ParC#: Parallel Computing with C# in .Net*

João Fernando Ferreira and João Luís Sobral

Departamento de Informática - Universidade do Minho,
4710 - 057 BRAGA – Portugal
{joaoferreira, jls}@di.uminho.pt

Abstract. This paper describes experiments with the development of a parallel computing platform on top of a compatible C# implementation: the Mono project. This implementation has the advantage of running on both Windows and UNIX platforms and has reached a stable state. This paper presents performance results obtained and compares these results with implementations in Java/RMI. The results show that the Mono network performance, critical for parallel applications, has greatly improved in recent releases, that it is superior to the Java RMI and is close to the performance of the new Java nio package. The Mono virtual machine is not yet so highly tuned as the Sun JVM and Thread scheduling needs to be improved. Overall, this platform is a new alternative to explore in the future for parallel computing.

1 Introduction

Traditional parallel computing is based on languages such as C/C++ and Fortran, since these languages provide a very good performance. Message passing libraries such as MPI and PVM are also very popular, since there are bindings for several languages and implementations for high performance networks, like Myrinet and Infiniband. These message passing libraries support the CSP model, where parallel applications are decomposed into a set of processes that communicate through message passing. It has been recognised that this programming model is not the most appropriated for object-oriented applications [1], since the natural mechanism for communication on these applications is the method invocation. Several extensions to C++ have been proposed [2] that use the object as the base unit of parallelism (instead of process) and objects communicate through remote method invocations (instead of message passing).

The Java programming language has gained an increasing acceptation in the last decade. It is a much cleaner object oriented language than C++, since it removes the burden of pointer management and memory allocation. It also has an increased portability, since it is based on a virtual machine and an application can run anywhere that has a virtual machine implementation. This approach also resolves the communication problem among heterogeneous machines, since the communication is always between virtual machines. These are also important advantages for the increasing popular GRID computing field. The Java language also includes support for threads, remote method invocation (RMI) and object serialisation. Object serialisation allows object copies to move between virtual machines, even when objects are not allocated on a continuous memory range or when they are composed

V. Malyshkin (Ed.): PaCT 2005, LNCS 3606, pp. 239 – 248, 2005.

by several objects. The serialisation mechanism can automatically copy the object to a continuous stream that can be sent to another virtual machine, which can reconstruct a copy of the original object structure on the remote machine.

Several works are based on the Java platform for parallel computing: performance improvements to the original RMI implementation [3], thread distribution among virtual machines [4][5], MPI bindings [6] and implementation of higher level programming paradigms [7], just to name a few.

Microsoft has proposed the .Net platform to compete against the Java success. In particular, the C# language closely resembles to Java: it is also based on a virtual machine; it relieves the programmer from memory allocation and pointer management issues; it includes thread support in the language specification and supports RMI. However, the C# language includes some improvements; namely, it provides support for asynchronous method invocation and several ways to publish remote objects, which will be discussed in more detail in the next section. The main Microsoft .Net platform drawback is the lack of support in other platforms besides Microsoft Windows. This may explain the limited number of research projects related to .Net platform on clusters, since clusters mainly run Linux operating systems or other UNIX variants.

The Mono project is a free .Net platform implementation that runs on several operating systems, including Linux machines. This paper describes the experience acquired when porting a parallel object oriented system to this platform. The rest of this paper is organised as follows. Section 2 presents a more detailed comparison of the supported concurrency and distribution mechanisms of MPI, Java and C#. Section 3 presents the proposed platform, including the programming model and its implementation on the Mono platform. Section 4 presents performance results. Section 5 closes the paper with suggestions for future work.

2 C# Remoting Versus MPI and JAVA RMI

The Message Passing Interface (MPI) is a collection of routines for inter process communication. The mechanisms for communication are based on explicit message send and receive, where each process is identified by its rank in the communication group. MPI has a large set of primitives to send and receive messages, namely, blocking and unblocking sends and receives; broadcasts and reductions. MPI requires explicit packing and unpacking of messages (i.e., a data structure residing in a non-continuous memory must be packed into a continuous memory area before being sent and must be unpacked in the receiver). A thread library such as Pthreads can be used to create multithreaded applications. However, most MPI implementations are not thread safe, increasing the application complexity, when several threads in the same process need to access to the MPI library.

