Estudo, Testes e Expansão do projeto MPI.NET

Fernando Afonso; Ismael Stangherlini; Nicolas Maillard
Federal University of Rio Grande do Sul
Informatics Institute
Bento Gonçalves Avenue, 9500 Porto Alegre, Brazil
{faafonso;istangherlini;nicolas}@inf.ufrgs.br

Resumo

MPI is considered an standard to write message passing applications in HPC. The modern programming languages are high-level object-oriented like C# and Java. Although MPI has bindings only for C, C++ and Fortran with a lack of high-level languages. Some extensions provide MPI support for such languages like Java (JavaMPI, MPIJava, PJMPI), Python (pympi), Ruby (ruby-mpi). The Open Systems Laboratory at Indiana University proposed a high-level C# binding called MPI.NET, which implementation is straightforward, and reasonable performance. However, the tests are partial and do not cover collective and non-blocking communications. This paper evalutes the MPI implementation of Indiana University for C# language through micro-benchmarks on collective and non-blocking communications.

1. Introdução

MPI *(Message Passing Interface)* é a especificação padrão da indústria para o desenvolvimento de aplicações de alta-performance com comunicação via troca de mensagens principalmente em ambientes com memória distribuída. Por se tratar de uma especificação e ser uma biblioteca e não uma linguagem de programação, existem diferentes implementações, porém implementações padrões e fortemente suportadas existem somente para as linguagens C, Fortran e C++ [3].

Novas linguagens de programação surgiram nos últimos anos, dentre elas está C#, uma linguagem de programação de alto nível orientada a objetos, criada sobre a plataforma .Net e baseada nas linguagens C++ e Java. A linguagem C# faz parte da plataforma .Net da Microsoft, a qual oferece suporte a diversas linguagens de programação, compilando o código dessas linguagens para uma linguagem intermediária chamada CLS *(Common Language Specification)*, garantindo a interoperabilidade entre essas linguagens. Os programas executam sobre uma máquina virtual chamada CLR *(Common Language Runtime)* [4].

O framework .Net juntamente com a linguagem C# permitem o desenvolvimento rápido de aplicações [4], fazendo com que C# seja uma linguagem de apelo aos programadores pela simplicidade de desenvolvimento das aplicações. Além disso, devido a flexibilidade da linguagem, também é possível acessar bibliotecas já existentes em C, tornando muito convidativo o uso das bibliotecas nativas de MPI.

Devido a grande aceitação do padrão MPI, existem inúmeros projetos que tratam da implementação de bibliotecas MPI para linguagens de alto nível ou que simplesmente realizam chamadas a bibliotecas já implementadas para C como por exemplo Java (JavaMPI, MPIJava, PJMPI) [1] [10], Python (pympi) e Ruby (ruby-mpi).

O objetivo desse trabalho é estudar, testar e verificar possibilidades de expansão para a biblioteca desenvolvida pela Universidade de Indiana que oferece suporte MPI sobre C#, chamada MPI.NET [2] [11]. Essa biblioteca possui algumas características que facilitam a programação como, por exemplo, a primitiva MPI_Send, a qual recebe como parâmetros a mensagem a ser enviada, o destino e o tag, diferentemente da biblioteca nativa para C a qual necessita de seis parâmetros. Dessa forma, essa primitiva pode transmitir tanto um objeto serializável como um vetor com diversos valores ou objetos sem a necessidade de passar nenhum parâmetro extra. Essas características são interessantes pois demonstram que, através da simplicidade de programação do C#, a interface com o usuário para o ambiente de programação MPI pode possuir um nível de abstração maior.

2. C#

C# é uma nova linguagem de alto nível orientada a objetos e interpretada. Essa linguagem é de grande apelo para programadores e empresas devido a seu nível de abstração elevado e por ser uma linguagem de desenvol-

vimento rápido. Ela foi criada combinando as melhores características de C++ e Java [6] [4].

C# é uma linguagem segura. Assim sendo, ponteiros e aritmética sem checagem só podem ser utilizados em um modo chamado *unsafe*. Objetos e variáveis são automaticamente desalocados pelo coletor de lixo quando não há mais referências para os mesmos. Além disso, existem finalizadores, porém, sua execução não é imediata. Para que objetos sejam desalocados instantâneamente existe uma interface chamada *Disposable* que, quando utilizada, permite que os objetos contidos dentro de um bloco delimitado por *using block* liberem seus recursos alocados instantâneamente quando não mais necessários [6].

Não é permitido utilizar herança múltipla, porém é possível implementar várias interfaces. É uma linguagem altamente tipada e diferencia-se do C++ pelo fato de não existirem ponteiros nulos, pela necessidade de conversões definidas pelo usuário serem marcadas explícitamente e pelo fator de que somente conversões de tipo seguras são implícitas. Em relação a organização, não existem variáveis ou funções globais. Dessa forma, todos os métodos devem estar definidos dentro de classes. Uma alternativa é utilizar métodos e variáveis estáticos dentro de classes públicas. Os programas são compilados em tempo de execução [6] [4].

