UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

FERNANDO ABRAHÃO AFONSO

MPI2.NET: Criação Dinâmica de Tarefas com Orientação a Objetos

Dissertação apresentada como requisito parcial
para a obtenção do grau de
Mestre em Ciência da Computação

Prof. Dr. Nicolas Maillard
Orientador

Porto Alegre, maio de 2010

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Afonso, Fernando Abrahão

MPI2.NET: Criação Dinâmica de Tarefas com Orientação a Objetos / Fernando Abrahão Afonso. – Porto Alegre: PPGC da UFRGS, 2010.

77 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2010. Orientador: Nicolas Maillard.

1. Criação Dinâmica de Tarefas. 2. MPI. 3. Processamento Paralelo. 4. Programação de Alto Desempenho. I. Maillard, Nicolas. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Neto
Pró-Reitor de Coordenação Acadêmica: Prof. Pedro Cezar Dutra Fonseca
Pró-Reitor Adjunto de Pós-Graduação: Prof. Aldo Bolten Lucion
Diretor do Instituto de Informática: Prof. Flávio Rech Wagner
Coordenador do PPGC: Prof. Álvaro Freitas Moreira
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

“The pessimist complains about the wind;
The optimist expects it to change;
The realist adjusts the sails.”
— WILLIAM GEORGE WARD

AGRADECIMENTOS

Primeiramente agradeço a meus pais por todo apoio e incentivo que me deram. Agradeço a minha esposa Manoella por ter me apoiado e aguentado firme ao meu lado durante essa jornada.

Agradeço, em especial, meu orientador Nicolas Maillard por todo apoio, orientação e especialmente confiança depositados em mim. Suas orientações foram fundamentais para meu crescimento pessoal e profissional.

Agradeço, com carinho muito especial, meus colegas Alexandre, Cláudio, Cristian, João e Stéfano que passaram a ser grandes amigos. O apoio e incentivo oferecidos por eles foram fundamentais para que essa jornada se concretiza-se.

Por fim, agradeço ao CNPq pelo apoio financeiro.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
BSL	*Boost Serialization Library*
CLR	*Common Language Runtime*
E/S	*Entrada e Saída*
JIT	*Just-In-Time*
JNI	*Java Native Interface*
MPI	*Message Passing Interface*
MPMD	*Multiple Program Multiple Data*
MSIL	*Microsoft Intermediate Language*
NFS	*Network File System*
NIS	*Network Information Service*
RMI	*Remote Method Invocation*
SPMD	*Single Program Multiple Data*
STL	*Standard Template Library*
TPL	*Task Parallel Library*
TSP	*Travelling Salesman Problem*
URL	*Uniform Resource Locator*
WCF	*Windows Communication Foundation*

LISTA DE FIGURAS

RESUMO

Message Passing Interface (MPI) é o padrão *de facto* para o desenvolvimento de aplicações paralelas e de alto desempenho que executem em *clusters*. O padrão define APIs para as linguagens de programação Fortran, C e C++. Por outro lado a programação orientada a objetos é o paradigma de programação dominante atualmente, onde linguagens de programação como Java e C# têm se tornado muito populares. Isso se deve às abstrações voltadas para facilitar a programação oriundas dessas linguagens de programação, permitindo um ciclo de programação/manutenção mais eficiente. Devido a isso, diversas bibliotecas MPI para essas linguagens emergiram. Dentre elas, pode-se destacar a biblioteca MPI.NET, para a linguagem de programação C#, que possui a melhor relação entre abstração e desempenho. Na computação paralela, o modelo utilizado para o desenvolvimento das aplicações é muito importante, sendo que o modelo Divisão & Conquista é escalável, aplicável a diversos problemas e permite a execução eficiente de aplicações cuja carga de trabalho é desconhecida ou irregular. Para programar utilizando esse modelo é necessário que o ambiente de execução suporte dinamismo, o que não é suportado pela biblioteca MPI.NET. Desse cenário emerge a principal motivação desse trabalho, cujo objetivo é explorar a criação dinâmica de tarefas na biblioteca MPI.NET. Ao final, foi possível obter uma biblioteca com desempenho competitivo em relação ao desempenho das bibliotecas MPI para C++.

Palavras-chave: Criação Dinâmica de Tarefas, MPI, Processamento Paralelo, Programação de Alto Desempenho.

MPI2.NET: Dynamic Tasks Creation with Object Orientation

ABSTRACT

Message Passing Interface (MPI) is the *de facto* standard for the development of high performance applications executing on clusters. The standard defines APIs for the programming languages Fortran C and C++. On the other hand, object oriented programming has become the dominant programming paradigm, where programming languages as Java and C# are becoming very popular. This can be justified by the abstractions contained in these programming languages, allowing a more efficient programming/maintenance cycle. Because of this, several MPI libraries emerged for these programming languages. Among them, we can highlight the MPI.NET library for the C# programming language, which has the best relation between abstraction and performance. In parallel computing, the model used for the development of applications is very important, and the Divide and Conquer model is efficiently scalable, applicable to several problems and allows efficient execution of applications whose workload is unknown or irregular. To program using this model, the execution environment must provide dynamism, which is not provided by the MPI.NET library. From this scenario emerges the main goal of this work, which is to explore dynamic tasks creation on the MPI.NET library. In the end we where able to obtain a library with competitive performance against MPI C++ libraries.

Keywords: Dynamic Tasks Creation, High Performance Computing, MPI, Parallel Computing.

1 INTRODUÇÃO

A computação paralela tem sido utilizada em diversas áreas para o desenvolvimento de aplicações científicas e comerciais que demandem grande poder computacional. A arquitetura das plataformas paralelas sofreu diversas modificações ao longo dos anos, partindo de máquinas vetoriais até *clusters* e mais recentemente *grids* e processadores *multi-core*. Um *cluster* ou agregado de computadores (NAVAUX; ROSE, 2003) consiste de uma plataforma composta por PCs, os quais possuem um ou mais processadores, interconectados por rede de alto desempenho e estão fisicamente próximos. Um *cluster* pode ser conectado a outros *clusters* formando assim um *cluster* de *clusters*. Essas arquiteturas permitem que programas desenvolvidos para elas obtenham ganhos de desempenho significativos.

No entanto, programar para arquiteturas paralelas não é uma tarefa trivial, demandando um nível técnico elevado. Uma vez que *clusters* utilizam memória distribuída, é necessário um padrão de programação que explore tanto a comunicação quanto a concorrência. O padrão *de facto* para o desenvolvimento de aplicações paralelas para *cluster* é a norma *Message Passing Interface* (MPI). MPI é uma API para programação baseada em troca de mensagens, a qual, possui especificada como seus mecanismos devem se comportar em qualquer distribuição (GROPP et al., 1996). Através dela, programas podem ser divididos em processos distribuídos, os quais executam em paralelo, e se comunicam através das funções contidas nela. Para desenvolver programas MPI podem ser utilizados diversos modelos de programação, por exemplo:

- *Bag of Tasks*: Tarefas que não possuem dependências entre si são executadas concorrentemente;
- *Bulk Synchronous Parallel*: Faz uso de *supersteps* globais, os quais se subdividem em três etapas: computação paralela, comunicação e sincronização;
- *Pipeline*: As tarefas são divididas em partes, onde os dados de entrada são colocados no primeiro processador e repassados entre os processadores após a execução de cada parte;
- Mestre/Escravo: O mestre distribui para os escravos o trabalho a ser executado. Uma vez executado, o escravo retorna o resultado e pede mais trabalho ao mestre;
- Divisão & Conquista: Uma tarefa inicial se subdivide em outras tarefas, as quais se subdividem recursivamente, e de forma paralela, até alcançarem um tamanho de problema trivial, resolvendo-o e combinando os resultados paralelamente até obter-se a solução para o problema original.

Divisão & Conquista pode ser utilizado para programar uma grande gama de algoritmos. Ele permite que programas se adaptem a maiores ou menores quantias de recursos computacionais sem a necessidade de modificar o código ou com modificações mínimas. Nesse modelo, as comunicações são limitadas, sendo que uma tarefa irá comunicar somente com as tarefas criadas por ela e com a tarefa que a criou, simplificando o desenvolvimento e diminuindo a probabilidade de *bugs* e *deadlocks*. Além disso, esse modelo permite executar de maneira eficiente aplicações nas quais a carga de trabalho é conhecida somente em tempo de execução ou onde a carga de trabalho é irregular entre as diferentes tarefas, uma vez que permite dividir as tarefas até que seja atingido o caso trivial, e alocá-las de forma homogênea sobre os recursos computacionais. A criação dinâmica de tarefas pode ser utilizada como uma maneira de programar esse modelo em sistemas paralelos de maneira a manter seu código simples e similar ao sequencial.

MPI-2 oferece suporte a criação dinâmica de processos, podendo ser utilizada para o desenvolvimento de algoritmos Divisão & Conquista com criação dinâmica de tarefas (as quais são mapeadas para processos). Uma vez que a norma MPI especifica APIs para as linguagens de programação Fortran, C e C++, somente estas linguagens de programação possuem implementações de bibliotecas MPI consolidadas e amplamente suportadas. Projetos de bibliotecas MPI para linguagens de programação como Java e C# em geral possuem uma API com maiores níveis de abstração, diminuindo a quantidade de detalhes com os quais o programador deve se preocupar, e, por consequência, aumentando a produtividade e diminuindo erros de programação. Além disso, é possível tirar proveito das características presentes nessas linguagens de programação, como *Reflection*, genéricos, gerência automática de memória e serialização automática (BUDD, 2001).

