrupted. Because the OPI library uses the FCall interface to call into the runtime, it is essential that the message passing library adheres to the FCall rules. As introduced in Section 6, the SSCLI garbage collector requires all threads to regularly poll the garbage collector. Managed code polling is compiled in by the JIT, whereas FCalls must *explicitly* and *regularly* poll for collection.

Therefore, Motor MPI operations poll at least twice during an operation:

- (i) On entry to the FCall, before the operation has commenced.
- (ii) Immediately before exiting the FCall, after the operation has been completed.

In addition, blocking operations must poll *during* the operation:

- (iii) Threads that enter the message passing library never block when waiting for a transport operation to be completed. Rather they enter a *polling-wait* state to wait for the operation to be completed. To achieve this, blocking system calls in the original MPICH2 were replaced with a *polling-wait*, which periodically releases and polls the garbage collector.

8.3.2. Object pinning

To provide automatic pinning and unpinning of send and receive objects, the Motor MPI implementation applies a *pinning policy*, which decides whether automatic pinning is required for each message passing operation. The basis of this policy is that pinning is not necessary for *every* MPI operation. Rather, it is only required under both of the following conditions:

- If garbage collection might occur.
- If the object has the potential to be moved during that collection.

A pinning policy *cannot* reduce the chances that a garbage collection will occur or that objects will require pinning during that collection. After all, pinning will always be necessary to protect objects that *might* be moved during an impending collection. If a garbage collection occurs then objects that are being accessed by the underlying transport must be pinned. However, a pinning policy does mean that objects are not pinned with *each* MPI operation. Rather, an object is only pinned if it is at risk of being moved and there is a chance of an impending collection. Because MPI is implemented alongside the garbage collector, Motor can determine whether the necessary conditions are satisfied, in which case the object requires pinning. This policy relies on access to the object internal memory location and knowledge of the internal mechanics of the SSCLI two-generational garbage collector (Section 6.2). The pinning policy is different for blocking operations and non-blocking operations.



8.3.3. *Pinning policy for blocking operations*

The following describes the Motor pinning policy for blocking operations:

- The sending or receiving object is checked whether it is at risk of being moved during a collection by checking its resident generation.
- If the object exists in the elder generation then it is *not* at risk of being moved during collection. The transport can continue without pinning.
- On the other hand, if the object has been established as a resident of the younger generation, it might be moved during collection.

Depending on the underlying transport, many operations are completed very quickly. If transport is completed before the thread yields to garbage collection, then pinning is unnecessary. Therefore, pinning can be deferred until the transport operation has been commenced and the thread is about to enter a polling-wait. By the commencement of the polling-wait, the transport operation will either be complete or ongoing:

- Fast operations can complete almost immediately and thus will be complete as soon as the operation is about to enter the polling-wait. At this point, MPI operations can be completed without pinning because the transport operation has been completed and no longer accesses the object.
- Ongoing operations need to pin the object, yield to garbage collection and continue waiting for the transport to be completed.

Deferring the decision to pin until the last moment before a possible collection allows quick operations to be completed without pinning.

8.3.4. *Pinning policy for non-blocking operations*

Non-blocking operations return immediately and therefore they do not provide an opportunity to defer pinning. Furthermore, implementing pinning for non-blocking operations is more problematic because it is not clear when the transport operation has been completed and when the object can be unpinned. Existing external implementations of the MPI, such as the Indiana .NET bindings, do not describe a solution to this problem. The MPJ specification does not detail a solution because, similar to MPI, it defines an MPI and does not stipulate the internal workings. The Motor solution to ensuring that objects are unpinned after transport completion is to augment the garbage collector, which allows for pinning operations that are dependant on the status of an MPI operation. This is described in the following pinning policy for non-blocking operations:

- Objects in the elder generation are not pinned, as in the pinning policy for blocking operations.
- Younger generation objects are pinned immediately before transport is commenced. Then the non-blocking operation returns.

The pinning request is stored by the garbage collector until the next collection. Immediately after the mark phase of collection, when the garbage collector checks for unreferenced memory, the collector iterates through a list of pinning requests and checks the status of non-blocking operations. This causes the collector minimal extra work during this phase of collection. Objects



used in *completed* non-blocking operations do not need to be pinned during collection and the object can be released from further pinning. On the other hand, objects used in *incomplete* non-blocking operations are marked as pinned to guarantee that they are not moved during impending collection.

8.3.5. Motor structured data transport implementation

The OPI SDTO perform serialization to a buffer and then interfaces to the underlying MPC. Before sending the serialized buffer, the library performs an MPI send, announcing the size of the serialized buffer^{††}. Using this size, the receiver allocates an adequate buffer, receives the serialized buffer and de-serializes it. Pre-announcing the size of the buffer decreases performance; however, it provides the receiver an opportunity to allocate a buffer of adequate size.

Transportable attribute: The Transportable attribute is implemented as a CLI custom attribute, which provides a declarative mechanism to decorate CLI classes. This allows the serialization library to recursively include references that are to be propagated. CLI custom attributes can be introspected using the CLI reflection library. However, the reflection library is relatively slow because it accesses type metadata. Therefore, we decided to augment the Motor class/object model to keep track of transportable attributes. The SSCLI `FieldDesc` structure provides runtime details about each class field. Motor extends the SSCLI `FieldDesc` (Section 6.3) structure to include a transportable bit. This provides an efficient mechanism to determine which object references are transportable. For each object being serialized, the serialization library iterates through all the `FieldDesc` entries for that type. Entries that are Transportable and not null are also serialized.

Serialization and buffering: The Motor serialization mechanism produces a flat object-tree representation with two parts:

- A type table that consists of the fully qualified class names of the transported objects, providing the receiver with information to instantiate the transported objects.
- Object data, which consist of transported objects, laid out side-by-side, each prefixed with an internal-type reference. Serialized objects are copied to the serialization buffer and the object header is replaced with a reference to the specific type in the type table.