Essas características de linguagem segura e suas semelhanças com Java fazem com que C# seja uma linguagem popular e esteja na mira dos projetos de programação de alto desempenho [9] [2].

3. MPI.NET

O projeto MPI.NET foi criado na Universidade de Indiana [2] com o intuito de combinar a alta performance de C com o alto nível de abstração de C# como classes, genéricos, checagem de limites, coletor automático de lixo, etc. Essas características, além de tornarem a programação mais simples através da abstração de conceitos, eliminam erros comuns de programação.

As implementações MPI para Java, por exemplo, fazem uso da biblioteca MPI sem manter o estilo de programação alto nível natural da linguagem [1], uma vez que simplesmente mapeiam as chamadas MPI nativas para dentro do ambiente de programação Java. No caso do MPI.NET, são realizadas as mesmas chamadas para o ambiente, porém de uma forma transparente ao usuário, que desconhece o baixo nível das chamadas MPI nativas, uma vez que são escondidos grande parte dos parâmetros que podem ser inferidos. Através desse nível de abstração, permite-se uma programação mais limpa, sem o uso de estruturas como, por exemplo, ponteiros.

Diferente do extremo visto em [10], artigo que propõe uma biblioteca MPI totalmente escrita em Java, MPI.NET não implementa uma nova biblioteca puramente C#. A implementação, na verdade, simplesmente aproveita o alto desempenho da biblioteca nativa e transparece ao usuário os aspectos de baixo nível, oferecendo um bom desempenho em um ambiente de programação que possui um consideravel nível de abstração.

Para implementar a biblioteca foi utilizado o suporte ao C oferecido pela plataforma .Net, sendo que o código deve ser marcado como unsafe. Além disso, para informar a biblioteca nativa que será acessada utiliza-se uma função da linguagem C# chamada DllImport a qual permite que o usuário acesse uma biblioteca escrita em C. Com isso é possível que a biblioteca MPI nativa utilizada seja alterada modificando-se algumas chamadas na classe Unsafe e recompilando o código. Isso se deve ao fato de a biblioteca MPI ser importada através da chamada DllImport, bastando modificar a biblioteca a ser chamada. A biblioteca MPI nativa utilizada foi a da Microsoft, a qual é baseada na biblioteca MPICH2 do Argonne National Laboratory [5] [7] [2].

Na biblioteca MPI.NET as primitivas MPI foram distribuídas dentro de classes. A classe mais comum é a Communicator, a qual possui as primitivas de comunicação. Para realizar as comunicações é necessário instanciar um objeto do tipo Communicator e fazer o acesso a essas primitivas através desse objeto.

As primitivas tiveram o número de parâmetros significativamente reduzido. Como exemplo, temos a primitiva Send que em MPI nativo possui seis parâmetros. Já na solução MPI.NET, ela passou a ter somente três. Esse resultado pôde ser obtido pela inferência dos demais parâmetros antes do chamamento da biblioteca nativa. Assim sendo, o usuário, através da primitiva Send, apenas informa a mensagem, o destino e um tag. A partir daí, a biblioteca descobre o tipo, a quantidade de dados e o canal de comunicação. Dessa forma, é possível perceber que a redução do número de parâmetros necessário para as primitivas simplifica bastante a programação e evita erros.

A implementação da primitiva Send se dá da seguinte maneira: primeiramente é verificado se o que está sendo enviado é um tipo primitivo, como, por exemplo, um inteiro. Caso seja um tipo primitivo, o Send da biblioteca MPI nativa é chamado e o tipo é enviado preenchendo o campo quantidade com 1 e o campo datatype com o tipo detectado. Caso não seja um tipo primitivo como, por exemplo, um objeto, então o mesmo é serializado através da classe BinaryFormatter. A partir daí, uma mensagem é enviada contendo o cabeçalho para desserialização e um tag interno. Após isso, a mensagem é enviada contendo um *stream* de *bytes* equivalente ao objeto serializado, a quantidade de *bytes* desse *stream*, o tipo MPI_BYTE, o destino, a tag do usuário e o comunicador.