A biblioteca MPI.NET (GREGOR; LUMSDAINE, 2008), escrita na linguagem de programação C#, apresenta uma combinação de abstração/desempenho permitindo que as funcionalidades MPI sejam utilizadas de forma mais simples do que na API MPI para C++, mantendo desempenho similar. Além disso, essa biblioteca permite que objetos sejam comunicados da mesma forma que tipos primitivos, algo que não é suportado pela norma MPI para C++ e é importante ao programarmos com orientação a objetos. A MPI.NET implementa as funcionalidades contidas na norma MPI-1. No entanto a criação dinâmica de processos é especificada na norma MPI-2.

Este trabalho propõe explorar a criação dinâmica de processos da norma MPI-2 na biblioteca MPI.NET. Ele descreve a implementação desses mecanismos na biblioteca MPI.NET, bem como propõe e implementa um algoritmo voltado para otimizar o desempenho desse mecanismo nessa biblioteca. O modelo de programação Divisão & Conquista foi utilizado para desenvolver as aplicações de teste da biblioteca proposta. A principal contribuição deste trabalho é uma **biblioteca de alto desempenho voltada para a criação dinâmica de processos na plataforma .NET**.

1.1 Objetivo

O principal objetivo do presente trabalho é investigar a programação MPI com criação dinâmica de processos na biblioteca MPI.NET. A concretização esperada é uma biblioteca que possua criação dinâmica de processos através de uma API simples e que possua desempenho similar ao oferecido pelo MPI em C++.

1.2 Estrutura do Texto

O Capítulo 2 trata sobre a norma MPI e seu contexto em diversas linguagens de programação orientadas a objetos. Nesse capítulo é realizada uma discussão sobre o modelo de programação com MPI, suas inovações ao longo dos anos, e as melhorias propostas para a norma MPI para a linguagem de programação C++. Além disso são demonstrados diversos projetos de bibliotecas MPI para linguagens de programação orientadas a objetos não suportadas pela norma MPI, enfatizando as melhorias que elas propõem para a API MPI.

Logo após, o Capítulo 3 trata dos mecanismos básicos do Framework .NET. A máquina virtual bem como o gerenciamento de memória são explorados, além de diversas outras ferramentas voltadas para abstração. O capítulo também discorre sobre as ferramentas para programação paralela e distribuída contidas no Framework .NET, deixando claro porque é necessário uma biblioteca MPI para o mesmo.

Na sequência, o Capítulo 4 trata do mecanismo proposto pelo presente trabalho, discutindo seu projeto e implementação na biblioteca MPI.NET. Além disso é discutida a importância da criação dinâmica de processos. Por fim é realizada uma breve avalização de desempenho do mecanismo proposto.

O Capítulo 5 apresenta o algoritmo proposto para otimizar o desempenho do mecanismo proposto pelo presente trabalho, sua implementação e seu desempenho. Também são discutidas soluções alternativas para otimizar o desempenho.

Em seguida, o Capítulo 6 apresenta brevemente os *benchmarks* sintéticos utilizados para medir o desempenho do mecanismo de criação dinâmica de processos proposto. É demonstrado como foi realizada a implementação dos *benchmarks* bem como o que cada um deles testa. Após são demonstrados e discutidos os resultados obtidos nas execuções dos *benchmarks* sintéticos.

Por fim, no Capítulo 7 são apresentadas as conclusões finais desse trabalho, as publicações e as propostas de trabalhos futuros.

2 CONTEXTO CIENTÍFICO: MPI E ORIENTAÇÃO A OBJETOS

Este capítulo discute a programação paralela orientada a objetos com MPI. É introduzida a evolução da norma MPI, bem como o suporte a suas funcionalidades em linguagens de programação orientadas a objetos. São apresentados os conceitos básicos sobre a programação com MPI, bem como alguns exemplos de programas MPI. Também são vistos alguns trabalhos que propõem modificações à API MPI para C++ a qual é discutida por diversos trabalhos.

2.1 A Norma MPI

Message Passing Interface (MPI) é uma especificação de biblioteca para o desenvolvimento de aplicações paralelas, baseadas no modelo troca de mensagens (forma de comunicação que consiste no envio e recebimento de mensagens). Inicialmente desenvolvida para as linguagens de programação C e Fortran, a norma MPI é mantida pelo *MPI Forum* (FORUM, 1994), um grupo composto por representantes de diversas organizações.

A norma MPI define uma API, a qual é, implementada por diversas bibliotecas (ou distribuições) MPI livres ou proprietárias. Inicialmente, a norma definiu API para as linguagens de programação C e Fortran. Mais tarde, em suas versões 1.2 e 2, foi adicionado o suporte à linguagem de programação C++.

São definidas diversas funções de comunicação (ponto-a-ponto, coletivas, criação e gerência de grupos, entre outros), sendo que, toda comunicação entre diferentes processos é realizada através da troca de mensagens. Os modelos de programação utilizados no desenvolvimento de aplicações MPI podem ser dos tipos *Single Program Multiple Data* (SPMD) (DAREMA et al., 1988) ou *Multiple Program Multiple Data* (MPMD) (FLYNN, 1972) (FORUM, 1997).

Na norma MPI 1.2 foi introduzida a API MPI para a linguagem de programação C++. Através dela passou a ser possível programar MPI em C++ com orientação a objetos. Ela é discutida por diversos trabalhos dos quais alguns serão discorridos em 2.2.3.1, uma vez que autores acreditam que ela não é adequada ao modelo de programação da linguagem C++.

2.1.1 A Norma MPI 2

Esta extensão da norma MPI apresentou mudanças significativas em relação às versões anteriores, incluindo novas funcionalidades. Elas são divididas em: entrada e saída (E/S) paralela, operações remotas de memória, criação dinâmica de processos e suporte a múltiplas *threads* (GROPP; LUSK; THAKUR, 1999).

2.1.1.1 E/S Paralela

Além das operações básicas de arquivos poderem ser executadas em paralelo, foram incluídas operações avançadas para permitir acesso não contíguo à memória e disco, E/S não bloqueante, ponteiros de arquivos individuais e compartilhados, operações de E/S coletivas. Dessa forma é possível que programas realizem operações paralelas em arquivos.

2.1.1.2 Operações em Memória Remota

Com essa funcionalidade é possível utilizar o modelo de memória compartilhada em ambientes MPI. Esse modelo permite que processos criem e exponham janelas de memória compartilhada, as quais podem ser acessadas por outros processos sem a necessidade de intervenções do hospedeiro.

2.1.1.3 Criação Dinâmica de Processos

Essa funcionalidade permite que programas MPI criem e estabeleçam conexão com novos processos MPI, em tempo de execução. Foram adicionadas funcionalidades para permitir que seja estabelecida comunicação com processos MPI já existentes e funções que permitem criar dinamicamente processos de um ou mais programas MPI.

A programação com criação dinâmica de processos facilita o paralelismo de algoritmos do tipo Divisão & Conquista (CORMEN et al., 2001), uma vez que cada divisão da entrada pode ser associada a criação de novos processos, que realizarão o processamento daquela parte dos dados. Além disso, a criação dinâmica de processos permite melhorar o balanceamento de carga em aplicações com carga de trabalho irregular, uma vez que distribui os processos entre os recursos em tempo de execução, uniformizando a carga do ambiente de execução.

No modelo de programação MPI com criação dinâmica de processos, um ou mais processos criam dinamicamente novos processos. Os programas utilizados para resolver um problema podem ser divididos de acordo com a tarefa que realizam, por exemplo, para executar uma ordenação mergesort, podem ser criados três programas: um para fazer a leitura e saída dos dados, um responsável por particionar os dados e o terceiro responsável por ordenar as partes. Esse modelo não é obrigatório mas aumenta a reusabilidade de código uma vez que podem ser reaproveitadas determinados programas para resolver outro problema, e além disso, permite uma melhor organização do código, uma vez que o separa em programas que executam determinada tarefa.

2.1.1.4 Suporte a Múltiplas Threads

Em MPI-2 os usuários podem especificar o nível de segurança desejado entre *threads*. Dessa forma a biblioteca MPI passa a saber o nível de segurança de *threads* a ser aplicado entre funções MPI a fim de evitar condições de corrida e manter o desempenho adequado conforme a aplicação do usuário. Os níveis de segurança de *threads* são os seguintes:

- MPI_THREAD_SINGLE: Somente uma *thread* irá executar;
- MPI_THREAD_FUNNELED: Pode haver múltiplas *threads*, no entanto, somente a *thread* principal irá realizar chamadas MPI;
- MPI_THREAD_SERIALIZED: Pode haver múltiplas *threads*, e todas podem chamar funções MPI, porém, uma por vez, necessitando sincronização por parte do usuário;

- MPI_THREAD_MULTIPLE: Pode haver múltiplas *threads* chamando as funções MPI indiscriminadamente.

2.1.2 Programação MPI em linguagem C

Nessa subseção serão vistos alguns exemplos de programação MPI. A linguagem de programação adotada pelos exemplos é a linguagem C, porém os conceitos sobre o modelo de programação se mantém para as outras linguagens.

Cada processo MPI é identificado por um *rank* (sendo esse um número único). A biblioteca MPI atribui a cada processo (durante sua inicialização) esse *rank* (do tipo `int`) e um comunicador. O *rank* é sequencial, inicializando em zero. Através do *rank* o usuário pode especificar a origem e o destino de mensagens e também realizar tarefas distintas em cada processo.

No início da execução, um processo MPI possui dois comunicadores, o `MPI_COMM_WORLD` (contendo o processo e seus irmãos) e o `MPI_COMM_SELF` (contendo somente o próprio processo). Um comunicador define o universo de comunicação entre os processos, sendo que é possível criar novos comunicadores e agrupar processos neles. Um processo pode ter *ranks* diferentes em diferentes comunicadores. Para descobrir qual seu *rank* em determinado comunicador o processo deve acessar a função `MPI_Comm_rank()`, passando para ela, como argumento, o comunicador no qual deseja obter seu *rank* . Tipicamente os processos são agrupados de acordo com a tarefa que irão realizar, sendo essa uma forma (não obrigatória) do programador organizar seu programa.

