Scientia
Interdisciplinary Studies in Computer Science
20(1): 53-64, January/June 2009

Implementação da criação dinâmica de tarefas na biblioteca MPI.NET

Fernando Abrahão Afonso, Nicolas Maillard

Universidade Federal do Rio Grande do Sul
Avenida Bento Gonçalves, 9500, Porto Alegre, RS, Brasil

{faafonso, nicolas}@inf.ufrgs.br

Resumo
MPI é o padrão mais utilizado para o desenvolvimento de aplicações paralelas de alto desempenho e sua versão MPI-2 oferece suporte à criação dinâmica de tarefas. A norma MPI provê especificações para Fortran, C e C++. Surgiram algumas tentativas de suportá-la em várias outras linguagens de programação. Dessas tentativas, pode-se destacar a biblioteca MPI.NET (para o Framework. Net), a qual demonstrou uma API com maiores níveis de abstração do que as *Application Programming Interface* (API's) da norma, bem como desempenho competitivo em relação a MPI-C. No entanto, essa biblioteca possui uma lacuna: o suporte à criação dinâmica de tarefas. O objetivo deste trabalho é preencher esta lacuna, estudando a utilização da biblioteca MPI.NET para a criação e execução de aplicações paralelas que utilizem criação dinâmica de tarefas. Ao final, o estudo demonstra que o grande problema de desempenho está na inicialização das tarefas.

PALAVRAS-CHAVE: programação paralela, processamento de alto desempenho, MPI.

Abstract
Implementation of dynamic tasks creation on the MPI.NET library. MPI is the most used standard for the development of parallel high-performance applications and the MPI-2 version supports dynamic creation of tasks. 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.

KEY WORDS: parallel programming, high performance computing, MPI.

1 Introdução

MPI (*Message Passing Interface*) é a norma mais utilizada para o desenvolvimento de aplicações paralelas de alto desempenho. Ela especifica uma API (*Application Programming Interface*), a qual é implementada por diferentes bibliotecas MPI. A norma especifica API's somente para as linguagens de programação Fortran, C e C++, portanto, somente essas linguagens possuem implementações de bibliotecas MPI consolidadas e amplamente suportadas (Gregor e Lumsdaine, 2008). A norma evoluiu para MPI-2, inserindo novas funcionalidades na interface MPI, a qual passou a oferecer suporte à criação dinâmica de tarefas, E/S paralela e operações remotas de memória. Também foi incluída a API para a linguagem de programação C++ (Gropp *et al.*, 1999).

A API da norma MPI para a linguagem C++ é constantemente criticada, por não suportar que objetos sejam enviados da mesma maneira que tipos primitivos, e por não suportar outras abstrações contidas na linguagem C++ (McCandless *et al.*, 1996; Kambadur *et al.*, 2006; Gregor e Troyer, 2003). A maior parte dos trabalhos que a criticam propõem modificações com o intuito de aumentar o grau de abstração da API e oferecer suporte simplificado ao envio de objetos.

Surgiram diversos projetos de bibliotecas MPI para linguagens de programação como Java (JavaMPI, MPI-Java, PJMPI) (Getov *et al.*, 1999; WenSheng, 2000), C# (MPI.NET) (Gregor e Lumsdaine, 2008), Python (pyMPI) (PYM, 2005) e Ruby (MPI Ruby) (Cilibrasi, 2001). Grande parte dessas bibliotecas possui uma API mais simples, de utilização simplificada, que elimina o emprego de ponteiros e parâmetros redundantes (porém necessários nas linguagens suportadas pela norma). Esses parâmetros podem ser eliminados por intermédio da capacidade de introspecção (Flanagan, 1998) dessas linguagens de programação. Além disso, a maior parte dessas bibliotecas permite que o envio de objetos seja feito da mesma maneira que o envio de tipos primitivos.

Apesar de as linguagens de programação mais recentes (Java, C#, Ruby, Python etc.) possuírem novos mecanismos de comunicação com maiores níveis de abstração (por exemplo, RMI (Eckel, 2006), Web Services (Kalin, 2009; McMurtry *et al.,* 2008) e Remoting (Rammer, 2002)), esses mecanismos não têm a capacidade de substituir a norma MPI, por não possuírem desempenho aceitável para programar aplicações paralelas de alto desempenho (Morin *et al.*, 2002; Schwarzkopf *et al.*, 2008; Juric *et al.*, 2004). As bibliotecas MPI desenvolvidas para essas linguagens de programação mostram-se uma boa alternativa para programar esse tipo de aplicações.

As bibliotecas MPI desenvolvidas para linguagens de programação não suportadas pela norma implementam somente os recursos da norma MPI-1, ou algum subgrupo desses recursos. No entanto, algoritmos baseados no modelo Divisão e Conquista (D&C), bem como algoritmos nos quais a carga de trabalho é desconhecida no início da execução, obtêm vantagem na utilização de criação dinâmica de tarefas, a qual já era utilizada no PVM (El-Rewini e Abd-El-Barr, 2005), por simplificar a programação e permitir melhor balanceamento da carga de trabalho. Em síntese, é necessário suportar os recursos da norma MPI-2.

As características discutidas para a interface MPI-C++ estão presentes na biblioteca MPI.NET. Ela foi criada a partir de um projeto que discute a norma MPI para C++, o projeto BOOST.MPI (Gregor e Troyer, 2003). Essa biblioteca está sendo escrita na linguagem de programação C# e pode ser utilizada por qualquer linguagem de programação que faça parte do Framework .Net. Ela demonstrou bons resultados em seus testes de desempenho (Gregor e Lumsdaine, 2008) e tem implementado a maior parte dos recursos presentes na norma MPI-1.

Este artigo trata do estudo e da implementação do mecanismo de criação dinâmica de tarefas na biblioteca MPI.NET.

A seção 2 deste artigo discute o estado da arte do suporte à MPI em linguagens de programação não suportadas pela norma MPI. Na seção 3, é discutida a biblioteca utilizada para a implementação desse trabalho. Na seção 4, discute-se a proposta de inclusão da criação dinâmica de tarefas na biblioteca MPI.NET, bem como o desempenho obtido nos testes. Por fim, na seção 5, são apresentadas as conclusões e as propostas para trabalhos futuros.

2 Estado da arte

Diversos projetos propõem bibliotecas MPI para linguagens de programação não suportadas pela norma. Dentre esses projetos, alguns criam uma biblioteca MPI totalmente nova, escrita na linguagem de programação na qual se quer utilizá-la. Outros proveem uma API que permite acessar bibliotecas MPI já existentes, escritas em C. Esta seção discorre sobre os conceitos básicos de MPI e, após, discute alguns destes projetos. A biblioteca escolhida para a realização deste trabalho será discutida na próxima seção.