As outras primitivas, inclusive as de comunicação co-

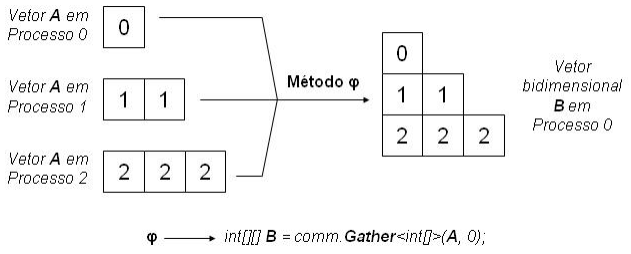

Figura 1. Figura ilustrando o método Gather

letiva, seguem o mesmo princípio encontrado na primitiva Send. A classe Communicator pode ser utilizada também para criar objetos das classes Intercommunicator e Intracommunicator. Dessa forma, os comunicadores são objetos e as primitivas de comunicação são métodos desses objetos.

O usuário pode utilizar qualquer primitiva de comunicação para enviar objetos, com a condição que sejam de classes serializáveis. As primitivas para comunicação coletiva também tiveram o número de parâmetros reduzido, utilizando as propriedades da linguagem para preencher os campos ocultos. As comunicações não-bloqueantes possuem um tipo Request no Receive, o qual deve ser testado para verificar se a comunicação foi realizada com sucesso. Se obteve-se sucesso na comunicação, então é possível obter a mensagem transmitida através do objeto Request.

A figura 1 ilustra a simpliciade de utilização do método Gather. No MPI.NET, o usuário simplesmente utiliza uma matriz de qualquer tamanho para receber os diversos vetores obtidos em uma função Gather. Além disso, cada linha da matriz será inicializada corretamente de acordo com o tamanho do vetor que lhe for repassado, economizando espaço em memória e oferecendo uma solução elegante para o programador, uma vez que o mesmo não precisa se preocupar com o tamanho dos vetores que irá transmitir/receber.

O artigo [2] trata de diversos aspectos do projeto e detalha a implementação. Ele também detalha as comunicações coletivas e como a primitiva MPI_Reduce foi implementada. Porém, os testes de desempenho presentes no artigo estão muito pobres e não apresentam comparações detalhadas entre as primitivas C versus C#. Segundo o artigo, os resultados de desempenho obtidos foram muito bons e é satisfatória a utilização da linguagen C# para o desenvolvimento de aplicações MPI, por ser uma linguagem de alto nível.

4. Testes e possíveis expansões

Foram realizados alguns testes sobre a biblioteca MPI.NET, sendo alguns deles para verificar a funcionalidade das comunicações e outros para verificar o desempenho comparado a biblioteca MPI para linguagem C. A biblioteca utilizada nos testes foi a da Microsoft sendo a mesma utilizada para C#.

Primeiramente testamos as primitivas Send/Receive. Após isso, partimos para as primitivas relativas a comunicação coletiva, obtendo êxito em todos os testes. Em seguida, testamos as comunicações não-bloqueantes, que também demonstraram êxito. Realizamos também alguns testes através do envio de objetos, verificando a simplicidade de realização essa operação. Implementamos também todos os programas presentes em [8], obtendo êxito nos testes.

Após verificar que tudo estava funcionando, resolvemos passar para alguns testes de desempenho em comparação a linguagem C. No primeiro teste fizemos um programa simples em MPI-C e MPI C#. O programa utiliza 2 processos. Como definição, ele realiza um loop de 10000000 iterações, sendo que em cada iteração o processo de rank 0 realiza o Send de um inteiro para o processo de rank 1 e faz um Receive de um inteiro deste último processo. Similarmente, o processo de rank 1 faz um Receive de um inteiro do processo de rank 0 e um Send de um inteiro para este último processo. Os resultados encontrados foram satisfatórios, uma vez que o tempo médio das execuções em C foi de 41s e em C# de 46s. Ou seja, mostrou-se que o C# executou somente 12% mais lento.

Para o segundo teste modificamos o programa substituindo o inteiro por um vetor com 10 inteiros. Nesse teste, a média de valores encontrados para o programa em C ficou em 45s e a média para o C# ficou em 56s. Ou seja, os resultados mostraram que a execução foi 25% mais lenta, o que mostra que o overhead introduzido pela biblioteca começou a se tornar significativo. Já para os testes com um vetor de 1000 inteiros o tempo do C# ficou bem próximo do tempo do C, porém ambas linguagens variando muito o tempo de execução entre 110s e 80s e entrelaçando os resultados.

Em seguida desenvolvemos uma aplicação para testar como as linguagens se portam com vários processos em uma mesma máquina. A aplicação segue enviando um inteiro de um processo de rank i para o outro de rank i + 1. Ao se chegar no processo de último rank, ele envia a mensagem para o processo de rank 0, completando o ciclo. Assim sendo, foi desenvolvida um programa que implementava um loop com 10000 voltas no ciclo. Com 20 processos o programa em C levou 15s em média para completar as 10000 voltas e o programa em C# levou uma média de 20s, ou seja, 33% mais lento. Para esse teste medimos também a ocupação de memória. Como resultado, obtivemos que cada processo C ocupou em média 2200kb. Já os processos C# ocuparam em média 3300kb cada. Para o teste com 40 processos em C a execução levou em média 71s e em C#105s, ou seja, 48% mais lento.