The Java language specification includes support for multithreaded applications through the Thread class and the Runnable interface. The thread method start initiates the execution of a new thread that executes the run method of an object implementing the Runnable interface. Synchronised methods prevent two threads from simultaneously executing code in the same object, avoiding data races. The Java RMI provides remote method invocations among Java virtual machines. Using RMI involves several steps, which considerably increase the burden to use it:

1. Server classes must implement an interface, which must extend the *Remote* interface, and its methods must throw a *RemoteException*.
2. Each server object must be manually instantiated (by introducing a *main* method on the server class), exported to be remotely available and registered in a name server to provide remote references to it;
3. Client classes must contact a name server to obtain a local reference to a remote object;
4. Each remote call must include a *try { … } catch* statement to deal with *RemoteExcetions*;
5. For each server class it is required to run the *rmic* utility to generate proxies and ties that are, respectively, used by the client and server class in a transparent way.

Fig. 1 illustrates these required transforms for a simple remote class that performs a division of two numbers.

```
public class DServer {
  public double divide(double d1, double d2) {
    return d1 / d2;
  }

public class DivideClient {
  public static void main(String args[]) {
    DServer ds = new DServer();
    double d1 = Double.valueOf(args[0]).doubleValue();
    double d2 = Double.valueOf(args[1]).doubleValue();
    double result = ds.divide(d1, d2);
  }
}
```

```
  public interface IDServer extends Remote {
    double divide(double d1, double d2) throws RemoteException;
  }

  public class DServer extends UnicastRemoteObject implements IDServer {
    public double divide(double d1, double d2) throws RemoteException {
      return d1 / d2;
    }                                                                    ❶
    public static void main(String args[]) {
      try {
        DServer dsi = new DServer();
        Naming.rebind("rmi://host:1050/DivideServer",dsi);
      } catch(Exception e) { e.printStackTrace(); }                      ❷
    }
  }

  public class DivideClient {
    public static void main(String args[]) {
      try {
        IDServer ds; // Obtains a reference to the remote object
        ds = (IDServer) Naming.lookup("rmi://host:1050/DivideServer");   ❸
        double d1 = Double.valueOf(args[0]).doubleValue();
        double d2 = Double.valueOf(args[1]).doubleValue();
        double result = ds.divide(d1, d2);
      } catch(RemoteException ex) { ex.printStackTrace(); }              ❹
  }
```

Fig. 1. Conversion of a Java class to a remote class

With RMI the only binding between the client and the server is the registered name of the server object (*host:1050/DivideServer* in the figure), which truly provides location transparency. All objects passed among remote classes should implement the

interface *serializable*, providing a way to automatically send object copies among virtual machines.

The .Net platform implements threads in a way similar to Java, but the use of remote method invocations has become simpler and several improvements have been added. One important difference is the various alternatives to publish remote objects (step 2 from the previous list). In addition to publish objects explicitly instantiated, it is possible to register an object factory that instantiates objects at request. This object factory has two alternatives to instantiate objects:

1. singleton - all remote calls are executed by the same object instance;
2. singlecall – each remote call may be executed by a different instance (i.e., object state is not maintained between remote calls).