2.1 Programação com MPI

A comunicação entre processos de um programa MPI é realizada por meio de troca de mensagens explícitas. Para isso, MPI especifica diversas primitivas de comunicação (ponto a ponto, coletivas, bloqueantes e não bloqueantes). Para realizar a troca de uma mensagem ponto a ponto, a primitiva `MPI_Send` e suas variações são utilizadas. As interfaces do `MPI_Send` para C e C++ são:

- **C**: `int MPI_Send(void *buf, int count, MPI_Datatype data- type, int dest, int tag, MPI_Comm comm);`
- **C++**: `void Comm::Send (void* buf, int count, MPI_Datatype datatype, int dest, int tag);`

A diferença entre a interface, nessas duas linguagens, está no comunicador, o qual, em C, é instanciado como uma variável do tipo `MPI_Comm` e passado como parâmetro para a função. Já em C++, um objeto do tipo `MPI::Comm` deve ser instanciado, visto que ele encapsula os métodos de comunicação. Os outros parâmetros comuns às duas linguagens são:

(i) `void *buf`: Os dados a serem enviados;
(ii) `int count`: A quantidade de dados a serem enviados;
(iii) `MPI_Datatype datatype`: O tipo dos dados que estão sendo enviados;
(iv) `int dest`: O destino para qual os dados estão sendo enviados.
(v) `int tag`: Uma *tag* para o programador diferenciar as mensagens.

A estrutura `MPI_Datatype` contém os tipos primitivos presentes nas linguagens suportadas pela norma. É por meio dessa primitiva que a biblioteca MPI distingue o tipo de dado que está sendo transmitido. Por exemplo, ao transmitir um inteiro, passa-se como parâmetro `MPI_Int` para o `datatype`. Dados serializados (convertidos de um tipo definido pelo usuário para um *buffer* de *bytes*) podem ser transmitidos como dados do tipo `MPI_Byte`.

2.2 Pure Mpi.NET

A Pure Mpi.NET (PMP, 2008) é uma biblioteca MPI totalmente escrita em C#, a qual pode ser utilizada pelas linguagens suportadas pelo Framework .Net (C#, Visual Basic, J#, etc.). Ela utiliza WCF (*Windows Communication Foundation*) para realizar as comunicações entre os processos (McMurtry *et al.*, 2008). WCF é a tecnologia de Web Services do Framework .Net.

Sua API possui maiores níveis de abstração em relação às API's MPI suportadas pela norma. Por exemplo, o envio de tipos primitivos pode ser feito da mesma forma que o envio de objetos, sendo, para isso, necessários apenas três parâmetros: destino, *tag* e dados, ao contrário da interface MPI para C, que, para enviar tipos primitivos além desses parâmetros, necessita que o usuário especifique o tipo dos dados a serem enviados, a quantia de dados a serem enviados e o comunicador a ser utilizado. O envio de tipos abstratos em C exige a serialização manual dos dados e o cálculo manual do tamanho do buffer a ser enviado.

Essa biblioteca possui algumas funcionalidades adicionais às suportadas pela norma MPI. Por exemplo, é possível especificar um método de *callback* ao fazer um `Send`. A biblioteca implementa as comunicações ponto a ponto bloqueantes e não bloqueantes, e algumas comunicações coletivas.

Todas as comunicações da biblioteca possuem a opção de serem chamadas com limite de tempo, fazendo com que seja possível gerenciar exceções, dentro da aplicação, quando uma operação de envio não é concluída dentro de determinado tempo. Mesmo que o usuário opte por não especificar um tempo limite, a biblioteca utiliza um limite padrão. Os erros são gerenciados via exceções, permitindo que a aplicação do usuário trate erros que possam ocorrer.

O desempenho da biblioteca não é satisfatório pelo fato de ela utilizar WCF em suas comunicações. Web Services não possuem o desempenho apropriado para o uso no desenvolvimento de aplicações paralelas de alto desempenho (Gupta, 2007). Além dos problemas de desempenho, a biblioteca implementa um subgrupo de chamadas MPI-1 e não suporta a norma MPI-2.

2.3 PMPI

Uma outra biblioteca MPI, totalmente escrita em C#, é a biblioteca PMPI (Saifi, 2006). Essa biblioteca utiliza .Net Remoting (Rammer, 2002) para realizar suas comunicações. Remoting é a tecnologia de acesso a chamadas remotas do Framework .Net.

Sua API é muito similar à API MPI para C; não tira proveito da capacidade de abstração oferecida pela linguagem de programação C#. Para programar nessa biblioteca, o usuário deve instanciar um objeto do tipo MPI, o qual contém as chamadas MPI iguais às da norma para C encapsuladas. Não faz sentido uma biblioteca MPI para uma linguagem orientada a objetos seguir a norma MPI para C e não para C++, uma vez que C++ é uma linguagem bem mais próxima da que usa a programação C#. Além disso, essa linguagem de programação permite abstrair as chamadas MPI, eliminando parâmetros redundantes.

Após o usuário instanciar um objeto do tipo MPI, deve utilizá-lo da seguinte maneira para enviar uma mensagem: `objMPI.MPI_Send(void *buf, int count, MPI_Datatype, int dest, int tag, MPI_Comm comm)`. Essa interface é a mesma utilizada na linguagem C, na qual o usuário passa o comunicador como um dos parâmetros. Para uma linguagem orientada a objetos, tomando como base a norma para C++, bem como os trabalhos que a discutem, faz mais sentido que o comunicador seja encapsulado dentro de um objeto do tipo "comunicador", como na API MPI para C++. Em uma biblioteca MPI escrita totalmente em C#, não é necessário especificar o tipo e a quantia de dados a serem enviados, uma vez que essas informações podem ser facilmente inferidas via introspecção.

Além dos problemas de interface, a biblioteca possui um desempenho 70% inferior à biblioteca MPICH2 (Saifi, 2006) fato que desmotiva a sua utilização.

2.4 PJMPI

O projeto PJMPI (WenSheng, 2000) propõe uma biblioteca MPI totalmente escrita em Java. As comunicações da biblioteca utilizam *sockets*, formando canais

de comunicação ponto a ponto entre os processos. Para realizar comunicações não bloqueantes, são utilizadas duas *threads* com duas filas de mensagens: uma para dados a serem enviados e outra para dados a serem recebidos.

Para enviar tipos de dados definidos pelo usuário, é necessário que eles derivem da classe abstrata chamada `Datatype`. Essa classe possui dois métodos abstratos, os quais devem ser sobrescritos; um deles serve para converter os dados em um vetor de *bytes* para ser transmitido; o outro, para converter os *bytes* do vetor para os dados do usuário novamente.