The Motor serialization mechanism is recursive and can be described as follows:

- (1) The object is copied to the serialization buffer and the object header is replaced with a reference to the fully qualified class name in the type table.
- (2) Each field of the object is checked by iterating through the `FieldDesc` entries for that type and checking the transportable bit. Object references are treated differently according to the following rules:
 - A reference to an object that has already been serialized is exchanged to its local serialized reference.
 - A reference to an object that is non-transportable is swapped to null; and
 - A reference to an object that is transportable, but not yet serialized, is recursively serialized.

^{††}This technique is also used by mpiJava.



To support collective communication operations such as scatter, the serialization mechanism can produce a split representation. A single split representation is constructed of many regular representations, each with an individual-type table and each individually deserializable 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 automatically provides buffers for SDTO. These are allocated from static runtime memory, which means that the buffer is not subject to the requirements of pinning. Motor buffers are created on demand and stored in a stack for later use. Each time a buffer is used, it is noted as recently used. The Motor garbage collector has been augmented to check for buffers that have not been recently used and unallocates them.

9. DEMONSTRATION AND PERFORMANCE OF THE MOTOR APPLICATION-RUNTIME

The goal of this section is to measure the performance of the Motor environment. Section 9.1 discusses the execution, results and analysis of a set of micro-benchmarks, and Section 9.2 presents the implementation and performance of a scientific model, the shallow water equations, developed for the Motor environment.

Throughout these tests, all C# applications were compiled to the CLI IL using Visual Studio 7.1.3088. Similarly, C++ applications were compiled to Win32 native using Visual Studio 7.1.3088 with the speed optimization turned on. Standard C applications were compiled to the CLI IL using lcc.NET. Java applications were compiled using the Sun Java JDK 1.5.0.04 javac compiler and executed using the Sun Java JDK 1.5.0.04 runtime. MPICH-2 v1.0.2 was the common MPI implementation used by *all* applications, apart from Motor applications. However, Motor itself is a product of the same version of MPICH-2. The Indiana C# bindings and mpiJava were all originally implemented over MPICH, which does not perform as well as MPICH-2. To provide a fair comparison, we re-implemented them over MPICH-2 v1.0.2. All applications tested, including both the micro-benchmarks and the shallow water applications, used the MPICH2 socket channel for communication—the only implementation of the channel interface currently implemented within Motor.

9.1. Motor micro-benchmarks

To evaluate the performance of the Motor MPI library we compared a number of applications that implement a Ping-Pong algorithm, where two processes take turns to send and receive a block of data. Our intention is to measure the efficiency of the message passing facilities provided by the Motor environment and purposefully avoid measuring the performance of the environment or data processing. In contrast, Section 9.2 is devoted to presenting the measurements and analyses of the performance of the overall Motor environment.

Both the MPI and OPI operations (Section 9.1.1) and the OPI structured data transport (Section 9.1.2) operations were tested.



9.1.1. The MPI/OPI micro-benchmark

To test the performance of the MPI/OPI operations, the following implementations of the Ping-Pong code were tested:

- (1) A native C++ application using MPICH-2 v1.0.2.
- (2) A C application compiled for Motor and using the Motor MPI library.
- (3) A C# application compiled for Motor and using the Motor OPI library.
- (4) A C# application using the Indiana C# .Net bindings, hosted by the SSCLI.
- (5) A C# application using the Indiana C# .Net bindings, hosted by .Net version 1.1.
- (6) A Java application using mpiJava version 1.2.5.

For the purposes of this experiment, a single iteration is defined as the round trip between the two participating processes. Each experiment performed 200 iterations, the last 100 of which were timed. This ensured that the overhead of environment startup, communication startup and JIT compilation was avoided in the performance measurement. A range of buffer sizes were tested and each buffer size was tested five times and the average time in microseconds per iteration was calculated. All tests were performed on a Pentium M 1.7 GHz, with 1 GB RAM, running Windows XP SP2. A single node was used because we were interested only in the performance of the message passing implementation, rather than the underlying transport.

Figures 8 and 9 show the performance results for small and large buffer sizes, respectively. In addition, Figure 10 shows the bandwidth measured by the Ping-Pong micro-benchmark. As expected, the native C++ application shows the best performance. Significantly, the two applications executed by Motor show the next best performance. In comparison to the native C++ implementation, the C (Motor) application performs 3% slower across all buffer sizes and 9% slower across buffer sizes less than 8192 bytes. The C# (Motor) application performs 8% slower across all buffer sizes and 17% slower across buffer sizes less than 8192 bytes.

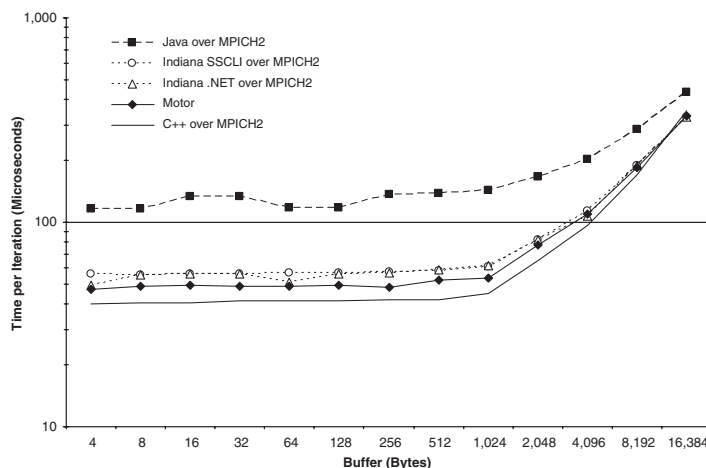


Figure 8. Ping-Pong comparison of MPI operations showing time per iteration (4–16 384 bytes).