Fig. 2 presents the code in Fig. 1 converted to C#.

```
public interface IDServer {
  double divide(double d1, double d2);
}

public class DServer : MarshalByRefObject, IDServer {
  public double divide(double d1, double d2) {
    return d1 / d2;
  }
  public static int Main (string [] args) {
    TcpChannel cn = new TcpChannel (1050);
    ChannelServices.RegisterChannel(cn);
    RemotingConfiguration.RegisterWellKnownServiceType( typeof(DServer),
      "DivideServer", WellKnownObjectMode.Singleton);
  }
}

public class DivideClient {
  public static int Main (string [] args) {
    TcpChannel cn = new TcpChannel();
    ChannelServices.RegisterChannel(cn);
    IDServer ds = (IDServer) Activator.GetObject( typeof(DivideServer),
      "tcp://localhost:1050/DivideServer");
    double d1 = Convert.ToDouble(args[0]);
    double d2 = Convert.ToDouble(args[1]);
    double result = ds.divide(d1, d2);
  }
}
```

Fig. 2. Remote class in C#

There are two important differences: no *RemoteException* needs to be thrown/caught and the server code only publishes the object factory (*RemotingConfiguration* line), not an object instance. Conversely to the Java version it is not required to generate proxy and ties, since they are automatically generated.

C# Remoting also includes support for asynchronous method invocation through *delegates*. A delegate can perform a method call in background and provides a mechanism to get the remote method return value, if required. In Java, a similar functionality must be explicitly programmed using threads.

3 The Platform

The ParC# is a SCOOPP (Scalable Object Oriented Parallel Programming) implementation [8], which has been previously implemented in C++/MPI

(implementation called ParC++). The C# implementation is much simpler, since the C++ version must contain code to explicitly pack/unpack method tags and parameters into MPI messages, required to implement synchronous or asynchronous remote method invocations. This section shortly reviews the programming model and details the main differences between these two implementations.

3.1 Programming Paradigm

SCOOPP is based on an object oriented programming paradigm supporting active and passive objects. Active objects are called parallel objects and they specify explicit parallelism, having its own thread of control. These objects model parallel tasks and are automatically distributed among processing nodes. They communicate through either asynchronous (when no value is returned) or synchronous method calls (when a value is returned).

References to parallel objects may be copied or sent as a method argument, which may lead to cycles in a dependence graph. The application's dependence graph becomes a DAG when this feature is not used.

Passive objects are supported to make easier the reuse of existing code. These objects are placed in the context of the parallel object that created them, and only copies of them are allowed to move between parallel objects.

SCOOPP removes parallelism overheads at run-time by transforming (packing) parallel objects in passive ones and by aggregating method calls [8]. These run-time optimisations are implemented through:

- method call aggregation: (delay and) combine a series of asynchronous method calls into a single aggregate call message; this reduces message overheads and per-message latency;
- object agglomeration: when a new object is created, create it locally so that its subsequent (asynchronous parallel) method invocations are actually executed synchronously and serially.

3.2 Implementation

The ParC++ implementation supports some extensions to C++. It includes a pre-processor, several C++ support classes and a run-time system. The pre-processor analyses the application - retrieving information about the declared parallel objects - and generates code for remote object creation and remote method invocation.

The ParC++ run-time system (RTS) is based on three object classes: proxy objects (PO), implementation objects (IO) and server objects (SO).

A PO represents a local or a remote parallel object and has the same interface as the object it represents. It transparently replaces remote parallel objects and forwards all method invocations to the remote parallel object implementation (IO/SO in Fig. 3). A PO maintains the remote identification of its IO and SO. On inter-grains method calls the PO forwards the call to a remote SO, which activates the corresponding method on the IO (calls a in Fig.3). On intra-grain calls, the PO directly calls the corresponding method on the local IO (call b in Fig.3).

Converting the ParC++ prototype to C# removed a large amount of code from PO objects, since most of its functionality is already implemented by C# remoting.

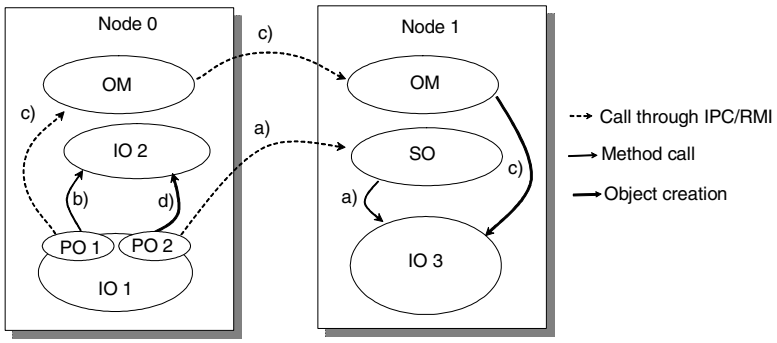

Fig. 3. Inter-grains a) and intra-grain b) method calls; RTS c) and d) direct object creation