Todos os dados são sempre convertidos para um vetor de *bytes*, primitivos ou não. Nas comunicações, o vetor é a única estrutura a ser enviada e, para recuperar os dados enviados, é utilizado um método chamado `getDataType`, o qual é responsável por descobrir o tipo do dado recebido e converter o vetor de *bytes* para o seu tipo de dado correspondente.

Nos testes de desempenho, os resultados obtidos foram significativamente mais lentos em relação à biblioteca MPI-C (WenSheng, 2000) . A causa apontada para o grande *gap* de desempenho é o fato de operações em vetores serem muito lentas em Java. Um outro problema encontrado está no envio de vetores grandes, o que consome muita memória e é muito lento, em vista do processo de serialização dos dados. A biblioteca também não oferece comunicações coletivas.

2.5 JMPI

O projeto JMPI (Morin *et al.*, 2002) propõe a criação de uma biblioteca MPI totalmente escrita em Java. As comunicações são realizadas por meio de RMI (Eckel, 2006). RMI é a tecnologia que permite o acesso a chamadas remotas em Java. Uma das vantagens de se utilizar RMI para a transmissão de objetos é que um objeto com referência para outros, ao ser serializado, serializará o grafo completo de referências e transmitirá todos os objetos automaticamente.

A biblioteca possui três diferentes camadas:

(i) *A API MPI:* o núcleo das funções MPI a serem utilizadas pelas aplicações, seguindo o modelo proposto pelo projeto (Baker *et al.*, 1999);
(ii) *A camada de comunicação:* o núcleo de comunicação, que contém todas as comunicações necessárias para a implementação da API MPI;
(iii) *A máquina virtual Java:* responsável por compilar e executar as aplicações.

Uma das vantagens de se utilizar Java está na transmissão de matrizes, a qual se realiza por meio da utilização de introspecção, o que permite, dinamicamente, determinar o tamanho de cada linha de uma matriz, bem como possibilita determinar o tipo de um objeto. Outra vantagem está no tratamento de erros, em que exceções são disparadas, o que permite ao usuário especificar ações para cada exceção.

Para testar o desempenho da biblioteca, foram utilizadas uma aplicação PingPong (aplicação que mede o tempo de enviar uma mensagem e recebê-la de volta) e uma aplicação que calcula o fractal de Mandelbrot. A performance da biblioteca foi comparada com a performance da biblioteca mpiJava (Baker *et al.*, 1999). Foram utilizadas duas versões do RMI para realizar essa comparação, o RMI da Sun e o KaRMI (Nester *et al.*, 1999).

Enquanto os testes apresentados demonstraram que o JMPI/KaRMI possui um desempenho melhor que o JMPI/RMI, o mpiJava revelou possuir um desempenho consideravelmente superior em relação às duas versões do JMPI. Sua interface de programação é igual à da biblioteca mpiJava, porém seu desempenho é consideravelmente pior que o desta, que já possui certo *overhead* em relação às grandes distribuições MPI (Morin *et al.*, 2002). A biblioteca implementa comunicações ponto a ponto e algumas comunicações coletivas.

2.6 mpiJava

O projeto mpiJava (Baker *et al.*, 1999) propõe uma biblioteca MPI para a linguagem de programação Java. Sua API possui a estrutura de classes baseada na estrutura de classes da interface MPI para a linguagem C++. A biblioteca é implementada mediante a utilização da interface JNI (Liang, 1999), a qual permite a realização de chamadas a bibliotecas MPI para C. As funções MPI oferecidas pela biblioteca chamam suas funções equivalentes na biblioteca MPI nativa, que as executa.

Essa biblioteca implementa a interface da maioria das chamadas MPI-1. No entanto, a biblioteca não abstrai as chamadas; ela simplesmente insere as funções MPI-C dentro do ambiente de programação Java (o que não é bom, pois a linguagem Java permite inferir grande parte dos parâmetros das chamadas MPI para C, eliminando a redundância). Sua interface é muito similar à da biblioteca PMPI, na qual se instancia um objeto da classe MPI, em que se encontram as chamadas MPI iguais às disponíveis na API MPI-C.

No modelo de programação Java, não se utilizam ponteiros. No entanto, o retorno das funções MPI é feito da mesma maneira que no MPI-C: recupera-se, via referência, na chamada da função, em vez de utilizar o `return`. Como Java não utiliza ponteiros, foi empregado o artifício de criar um vetor de somente 1 (uma) posição para mandar dados

singulares, e um vetor de mais posições para mandar mais dados. Isso se deve ao fato de vetores serem passados por referência em Java.

Os testes de desempenho apresentados em (Baker *et al.*, 1999) demonstraram um desempenho competitivo, mas existe um pequeno *overhead* (constante) devido à utilização da JNI. A biblioteca deixa a desejar uma melhor interface, a qual combine com o paradigma de programação orientada a objetos da linguagem Java. A biblioteca também não suporta as funcionalidades presentes na norma MPI-2.

2.7 Comparativo

As bibliotecas estudadas nesta seção possuem características distintas. Entre essas, estão: (i) o desempenho, (ii) a interface de programação, (iii) a tecnologia utilizada para sua implementação e (iv) as funcionalidades MPI que cada uma delas implementa.

Um quadro comparativo das bibliotecas estudadas nesta seção pode ser visto na Tabela 1, a qual compara as principais características presentes nessas bibliotecas. Esse quadro demonstra que a biblioteca MPI.NET atinge todos os requisitos esperados de uma biblioteca MPI alto-nível.

3 Projeto MPI.NET

O projeto MPI.NET foi criado na Universidade de Indiana (Gregor e Lumsdaine, 2008) com o intuito de combinar o alto desempenho do MPI-C com o alto nível de abstração da linguagem de programação C#. Essa biblioteca torna a programação MPI mais simples, mediante a abstração das chamadas, eliminando a necessidade de passar parâmetros redundantes ou que possam ser inferidos para as chamadas MPI. Isso elimina erros comuns de programação, como, por exemplo, o cálculo de maneira errônea do tamanho de um *buffer* de dados.

A plataforma .Net oferece suporte à iteração entre diversas linguagens, inclusive entre as que não rodam dentro de sua máquina virtual, como C e C++. A biblioteca MPI.NET utiliza esse suporte para rodar sobre a biblioteca MPI nativa escrita em C. As chamadas MPI implementadas pela biblioteca realizam chamadas à biblioteca nativa, não sendo necessário implementar novas chamadas MPI. Isso provê flexibilidade para o usuário trocar a biblioteca MPI a ser utilizada como base e, além disso, a possibilidade de utilizar uma biblioteca MPI consolidada e fortemente suportada.