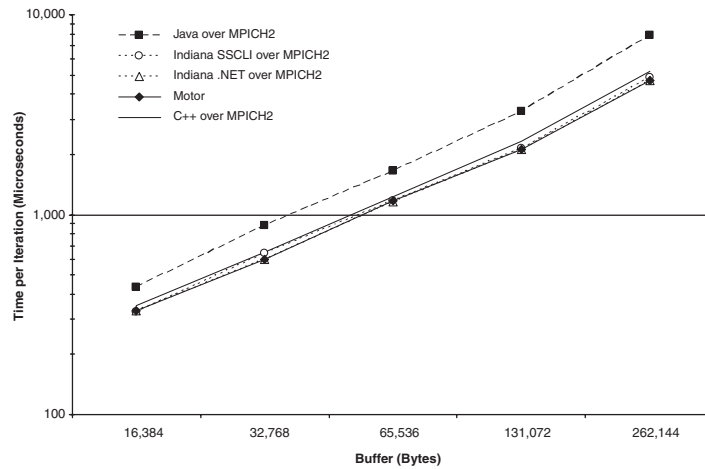


Figure 9. Ping-Pong comparison of MPI operations showing time per iteration (16 384–262 144 bytes).

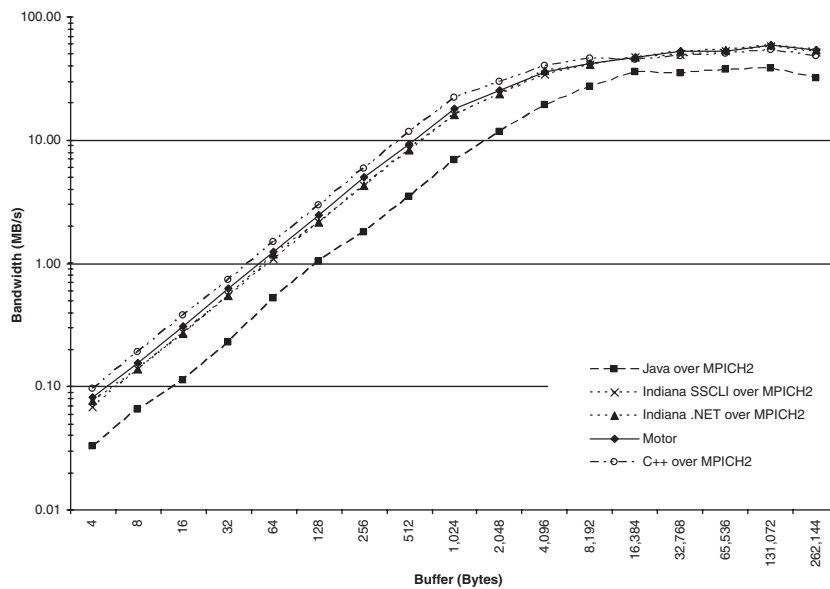


Figure 10. Ping-Pong bandwidth by message size.

In our tests, mpiJava performs the least favorably. We have examined published mpiJava performance results [11]; however, recreating the same results that were published was problematic for a number of reasons. First, the machines used are obviously different. Second, [11] does not specify the exact version of the Java virtual machine employed. Third, [11] indicates that the overhead of



mpiJava varies depending on the underlying MPI implementation. As discussed in Section 9, we have re-implemented mpiJava over MPICH-2 because using the original MPICH does not provide a fair comparison—all other environments we tested use the more highly tuned MPICH-2.

In particular, C# (Motor) performs noticeably better than the C# application using the Indiana bindings hosted by the SSCLI using MPICH-2: 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 that Motor is a synthesis of these same major components: the SSCLI and MPICH-2. We attribute these results to the fact that the Motor MPI library uses the fast InternalCall mechanism, which is a *trusted* internal call mechanism and does not perform data marshaling. Therefore, the InternalCall mechanism imposes a smaller overhead than managed-to-native call mechanisms such as P/Invoke.

9.1.2. The structured data transport micro-benchmark

To test the performance of the Motor OPI SDTO we also used a Ping-Pong code similar to the previous test. However, this experiment measured the influence of sending and receiving structured data, serialized and deserialized using the particular mechanism supported by each system. The following implementations of the Ping-Pong code were compared:

- (1) A C# Motor application, using the Motor SDTO;
- (2) a Java mpiJava application, using the mpiJava MPI.Object data type, which uses the Java serialization mechanism to transport structured data and
- (3) an Indiana C# .Net application, hosted by both the SSCLI and commercial .NET. To provide structured data transport, we used the CLI binary serialization mechanism to produce a buffer to be transported using the standard MPI routines.

Procedural C applications were not included in this test because standard MPI does not define a facility to *automatically* transport structured data. The structured data were in the form of a linked list, with each list element containing a buffer. Figure 11 shows the LinkedList data structure written in C#, which was used in both the Motor and Indiana applications. The Java application transported an identical data structure.

The total data buffer was 4096 integers, evenly distributed over the entire linked list. The number of linked list elements varied from 1 to 4096^{§§}. Each experiment was executed 10 times and the average time in microseconds per iteration was calculated.

Figure 12 plots the results of the structured data transport micro-benchmark. The Motor OPI serialization mechanism performs best throughout the range of linked list lengths (it executes an average of 65% faster than the .Net serialization mechanism). We found the bump in mpiJava results to be a consistent feature and we believe it might suggest that Java version 1.5.0_04 employs different serialization algorithms or data structures to serialize small and large numbers of objects. The 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. It is clear from the results that the serialization mechanism

^{§§}The total number of objects transported was twice the number of linked list elements because the array referenced by each linked list element is itself an individual object.



```
class LinkedList
{
    public int[] array = null;
    public LinkedList next = null;
}
```

Figure 11. The LinkedList class.