However, PO objects are still required, since they perform much of the grain-size adaptation. The main simplification of PO objects arises from the elimination of code required to pack a method tag and method arguments into a MPI message. This code is directly replaced by a direct call to the corresponding method in the IO, using C# remoting. This change also allowed PO objects to transparently use remote objects or local objects (i.e., those objects created directly, when performing object agglomeration). Implementing asynchronous method invocation was simpler, since it only required the use of delegates. During the preprocessing phase, the original parallel object classes are replaced by generated PO classes.

Fig. 4 presents a simple source code and the PO code generated by the preprocessor. This code calls the *process* method asynchronously, using a delegate.

```
public class PrimeServer : PrimeFilter {
  public void process(int[] num) {
  ...
  }
}
```

```
public class PrimeServer : PrimeFilter {  // PO object
  public delegate void RemoteAsyncDelegate (int[] num );   // delegate decl.
  PrimeServerImpl obj;                              // reference to IO object
  ...
  public void process(int[] num) {   // asynchronous call using delegates
    RemoteAsyncDelegate RemoteDel= new RemoteAsyncDelegate(obj.process);
    IAsyncResult RemAr=RemoteDel.BeginInvoke(num,null,null);
  }
}
```

Fig. 4. PO object using delegates

The ParC++ RTS provides run-time grain-size adaptation and load balancing through cooperation among object managers (OM) and POs. The application entry code creates one instance of the OM on each processing node. The OM controls the grain-size adaptation by instructing PO objects to perform method call aggregation and/or object agglomeration.

When a parallel object is created in the original code, the generated code creates a PO object instead. The first task of the newly created PO is to request the creation of

the IO. When parallelism is not being removed, the OM selects a processing node to create a new IO (according to the current load distribution policy), selects or creates the associated SO and returns their identifier to the PO (calls c in Fig.3). When the RTS is removing excess of parallelism, the PO directly creates the parallel object, by locally creating the IO (call d in Fig.3) and notifying the RTS. In ParC# this generated code is very similar to the ParC++ code and it is placed on the PO object constructor. shows the PO generated constructor from the example in Fig. 4.

```
public class PrimeServer : PrimeFilter {  // PO object
  ...
  PrimeServerImpl obj;                              // reference to IO object
  ...
  public PrimeServer() {
    if (aglomerateObj) {                            // perform agglomeration?
      obj = new PrimeServerImpl();            // intra-grain object creation
      ... // notify local OM
    }
    else {
      ... // contact OM to get a (host) and tcp (port) for the new object
      string uri="tcp://"+host+":"+port+"/factory.soap";
      // gets a reference to the remote factory (rf)
      rf =(RemoteFactory)Activator.GetObject(typeof(RemoteFactory),uri);
      obj=(PrimeServerImpl)rf.PrimeServer(); // request remote object creation
    }
  }
}
```

Fig. 5. PO generated code for IO object creation

```
public class PrimeServerImpl : MarshalByRefObject {
  ...
  public void process(int[] num) {
    ... // copy of the original method implementation
  }
}
// object factory
public class RemoteFactory : MarshalByRefObject  {
  ...
  public PrimeServerImpl PrimeServer() {
    return new PrimeServerImpl();
  }
}
// main code the register the factory
public static void Main (string [] args) {
  ...
  RemotingConfiguration.RegisterWellKnownServiceType(typeof(RemoteFactory),
        "factory.soap",WellKnownObjectMode.Singleton);
}
```

Fig. 6. IO code and the corresponding factory

In ParC++ SO objects are active entities (i.e., threads) that continuously receive messages from PO objects, calling the requested method on local IO and, if needed, returning the result value to the caller. The ParC# implementation no longer requires SO objects and the corresponding message loop to receive external messages, since this loop is implemented by the C# remoting.

