Dynamic Processes Creation in MPI.NET

Fernando Abrahão Afonso, Nicolas Maillard
Universidade Federal do Rio Grande do Sul
Instituto de Informática
Porto Alegre, RS, Brasil
{faafonso, nicolas}@inf.ufrgs.br

Resumo

MPI is the most used standard for the development of parallel high-performance applications and the MPI-2 version supports dynamic creation of processes. The MPI standard provides bindings only for C, Fortran and C++, but many works support it in many other programming languages. Among this works we can highlight the MPI.NET library for the .Net Framework. This library provides an API with higher levels of abstraction. It also has competitive performance. However, it does not provide dynamic task creation support. The aim of this work is to implement this support and study how this library will respond to it. In the end, our experiments support the conclusion that the main performance problem is at the tasks initialization.

1. Introduction

MPI *(Message Passing Interface)* is the most used specification for the development of high-performance parallel aplications. The MPI standard specifies an API (Application Programming Interface) wich is implemented by diferent MPI libraries. The standard specifies bindings only for C, C++ and Fortran programming languages, so these are the only programming languages that have consolidated and strongly suported MPI libraries [8]. The MPI specification evolved to MPI-2 wich offers support to dynamic process creation, remote memory operations and parallel I/O [9].

Along the years some projects emerged to create MPI libraries on others programming languages like Java, C#, Python and Ruby [8, 1, 7, 17, 4]. Most of these libraries have a simpler programming API due to eliminating the use of pointers and redundant parameters (necessary in the languages supported by the standard). These parameters can be eliminated through the ability of introspection [6] of these programming languages. Moreover most of these libraries allows the transmission of objects to be done in the same way that primitive types. These libraries implements only the features of MPI-1 standard, or some subset of these features.

This article deals with the study and implementation of dynamic processes creation mechanism in the MPI.NET library. In the next section the related works will be discussed. Section 3 discusses the MPI.NET library. In sections 4 and 5 our approach is discussed. We conclude in section 6.

2 Related Work

Several projects proposed MPI libraries for programming languages not supported by the MPI standard. Among these projects, some create a completely new MPI library. Others provide an API which allows access to existing MPI libraries, written in C. None of the studied projects supports dynamic process creation.

2.1 PJMPI

The PJMPI project [17] proposes a MPI library written entirely in Java. The communications uses *sockets*, forming channels for point to point communication between processes. This library allows the point-to-point blocking and non blocking communications.

This library allows objects and user defines data types to be sent in a simplified way. This requires that they derive from the abstract class called `Datatype`. This class has two abstract methods, which should be overwritten. These methods describe how to serialize and de-serialize objects. During communication, all data (primitive or not) is serialized. The vector is the only transmitted structure, and to recover sent data, a method called `getDataType` is used. That method is responsible for discovering the type of received data.

In performance tests the results were significantly slower compared to MPI-C library [17]. The large performance gap is because vector operations are very slow in Java. Another problem is large data transmission, which consu-

mes too much memory and is very slow due to the serialization process.

2.2 JMPI

The project JMPI [14] proposes the creation of a MPI library written entirely in Java. The communications are performed through RMI (technology that allows access to remote calls in Java) [5]. One of the advantages of using RMI is in the transmission of objects. When an object with reference to others is transmitted, the complete graph of references will be transmitted automatically.

The transmission of matrices are performed through the use of introspection, which allows, dynamically determining the size of each row of a matrix, as well as it's type. The treatment of errors uses exceptions, allowing the user to specify actions for each exception.

The performance of the library was compared with the performance of the library [3]. The library mpiJava has a considerably higher performance than JMPI [14].

2.3 mpiJava

The project mpiJava [3] offers a MPI library for the Java programming language using the JNI interface (interface that allows access to native libraries) [11]. The functions implemented in this library call their equivalent functions in the native MPI library.

This library implements the interface of most MPI-1 calls. It incorporates the functions MPI-C within the Java programming environment. To program with the library the user must instantiate an object of the class `MPI` where the calls are equal to those available in the MPI-C API.