Em ambiente Windows, a biblioteca MPI.NET funciona sobre a MPI da Microsoft (baseada no MPICH2); já no Linux, qualquer biblioteca MPI pode ser utilizada como base. Para a implementação do Spawn (chamada responsável por criar tarefas dinamicamente) realizada, neste trabalho, em ambiente Windows, a biblioteca MPI utilizada como base foi trocada pelo MPICH2 devido à MPI da Microsoft não suportar a chamada Spawn da norma MPI-2.

Para uma biblioteca permitir a utilização de MPI dentro da linguagem C#, ela não deve simplesmente importar as chamadas nativas da biblioteca MPI-C e torná-las disponíveis para o programador C#; é desejável prover uma interface com maiores níveis de abstração, que respeite a linguagem e também seus paradigmas. Objetos devem ser transmitidos naturalmente como se fossem tipos primitivos e a biblioteca deve ser simples e intuitiva para o programador C#, de modo a eliminar a necessidade de passar dados redundantes, uma vez que podem ser inferidos, mediante métodos de introspecção, padrões da linguagem.

A biblioteca MPI desenvolvida nesse projeto funciona para todas as linguagens que executam sob a máquina virtual da plataforma .NET. As chamadas de métodos requerem um número de parâmetros consideravelmente

Tabela 1. Comparativo entre as bibliotecas MPI para Java e .Net.
Table 1. Comparison between the MPI libraries for Java and .Net.

	Desempenho	Linguagem de programação	Interface de programação	Tecnologia	Funcionalidades
Pure Mpi.NET	Baixo	.Net	Simples e orientada a objetos	WCF	Comunicações ponto a ponto e coletivas
PMPI	Baixo	.Net	MPI-C	Remoting	Comunicações ponto a ponto
PJMPI	Baixo	Java	Simples e orientada a objetos	Sockets	Comunicações ponto a ponto
JMPI	Baixo	Java	MPI-C	RMI	Comunicações ponto a ponto e coletivas
mpiJava	Alto	Java	MPI-C	Interface nativa	Norma MPI-1
MPI.NET	Alto	.Net	Simples e orientada a objetos	Interface nativa	Norma MPI-1

menor, eliminando redundâncias. Mesmo assim, essa interface permite a utilização plena das funções MPI e, além disso, possui suporte para envio de tipos definidos pelo usuário (estruturas, objetos etc.) como se fossem tipos primitivos.

As comunicações coletivas também são consideravelmente simplificadas. Por exemplo, o método `Gather` encapsula tanto o `MPI_GATHER` quanto o `MPI_GATHERV`; em uma única chamada, retorna uma matriz cujo tamanho de cada linha equivale ao tamanho do vetor enviado pelos outros processos, um a um, sem que seja necessário especificar a quantidade de dados a serem enviados nem a quantidade de dados a serem recebidos. Simplesmente, especificam-se os dados a serem enviados e quem os receberá, conforme se vê na Figura 1. No C e no C++, é necessário especificar, dentre diversas outras informações, a quantidade de dados a serem enviados e o número de dados a serem recebidos. No caso do `MPI_GATHERV`, a quantidade de dados a ser recebida de cada processo deve ser discriminada em um vetor que informa a quantia que cada processo enviará.

Outro exemplo de simplicidade da biblioteca MPI.NET, em relação à biblioteca MPI para C ou C++, está no envio de estruturas abstratas como um objeto. No C#, basta enviar o objeto. No C, é necessário serializar manualmente os dados para um *buffer*, calcular o tamanho desse *buffer*, para, finalmente, enviá-lo. Do outro lado é necessário fazer o processo inverso. Um exemplo de envio de objeto é: `comm.Send(obj, dest, tag)`, que é feito da mesma forma que o envio de tipos primitivos: `comm.Send(1, dest, tag)`. Para que isso seja possível, a biblioteca serializa automaticamente o dado criado pelo usuário para um vetor de *bytes* e o envia como um vetor do tipo `MPI_BYTE` para a biblioteca MPI-C, de forma transparente para o usuário.

Os testes de desempenho foram realizados mediante o uso do benchmark NetPIPE (Snell *et al.*, 1996), o qual mede o *throughput* da rede baseado em uma aplicação PingPong, que utiliza diversos tamanhos de mensagens. Um dos resultados pode ser visto na Figura 2, a qual demonstra a eficiência do NetPIPE do C#, em relação ao NetPIPE do C, utilizando tipos primitivos em memória compartilhada. O desempenho da biblioteca MPI.NET é bem similar ao desempenho da biblioteca nativa; é somente de 1 a 2 % (Gregor e Lumsdaine, 2008) mais lenta para mensagens pequenas e varia de 15% mais lenta, a 10% mais rápida, para mensagens grandes. Poderia ser usadas: *menores* maiores em vez de pequenas e *grandes*?

O fato de esta biblioteca oferecer um bom desempenho, combinado com uma interface de programação característica da linguagem de programação C#, bem como a possibilidade de utilizar uma biblioteca MPI consolidada para realizar as comunicações, motivaram a escolha desta para a realização deste trabalho. Além disso, uma outra vantagem oferecida por esta biblioteca é a capacidade de ser utilizada pelas diversas linguagens de programação que executam dentro do Framework .Net, inclusive uma linguagem muito similar à Java, a linguagem J#.

4 MPI.NET-Spawn

Embora a biblioteca MPI.NET implemente a norma MPI-1 por completo, ela não implementa a norma MPI-2. Tendo em vista que a utilização da criação dinâmica de tarefas traz vantagens para a programação de certos tipos de aplicações como, por exemplo, as aplicações em que se desconhece a carga de trabalho no início da aplicação e as aplicações baseadas em divisão e conquista, como também o balanceamento da carga em ambientes heterogêneos, motivou-se este trabalho o qual implementa e testa as funções MPI responsáveis por criar e gerenciar tarefas dinamicamente

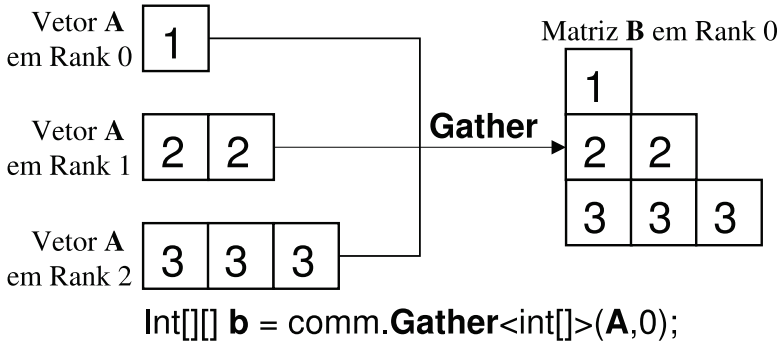

Figura 1. Exemplo de uso do Gather.
Figure 1. Example of the Gather use.