Para realizar a troca de uma mensagem ponto-a-ponto, a função `MPI_Send()` e suas variações são utilizadas para realizar o envio da mensagem. Ela possui a seguinte interface: `int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`. Os parâmetros dessa função são:

1. `void *buf`: Os dados a serem enviados;
2. `int count`: A quantidade de dados a serem enviados;
3. `MPI_Datatype datatype`: O tipo dos dados que estão sendo enviados;
4. `int dest`: O destino para qual os dados estão sendo enviados;
5. `int tag`: Uma tag para o programador diferenciar as mensagens;
6. `MPI_Comm comm`: O comunicador que será utilizado para o envio da mensagem.

Já para realizar o recebimento da mensagem são utilizadas a função `MPI_Recv()` e suas variações. Sua interface é a seguinte:

1. `void *buf`: Os dados a serem recebidos;
2. `int count`: A quantidade de dados a serem recebidos;
3. `MPI_Datatype datatype`: O tipo dos dados que estão sendo recebidos;
4. `int dest`: A origem da qual os dados estão sendo recebidos;
5. `int tag`: Uma tag para o programador diferenciar as mensagens;

6. `MPI_Comm comm`: O comunicador que será utilizado para o recebimento da mensagem.

7. `MPI_Status *stat`: Uma estrutura contendo diversas informações sobre a mensagem.

A estrutura `MPI_Datatype` contém os tipos de dados pré-definidos pela norma. É por meio dessa primitiva que uma biblioteca MPI distingue o tipo de dado que está sendo transmitido. Por exemplo, ao transmitir um inteiro, passa-se como argumento `MPI_INT` como `datatype`. Dados serializados (convertidos de um tipo definido pelo usuário para um buffer de *bytes*) podem ser transmitidos como dados do tipo `MPI_Byte`. Um exemplo básico de um programa MPI escrito em C pode ser visto no programa 1. Ao ser executado com **n** processo o programa irá fazer com que, com exceção do processo de *rank* 0, cada processo receba um inteiro do processo 0 e logo após envie um inteiro para o processo 0. Já o processo 0 irá enviar e, em seguida receber um inteiro de cada um dos outros processos.

Programa 1 Um exemplo de ping-pong em MPI.

```
int main(int argc, char * argv[])
{
   int i, r, p, mensagem;
   MPI_Status stat;

   // INICIALIZA A BIBLIOTECA MPI
   MPI_Init(&argc, &argv);
   // DESCOBRE O RANK DO PROCESSO
   MPI_Comm_rank(MPI_COMM_WORLD, &r);
   // DESCOBRE A QUANTIDADE DE PROCESSOS
   MPI_Comm_size(MPI_COMM_WORLD, &p);

   if (r==0)
   {
     for(i=0; i<p; i++)
     {
       // ENVIA UM INTEIRO PARA O PROCESSO DE RANK i
       MPI_Send(1, 1, MPI_INT, i, 5, MPI_COMM_WORLD);
       // RECEBE NA VARIÁVEL MENSAGEM UM INTEIRO DO PROCESSO DE RANK i
       MPI_Recv(&mensagem, 1, MPI_INT, i, 10, MPI_COMM_WORLD, &stat);
     }
   }
   else
   {
     // RECEBE NA VARIÁVEL MENSAGEM UM INTEIRO DO PROCESSO DE RANK 0
     MPI_Recv(&mensagem, 1, MPI_INT, 0, 5, MPI_COMM_WORLD, &stat);
     // ENVIA UM INTEIRO PARA O PROCESSO DE RANK 0
     MPI_Send(2, 1, MPI_INT, 0, 10, MPI_COMM_WORLD);
   }

  // FINALIZA A BIBLIOTECA MPI
   MPI_Finalize();
}
```

Criar programas MPI apenas com troca de mensagens ponto-a-ponto, embora suficiente para programar muitas aplicações paralelas com MPI, não é a maneira mais eficiente de se programar. A norma MPI possui diversos mecanismos que permitem realizar comunicações coletivas e de redução, além de permitir a criação de grupos de processos em diferentes comunicadores.

Um exemplo de comunicação coletiva é a função `MPI_Bcast()`, a qual envia uma mensagem em *broadcast* de um determinado processo para todos outros processos de seu

grupo. Existem diversas funções além dessa, como por exemplo, as funções `MPI_Gather()` (reúne valores de um grupo de processos), `MPI_Scatter()` (envia dados de um processo para outros em um grupo de processos), `MPI_Alltoall()` (envia dados de todos processos de um grupo para todos processos do mesmo grupo), *etc*.

Ao tratarmos de funções de redução, um exemplo é a função `MPI_Reduce()`, a qual realiza determinada operação de redução, descrita por uma `MPI_Op`, (por exemplo, `MPI_SUM`), para uma variável em todos processos de um grupo. Além dessa função de redução existem outras, como por exemplo, `MPI_Allreduce()` (combina valores de diferentes processos de um grupo e distribui o resultado) e `MPI_Reduce_scatter()` (realiza um `MPI_Reduce()` e, em seguida um `MPI_Scatter()`). Por fim, existem diversas funções que permitem criar, remover, dividir, aumentar, diminuir e comparar grupos de processos MPI.

O programa 2 ilustra um exemplo de utilização das comunicações coletivas e de redução. Ao invés de enviar a variável `dartsPerProcessor` para cada processo individualmente, é realizado um `MPI_Bcast()` o qual o faz. Ao receber o resultado, utiliza-se uma chamada à função `MPI_Reduce()`, que além de receber o resultado de cada um dos processos, realiza a soma dos resultados, retornando-o na variável `totalDarts`.

Programa 2 Um exemplo de utilização das comunicações coletivas e de redução.

```
int main (int argc, char **argv)
{

  // ALOCAÇÃO E INICIALIZAÇÃO DAS VARIÁVEIS E DA BIBLIOTECA MPI

  MPI_Bcast(&dartsPerProcessor, 1, MPI_INT, 0, MPI_COMM_WORLD);

  for (i = 0; i < dartsPerProcessor; ++i)
  {

    // EXECUÇÃO DO MONTECARLO

  }

  MPI_Reduce (&dartsInCircle, &totalDarts, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

  if(Rank==0)
  {

    // O PROCESSO 0 CALCULA E IMPRIME O RESULTADO

  }

  MPI_Finalize();
}
```

Uma vez explorada a programação MPI-1, a próxima sub-subseção trata da programação MPI-2 com criação dinâmica de processos.

2.1.2.1 Programação MPI-2: Criação Dinâmica de Processos

Na programação MPI com criação dinâmica de processos, novos processos podem ser criados em tempo de execução. Esse tipo de programação permite o mapeamento direto de algoritmos baseados em Divisão & Conquista para programas MPI, uma vez que a cada divisão pode ser traduzida para a criação de novos processos e o retorno do trabalho realizado por esses processos pode ser devolvido via troca de mensagens.

A função utilizada para criar novos processos dinamicamente em tempo de execução no MPI é `MPI_Comm_spawn()`. Sua API é: `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[])`.

Essa função permite criar `maxprocs` novos processos de um determinado programa, definido no parâmetro `command`. A função devolve um novo comunicador para o processo que a chamou através do parâmetro `intercomm`. Através desse comunicador são realizadas as comunicações entre os processos criados (filhos) e o processo que os criou (pai). Existe também a função `MPI_Comm_spawn_multiple()` a qual permite que sejam criados múltiplos processos de múltiplos programas MPI.

Para poder trocar mensagens com o processo pai, os filhos devem chamar a função `MPI_Comm_get_parent()`. Essa função devolve um comunicador MPI o qual permite a troca de mensagens com o processo pai. Um exemplo básico de programa MPI que utiliza a criação dinâmica de processos pode ser visto nos programas 3 e4:

Programa 3 Programa pai

```
/* PROGRAMA PAI */
int main(int argc, char **argv)
{
  long n;
  char cmd[] = "./filho";
  MPI_Comm intercomm;

  MPI_Init(&argc, &argv);

  // CRIA UM PROCESSO DINAMICAMENTE DO PROGRAMA FILHO
  MPI_Comm_spawn(cmd, MPI_ARGV_NULL, 1, MPI_INFO_NULL, 0,
                 MPI_COMM_SELF, &intercomm, MPI_ERRCODES_IGNORE);
  MPI_Send(&n, 1, MPI_LONG, 0, 0, intercomm);
  MPI_Recv(&n, 1, MPI_LONG, 0, 1, intercomm, MPI_STATUS_IGNORE);

  MPI_Finalize();
}
```

O programa ‘pai’ ao ser executado cria um processo do programa ‘filho’, uma vez que a chamada ao `MPI_Comm_spawn()` recebe 1 como argumento no parâmetro `maxprocs`. Logo em seguida ele envia e recebe uma mensagem do programa ‘filho’ através do comunicador `intercomm`.

Programa 4 Programa filho

```
/* PROGRAMA FILHO */
int main(int argc, char **argv)
{
  long n;
  MPI_Comm parentcomm;

  MPI_Init(&argc, &argv);

  // RECUPERA O COMUNICADOR COM O PROCESSO PAI
  MPI_Comm_get_parent(&parentcomm);
  MPI_Recv(&n, 1, MPI_LONG, 0, 0, parentcomm, MPI_STATUS_IGNORE);
  MPI_Send(&n, 1, MPI_LONG, 0, 1, parentcomm);

  MPI_Finalize();
}
```

O programa ‘filho’ recupera o comunicador do pai na variável `parentcomm` através da função `MPI_Comm_get_parent()`. Em seguida ele recebe e envia uma mensagem

para o programa ‘pai’ utilizando o comunicador `parentcomm`. Esse programa básico ilustra o modelo de programação com criação dinâmica de processos no MPI.