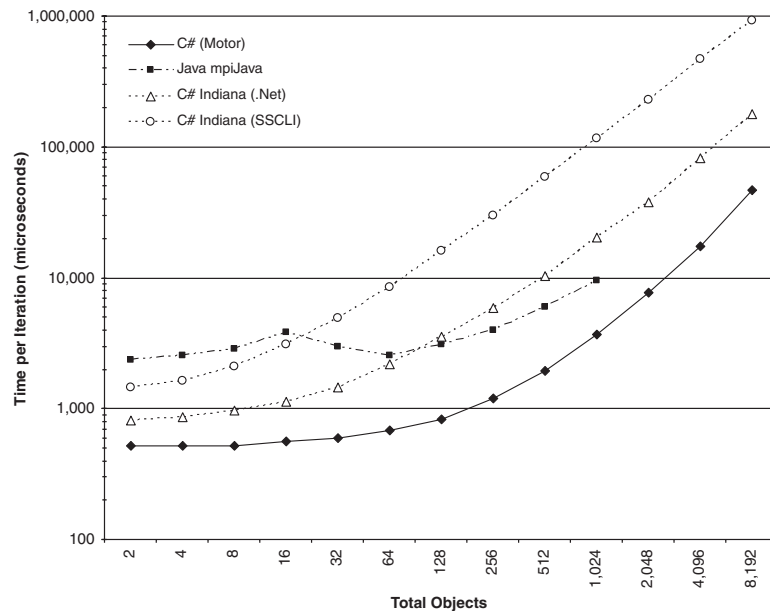


Figure 12. Ping-Pong comparison of structure data transport operations.

employed in the SSCLI is inferior to that provided by .Net. Examining the SSCLI source code, it becomes apparent that the SSCLI uses the reflection library to introspect objects using metadata. We believe it is likely that the .Net serialization mechanism is embedded within the runtime, meaning that it has faster access to object information.

9.2. Shallow water case study

This section demonstrates the application of the Motor runtime and analyzes its suitability for hosting high-performance e-Science applications. To achieve this we have developed a Motor implementation of the shallow water equations [41], which are a computational fluid model commonly used in atmospheric science codes to simulate the horizontal flow of fluid over a number of time steps. Analyzing an implementation of the shallow water model is particularly interesting because it is a relatively simple example of a computationally intensive fluid dynamics model.



```
struct Res {  
    double row[m];  
    int    indx;  
};
```

Figure 13. The Res structure.

The parallel version of the shallow water model we implemented uses a master–slave design and works as follows. The master initializes a set of multi-dimensional arrays that represent a geometric space and its attributes. It sends a complete copy of these data structures to each slave process, which commence processing a particular slice of the data. Processing is performed in a time-step loop, where for each iteration, each slave performs computations on its data slice, and then synchronizes the edges of its slice with its immediate neighbors. After the time-step loop, the slaves copy their slice of the data back to the master process, which recomposes the original buffer and performs final processing.

Our C and C# implementations of the parallel shallow water model is based on an earlier implementation, which is written in C [42]. This version uses regular MPI.Send and MPI.Recv operations for the majority of communication between slaves. To communicate results back to the server, it defines an MPI custom data type to send and receive a simple data structure of type Res (Figure 13), which represents a data block and its position within the overall data set.

The advantage of parallelizing the shallow water model has been documented [42] and this paper does not investigate or demonstrate speedup—rather we compare different implementations of the parallel shallow water model in order to understand Motor performance.

9.2.1. Implementing the shallow water model for the Motor runtime

We implemented two versions of the shallow water model. The first is a C implementation that is essentially identical to the earlier C version and requires no porting.

The second is a C# implementation. Porting the C shallow water implementation to C# using the Motor application-runtime involved a number of steps:

- (1) Porting the overall program structure to a C# object-oriented environment. This involved making changes to implement an overall class structure. Simple data types were mapped to their C# equivalent. For example, `char*` was mapped to the C# `string` class.
- (2) Porting the shallow water computational processing routines. C# uses C syntax and therefore, changes to the processing code were minimal. For example, C# uses a different syntax to access multidimensional array elements (`[index1, index2]` versus the original C `[index1] [index2]`). Mathematical library calls were mapped to their CLI System library equivalent. Other library calls, such as console output, were similarly mapped.
- (3) Porting MPI functionality to the Motor supported MPI library. Regular MPI primitives, such as MPI.Send and MPI.Recv, were mapped to their Motor equivalent, MP.Send and MP.Recv. This involved removing data type and count parameters because they have been removed from the Motor MPI interface. In the case of array subset transfers, the offset and length parameters were used.



- (4) The original C code uses a custom data type to send and receive the `Res` data structure. In the Motor shallow water implementation we used the SDTO to automatically transport `Res` objects.

In total, the C# Motor shallow water model consists of around 1100 lines of code.

9.2.2. Case study execution

In order to analyze the execution of the Motor shallow water model we ran and compared three versions of the shallow water model:

- (1) A Windows native port of the native shallow water application. This implementation uses regular MPI buffer-to-buffer send and receive operations for communication between slaves, and an MPI custom data type to transport the `Res` data structure back to the master.
- (2) A C implementation, targeting the Motor application-runtime. This implementation uses the same communication primitives as the native port, but interfaces to the Motor MPI library.
- (3) A C# implementation, targeting the Motor application-runtime. As introduced in the preceding section, this implementation uses regular object-to-object MPI send and receive operations for communication between slaves. It uses SDTO to automatically transport the `Res` data type back to the master.
- (4) A CLI implementation of the shallow water equations, which is written in C# and is identical to the Motor shallow water implementation, apart from the use of the Indiana C# bindings for communication. We executed this version of shallow water model using both the commercial .Net and the SSCLI. This implementation uses the regular MPI send and receive operations for communication between slaves. It uses the standard CLI serialization mechanism, as originally described in Section 9.1.2, to automatically transport the `Res` data type.