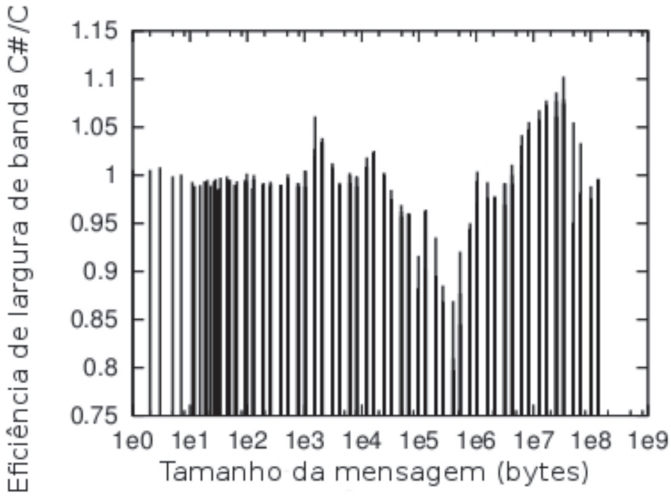

Figura 2. Eficiência do NetPIPE do C# em relação ao NetPIPE do C (Gregor e Lumsdaine, 2008).
Figure 2. C#'s NetPIPE efficiency in relation to C's NetPIPE (Gregor and Lumsdaine, 2008).

(`MPI_Comm_spawn` e `MPI_Comm_get_parent`) na biblioteca MPI.NET.

Nessa implementação, foi tomada como base a API MPI-2 para C++ a qual implementa as funções dentro da classe `Comm` (classe que contém as funções de comunicação MPI). Na implementação para C#, as funções foram implementadas na classe `Communicator`, a qual é equivalente à classe `Comm` do C++.

Primeiramente, foi necessário criar as interfaces para as chamadas nativas em C, as quais foram alocadas na classe responsável por mobilizar todas as interfaces para as chamadas em C na biblioteca MPI.NET, a classe `Unsafe`. Foi necessário especificar o tipo de dado mais compatível do C# em relação a cada um dos argumentos da chamada em C, por exemplo, `char *command` da função em C recebe `byte[] command` e `char *argv[]` recebe `string[] argv`.

Em seguida, foi necessário inserir os métodos do Spawn na classe `Communicator`, já que as interfaces desses métodos devem ser mais simples que a interface oferecida pelo C e C++, e, ao mesmo tempo, permitem que o usuário utilize todos os parâmetros possibilitados pelo C e C++. Para isso, foram escritas cinco sobrecargas de método, a saber:

(i) `public Communicator Spawn (String command)`: permite ao usuário criar uma nova tarefa do programa `command`;
(ii) `public Communicator Spawn (String command, int maxprocs)`: permite que o usuário especifique a quantidade máxima de tarefas a serem criadas por meio do parâmetro `maxprocs`;
(iii) `public Communicator Spawn (String command, int maxprocs, int root)`: permite que o usuário especifique qual dos processos MPI.NET criará a nova tarefa pelo parâmetro `root` (nas outras chamadas o padrão é 0);
(iv) `public Communicator Spawn (String command, String[] argv, int maxprocs)`: permite que o usuário envie argumentos à nova tarefa;
(v) `public Communicator Spawn (String command, String[] argv, int maxprocs, int root)`: combina o conteúdo de todas outras chamadas.

Esses métodos retornam o novo comunicador via `return`, assim como no C++, porém diferentemente do C, em que o usuário deve alocar um novo comunicador e passá-lo como ponteiro para a função. Como o Spawn utilizado pela biblioteca é o provido pela interface MPI para C e não para C++, primeiramente, é alocado um novo comunicador nativo do MPI, o qual é passado como `out MPI_Comm newComm` para a interface C, lembrando-se que `out` especifica que a variável `newComm` será modificada pelo C. Após receber o novo comunicador, é necessário convertê-lo para um comunicador MPI.NET, o qual contém todos os métodos de comunicação. Para isso, um novo `Intercommunicator` é alocado e tem suas variáveis sobrescritas pelas informações contidas no comunicador MPI. Após essa operação, o novo comunicador MPI.NET é retornado e pode ser utilizado pelo usuário como qualquer outro comunicador MPI.NET.

As interfaces do Spawn para C e C++ são, respectivamente:

- int MPI_Comm_spawn (char *command, char *argv[], int maxprocs, MPI_Info info, int root, MPI_Comm comm, MPI_Comm *intercomm, int array_of_errcodes[]);
- Intercommunicator MPI::Comm.Spawn (char *command, char *argv[], int maxprocs, MPI_Info info, int root).

O parâmetro `array_of_errcodes[]` da interface para C não deve aparecer em interface C#, uma vez que é utilizado para retornar um vetor com os códigos dos erros que ocorreram. Em C#, os erros são tratados via exceção, por essa razão não faz sentido disponibilizar um vetor de códigos de erros para o usuário.

Algumas modificações devem ser feitas nos parâmetros passados pelo usuário nos métodos spawn. É necessário concatenar a string contida no parâmetro `command` com **\0** (para indicar ao C que é o fim da linha) e, após, convertê-la em um vetor de *bytes* que contém o código ASCII de cada um dos caracteres da string. Esse vetor de *bytes* deve ser protegido do coletor de lixo, pois esse desfragmenta constantemente a memória, ao mover os dados. Para protegê-lo do coletor de lixo, um objeto do tipo `GCHandle` (estrutura do .Net que permite que um objeto gerenciado seja acessado em memória não gerenciada) (Microsoft, 2007) deve ser criado, com uma das diversas inicializações para o `GCHandle`:

(i) *Weak:* permite controlar o objeto, porém ele pode ser coletado. Caso isso ocorra, o handler é zerado;
(ii) *WeakTrackResurrection:* similar ao weak, porém, se o objeto é ressuscitado (técnica que permite reutilizar um objeto finalizado) na finalização, ele o recupera;
(iii) *Normal:* protege o objeto de ser coletado, porém não permite recuperar seu endereço;
(iv) *Pinned:* similar ao normal, porém permite que o endereço da variável seja recuperado.

Para os dados tratados na biblioteca, o `GCHandle` deve ser inicializado com o vetor que contém os dados e o tipo `GCHandleType.Pinned`, permitindo, assim, que o C recupere o endereço das variáveis como se fossem um ponteiro, e, desse modo, proteja os dados do coletor de lixo. Após a função C ser chamada, o `GCHandle` é liberado e os dados voltam a ser gerenciados pelo coletor. O mesmo procedimento deve ser repetido para o `argv`.