Algumas observações são importantes sobre o comunicador `parentcomm`: os *ranks* dos filhos são organizados de 0 a `maxprocs`-1. Os filhos sempre devem enviar suas mensagens para o *rank* do (pai) e o pai pode enviar mensagens para os filhos de 0 a `maxprocs`-1. Para que os filhos se comuniquem entre si deve ser utilizado o comunicador `MPI_COMM_WORLD`, o qual inclui somente estes.

2.2 Orientação a Objetos e o Paralelismo

Orientação a objetos é o paradigma de programação dominante (BUDD, 2001). As linguagens de programação orientadas a objetos possuem diversos recursos voltados a aumentar seu nível de abstração, diminuindo o ciclo de programação/manutenção de software. No contexto de linguagens de programação, de acordo com (SEBESTA, 1998) “abstração significa a capacidade de definir e, depois, de utilizar, estruturas ou operações complicadas de uma maneira que permita ignorar muitos dos detalhes”.

2.2.1 Características da Orientação a Objetos

Dentre as inúmeras características da orientação a objetos voltadas para abstração e para a eficiência do ciclo de desenvolvimento podemos citar tipos genéricos (ou *templates*), polimorfismo, encapsulamento, herança, *etc*. Além disso, as mais modernas linguagens orientadas a objetos possuem mecanismos que permitem deixar a cargo da linguagem de programação gerenciar a alocação e desalocação de memória e serializar objetos. Esses mecanismos permitem que o programador construa sistemas de maneira modular, hierárquica e organizada.

A herança traz a possibilidade de um objeto acessar e utilizar métodos e estruturas de outro objeto como se fossem seus. Quando, por exemplo, existirem dois objetos semelhantes pode-se codificar os métodos e estruturas que eles tem em comum somente uma vez e utilizá-los nos dois objetos. Dessa forma é possível diminuir a quantidade de código necessária em determinados problemas além de aumentar a capacidade de reutilização de código.

O recurso de encapsulamento pode ser conceituado como a capacidade de codificar os dados e os métodos que manipulam esses dados em um único objeto. Além disso, é possível criar métodos acessíveis somente dentro desse objeto, ou pelos objetos que o herdam, permitindo modificar a estrutura interna de um objeto sem alterar sua estrutura externa. A utilização desse recurso permite maior organização dos dados uma vez que os métodos que os manipulam estão encapsulados junto aos mesmos.

Os recursos de *reflection* (reflexão) e *introspection* (introspecção) se referem à habilidade de um programa aprender algo sobre si mesmo em tempo de execução. Através desses mecanismos, um programa consegue examinar seu estado interno em detalhes, podendo, inclusive, adicionar novos comportamentos em si mesmo durante o tempo de execução. Essa habilidade é útil na construção de bibliotecas pois permite analisar dados e objetos desconhecidos durante o tempo de compilação. Nem todas linguagens de programação orientadas a objetos suportam esses recursos (como o caso de C++) uma vez que estão fortemente associados a uma máquina virtual.

Segundo (BUDD, 2001), podemos diferenciar *reflection* de *introspection* da seguinte maneira:

- *Reflection*: Refere-se a habilidade do programa estudar seu estado interno, por

exemplo, descobrir o tipo de um objeto e seus métodos.

- *Introspection*: Refere-se a habilidade do programa se modificar durante o tempo de execução, por exemplo, carregar uma classe dinamicamente.

Outro recurso poderoso da orientação a objetos é o polimorfismo (um nome, vários significados diferentes). Existem pelo menos quatro formas de polimorfismo(BUDD, 2001):

- *Overloading (sobrecarga)*: Utilizado para descrever vários métodos de uma classe os quais possuem o mesmo nome, no entanto com implementações diferentes (programa 5).
- *Overriding (sobrescrita)*: Utilizado para sobrescrever métodos em uma classe que herde outra (programa 6).
- *Polymorphic variable*: É uma variável a qual é declarada de um tipo mas possui o valor de outra (programa 7).
- *Genéricos ou Templates*: Provê uma maneira de criar mecanismos e especializá-los para situações específicas(programa 8).

Programa 5 Overloading

```
class Overloading
{
  public void metodo(int x) {...}
  public void metodo(string s) {...}
}
```

Programa 6 Overriding

```
class Pai
{
  public int metodo(int x)
  {
    return x+1;
  }
}

class Filho extends Pai
{
  public int metodo(int x)
  {
    return x-1;
  }
}
```

Programa 7 Polymorphic variable

```
Pai p = new Filho();
```

Esses mecanismos aumentam a reusabilidade de código bem como o tornam menor. Por exemplo, se houverem dois métodos que realizam a mesma operação sobre dados de

Programa 8 Genéricos ou Templates

```
template <class T> T max (T esquerda, T direita)
{
  if(esquerda < direita)
    return direita;
  return esquerda;
}
```

tipos diferentes é possível criar os dois métodos com o mesmo nome ao invés de utilizar dois nomes diferentes para diferenciar os tipos dos dados. Os tipos genéricos são úteis na criação de bibliotecas pois permitem que sejam passados valores de tipos não conhecidos em tempo de compilação da biblioteca para seus métodos, eliminando a necessidade de criar um método para cada tipo de dado existente.

2.2.2 Programação Orientada a Objetos Distribuída e Paralela

Processamento paralelo e programação orientada a objetos emergiram como tecnologias promissoras e populares (KALE; KRISHNAN, 1993). Essas tecnologias tem tido sua utilização extensiva nos últimos anos. Na era *multi-core*, a programação paralela é essencial para programação eficiente desse tipo de máquina. Por outro lado, a programação orientada a objetos é dominante no desenvolvimento de aplicações, atualmente. Portanto, faz sentido que essas tecnologias sejam fundidas a fim de combinar seus benefícios.

O encapsulamento pode ser visto como aspecto chave para a programação paralela orientada a objetos. Através dele é possível que dados e métodos sejam agrupados em um único objeto, obtendo, dessa forma, uma modularidade onde a tarefa e os dados que ela manipula estejam agrupados. De fato, diversas bibliotecas voltadas para a programação paralela orientadas a objetos fazem uso desse artifício, permitindo que um método de um objeto seja visto como uma tarefa a qual pode ser executada em paralelo sobre dados encapsulados em conjunto com ela, *e.g*:

- *Task Parallel Library*: Consiste em uma biblioteca voltada para o paralelismo em nível de tarefas, é totalmente orientada a objetos e permite que métodos sejam encarados em forma de tarefas, as quais podem ser criadas dinamicamente, sendo executados de forma concorrente;

- *Thread Building Blocks*: É uma biblioteca C++ a qual permite implementar paralelismo baseado em tarefas, criadas dinamicamente, onde uma tarefa é a invocação paralela de um método de um objeto;

- Cilk++: Trata de uma extensão da linguagem de programação C++ voltada para o paralelismo de tarefas criadas dinamicamente, as quais executam métodos em paralelo;

No entanto, essas bibliotecas se concentram na programação paralela em memória compartilhada. No contexto da programação paralela em memória distribuída, é necessário utilizar um mecanismo de comunicação eficiente entre os diferentes processadores.

Os mecanismos de comunicação presentes nas linguagens de programação orientadas a objetos modernas, dentre eles RMI, Web Services e Remoting facilitam o desenvolvimento de programas com objetos distribuídos, permitindo que objetos sejam criados e acessados remotamente através de chamadas simples, além de permitir a interação de

programas distribuídos desenvolvidos em diferentes linguagens de programação através de Web Services.

Outrossim, esses mecanismos de comunicação não oferecem desempenho satisfatório para sua utilização no desenvolvimento de aplicações paralelas de alto desempenho. Para desenvolver esse tipo de aplicação é crucial que o mecanismo utilizado para a comunicação seja o mais eficiente possível. Para obter essa eficiência é necessário que seja utilizada uma biblioteca MPI. Todavia, ao tratarmos de MPI para linguagens de programação orientadas a objetos temos a API MPI para C++ a qual é constantemente criticada por não oferecer suporte a diversos recursos presentes na linguagem de programação C++. Por outro lado existem diversos projetos independentes de bibliotecas MPI para C++ e para outras linguagens de programação orientadas a objetos não suportadas pela norma MPI.

Dentre os projetos de bibliotecas MPI para linguagens de programação não suportadas pela norma, alguns criam bibliotecas totalmente novas, escritas na linguagem de programação na qual se deseja utilizá-las. Outros provêem *bindings* que permitem acessar bibliotecas MPI já existentes. A maior parte desses projetos propõem melhorias à API do MPI, diminuindo a quantidade de parâmetros necessária pelas funções e também oferecendo suporte a comunicação de objetos e outras abstrações presentes nessas linguagens de programação. Algumas linguagens de programação não suportadas pela norma MPI que possuem bibliotecas MPI são C#, Delphi, Eiffel, Haskell, Java, Prolog, Python, Ruby, *etc*.

Vistas algumas características que demonstram a importância do paradigma de programação orientada a objetos, bem como a importância da utilização de MPI na criação de programas de alto desempenho, nas próximas subseções será discutida a norma MPI em linguagens de programação orientadas a objetos.

2.2.3 MPI e Orientação a Objetos

A linguagem de programação orientada a objetos suportada pela norma MPI é a linguagem C++. Nessa sub-seção será discutida a API MPI para C++, APIs alternativas à ela, as quais propõem diversas melhorias e, por fim, bibliotecas MPI para outras linguagens de programação orientadas a objetos, as quais tem demonstrado grande potencial.