The performance tests presented in [3] demonstrated a competitive performance having a small overhead because of the JNI interface.

2.4 Pure Mpi.NET

The Pure Mpi.NET [2] library is written entirely in C#. It can be used by the languages of the .Net platform (C#, Visual Basic, J#, etc..). It uses WCF (Windows Communication Foundation) [12] for communications between the processes. WCF is the Web Services technology .Net platform.

The transmission of objects can be done in the same way as the transmission of primitive types. Only three parameter are used to send a message: `destination`, `tag` and `data`). All communications of the library have the option to be called with a time limit, making possible to manage within the application when an operation of transmission is not completed within a certain time.

The library implements the point-to-point (blocking and non blocking) and some collective communications. The performance of the library is not satisfactory because the use of WCF. Web Services do not have suitable performance for use in high-performance parallel applications [10].

2.5 PMPI

The PMPI library [16] is a MPI library for the .Net platform, written entirely in C#. It uses .Net Remoting (technology to access remote methos) [15] to achieve the processes communications.

Its API is very similar to the MPI API for C. To programm with the library the user must instantiate an object of class `MPI`, which contains MPI calls very similar to the MPI-C MPI calls. The library implements point-to-point communications. Its performance is 70 % less than MPICH2 library [16].

3 MPI.NET Project

The MPI.NET [8] project is a MPI library for the .Net Framework written in C#. This library makes MPI programming easier by abstraction of calls, eliminating the need to pass parameters that can be inferred by introspection methods.

The .Net supports iteration between different programming languages, including languages that do not run from within your virtual machine, such as C and C ++. The MPI.NET library uses this support to run over the native MPI library written in C. The user can change the MPI library used as base having the possibility of using an consolidated and strongly supported MPI library.

User defined types (structures, objects, etc..) can be transmitted in the same way as primitive types. The colletive communications are also considerably simplified by using a smaller amount of parameters in their interfaces.

The performance of the library MPI.NET is very similar to the performance of the native library, which is only 1 to 2% slower for small messages and ranges from 15% slower to 10% faster for large messages [8].

The MPI.NET performance, combined with the programming interface features of the programming language C# and the possibility of using a consolidated MPI libraries to carry the communications motivated the choice of this library for this work. Another advantage offered by the library is the ability to be used by various programming languages that run inside the platform .Net.

4 MPI.NET-Spawn

This section deals with the implementation of the dynamic process creation in the MPI.NET library. The functi-

ons responsible for creating and managing processes dynamically in MPI are: `MPI_Comm_spawn` (creates new process dinamically) and `MPI_Comm_get_parent` (recovers the parent communication in the created process).

We took a base for the implementation the MPI-2 C++ API, which implements these functions in the class `Comm`. In our implementation, the functions are implemented in the class `Communicator` which is equivalent to the class `Comm`. As the library MPI.NET has versions for Windows and Linux, the implementation was done in those two versions.

The native interfaces were created to allow the calls to the MPI-C library. For each of the parameters used by the implemented calls, the most compatible C# type with the parameter of the call in native C was specified. The implementation of the Spawn methods were included in the class `Communicator`, calling the native interface to execute them. The interfaces of these methods are simpler than the interface offered by C and C++, while allowing the user to use all the parameters allowed by C and C++. To make this possible, five method overloads was written.

The parameters to be passed to the native library must be protected from the Garbage Collector. That must be done by allocating a GCHandle [13] object. This object is used to protect the data and to recover the reference to data that must be passed as reference to the native library. The Strings must be modified as well with the add of `\0` at the end of the lines so the native library knows the line ends there.