A implementação do método `GetParent` foi realizada de maneira bem similar à implementação do Spawn, porém de forma mais simples, pois seu único parâmetro é o novo comunicador. Sua interface para C# ficou da seguinte maneira: `public static Communicator GetParent()`. As chamadas ao `MPI_Comm_spawn` e ao `MPI_Comm_get_parent` devem ser realizadas dentro de uma área demarcada como `unsafe` (modificador do C# que permite o uso de código não seguro) devido à utilização de ponteiros e ao fato de estar chamando uma biblioteca em C; enfatiza-se que isso tudo ocorre de forma transparente para o usuário da biblioteca.

5 Resultados experimentais

Foram realizadas medições de desempenho da função Spawn a fim de verificar o *overhead* introduzido pelo tratamento dos dados antes e depois da chamada MPI. Também foi testado o desempenho em relação ao MPI para C++, bem como avaliado o comportamento da biblioteca em execuções de benchmarks sintéticos.

As bibliotecas MPI testadas em conjunto com a biblioteca implementada foram: OpenMPI, LAM, e MPICH2. Dentre essas bibliotecas, a MPICH2 demonstrou melhor estabilidade durante a criação de muitas tarefas. Para isso, o gerenciador de processos smpd deve ser utilizado. As outras duas bibliotecas não conseguem criar muitas tarefas dinamicamente, pois acabam esgotando os recursos muito rapidamente. Por exemplo, ao testar-se a aplicação que calcula o i-ésimo número da sequência de Fibonacci na LAM e na OpenMPI, o maior número da sequencia calculado foi o 8º, utilizando um cluster de 8 máquinas e o tamanho de grão sequencial igual a 1. Já com o MPICH, foi possível calcular o Fibonacci de 20.

Primeiramente, foram realizadas diversas execuções de uma aplicação que faz o Spawn de uma tarefa MPI.NET vazia. A biblioteca foi modificada para capturar o tempo de execução da preparação dos dados até a chamada do Spawn, o que inclui a conversão e inicialização das variáveis, e a criação e inicialização dos `GCHandle`. Após, foi medida somente a chamada do Spawn. Para concluir foi medida somente a finalização dos dados, o que inclui a recuperação do comunicador e a liberação da memória.

É possível verificar que 92,5% do tempo de uma chamada Spawn na biblioteca é o tempo de execução da chamada Spawn em C. O *overhead* de tratamento dos dados em um Spawn ocupa 7,5% do tempo de execução; 6,5%, para a inicialização e 1%, para a finalização, ou seja, em um Spawn que leva 0,0426s para ser executado, 0,0028s é o tempo de inicialização, e 0,0004s é o tempo de finalização; ainda sobram 0,0394s, que são executados pela biblioteca MPI em C. Tendo em vista que esse tempo é fixo e, em geral, o Spawn é chamado para lançar ao menos

duas tarefas em modelos de D&C, esse tempo pode ser considerado curto.

No segundo teste, foi executado um benchmark batizado de Spawn-n, o qual cria n tarefas dinamicamente por meio de uma chamada da função Spawn, ou seja: `Spawn(command,n)`. Foram realizadas 50 execuções para cada caso. A plataforma de testes utilizada foi um cluster com a seguinte configuração:

- Número de nós alocados: 8;
- Processadores por Nó: 2 Pentium III 1266 Mhz;
- Memória por Nó: 512 Mb;
- Rede: Fast Ethernet;
- MPI: MPICH2 1.0.8p1;
- Máquina virtual .Net: Mono 2.4.

O benchmark compara o tempo que um programa C# e um C++ gastam para criar n processos MPI-C++ e MPI-C#. É importante verificar o tempo que a biblioteca MPI.NET leva para criar n processos C++, bem como o tempo que o MPI-C++ leva para criar n processos C#, pois, com essa comparação, é possível verificar o verdadeiro *overhead* introduzido pela biblioteca sobre a chamada Spawn. Os resultados do teste podem ser visualizados na Figura 3.

Foi possível visualizar que o tempo de inicializar um novo processo em C# é muito alto. Isso se deve ao fato do mono inicializar uma nova máquina virtual para cada processo C#, o que demanda um tempo considerável de processamento. No entanto, esse tempo de criação de uma nova máquina virtual pode se tornar pequeno quando o tamanho do grão do processo criado for exponencialmente maior que esse tempo.

Outra característica observada nesse teste é que a biblioteca não introduz *overhead* significativo na criação de processos. É possível visualizar que a criação de um processo C++ a partir da biblioteca utiliza praticamente o mesmo tempo que um programa MPI-C++ precisa para criar um processo C++, sendo majoritário o tempo gasto pela biblioteca MPI nativa para criar a tarefa.

Por fim, o gráfico demonstra que, ao criar processos MPI.NET a partir de um programa MPI-C++, os tempos são muito semelhantes aos de criação de um processo MPI.NET a partir da biblioteca. Novamente, o tempo da biblioteca MPI nativa é majoritário, fato a demonstrar que o problema de desempenho está na inicialização do novo processo C#.

O terceiro teste executado foi o cálculo do i-ésimo número da sequência de Fibonacci. Primeiramente, foram feitas execuções da versão sequencial para medir o *overhead* introduzido pelo uso do C# em relação ao C++, uma vez que a linguagem executa sobre máquina virtual. A plataforma utilizada foi um Intel T7250 com 2GB de memória para a execução do Fibonacci sequencial exibido na Figura 4. Por intermédio da Figura, é possível visualizar que o tempo de execução do programa em C# é bem similar ao tempo de execução do programa em C++.

Com base na diferença dos tempos de execução entre o C# e o C++ da versão sequencial do i-ésimo número da sequência de Fibonacci, foram feitas execuções paralelas, a fim de verificar o comportamento da biblioteca. O tamanho do grão (ponto a partir do qual se executa

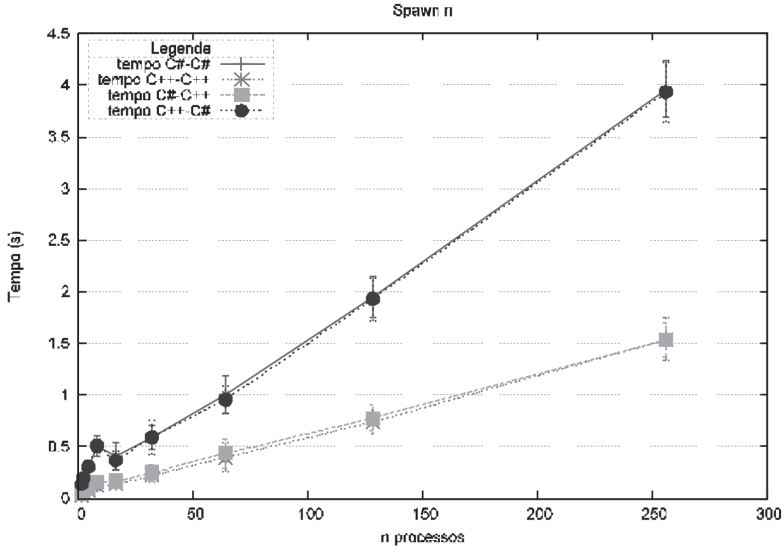

Figura 3. Execução do programa Spawn-n.
Figure 3. Program implementation Spawn-n.

sequencialmente) utilizado foi 38, devido ao tempo de execução do Fibonacci sequencial de 38 ser diversas vezes maior que o tempo de realização de um Spawn. Na Figura 5, foram plotados os dados da execução do Fibonacci paralelo C# e C++ e o Fibonacci sequencial em C++ para comparação. O speedup obtido em relação à versão sequencial em C++, para cada Fib(n), utilizando 16 processadores, é ilustrado na Figura 6. A plataforma de execução foi a mesma utilizada para o benchmark Spawn-n.