During the experiment each application was executed for 100 time steps. The shallow water matrix sizes were varied, from 8×8 double-length floating point numbers to 512×512 double-length floating point numbers. Each application was executed 20 times for each matrix size and the average result was calculated. All tests described in this section were performed on a computer with dual central processing unit Pentium III 700 MHz processors, with 256 MB of RAM, running Windows 2000 SP2. In each case, the application was executed with three processes, the master and two slaves.

In order to profile the execution of these applications in detail, the slave processes were developed to track processing and communication times separately.

9.2.3. Shallow water results

Figure 14 shows the overall shallow water master execution times. On average, the native implementation performs the best, followed by .Net, then Motor and then the SSCLI. The C (Motor) implementation was only executed up to 128×128 double-length floating point numbers because greater buffer sizes caused a stack overflow exception^{¶¶}.

^{¶¶}This is due to the fact that the lcc.NET compiler allocates memory on the stack, and very large buffer sizes exceed the maximum stack size possible on the target machine. Static allocation of large buffers also failed.

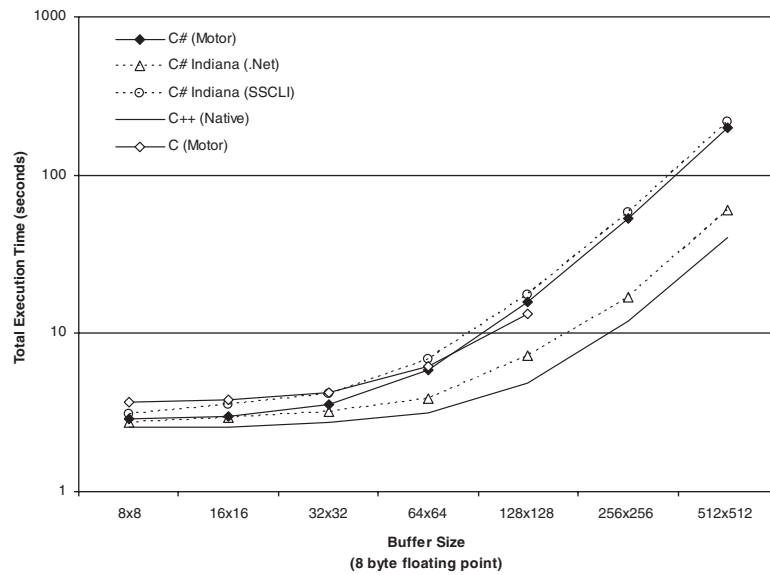


Figure 14. ShallowWater total execution times.

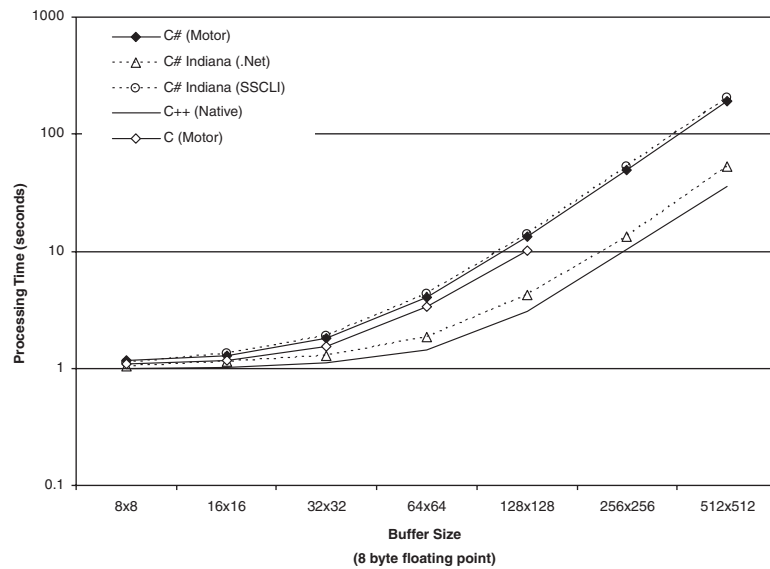


Figure 15. ShallowWater slave processing times.

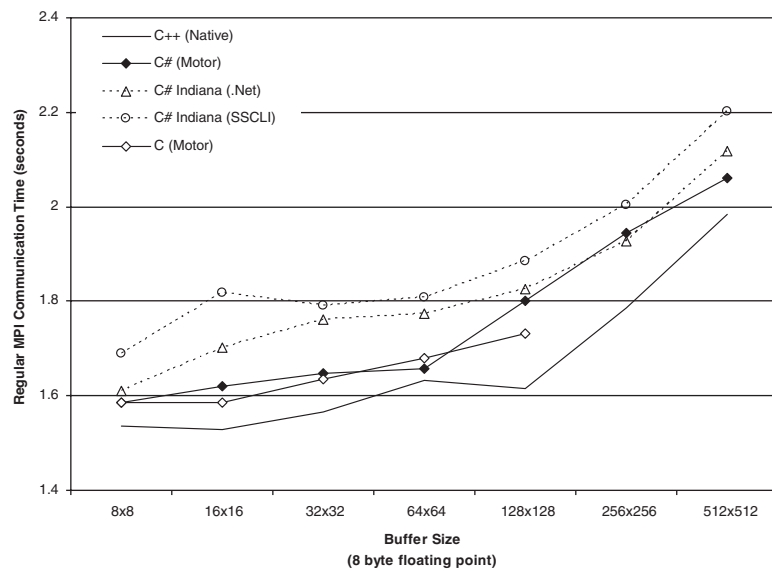


Figure 16. ShallowWater slave regular MPI communication times.