2.2.3.1 Bibliotecas MPI para C++

A API MPI para C++ difere na sua organização em relação as APIs para C e Fortran, uma vez que as funções MPI passam a ser encapsuladas em classes. No entanto, não existem divergências significativas nos cabeçalhos das funções. Os comunicadores passaram a ser objetos, os quais, encapsulam as funções de comunicação. Algumas funções passaram a ter menos parâmetros, além de devolverem seu resultado via `return`, diminuindo a quantidade de ponteiros utilizados no desenvolvimento dos programas.

As funções e os objetos MPI se encontram dentro do *namespace* `MPI`, sendo que os comunicadores passam a ser objetos encapsulados pela classe `Comm`. Para enviar uma mensagem a um processo MPI o qual faz parte de determinado comunicador, seleciona-se o comunicador e nele chama-se o método `Send()`, ao contrário do que ocorre em MPI-C aonde chama-se a função `MPI_Send()` e passa-se o comunicador como argumento, uma vez que esse método atua sobre o comunicador, devendo ser encapsulado por sua classe. Os métodos `Get_rank()` e `Get_size()` retornam seus resultados via `return`, eliminando a necessidade de passar um ponteiro para a variável na qual deseja-se obter o resultado e eliminando a necessidade de se instanciar uma variável para obter essa informação. Além disso, também são encapsulados pela classe `Comm`, pois atuam

sobre um comunicador.

No entanto, a API MPI para C++, embora seja organizada em classes e métodos, não suporta o uso pleno dos recursos da linguagem C++, uma vez que objetos não podem ser enviados de forma automática e não há suporte à *Standard Template Library* (STL). É natural que em uma linguagem de programação orientada a objetos, deseje-se comunicar objetos. De fato, a API MPI para C++ não passa de *bindings* um-para-um da API MPI para C (FORUM, 2009) encapsulados por classes. Esses problemas enfatizam a necessidade de expandir a API MPI para oferecer uma melhor API para C++.

2.2.3.2 APIs MPI Alternativas para C++

Diversos trabalhos propõem melhorias a API MPI para C++, visando obter maiores níveis de abstração eliminando parâmetros redundantes nas funções MPI (os quais são oriundos da API ser programada sobre a API MPI-C) e adicionando suporte a comunicação de objetos. Enquanto C e Fortran são linguagens parecidas quanto aos níveis de abstração, C++ suporta maiores camadas de abstração como objetos e containers genéricos. Essas características permitem a construção de interfaces de programação mais abstratas. No entanto, a API MPI para C++ não faz o uso de todo potencial de abstração provido pela linguagem C++, e teve seu desenvolvimento descontinuado na norma MPI 2.2 (FORUM, 2009).

2.2.3.2.1 Object Oriented MPI

O projeto OOMPI (MCCANDLESS; SQUYRES; LUMSDAINE, 1996) propõe a modernização da interface MPI para a linguagem C++ através da criação de uma biblioteca para MPI. Essa permite a utilização completa dos recursos da linguagem C++ como polimorfismo, herança, *etc*.

A biblioteca foi implementada como uma camada sobre as interfaces MPI para C e C++ sendo que todas suas primitivas executam funções MPI contidas nessas interfaces. Os nomes das funções são consistentes com os nomes das funções MPI invocadas por ela. Objetos podem ser comunicados da mesma forma que tipos primitivos e ela não insere sobrecusto significativo sobre a interface MPI para C ao comunicar tipos primitivos.

A fonte e o destino de uma mensagem MPI são representados por um objeto da classe `OOMPI_Port`. Essa abstração permite definir que todas as mensagens enviadas e recebidas através daquele objeto terão um destino fixo, eliminando a necessidade de utilizar o `rank`. De fato um objeto do tipo `Port` é uma abstração para encapsular um comunicador e um `rank`. Essa abstração realmente simplifica a comunicação uma vez que uma mensagem pode ser enviada através de um simples comando `Port.Send(msg)`, onde todas as outras informações necessárias para realizar o envio da mensagem estão encapsuladas no objeto.

Para encapsular os objetos `Port` de um determinado comunicador, foi criada a classe `OOMPI_Comm`, a qual possui uma coleção de objetos do tipo `Port`. Dessa forma é possível encapsular todos os objetos `Port` de um determinado comunicador em um único objeto, sendo que o acesso a eles é realizado através do operador '`[]`' da seguinte maneira: `Comm[i].Send(msg)`. É possível obter um determinado `Port` com o comando `Port=Comm[i]`.

Como a linguagem C++ possui como padrão o mecanismo de *Streams* para realizar E/S é natural que essa sintaxe seja utilizada para o envio e recebimento de mensagens. Um exemplo de utilização desse mecanismo é a operação `Port « x` a qual realiza o envio de `x` enquanto que `Port » x` realiza o recebimento de `x`.

A classe `OOMPI_Message` possui construtores para todos os tipos MPI primitivos. Através dessa técnica é possível definir qual tipo de dado está sendo comunicado uma vez que o construtor equivale ao tipo da variável passada como parâmetro será invocado. Todas as operações de comunicação da biblioteca são definidas em termo de objetos do tipo `OOMPI_Message`. Dessa forma é possível comunicar tipos primitivos sem a necessidade de especificar o tipo de dado a ser transmitido, sendo isso realizado pela biblioteca e informado ao MPI-C.

O envio de vetores é realizado da mesma maneira que o envio de uma única variável, possuindo um parâmetro extra o qual informa o tamanho do vetor. Para esse tipo de mensagem é utilizada a classe `OOMPI_Array_message` a qual instancia um objeto desse tipo de forma transparente toda vez que uma comunicação é chamada sobre um vetor de algum tipo primitivo.

Para comunicar objetos de classes do usuário existe a classe `OOMPI_User _type` a qual deve ser herdada por essas classes. Essa classe possui uma interface para as classes `OOMPI_Message` e `OOMPI_Array_message` permitindo assim que o envio de objetos da classe `OOMPI_User_type` siga a mesma sintaxe de envio de objetos dessas classes. No entanto, é necessário seguir alguns passos para que objetos criados pelos usuários possam ser comunicados. Esses passos são:

- Herdar a classe `OOMPI_User_type`;
- Conter um membro estático do tipo `OOMPI_Data_type`;
- O construtor da classe deve inicializar o objeto da seguinte maneira: `OOMPI_User _type(OOMPI_Data_type &type, USER_TYPE *this, int tag);`
- Os dados a serem comunicados devem ser identificados da seguinte maneira: `type « a « b « c « ... « h;` dentro do construtor;

O gerenciamento de erros foi melhorado pela biblioteca sendo que é possível gerenciar erros de três maneiras distintas: deixar a biblioteca MPI gerenciar o erro, disparar uma exceção ou setar a variável `OOMPI_errno`. Essas funcionalidades podem ser personalizadas para cada comunicador. Caso a função retorne após o tratamento de um erro ela irá retornar um valor inválido de acordo com o tipo de retorno adequado.

Os resultados dos testes de desempenho demonstraram que a biblioteca não introduz sobrecusto significativo, mesmo com a inserção das camadas de abstração.

Essa biblioteca introduziu uma API MPI alternativa a qual simplifica a programação MPI em C++ sem sacrificar o desempenho. A biblioteca demonstra o potencial oferecido pela linguagem C++ quando se trata de abstrair interfaces de programação, eliminando parâmetros desnecessários e permitindo o envio de objetos e tipos definidos pelo usuário da mesma maneira que o envio de tipos primitivos sem inserir sobrecusto, no entanto com a necessidade de realizar certas rotinas de serialização nos mesmos.

2.2.3.2.2 BOOST MPI

A biblioteca BOOST.MPI (GREGOR; TROYER, 2003) propõe a modernização da interface MPI para C++. Ela oferece suporte aos mecanismos modernos presentes na linguagem C++, incluindo suporte completo a tipos definidos pelo usuário, mantendo o maior desempenho possível. Essa biblioteca é implementada sobre as bibliotecas MPI já existentes para C e C++, colocando novas camadas de abstração sobre elas. Ela reimplementa a maior parte das funcionalidades da norma MPI-1.

Os recursos suportados pela linguagem como orientação a objetos, *containers*, genéricos e sobrecarga de operadores permitem uma interface de programação mais expressiva do que a presente na API MPI para C++. Segundo (KAMBADUR et al., 2006) interfaces MPI modernas devem seguir a essência do MPI, porém, oferecendo suporte para tipos definidos pelo usuário bem como *containers* e iteradores contidos na STL.

Diversas melhorias na interface de comunicações ponto-a-ponto foram realizadas. O parâmetro `MPI_Datatype` não precisa mais ser utilizado, uma vez que, essa informação pode ser inferida através do uso de *templates* do C++. O número de elementos sendo comunicados pode ser omitido quando são enviados tipos primitivos singulares, simplesmente passando o valor 1 para a biblioteca MPI-C no método responsável por enviar esse tipo de dados. A interface para o envio desse tipo de dados fica da seguinte forma: `send(int, int, const T &)`. O envio de vetores de tipos primitivos necessita que seja especificado o tamanho do vetor, omitindo o `MPI_Datatype`.

As comunicações coletivas utilizam as mesmas técnicas das comunicações ponto-a-ponto. Algumas coletivas recebem como um dos parâmetros um operador responsável por operar os dados obtidos dos diferentes processos. Elas permitem que um objeto de função seja passado no lugar do operador. A STL oferece diversos objetos funções para várias operações comuns de redução como soma e multiplicação. No entanto, os usuários são livres para definir suas próprias funções de redução e objetos funções para redução.