Os resultados demonstraram que a criação de um processo C# possui um *overhead* significativo em relação à criação de um processo C++. Combinado esse *overhead*, com a natureza dessa aplicação a qual cria muitos processos, o desempenho obtido foi bem inferior em relação ao desempenho de C++. No entanto, se a proporção grão do processo/número de processos fosse alterada, os resultados obtidos tenderiam a ser similares aos do teste sequencial. Porém, para isso, o grão deve ser grande o bastante para que o acúmulo de *overheads* da criação dos processos possa ser negligenciado.

Foram realizados testes de execução do Fibonacci de 50, variando o tamanho do grão de 39 até 50. Nesse teste, foi possível perceber que o tamanho de grão sequencial ideal para essa execução é em torno de 42, porque acima ou abaixo deste valor, o desempenho piora.

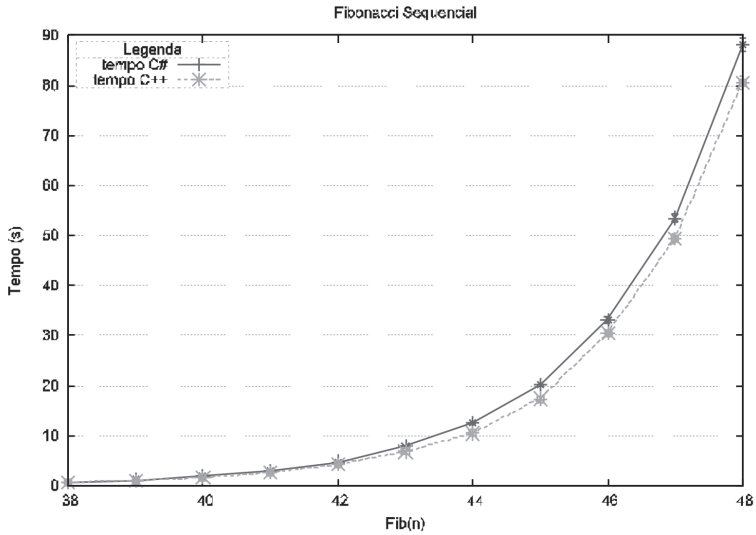

Figura 4. Execução do Fibonacci Sequencial.
Figure 4. The Fibonacci sequencial's implementation.

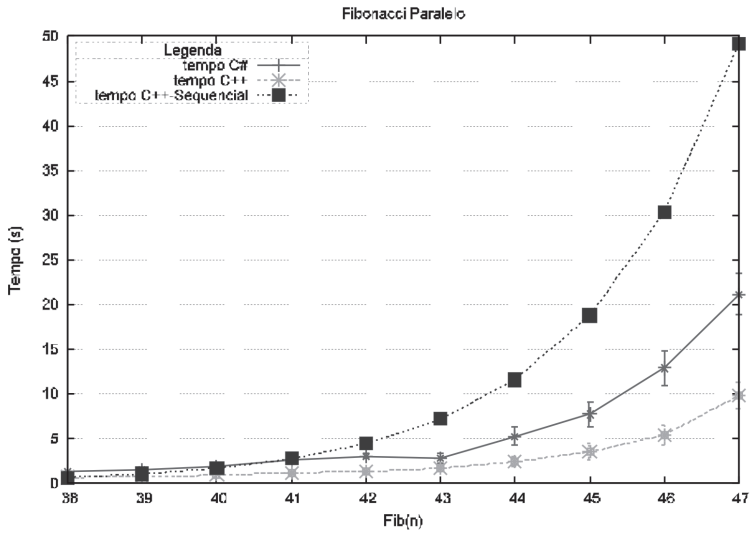

Figura 5. Execução do Fibonacci Paralelo.
Figure 5. Fibonacci parallel's implementation.

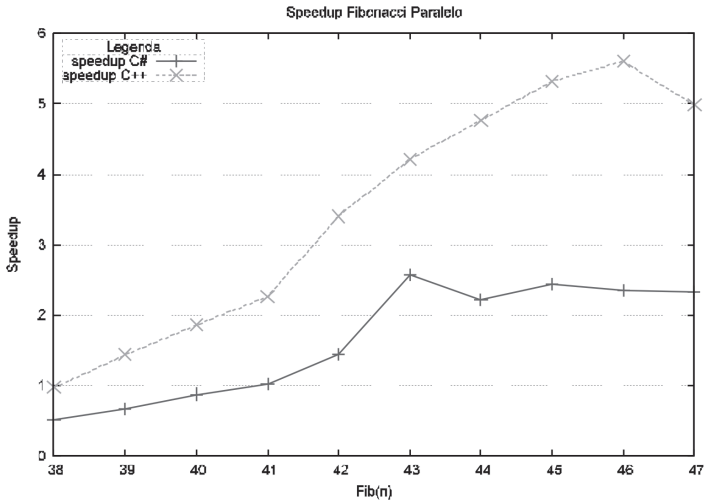

Figura 6. Speedup do Fibonacci Paralelo.
Figure 6. Fibonacci parallel's speedup.

As execuções com o grão de tamanho 38 ou menor não chegam ao final devido à grande quantidade de processos criados esgotar os recursos, em vista da quantidade de máquinas utilizadas.

6 Conclusões e trabalhos futuros

Este trabalho demonstrou que é viável suportar os recursos da norma MPI-2 em outras linguagens além de C, C++ e Fortran, oferecendo uma interface de programação com maiores níveis de abstração do que a API da norma.

Nos testes de desempenho realizados, foi possível concluir que o mecanismo de criação de processos oferecidos pela biblioteca não possui *overhead* significativo em relação a C++. No entanto, o tempo de criar um processo C# é bem maior que o necessário para criar um processo C++, evento que faz com que a biblioteca tenha obtido resultados ruins nos testes com o Fibonacci, devido ao acúmulo do *overhead* da criação de cada um dos processos, dada a grande quantidade de processos criados.