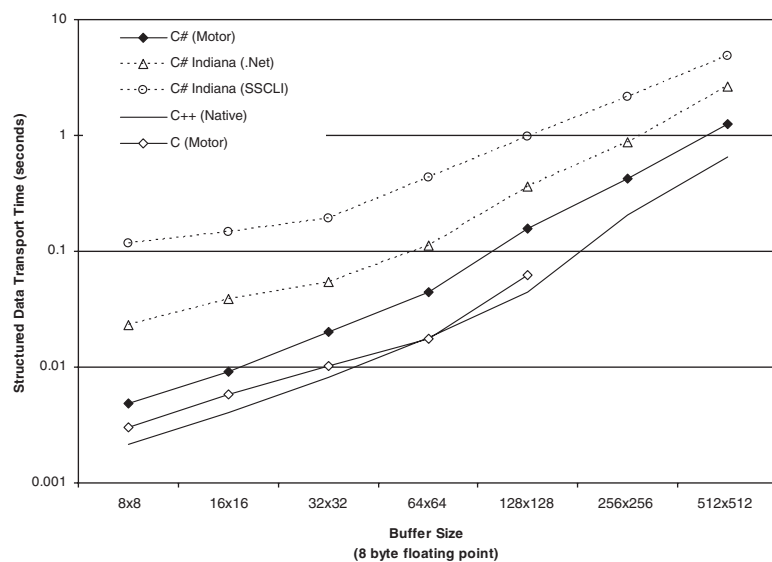


Figure 17. ShallowWater slave-structured data transport times.

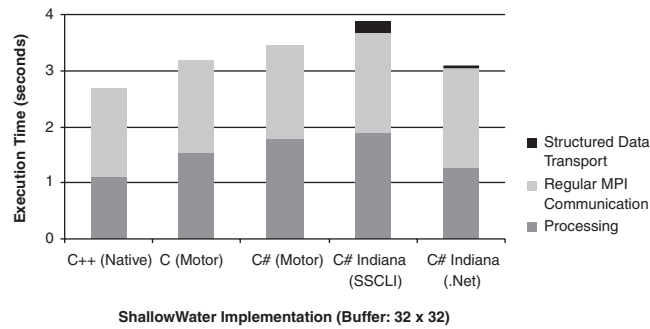


Figure 18. ShallowWater execution profile with a 32×32 buffer size.

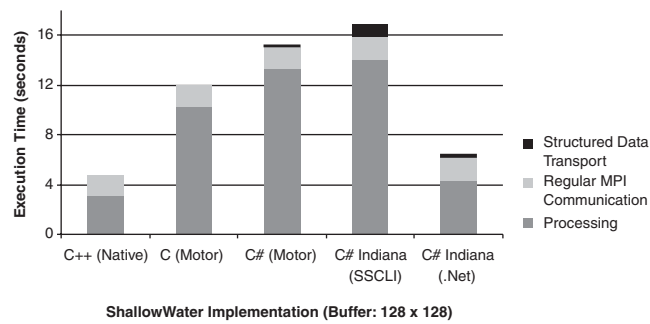


Figure 19. ShallowWater execution profile with 128×128 buffer size.

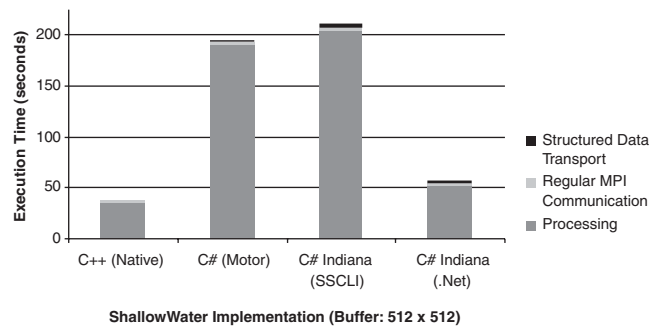


Figure 20. ShallowWater execution profile with 512×512 buffer size.



Figures 15–17 decompose the slave execution times into three parts:

- (1) Processing time (Figure 15), which includes all executions apart from communication;
- (2) regular MPI communication time (Figure 16), which includes regular MPI routines such as MPI_Send and MPI_Recv, or their environment-specific equivalent; and
- (3) structured data transport time (Figure 17), which encompasses the time required to transport the Res data structure, using the environment-specific mechanism.

In addition, Figures 18, 19 and 20 illustrate the overall communication and processing profile of each shallow water implementation, at buffer sizes 32×32 , 128×128 and 512×512 , respectively.

Figures 18–20 show that at large buffer sizes, the shallow water applications profiled are bound by processing. Furthermore, Figures 15 and 20 illustrate that processing time is the cause of the poorer performance demonstrated by both Motor and the SSCLI. Because Motor is based on the SSCLI, it uses the same JIT compiler and therefore, it is logical that they demonstrate similar processing performance.

On average across all measured buffer sizes, our Motor prototype performs significantly worse than the native implementation (the C implementation performs 83% slower, whereas the C# implementation performs 159% slower). This is because the SSCLI lacks a competitive JIT compiler [16]. Our processing performance results are consistent with published benchmarking results of various implementations of the CLI. These benchmarks illustrated a large difference in the performance of various JIT compilers and found that the foremost factor in runtime processing performance is the optimization of the JIT-generated code [16]. This benchmarking study also found that the SSCLI emitted poorer JIT-generated code in comparison with .Net and Mono. In comparison, .Net, which *does* support an optimized JIT compiler, achieves significantly better results—on average it performs only 28% worse than the native implementation. We believe that the .Net performance is a reasonable tradeoff considering the range of services that .Net provides, including platform independence, language neutrality, memory management and an object model. Although the current Motor processing time is not as good as that of .Net, we are confident that, considering that the SSCLI is a downgraded version of .Net, a more competitive SSCLI JIT implementation could achieve equal results. It is clear that the SSCLI JIT compiler is substandard [16] and optimizations [43,44] could be applied.