5 Performance Evaluation

A few performance tests where executed to see how the library would behave with synthetic benchmarks. This tests where done in a cluster with the following configuration:

- **Alocated machines**: 8;
- **Processors per machine**: 2 Pentium III 1266 Mhz;
- **Memmory per machine**: 512 Mb;
- **Nethwork**: Fast Ethernet.
- **MPI**: MPICH2 1.0.8p1;
- **.Net Virtual Machine**: Mono 2.4

The first benchmark was called Spawn-n (figure 1). This benchmark launches from 1 to n processes by calling the Spawn method. We compared the performance of our library against native MPI-C++. We also compared our library launching C++ processes and the C++ native library launching C# processes. The results showed that launching C# processes are much slower than launching C++ processes (due to the inicialization of a virtual machine for each process), and because of that, the Spawn function has a bad performance. The results also showed that our mechanism didn't introduced significative overhead as the time to launch C++ processes with our library is almost the same time that the native library takes.

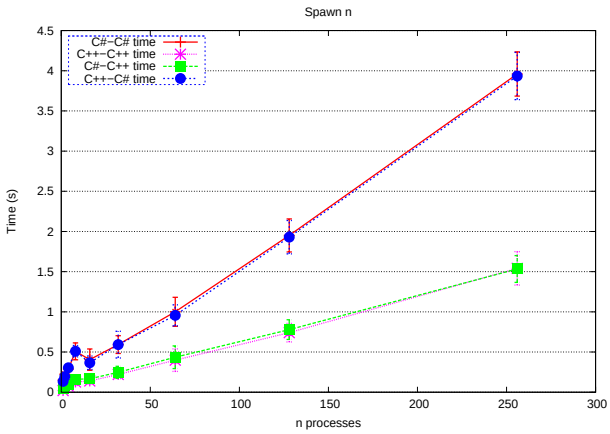

Figura 1. Execution of Spawn-n

The time to create a process consists of 6,5% to prepare the data, 92,5% for the native library to run the call and 1% to recover the communicator returned by the call.

The second benchmark is the parallel computation of the n number of the Fibonacci sequence (figure 2). This benchmark launches lots of processes with low computing time (only one small task per process). As we expected, the overhead due to processes creation was to creation was too high for this kind of application, so the performance was very slower than the C++ native MPI program. This is due to the low computation time of each task combined with the high ammount of dynamic created processes. We compared sequential versions of this benchmark and the C# performance was very similar to the C++ performance.

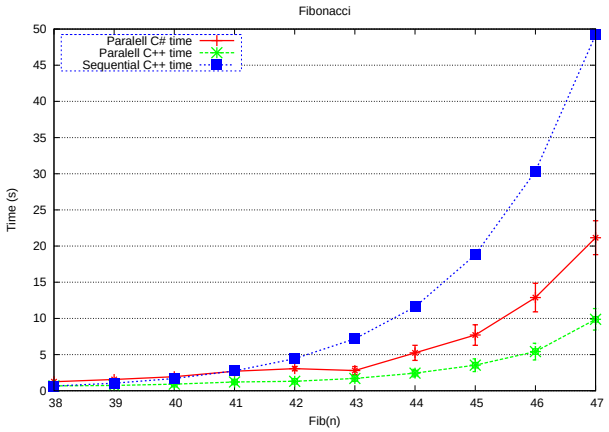

Figura 2. Execution of Fibonacci

The last benchmark was the parallel calculum of the Mandelbrot fractal (figure 3). For this program the time to process the task was much higher than the time to create it. Also, the amount of processes was smaller than in the Fibonacci program, so the performance of the C# version didn't lose as much performance as in the Fibonacci test. This is due to the high granularity of the tasks combined with the low amount of dynamic processes created.