A escolha do tamanho do grão sequencial das tarefas é de grande importância, uma vez que um grão muito pequeno pode tornar a criação de tarefas dinamicamente inviável, em vista do custo de se criar uma nova tarefa.

Um dos interesses no uso do .Net é sua capacidade de abstrair a heterogeneidade da plataforma de execução (hardware e sistema operacional). Portanto, utilizar MPI.NET, facilita o porte de aplicações MPI em clusters heterogêneos. Com a criação dinâmica de tarefas, isso se aplica a Grids.

Quanto aos trabalhos futuros, o próximo passo a ser seguido é a otimização do mecanismo de criação de tarefas (usando otimizações da VM). Também será realizada a execução de novos benchmarks sintéticos para estudar o comportamento da biblioteca com outros tipos de aplicações, além de recursivas, bem como será testado, mais a fundo, o impacto do tamanho do grão sequencial nas execuções.

Referências

BAKER, M.; CARPENTER, B., FOX, G., KO, S.H.; LIM, S. 1999. Mpijava: An object oriented java interface to mpi. *In:* IPPS/SPDP'99 WORKSHOPS, XI, Puerto Rico, 1999. *Anais*... London, Springer-Verlag, p. 748-762.

CILIBRASI, R. 2001. MPI ruby. Disponível em: http://mpiruby.sourceforge.net/. Acessado em: 08/05/2009.

ECKEL, B. 2006. *Thinking in Java*. Prentice Hall PTR, 1150 p.

EL-REWINI, H.; ABD-EL-BARR, M. 2005. *Advanced computer architecture and parallel processing*. New York, Wiley-Interscience, 288 p.

FLANAGAN, D. 1998. *Java in a Nutshell*. Cambridge, 1254 p.

GETOV, V.; GRAY, P.; SUNDERAM, V. 1999. MPI and java-mpi: contrasts and comparisons of low-level communication performance. *In:* ACM/IEEE CONFERENCE ON SUPERCOMPUTING, Rodes, Greece, 1999. *Anais*... New York, ACM, p. 21 [CD-ROM].

GREGOR, D.; LUMSDAINE, A. 2008. Design and implementation of a highperformance mpi for c# and the common language infrastructure. *In:* ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, XIII, Salt Lake City, 2008. *Anais*... New York, ACM, p. 133-142.

GREGOR, D.; TROYER, M. 2003. Boost.mpi. Disponível em: http://www.boost.org/doc/libs/1370/doc/html/mpi.html#mpi.intro. Acessado em: 10/11/2008.

GROPP, W., THAKUR, R.; LUSK, E. 1999. *Using MPI-2: Advanced Features of the Message Passing Interface*. Cambridge, MIT Press, 406 p.

GUPTA, S. 2007. A performance comparison of windows communication foundation (wcf) with existing distributed communication technologies. Disponível em: http://msdn.microsoft.com/enus/library/bb310550.aspx. Acessado em: 30/11/2008.

JURIC, M.B.; KEZMAH, B.; HERICKO, M.; ROZMAN, I.; VEZOCNIK, I. 2004. Java rmi, rmi tunneling and web services comparison and performance analysis. *SIGPLAN Not.*, **39**(5):58-65.

KALIN, M. 2009. *Java Web Services: Up and running*. New York, O'Reilly Media Inc., 316 p.

KAMBADUR, P.; GREGOR, D.; LUMSDAINE, A.; DHARURKAR, A. 2006. Modernizing the C++ interface to MPI. *In:* EUROPEAN PVM/MPI USERS' GROUP MEETING, XIII, Bonn, Germany, 2006. *Anais*... London, Springer, p. 266-274.

LIANG, S. 1999. *Java Native Interface: Programmer's guide and reference*. Boston, Addison-Wesley Longman Publishing Co. Inc., 315 p.

McCANDLESS, B.C.; SQUYRES, J.M.; LUMSDAINE, A. 1996. Object-oriented MPI (oompi): A class library for the message passing interface. *In:* MPI DEVELOPERS CONFERENCE, II, Notre Dame, 1996. *Anais...* Washington, DC, IEEE Computer Society, p. 87-94.

MCMURTRY, C.; MERCURI, M.; WATLING, N.; WINKLER, M. 2008. *Windows communication foundation 3.5 unleashed, second edition*. SAMS, Carmel, IN, USA, 768 p.

MICROSOFT. 2007. Gchandle structure. Disponível em: http://msdn.microsoft.com/en-us/library/system.runtime.interopservices.gchandle.aspx. Acessado em: 16/06/2009.

MORIN, S.; KOREN, I.; KRISHNA, C.M. 2002. Jmpi: Implementing the message passing standard in java. *In:* INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM, XVI, Fort Lauderdale, 2002. *Anais*...Washington, DC, IEEE Computer Society, p. 191.

NESTER, C.; PHILIPPSEN, M.; HAUMACHER, B. 1999. A more efficient rmi for java. *In:* ACM CONFERENCE ON JAVA GRANDE, I, San Francisco, 1999. *Anais...* New York, ACM, p. 152-159.

PYM. 2005. pympi. Disponível em: http://pympi.sourceforge.net/. Acessado em 05/04/2009.

PMP. 2008. Pure mpi.net. Disponível em: http://www.purempi.net/. Acessado em: 14/11/2008.

RAMMER, I. 2002. *Advanced .Net Remoting*. Berkely, Apress, 608 p.

SAIFI, M.M.E. 2006. *Pmpi: uma implementação mpi multiplataforma, multilinguagem.* São Paulo, SP. Dissertação de mestrado. Universidade de São Paulo, 117 p.

SCHWARZKOPF, R.; MATHES, M.; HEINZL, S.; FREISLEBEN, B.; DOHMANN, H. 2008. Java rmi versus .net remoting architectural comparison and performance evaluation. *In:* INTERNATIONAL CONFERENCE ON NETWORKING, VII, Plymouth, 2008. *Anais...* Washington, DC, IEEE Computer Society, p. 398-407.

SNELL, Q.O.; MIKLER, A.R.; GUSTAFSON, J.L. 1996. Netpipe: A network protocol independent performace evaluator. *In:* IASTED INTERNATIONAL CONFERENCE ON INTELLIGENT INFORMATION MANAGEMENT AND SYSTEMS, VI, Bahamas, 1996. *Anais*.... Calgary, IASTED, [CD-ROM].

WENSHENG, T.W.Y.H.Y. 2000. Pjmpi: pure java implementation of mpi. *In:* HIGH PERFORMANCE COMPUTING IN THE ASIA-PACIFIC REGION, 4, Beijing, 2000. *Anais...* Los Alamitos, IEEE, **1**:14-17.

Submitted on June 5, 2009.
Accepted on June 12, 2009.