With regard to regular MPI communication, Figure 16 shows that both applications hosted by Motor perform favorably against both .Net and the SSCLI. On average, C (Motor) performs 21.9% faster than C# (SSCLI) and 9.7% faster than C# (.Net). On average, C# (Motor) performs 6.9% faster than C# (SSCLI) and 3.7% faster than C# (.Net). On average, C (Motor) performs only 4.5% slower than the native implementation. These MPI communication times are consistent with our Ping-Pong micro-benchmarks.

Similarly, with regard to structured data transport of the Res type, Figure 17 demonstrates that Motor performs significantly better than .Net and the SSCLI, whereas the native implementation performs best. The native and C (Motor) shallow water implementations transport the Res type most efficiently because they both use a custom data type, which does not involve any serialization; C (Motor) performs 30% slower than the native implementation.

The Motor OPI SDTO are significantly faster than the regular serialization mechanisms offered by SSCLI and .Net. On average C# (Motor) performs nearly 8 times faster than the C# (SSCLI) and 2.6 times faster than C# (.Net).



In C/C++, using an MPI custom data type is possible because the C/C++ definition of the `Res` type is a single discreet structure and therefore the `Res` type can be transported with a single buffer-to-buffer transfer. However, in C# and Java, the `Res` object contains a reference to separate the array object and therefore a single object-to-object transfer is impossible. This demonstrates the fact that in a managed memory environment, the developer has less control over memory and it is difficult to organize memory to optimize communication. Therefore, an efficient serialization mechanism is essential to transport structured data.

From these results we can conclude that Motor communication performs favorably but its JIT compiler is poor. The performance of the .Net runtime demonstrates that an optimized JIT compiler is able to achieve reasonable results in comparison with a native C++ implementation. To make Motor a competitive environment would require implementing a more competitive JIT compiler or tuning the existing SSCLI compiler.

10. CONCLUSION

Motor is a unique environment for high-performance application development because it supports high-performance communication as part of its overall architecture. Motor places message passing alongside other CLI runtime libraries and services. This provides a guaranteed environment for HPC. It also gives the message passing library access to the entire internal runtime allowing us to implement an efficient object passing interface while guaranteeing object model integrity. Notably, because Motor is based on the CLI, it is language neutral, and it supports message passing libraries for both procedural and object-oriented languages.

We have demonstrated Motor performance using a micro-benchmark measuring regular buffer-to-buffer MPI and our own OPI object-to-object and structured data operations. In addition, we have implemented a scientific model for the Motor runtime and analyzed its performance. This has demonstrated that an embedded message passing architecture is feasible and offers good performance, in comparison with implementations that place message passing outside the runtime environment. We have also demonstrated the importance of a highly optimized JIT compiler, which is not supported by the SSCLI or Motor. However, examining .Net processing performance demonstrates that an optimized JIT compiler is able to provide good processing performance. A post-prototype implementation of Motor would need to support a more optimized JIT compiler.

The Motor architecture introduces and demonstrates the feasibility of developing an overall runtime for HPC. Although our starting point supports message passing, other services, such as shared memory, process management and mathematics libraries, are also important to parallel developers. For example, we believe that the detailed design of an embedded shared memory architecture would also be of significant value. This is particularly true considering the commoditization of multi-core microprocessors. Environments such as OpenMP are becoming more important for HPC developers—and all developers in general—who wish to take advantage of the thread-level parallelism offered by a multi-core architecture.

REFERENCES

1. *Standard ECMA-335: Common Language Infrastructure* (3rd edn). June 2005. <http://www.ecma-international.org/publications/standards/Ecma-335.htm> [11 January 2006].