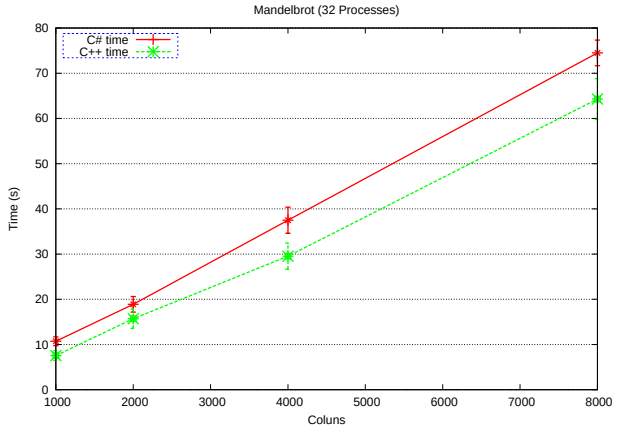

Figura 3. Execution of Mandelbrot

6. Conclusions and Future Work

In this work we showed that it is viable to implement dynamic process creation on MPI libraries not supported by the MPI standard.

The performance of C# process creation is worse than that of C++ process creation due to the cost of initializing of an individual virtual machine for each C# process.

Even with the high cost of dynamic creating C# processes some problems can have acceptable performance if the granularity/amount of process proportion is tuned. It's better to create lots of processes with one call to the `MPI_Comm_spawn` than to create the same ammount of processes with several calls to the `MPI_Comm_spawn` function. The overhead to prepare the data is fixed, so when creating one or several processes the only variation will be in the time taken by the native library.

As future work we plan to run new benchmarks and to improve the performance of dynamic process creation.

Referências

[1] pympi, 2005. Disponível em http://pympi.sourceforge.net/. Acesso em 5 de abr. de 2009.

[2] Pure mpi.net, 2008. Disponível em http://www.purempi.net/. Acesso em 14 de nov. de 2008.

[3] M. Baker, B. Carpenter, G. Fox, S. H. Ko, and S. Lim. Mpijava: An object-oriented java interface to mpi. In *Proceedings of the 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, pages 748–762, London, UK, 1999. Springer-Verlag.

[4] R. Cilibrasi. Mpi ruby, 2001. Disponível em http://mpiruby.sourceforge.net/. Acesso em 8 de mai. de 2009.

[5] B. Eckel. *Thinking in Java*. Prentice Hall PTR, 2006.

[6] D. Flanagan. *Java in a Nutshell*. Cambridge, 1998.

[7] V. Getov, P. Gray, and V. Sunderam. Mpi and java-mpi: contrasts and comparisons of low-level communication performance. In *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 21, New York, NY, USA, 1999. ACM.

[8] D. Gregor and A. Lumsdaine. Design and implementation of a high-performance mpi for c# and the common language infrastructure. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 133–142, New York, NY, USA, 2008. ACM.

[9] W. Gropp, R. Thakur, and E. Lusk. *Using MPI-2: Advanced Features of the Message Passing Interface*. MIT Press, Cambridge, MA, USA, 1999.

[10] S. Gupta. A performance comparison of windows communication foundation (wcf) with existing distributed communication technologies, 2007. Disponível em http://msdn.microsoft.com/en-us/library/bb310550.aspx. Acesso em 30 de nov. de 2008.

[11] S. Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[12] C. McMurtry, M. Mercuri, N. Watling, and M. Winkler. *Windows communication foundation 3.5 unleashed, second edition*. SAMS, Carmel, IN, USA, 2008.

[13] Microsoft. Gchandle structure, 2007. Disponível em http://msdn.microsoft.com/en-us/library/system.runtime.interopservices.gchandle.aspx. Acesso em 03 de mar. de 2008.

[14] S. Morin, I. Koren, and C. M. Krishna. Jmpi: Implementing the message passing standard in java. In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 191, Washington, DC, USA, 2002. IEEE Computer Society.

[15] I. Rammer. *Advanced .Net Remoting*. Apress, Berkely, CA, USA, 2002.

[16] M. M. E. Saifi. Pmpi: uma implementação mpi multiplataforma, multi-linguagem. Master's thesis, Universidade de São Paulo, 2006.

[17] T. W. Y. H. Y. WenSheng. Pjmpi: pure java implementation of mpi. In *High Performance Computing in the Asia-Pacific Region*, volume 1, pages 14–17, 2000.