Objetos e tipos definidos pelo usuário podem ser transmitidos da mesma forma que tipos primitivos singulares. Para isso esses objetos devem definir uma função de serialização utilizando à biblioteca *BOOST Serialization Library* (BSL) (RAMEY, 2002). Essa biblioteca facilita o processo de serialização/desserialização. Ao serem comunicados, os objetos são serializados/desserializados através de uma chamada a biblioteca BSL para um vetor de *bytes* realizada pelo método de comunicação. O envio de um objeto possui o custo do envio de uma mensagem informando o tamanho do vetor de *bytes* mais o envio do vetor em si. Dessa forma o receptor sabe a quantia de dados a serem recebidos sem a necessidade de ser especificado pelo usuário.

Os testes de desempenho utilizaram o *benchmark* NetPIPE (SNELL; MIKLER; GUSTAFSON, 1996). Eles demonstraram que a biblioteca não insere penalidades de desempenho para tipos primitivos. Para dados serializados existe uma certa penalidade devido a serialização, cópia de memória e um par Send/Receive extra para informar o tamanho dos dados a serem enviados.

Essa biblioteca evidenciou que é possível obter maiores níveis de abstração do que os níveis de abstração oferecidos pela interface MPI para C++, dentro dos limites oferecidos pela linguagem. O desempenho da biblioteca mesmo com as camadas de abstração não teve quedas em relação ao desempenho das bibliotecas MPI convencionais para C. Isso se deve a descoberta dos parâmetros não causar impacto considerável no desempenho. Quanto a interface de programação a mesma se mostrou em um nível de abstração esperado por programadores que utilizem técnicas modernas de programação em C++.

Além desses projetos, é importante citar as bibliotecas TPO++ (GRUNDMANN; RITT; ROSENSTIEL, 2000) e Para++ (COULAUD; DILLON, 1998) as quais também propõem modificações na API MPI para C++ com o intuito de prover maiores abstrações.

2.2.3.3 APIs MPI para C#

C# é uma linguagem de programação de alto nível orientada a objetos a qual executa sobre uma máquina virtual. Ela provê diversos recursos voltados a facilitar a programação como coletor de lixo, serialização automática e verificação de limites *(bounds checking)*,

além de permitir a interoperabilidade entre diferentes plataformas.

2.2.3.3.1 Pure Mpi.NET

A Pure Mpi.NET (PURE MPI.NET, 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 *Windows Communication Foundation* (WCF) 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 APIs 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 `Send<(T)>(Int32, String, T)`. A serialização dos objetos é feito de forma automática e transparente para o usuário.

Essa biblioteca possui algumas funcionalidades adicionais às suportadas pela norma MPI. Por exemplo, é possível especificar um método de *callback* ao realizar o envio de uma mensagem. Ela 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 ex: `Send<(T)>(Int32, String, T, TimeSpan)`, 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.

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 nenhuma funcionalidade da norma MPI-2.

2.2.3.3.2 MPI.NET

O projeto MPI.NET (GREGOR; LUMSDAINE, 2008) trata de uma biblioteca MPI para o framework.NET escrita em C#. Essa biblioteca torna a programação MPI mais simples através da abstração das chamadas, eliminando a necessidade de passar parâmetros redundantes ou que possam ser inferidos através de métodos de *reflection*.

A plataforma.NET oferece suporte à interação entre diversas linguagens de programação, inclusive entre linguagens que não rodam dentro de sua máquina virtual, como C e C++ (através de chamadas nativas). 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 a biblioteca MPI-C, não sendo necessário reimplementá-las em C#. Isso dá 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 para executar as chamadas MPI.

A biblioteca provê uma API MPI com maiores níveis de abstração do que MPI-C++, respeitando a linguagem C# e seus paradigmas. Tipos definidos pelo usuário (estruturas, objetos, *etc*.) podem ser transmitidos da mesma forma que tipos primitivos, sendo serializados de forma transparente.

As comunicações coletivas também são consideravelmente simplificadas, utilizando uma quantidade menor de parâmetros em suas interfaces. Um exemplo de simplicidade da biblioteca MPI.NET em relação a biblioteca MPI para C ou C++ está no envio de

estruturas abstratas como objetos. No C# basta enviar o objeto. Em C e C++ é necessário serializar manualmente os dados para um *buffer*, calcular o tamanho do *buffer*, enviar o tamanho do *buffer* e após enviar o *buffer*. Na biblioteca MPI.NET esse procedimento é realizado de forma transparente.

Diferentemente da biblioteca BOOST.MPI (GREGOR; TROYER, 2003), nessa biblioteca não é necessário especificar a quantia de dados em um vetor ao enviá-lo, pois isso pode ser inferido de maneira simples em C#. Outra diferença notável é no envio de objetos os quais são serializados de forma mais simples na linguagem de programação C#, processo que será discutido na Seção 3.4.

Essa biblioteca pode ser utilizada em qualquer ambiente que possua a máquina virtual.NET instalada. Em ambiente Windows ela funciona sobre a biblioteca MPI da Microsoft (baseada no MPICH). No Linux, qualquer biblioteca MPI pode ser utilizada como base por ela. O desempenho dela é similar ao desempenho da biblioteca nativa, sendo que é somente de 1 a 2 % mais lenta para mensagens pequenas e varia de 15% mais lenta a 10% mais rápida para mensagens grandes.

Devido a 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 dessa biblioteca para a realização deste trabalho. Outra vantagem oferecida pela biblioteca é a capacidade de ser utilizada pelas diversas linguagens de programação que executam dentro da plataforma.NET. Mais detalhes dessa biblioteca serão discutidos ao longo do Capítulo 4.

2.2.3.4 APIs MPI para Java

A linguagem de programação Java possui diversas bibliotecas MPI. Essa linguagem possui diversos recursos semelhantes a linguagem de programação C# possuindo o mesmo potencial para o desenvolvimento de bibliotecas MPI. No entanto, não existem bibliotecas MPI para Java que combinem desempenho com abstração, como a biblioteca MPI.NET.

2.2.3.4.1 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 `Datatype` definida pela biblioteca. 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 baixo desempenho é o fato de operações em vetores serem lentas em Java. Um outro problema encontrado está no envio de vetores grandes, o que consome muita memória e é lento,

em vista do processo de serialização dos dados. A biblioteca não oferece comunicações coletivas nem suporte a MPI-2.

2.2.3.4.2 JMPI

O projeto JMPI (MORIN; KOREN; KRISHNA, 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 métodos remotos em Java.

A biblioteca possui três diferentes camadas (MORIN; KOREN; KRISHNA, 2002):

1. A API MPI: O núcleo das funções MPI a serem utilizadas pelas aplicações, seguindo o modelo proposto pelo projeto mpiJava (BAKER et al., 1999);

2. A camada de comunicação: O núcleo de comunicação, que contém todas as comunicações necessárias para a implementação da API MPI;

3. A máquina virtual Java: Responsável por compilar e executar as aplicações.

A camada de comunicação possui três responsabilidades: inicializar a máquina virtual, rotear mensagens entre os diferentes processos e prover o núcleo de primitivas MPI. Como a biblioteca utiliza RMI, é necessário registrar uma instância do comunicador para cada processo no registro do Java. Para realizar o acesso aos comunicadores registrados utilizam-se URLs as quais seguem o seguinte padrão: `rmi://hostname:portno/Commx` onde `portno` representa o número da porta no qual o registro aceita conexões e `x` representa o *rank* MPI do processo. Após todos os processos serem inicializados e se registrarem é utilizada uma barreira responsável por verificar se todos os processos iniciaram corretamente e também por distribuir uma tabela contendo todas as URLs de todos os comunicadores entre todos os processos.

As mensagens são passadas entre diferentes processos como parâmetros de chamadas RMI. A mensagem é um objeto serializável o qual possui o *rank* da origem, o *rank* do destino, uma tag, o tipo do objeto e por fim os dados. Ao realizar a chamada RMI o processo origem registra a mensagem na fila de mensagens do processo destino e depois o notifica que há uma mensagem nova. Para o envio bloqueante o processo origem fica bloqueado até que o processo destino tenha recebido a mensagem, para isso são utilizados os métodos `wait()/notify()` do Java.

Para receber as mensagens o processo destino acessa sua camada de comunicação e recupera a primeira mensagem que atende os requisitos esperados. A fila é implementada através da utilização de um Vector(FLANAGAN, 1998) o qual permite remoção fora de ordem e também acesso concorrente. As comunicações coletivas são construídas sobre as comunicações ponto-a-ponto, porém utilizam filas distintas.

Embora não exista uma interface MPI oficial para Java, a interface (BAKER et al., 1999) foi seguida, sendo assim os programas escritos utilizando ela podem utilizar a biblioteca proposta sem nenhuma modificação no código fonte, tendo assim uma interface padrão entre as duas bibliotecas e quaisquer outras que sigam esse padrão.

Uma das vantagens de se utilizar Java está na transmissão de matrizes, a qual se realiza por meio da utilização de *reflection*, 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. Foram utilizadas duas versões do RMI para realizar essa comparação, o RMI padrão e o KaRMI (NESTER; PHILIPPSEN; HAUMACHER, 1999).

Enquanto os testes apresentados demonstraram que o JMPI/KaRMI possui um desempenho melhor que o JMPI/RMI, o mpiJava (discutido a seguir) (BAKER et al., 1999) 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 inferior ao desta, que já possui certo sobrecusto em relação às distribuições MPI para C (MORIN; KOREN; KRISHNA, 2002). A biblioteca implementa comunicações ponto-a-ponto e algumas comunicações coletivas, não oferecendo suporte a MPI-2.

2.2.3.4.3 mpiJava

O projeto mpiJava (BAKER et al., 1999) propõe uma biblioteca MPI para a linguagem de programação Java, 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-C nativa, a qual as executa.

Sua API possui a estrutura de classes baseada na estrutura de classes da interface MPI para a linguagem C++. Ela implementa a interface da maioria das funcionalidades contidas na norma MPI-1. A classe responsável por realizar as comunicações é chamada `Comm`, sendo composta por diversas sub-classes. A classe `Datatype` contém as correspondências entre os tipos primitivos de Java e os tipos primitivos MPI. Essa classe é utilizada nas comunicações para fazer a conversão de tipos.