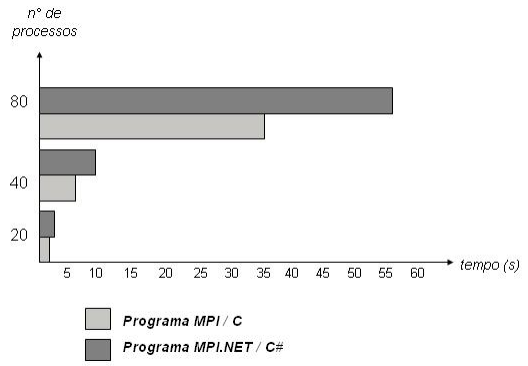

Figura 2. Figura ilustrando o aumento de tempo de acordo com o número de processos

Como os tempos de execução estavam ficando consideravelmente altos, resolvemos diminuir o laço para 1000 iterações. Os valores médios que obtivemos foram 2s com 20 processos para ambas linguagens, 10s para C# e 7s para C com 40 processos e 56s para C# e 35s para C com 80 processos, demonstrando que conforme cresce o número de processos maior a diferença de desempenho entre C# e C. Essa crescimento da diferença de tempo entre as duas linguagens pode ser vista na figura 2. Embora a linguagem C# demonstre uma queda de desempenho significativa nessas condições elas provavelmente não irão ocorrer em um programa real pois os diversos processos seriam distribuídos entre diversas máquinas.

Dentre as diversas primitivas MPI percebemos que as primitivas MPI_Bsend, MPI_BUFFER_ATTACH e MPI_BUFFER_DEATTACH não se encontram implementadas. Estamos estudando a melhor maneira de implementar essas primitivas dentro da biblioteca MPI.NET.

5. Conclusões

A biblioteca MPI se mostrou a alternativa ideal para o desenvolvimento de aplicações de alto desempenho que sejam baseadas no modelo SPMD *(Single Program Multiple Data)*. Por ser uma biblioteca tão popular acaba chamando a atenção dos desenvolvedores das mais diversas linguagens de programação que acabam por tentar oferecer suporte a ela.

Embora diversos projetos tentem transportar MPI para linguagens de programação de alto nível, a maioria não obteve êxito. Isso se deve tanto pelo fato do desempenho ser ruim como pelo fato de não mesclar o alto nível da linguagem com o MPI, oferecendo somente acesso às primitivas MPI em C.

O projeto MPI.NET se mostrou promissor, tendo resultados satisfatórios nos testes de desempenho, uma vez que se espera que uma linguagem que execute sobre uma máquina virtual tenha um desempenho bem inferior a linguagens nativas.

O modelo de programação oferecido por essa biblioteca foi identificado como o ponto mais brilhante desse projeto, pois a biblioteca mescla com harmonia as características do modelo de programação alto-nível da linguagem C# com a biblioteca MPI para C da Microsoft sem transparecer para o usuário os detalhes que ocorrem por baixo das chamadas.

Por fim, podemos concluir que este é um projeto promissor pois trabalhar com essa linguagem para desenvolver programas complexos facilita e organiza a programação, além disso, a orientação a objetos e a transparência quanto a estruturas de baixo nível oferecida por essa linguagem facilitam a organização do código.

Para trabalhos futuros pretendemos implementar as primitivas MPI_Bsend, MPI_BUFFER_ATTACH e MPI_BUFFER_DEATTACH dentro da biblioteca MPI.NET e estudar o modelo orientado a objetos para vermos o que mais poderia ser feito para aproveitar da melhor maneira possível esse paradigma de programação dentro da área de alto desempenho.

Referências

[1] 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.

[2] 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.

[3] W. Gropp. *Using mpi : portable parallel programming with the message-passing interface*. Cambridge : Mit, 1995.

[4] http://msdn.microsoft.com/en-us/library/aa645597(VS.71).as px.

[5] http://msdn.microsoft.com/en us/library/bb524831(VS.85).aspx.

[6] http://msdn.microsoft.com/pt br/library/yyaad03b.aspx.

[7] http://www.mcs.anl.gov/research/projects/mpich2/.

[8] http://www.osl.iu.edu/research/mpi.net/documentation/tutorial/.

[9] http://www.parallelcsharp.com/.

[10] T. W. Y. H. Y. WenSheng. Pjmpi: pure java implementation of mpi. In *High Performance Computing in the Asia-Pacific Region*, volume 1, pages 533 – 535, 2000.

[11] J. Willcock, A. Lumsdaine, and A. Robison. Using mpi with c# and the common language infrastructure. In *JGI '02: Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, pages 238–238, New York, NY, USA, 2002. ACM.