2. Foster I, Kesselman C. *The Grid: Blueprint for a New Computing Infrastructure* (1st edn). Morgan Kaufmann: Los Altos, CA, 1998.
3. Goscinski W, Abramson D. Application deployment over heterogeneous Grids using distributed ant. *First International Conference on e-Science and Grid Technologies (e-Science 2005)*, Melbourne, Australia, 2005; 361–368.
4. Goscinski W, Abramson D. Motor: A virtual machine for high performance computing. *The Fifteenth IEEE International Symposium on High-Performance Distributed Computing (HPDC2006)*, Paris, France, 2006.
5. Goscinski W, Abramson D. An infrastructure for the deployment of e-Science applications. In *High Performance Computing and Grids in Action*, Grandinetti L (ed.). *International Symposium on High Performance Distributed Computing 2007 (HPDC2007)*, Advances in Parallel Computing, vol. 16. IOS Press: Amsterdam, 2008; 131–149.
6. Abramson D. Applications development for the computational Grid (*Lecture Notes in Computer Science*, vol. 3841). Springer: Berlin, 2006.
7. Judd G, Clement M, Snell Q, Getov V. Design issues for efficient implementation of MPI in Java. *Proceedings of the ACM 1999 Java Grande Conference*, San Francisco, U.S.A., 1999.
8. Lee H-K, Carpenter B, Fox G, Lim SB. HPJava: Programming support for high-performance Grid-enabled applications. *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, NV, U.S.A., 2003.
9. Thurman D. JavaPVM. <http://www.isye.gatech.edu/chmsr/JavaPVM/> [12 January 2000].
10. Morin S, Koren I, Krishna C. JMPI: Implementing the message passing standard in Java. *International Parallel and Distributed Processing Symposium: IPDPS 2002 Workshops*, Fort Lauderdale, U.S.A., 2002.
11. Baker M, Carpenter B, Fox G, Ko SH, Lim S. mpiJava: An object-oriented Java interface to MPI. *International Workshop on Java for Parallel and Distributed Computing*, San Juan, Puerto Rico, 1999.
12. Carpenter B, Getov V, Judd G, Skjellum A, Fox G. MPJ: MPI-like message passing for Java. *Concurrency and Computation: Practice and Experience* 2000; **12**(11):1019–1038.
13. Bull J, Westhead M, Kambites M, Obdrzalek J. Towards OpenMP for Java. *Proceedings of Second European Workshop on OpenMP*, Edinburgh, 2000.
14. Dincer K. Ubiquitous message passing interface implementation in Java: jmpi. *Proceedings of the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, San Juan, Puerto Rico, 1999.
15. Mintchev S. Writing programs in JavaMPI. *Technical Report MAN-CSPE-02*, 1997.
16. Vogels W. HPC.NET—are CLI-based virtual machines suitable for high performance computing? *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, Phoenix, U.S.A., 2003.
17. Willcock J, Lumsdaine A, Robison A. Using MPI with C# and the common language infrastructure. *Concurrency and Computation: Practice and Experience* 2005; **17**(7–8):895–917.
18. The Java Grande Forum, 2003. <http://www.javagrande.org/> [11 April 2007].
19. MPICH2, 2006. <http://www-unix.mcs.anl.gov/mpi/mpich2/> [12 September 2006].
20. Shared Source Common Language Infrastructure 1.0 Release, 2003. <http://msdn.microsoft.com/net/sscli/> [13 January 2006].
21. Stutz D, Neward T, Shilling G. *Shared Source CLI Essentials* (1st edn). O'Reilly: Sebastopol, CA, Farnham, 2003.
22. Adabala S, Chadha V, Chawla P, Figueiredo RJO, Fortes JAB, Krsul I, Matsunaga AM, Tsugawa MO, Zhang J, Zhao M, Zhu L, Zhu X. From virtualized resources to virtual computing grids: The In-VIGO system. *Future Generation Computer Systems* 2005; **21**(6):896–909.
23. Zhao X, Borders K, Prakash A. SVGrid: A secure virtual environment for untrusted Grid applications. *Proceedings of the 3rd International Workshop on Middleware for Grid Computing*, Grenoble, France, 2005.
24. Krsul I, Ganguly A, Zhang J, Fortes JAB, Figueiredo RJ. VMPlants: Providing and managing virtual machine execution environments for Grid computing. *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, Pittsburgh, 2004.
25. Keahey K, Foster I, Freeman T, Zhang X, Galron D. Virtual workspaces in the Grid. *Eleventh International Euro-Par Conference*, Lisbon, Portugal, 2005.
26. Zhang X, Keahey K, Foster I, Freeman T. Virtual cluster workspaces for Grid applications. *ANL Technical Report, ANL/MCS-PI246-0405*, 2005.
27. Foster I, Freeman T, Keahey K, Scheftner D, Sotomayor B, Zhang X. Virtual clusters for Grid communities. *IEEE International Symposium on Cluster Computing and the Grid*, Singapore, 2006.
28. Figueiredo R, Dinda P, Fortes J. A case for Grid computing on virtual machines. *Proceedings of the 23rd International Conference on Distributed Computing Systems*, Washington, U.S.A., 2003.
29. Ganguly A, Agrawal A, Boykin P, Figueiredo R. WOW: Self-organizing wide area overlay networks of virtual workstations. *Proceedings of the Fifteenth International Symposium on High Performance Distributed Computing (HPDC-15)*, Paris, France, 2006.
30. Castain R, Woodall T, Daniel D, Squyres J, Barrett B, Fagg G. The open run-time environment (OpenRTE): A transparent multi-cluster environment for high-performance computing. *Euro PVM/MPI 2005*, Sorrento, Italy, 2005.



31. Migliardi M, Sunderam V, Geist A, Dongarra J. Dynamic reconfiguration and virtual machine management in the Harness metacomputing system (*Lecture Notes in Computer Science*, vol. 1505). Springer: Berlin, 1998.
32. Baker M, Carpenter B. Thoughts on the structure of an MPJ reference implementation. *Technical Report CRPC-TR99803*, 1999.
33. Java Native Interface, 2007. <http://java.sun.com/j2se/1.4.2/docs/guide/jni/>.
34. Clark J. Calling Win32 DLLs in C# with P/Invoke. *MSDN Magazine*, 2003. <http://msdn.microsoft.com/msdnmag/issues/03/07/NET/>.
35. Parrot Virtual Machine, 2006. <http://www.parrotcode.org/>.
36. Jones R, Lins R. *Garbage Collection, Algorithms for Automatic Dynamic Memory Management*. Wiley: New York, 1997.
37. Gropp W, Lusk E. Creating a new MPICH device using the channel interface. *Technical Report ANL/MCS-TM-213*, 1995.
38. Gropp W, Lusk E, Doss N, Skjellum A. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing* 1996; **22**:789–828. Available at: <http://citeseer.ist.psu.edu/gropp96highperformance.html> [20 September 2006].
39. Hanson DR. lcc.NET: Targeting the .NET common intermediate language from standard C. *Software—Practice and Experience* 2004; **34**:265–286.
40. Fraser CW, Hanson DR. A retargetable compiler for ANSI C. *SIGPLAN Notices*, 1991.
41. Vreugdenhil C. *Numerical Methods for Shallow-Water Flow*. Kluwer Academic Publishers: Dordrecht, 1994.
42. Abramson D, Dix M, Whiting P. A study of the shallow water equations on various parallel architectures. *Fourteenth Australian Computer Science Conference*, Sydney, Australia, 1991.
43. Ishizaki K, Kawahito M, Yasue T, Takeuchi M, Ogasawara T, Suganuma T, Onodera T, Komatsu H, Nakatani T. Design, implementation, and evaluation of optimizations in a just-in-time compiler. *Proceedings of the ACM SIGPLAN Java Grande Conference*, 1999.
44. Suganuma T, Yasue T, Kawahito M, Komatsu H, Nakatani T. Design and evaluation of dynamic optimizations for a Java just-in-time compiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 2005; **27**(4):732–785.