Os resultados das chamadas MPI são obtidos da mesma maneira que no MPI-C: recupera-se o resultado via referência na chamada da função. 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.

A interface para envio de mensagens ponto-a-ponto é similar a API MPI-C, sendo ela: `public void Send(Object buf, int offset, int count, Datatype datatype, int dest, int tag)` onde `buf` deve ser o vetor contendo o(s) dado(s) a ser(em) enviado(s), `offset` informa em qual elemento do vetor a mensagem inicia, `count` informa a quantidade de elementos do vetor, `datatype` o tipo de dado que está sendo enviado, `dest` o destino e por fim `tag` uma tag MPI.

Para realizar o envio de tipos criados pelo usuário existem duas opções: utilizar os construtores de `datatype` padrões do MPI os quais são providos pela biblioteca com a restrição que elementos `MPI_TYPE_STRUCT` devem ser todos do mesmo tipo. Outra abordagem é serializar o tipo do usuário para um buffer, transmitir o buffer e deserializálo, para isso, utilizando o tipo `Object` criado pela biblioteca. O envio de matrizes não pode ser feito diretamente, sendo que devem ter suas linhas copiadas para um vetor ou serializadas para o tipo `Object`.

Os testes de desempenho apresentados em (BAKER et al., 1999) demonstraram um desempenho competitivo, mas existe um pequeno sobrecusto (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. Ela não suporta as

funcionalidades presentes na norma MPI-2.

2.2.3.4.4 APIs MPI para Python

A biblioteca MPI for Python (DALCIN; PAZ; STORTI, 2005) provê *bindings* MPI para a linguagem de programação Python. Embora seja possível construir uma biblioteca MPI totalmente escrita em Python, favorecendo a portabilidade, a qualidade e complexidade das bibliotecas MPI já existentes, além da facilidade em conectar Python com C e C++, tornam mais atraente uma solução que chame as funções já desenvolvidas nessas bibliotecas MPI-C (DALCIN; PAZ; STORTI, 2005).

A classe `Comm` encapsula as funções de comunicação a qual por sua vez é encapsulada pela classe `mpi`. O conceito de utilizar `ports` para do projeto OOMPI (MCCANDLESS; SQUYRES; LUMSDAINE, 1996) foi implementado, permitindo que mensagens sejam enviadas via *streams* através dos operadores » e «. A interface das funções `send()` e `receive()` foi simplificada, sendo que para enviar uma mensagem é necessário passar como parâmetro somente a mensagem e o destino. Objetos podem ser comunicados sendo que para isso eles são automaticamente serializados para vetores de `char`. Para realizar comunicações coletivas foram implementadas e simplificadas as funções Bcast, Scatter, Gather, Allgather e Alltoall, além das operações globais de redução Reduce, Allreduce e Scan as quais não foram modificadas.

Os testes de desempenho demonstrados em (DALCIN; PAZ; STORTI, 2005) mostram que a largura de banda máxima obtida equivale a 85% da largura máxima de banda obtida em C para troca de mensagens ponto-a-ponto, e para as comunicações coletivas ela é em média 5% mais lenta para o *broadcast* e 20% mais lenta para o *alltoall*. Além desse projeto existem outros que criam bibliotecas MPI para Python como por exemplo a pyMPI (DRUMMOND et al., 2009), porém seu desempenho é inferior ao do projeto MPI for Python.

2.2.4 Programação MPI Orientada a Objetos

Uma vez explorada a vasta disponibilidade de bibliotecas MPI para linguagens de programação orientadas a objetos, essa sub-seção discorre sobre os conceitos de programação MPI nesse paradigma. Conforme visto anteriormente, a API MPI para C++ possui as funcionalidades encapsuladas por classes. No entanto, não suporta adequadamente a orientação a objetos tendo motivado a criação de novas APIs.

A utilização de APIs com maiores níveis de abstração em conjunto com linguagens de programação modernas permite diminuir o esforço de programação e, por consequência, aumentar a qualidade dos códigos produzidos diminuindo a quantidade de erros que o programador está sujeito a cometer. Além disso, a curva de aprendizado da programação paralela tende a ser menor ao utilizarmos APIs com maiores níveis de abstração.

Utilizar objetos para encapsular estruturas de dados complexas a serem comunicadas facilita a comunicação desse tipo de estrutura. Em MPI estruturas não alocadas em área contígua de memória não podem ser comunicadas sem antes serem serializadas. MPI utiliza um ponteiro para uma posição inicial de memória e a quantia de elementos daquele tipo que serão transmitidos para recuperar os dados a serem comunicados em uma mensagem. Para comunicar uma matriz, por exemplo, o programador pode enviar linha a linha ou então serializá-la para um vetor. Com a orientação a objetos, se essa matriz for encapsulada por uma classe, ao comunicar um objeto dessa classe ela será serializada pelo método de serialização da classe, o que em C# é realizado de forma automática e transparente e em C++ é feito de maneira mais simples com o uso do BOOST.MPI.

Visto que existe uma grande gama de algoritmos que necessitam comunicar matrizes, essa é uma contribuição importante da orientação a objetos. Além disso, ao utilizarmos criação dinâmica de tarefas, podemos comunicar os dados em conjunto com as operações a serem executadas sobre eles em uma única mensagem ao comunicarmos um objeto. Em C#, é possível que objetos totalmente desconhecidos em tempo de compilação sejam comunicados e tenham métodos invocados através dos métodos de reflexão da linguagem. De fato, a utilização de novas linguagens de programação oferece novas abstrações antes não existentes em MPI.

Em comparação com a API MPI para C++, a API da biblioteca MPI.NET possui grandes simplificações. Todos os métodos exigem quantias significativamente menores de parâmetros, porém possuem maior expressividade, uma vez que serializam e comunicam objetos de forma automática. Em comparação com o BOOST.MPI, as bibliotecas possuem níveis de abstração semelhantes, porém, como MPI.NET é escrito em uma linguagem de programação que executa sobre máquina virtual, existem maiores abstrações providas pela linguagem. A serialização em C# é totalmente transparente, no BOOST.MPI é necessário especificá-la manualmente. C# abstrai totalmente a heterogeneidades de *hardware* e sistema operacional. A gerência de memória é um fator de grande importância, por ser automática em C#, o usuário não precisa se preocupar com a desalocação de memória e, por consequência, não existem vazamentos de memória.

Em comparação com as outras APIs MPI vistas anteriormente, nenhuma oferece a combinação desempenho/abstração oferecida pela biblioteca MPI.NET. A biblioteca mpiJava oferece bom desempenho, no entanto, não abstrai de maneira alguma a API MPI-C. Por outro lado, as outras bibliotecas que abstraem a API, não possuem desempenho suficiente para serem utilizadas. A biblioteca MPI.NET se sobressai em relação às demais por abstrair de maneira significativa a API MPI-C++ (mas não deixando de respeitá-la) e ao mesmo tempo não onerar o desempenho de maneira a não poder ser utilizada.

2.3 Conclusões sobre o Capítulo

Nesse capítulo, foi apresentada uma introdução à norma MPI, onde sua evolução, API e modelo de programação foram discutidos. Também foi introduzida a programação MPI com criação dinâmica de processos. Sua API para C++ foi discutida, apresentando-se projetos que propõem melhorias a ela explorados. Além disso, foram vistos projetos que estendem a norma MPI para outras linguagens de programação orientadas a objetos. Por fim, foi realizada uma discussão sobre a programação MPI orientada a objetos.

A programação orientada a objetos traz para a programação paralela as mesmas vantagens que para a programação sequencial, além de permitir que objetos contendo os dados e os métodos que manipulam esses dados sejam distribuídos, executados em paralelo e, logo após, reagrupados. No entanto, a API MPI para a linguagem de programação C++ é amplamente discutida, uma vez que não suporta diretamente essa comunicação de objetos. Uma outra característica, constantemente criticada, é o seu grau de abstração o qual poderia omitir mais detalhes do programador. Essa API nada mais é do que o encapsulamento das funções MPI-C em classes C++. Diversos projetos demonstraram que é possível melhorar sua API suportando mais recursos da linguagem C++, tornando assim sua API mais abstrata.

Embora a linguagem de programação C++ possua mais recursos do que C e Fortran, linguagens como Java, C#, Python e Ruby possuem diversos recursos não suportados por C++, dentre eles gerência automática de memória, introspecção, independência de

plataforma, *etc*. Nesse capítulo estudou-se várias alternativas para suporte MPI nessas linguagens.

No entanto, em sua maior parte, elas possuem desempenho insatisfatório além de suportarem pequenos subgrupos de funcionalidades MPI-1, o que torna sua utilização inviável. No entanto algumas bibliotecas como a mpiJava e MPI.NET demonstraram desempenho aceitável, sendo que a API da biblioteca MPI.NET ao contrário da mpiJava, abstrai as chamadas MPI, diminuindo a quantia de parâmetros necessários e a complexidade de programação, além de permitir o envio de objetos. Portanto a biblioteca MPI.NET mostrou-se a que melhor combina abstração/desempenho.

Uma vez escolhida a biblioteca MPI.NET para o desenvolvimento desse trabalho, a qual é escrita na linguagem de programação C#, o próximo capítulo aborda a programação paralela em C#.

3 .NET E PROGRAMAÇÃO PARALELA

Este capítulo irá tratar sobre a programação paralela e distribuída na plataforma .NET. Serão discutidos aspectos fundamentais da plataforma bem como algumas características importantes para a realização desse trabalho. Os conceitos apresentados nesse capítulo servem para todas linguagens da plataforma .NET, porém serão apresentados sob a ótica da linguagem de programação C# uma vez que é a linguagem adotada pela biblioteca MPI.NET.