The object manager in the ParC++ implementation had the responsibility to perform load management and explicit object creation. A factory was generated for each class and instantiated on each node to implement this functionality. On the C#

prototype this functionality was separated form the OM code since object factories can be automatically registered in the boot code of each node. Fig. 6 shows the code of the generated IO from the previous example and also shows the generated object factory and the code to register this factory.

Aggregating several method calls in a single message required the introduction of a new method in the implementation object to process a pack of several method calls. The parameters of the several invocations are placed in an array structure that is constructed on the PO side and fetched from the array on the IO side. Fig. 7. presents the generated code for method call aggregation.

```
public class PrimeServer : PrimeFilter {    // PO with method call aggregation
  [Serializable]
  struct paramsprocess {
      public int[] num;
  }                                 // array structure for multiple invocations
  public ArrayList processList = new ArrayList();
  paramsprocess processStruct = new paramsprocess();

  public void process (int[] num) {
    if (currentCall++<maxCalls) {              // maxCalls = calls per message
      processStruct.num=num;
      processList.Add(processStruct);
      currentCall++;
    } else {
      obj.processN(processList, maxCalls);
    }
}
public class PrimeServerImpl : MarshalByRefObject {                // IO code
  ...
  public void processN (ArrayList a, int nInv) {
    paramsprocess b;
    for (int i=0;i<maxCalls;i++) {
    b=(paramsprocess)(a[i]);
    process(b.num);
  }
}
```

Fig. 7. Method call aggregation code

4 Performance Results

Performance evaluation was performed through low and high level tests. The low level evaluation measures the base communication latency and bandwidth. The high level evaluation measures the application performance with a simple application. These tests were run in a Linux cluster, connected through a 100 Mbit Ethernet. Each node is a dual Athlon MP 1800+ and has 512 MB of RAM.

Low-level performance was evaluated by a ping-pong test, where messages with several sizes are exchanged between two nodes. These tests compare the Mono Remoting (version 1.1.7) performance against an equivalent Java RMI (SDK 1.4.2) application and an MPI version (MPICH 1.2.6 and GNU g++ 3.2.2). Both Java and the Mono implementations use a remote object, where an array of integers is sent and received as the method parameter and return type. In these results the performance penalty introduced by the ParC# platform is not noticeable (results not shown). The MPI version uses the MPI_Send and MPI_Recv primitives.

Inter-node bandwidth (Fig. 8b) shows that the MPI bandwidth performance is superior to Java and Mono. This is explained by the high level nature of the remote

method invocation and the well-optimised version of MPI. Also, for large messages, the Mono performance lags behind the Java implementation. This may be explained by the fact that the Mono platform is relatively new, when compared to the other alternatives and it is not yet so well tuned.

Inter node latency in Mono (not shown) is between the Java RMI and the MPI latency (respectively, 520, 273 and 100us). This low latency is promising for parallel applications since it is in the same order as highly optimised Java RMI

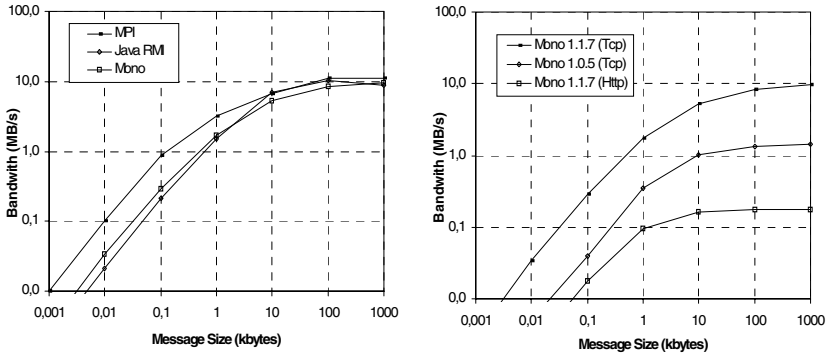

Fig. 8. Inter-node bandwidth a) Mono versus other; b) Mono implementations

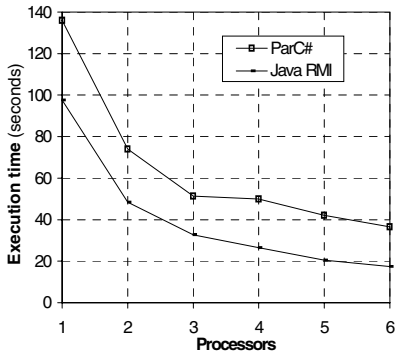

Fig. 9. Parallel Ray Tracer execution time

implementations [3]. This latency is very close to the performance of the Java nio package (introduced in Java 4). However, this Java package is more low level, based on message passing. Fig. 8b compares the performance of various Mono implementations; it shows that Mono performance has radically increased from release 1.0.5 and the low performance of an Http channel.

The high level evaluation was performed using a parallel Ray Tracer from the Java Grande Forum, converted to C#. This application was parallelised using a farming approach, where each worker renders several lines from the generated image.

Fig. 9 compares the execution times of Java and ParC# to render a scene with 500x500 pixels. The C# sequential execution time in this particular application is 40% superior to the Java version (using the Microsoft virtual machine, on a Windows machine, it is only 10% superior). This indicates that the Mono virtual machine is not as highly tuned as the JVM. However, running another application, a prime number sieve, the Mono execution time is about the same as the JVM.

The parallel Ray Tracer execution time in several processors is higher in ParC# mainly due to the higher sequential time and due to thread management. The Mono implementation uses a thread pool to reduce the thread creation cost; however limiting the number of running threads in parallel applications reduces the overlap among computation and communication and also produces starvation in some application threads.

5 Conclusion

This paper presented the implementation of a parallel programming paradigm on top of a C# and .Net platform. The experience with this implementation revealed that the platform greatly simplifies the implementation of the ParC++ and that it is possible to use C# and the .Net platform for parallel applications, both on Windows and UNIX machines. Code can be moved between these two platforms without any recompilation and it is even possible to use it simultaneously on both platforms (something that Java does since its appearance). However, performance gains would be achieved by a more performance tuned Mono implementation; specifically, the virtual machine JIT and the Thread scheduling policy should be improved.

References

[1] Yonezawa, A., Tokoro, M. (eds): Object-Oriented Concurrent Programming, MIT Press (1987)

[2] Wilson G. (ed): Parallel Programming Using C++, MIT Press (1996).

[3] Nester, C. ,Philippsen M., Haumacher, B. : A More Efficient RMI for Java, Proceedings of the ACM 1999 Java Grande Conference, San Francisco, June (1999)

[4] MacBeth, M., McGuigan, K., Hatcher**:** Executing Java threads in parallel in a distributed-memory environment, Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research, Cascon'98, Ontario, Canada, November (1998)

[5] Aridor, Y., Factor, M., Teperman, A.: cJVM: A Single System Image of a JVM on a Cluster, Int. Conference on Parallel Processing**,** Wakamatsu, Japan**,** September (1999)

[6] Baker, M., Carpenter, B., Fox, G., Ko, S., Lim, S.: MPIJAVA:An Object-Oriented JAVA Interface to MPI, International Workshop on Java for Parallel and Distributed Computing, Proceedings of the 11 IPPS/SPDP'99 Workshops, San Juan, Puerto Rico **,** April (1999*)*

[7] Philippsen, M., Zenger, M.: JavaParty – transparent remote objects in Java. Concurrency: Practice and Experience. v. n. 11, November (1997)

[8] Sobral, J., Proença, A.: Designing Scalable Object Oriented Parallel Applications, Proceedings of the 8th Int. European Conference on Parallel Processing (Euro-Par'02), Parderborn, Germany, August (2002)

---

*This work was supported by PPC-VM project POSI/CHS/47158/2002