A plataforma de software .NET é um ambiente com suporte a diversas linguagens de programação as quais compilam código fonte para uma linguagem intermediária chamada *Microsoft Intermediate Language* (MSIL). Os programas compilados para essa linguagem executam sobre uma máquina virtual chamada *Common Language Runtime* (CLR), a qual compila o código intermediário para código específico da plataforma de execução.

A utilização do código intermediário permite que programas escritos nas linguagens de programação que geram esse código interajam entre si. Dentre elas podemos citar as linguagens C#, Visual Basic .NET, Visual C++ .NET, Visual J# .NET, além de diversas linguagens de *scripting*, por exemplo, ASP.NET. As principais características da utilização da MSIL e da CLR serão discutidas na próxima seção.

3.1 CLR e MSIL

A linguagem MSIL assim como Java *byte code* é uma linguagem de baixo nível com uma sintaxe simples, baseada em códigos numéricos ao invés de texto, o que permite que seja traduzida para código de máquina muito rapidamente. A utilização dessa abordagem permite independência de plataforma, aumento de desempenho e interoperabilidade entre linguagens.

Um programa compilado para MSIL pode ser executado em qualquer plataforma que possua a máquina virtual, assim como um programa Java. Embora a máquina virtual da Microsoft seja somente para Windows, existe uma máquina virtual chamada Mono a qual oferece suporte para diversos sistemas operacionais.

A compilação da MSIL para código de máquina é realizada em tempo de execução pela CLR, técnica conhecida como compilação *Just-In-Time* (JIT). A MSIL é sempre compilada antes de executar, no entanto, a aplicação não é compilada inteiramente de uma só vez, uma vez que isso causaria atrasos no início da execução. Ao invés disso cada porção de código é compilada conforme necessário, sendo que o código nativo é armazenado durante toda a execução.

Uma vez que o código MSIL é compilado em tempo de execução, o tipo exato de processador que o executará é conhecido, permitindo que a CLR otimize o código para utilizar determinadas características e instruções oferecidas pelo processador. Dessa forma é

possível utilizar características particulares de determinado processador sem perder a portabilidade. Essa técnica pode ser realizada em linguagens de programação que compilam para código nativo, porém com o ônus de perder a capacidade de executar sobre qualquer processador de determinada plataforma, por exemplo, executar em qualquer processador de arquitetura x86, passando a executar somente em um x86 específico, por exemplo, um Nehalem.

Uma vez que as diferentes linguagens de programação da plataforma são compiladas para uma mesma linguagem intermediária, os códigos compilados de uma linguagem podem facilmente interagir com os compilados de outra linguagem, permitindo interoperabilidade entre as diferentes linguagens, permitindo que:

- Uma classe escrita em uma linguagem possa herdar de uma classe escrita em outra linguagem;
- Uma classe contenha objetos de uma classe escrita em outra linguagem;
- Um objeto chame métodos de outro objeto escrito em outra linguagem, *etc*.

Essas características permitem a reusabilidade de código já escrito em outras linguagens além de permitir integração de programas e bibliotecas escritos em diferentes linguagens. Por exemplo, a biblioteca MPI.NET pode ser utilizada por qualquer linguagem de programação que compile para MSIL.

Na MSIL valores e referências são distintos, sendo que variáveis armazenadas por referência são armazenadas em uma área de memória chamada *managed heap* já variáveis armazenadas por valor são armazenadas em uma área denominada *stack*.

Outro aspecto importante sobre a MSIL é que ela é baseada em variáveis fortemente tipadas. Em C é comum realizar o *cast* de ponteiros entre diferentes tipos. Essa técnica é excelente para desempenho, porém quebra a segurança de tipos. No entanto, em situações especiais, é permitido utilizar ponteiros para realizar esse tipo de operação, porém o código deve estar em um bloco denominado `unsafe` e o programa deve ser compilado com a *flag unsafe*. A utilização de ponteiros quebra a segurança de tipos pois a CLR não consegue testá-los.

Tendo visto os principais aspectos da MSIL e da CLR os quais são aspectos fundamentais da plataforma.NET, a próxima seção irá discorrer sobre outro aspecto fundamental: o gerenciador de memória.

3.2 Gerenciamento de Memória

Ao programar com C# o usuário não precisa se preocupar com o gerenciamento de memória, uma vez que o coletor de lixo trata da desalocação de variáveis não mais utilizadas. É importante entender o funcionamento do gerenciamento de memória para a integração de programas escritos em C# com programas escritos em linguagens de programação que não fazem parte da plataforma.NET, por exemplo, programas escritos em C, como distribuições MPI. Essa seção irá explicar como é feito o gerenciamento automático de memória no Framework .NET.

A alocação de variáveis de tipos primitivos (`int`, `float`, `char`, *etc*), e que não são alocadas dentro de um objeto é realizada na pilha (ou *stack*), da mesma maneira que em C e C++, sendo que essas variáveis são sempre acessadas por valor. A alocação de variáveis por referências (objetos, vetores, matrizes, *etc*) funciona de maneira diferente.

Essas variáveis são alocadas em um espaço de memória chamado *managed heap*, o qual diferentemente do *heap* de C++, é gerenciado pelo coletor de lixo.

O *managed heap*, criado pela plataforma .NET, utiliza é segmentado em gerações (ou *chunks* de memória), sendo elas 0, 1 e 2 as quais possuem respectivamente 256KB, 2MB e 10MB de tamanho inicial sendo esse tamanho ajustado durante o tempo de execução.

As variáveis são sempre alocadas na geração 0. Quando o limiar da geração 0 do *managed heap* é excedido, o coletor de lixo é executado na geração 0, removendo da memória objetos que possam ser descartados. Os objetos que persistirem, são movidos para a geração 1, compactando os dados em uma área contígua de memória, evitando a fragmentação.

Quando a geração 1 exceder seu limiar, o que somente ocorre quando uma coleta na geração 0 move dados para ela excedendo seu tamanho, é executada uma coleta na geração 1. Seus dados são movidos para a geração 2. Conforme o tempo de execução, objetos que possuam tempo de vida longo estarão localizados na geração 2 e objetos com tempo de vida curto estarão na geração 0. A geração 2 por sua vez, é coletada somente quando são movidos dados da geração 1 para ela, excedendo seu limiar, o que gera uma coleta completa de memória, uma vez que todas gerações são coletadas.

Objetos que possuam tamanho maior que 85KB são considerados grandes, e portanto, são tratados de maneira diferente, sendo colocados em um *heap* diferente, o qual é coletado somente quando a geração 2 é coletada, evitando que haja coletas de dados grandes frequentemente.

O coletor de lixo não realiza contagem de referências, uma vez que essa técnica causa sobrecusto significativo, ao invés disso, é construído um grafo de objetos no início da aplicação. A cada ciclo de coleta, o coletor de lixo constrói o grafo de objetos referenciados, onde objetos que não possuam pai podem ser descartados. Ele é construído à partir de referências raiz, sendo que objetos que não estiverem na subárvore de nenhuma raiz podem ser coletados (conforme visto na figura 3.1). Referências raiz são variáveis globais, estáticas e locais.

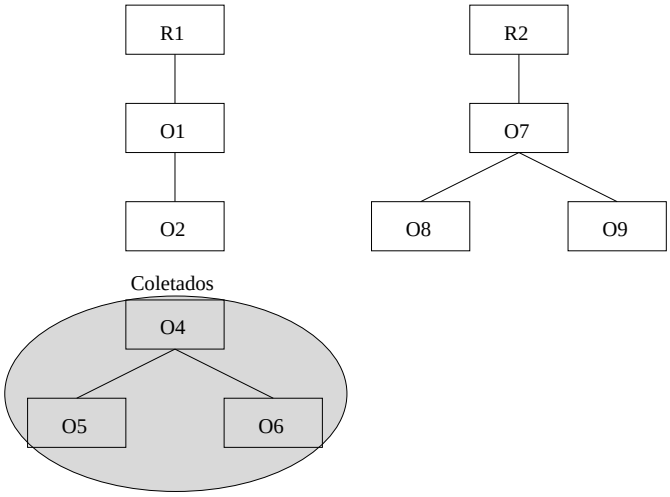

Figura 3.1: Grafo de objetos demonstrando objetos coletáveis.

Uma vez visto o funcionamento do coletor de lixo em áreas de memória gerenciadas, é importante tratar da utilização de memória não gerenciada para o entendimento da biblioteca MPI.NET. O Framework .NET provê diversas maneiras para permitir o acesso

e utilização de memória não gerenciada. Essa prática é essencial para a integração de aplicações .NET com aplicações escritas em C, as quais normalmente possuem diversas funções que recebem ponteiros como parâmetros.

Através da classe `UnmanagedMemoryStream` é possível instanciar uma *stream* de memória não gerenciada sobre a qual é possível executar operações de leitura e escrita, além de manter propriedades sobre a *stream* como, por exemplo, o seu tamanho. É possível serializar objetos para essa *stream*, a qual pode ser manipulada como um ponteiro, podendo ser passada para funções C que recebam ponteiros como argumento. Essa classe serve para alocar novas *streams* não gerenciadas, mantendo os dados serializados, porém é necessário copiar os dados para as mesmas.

Existe também a classe `GCHandle`, a qual previne que o coletor de lixo mova determinado objeto na memória. Ela marca o objeto como não coletável, fazendo com que o coletor de lixo não gerencie sua área de memória. Para isso, um objeto `GCHandle` deve ser instanciado no modo `Pinned`, o qual garante que o objeto a ser protegido permanecerá no mesmo endereço de memória até ser liberado. Através da classe `Marshal`, a qual detém diversos métodos para realizar operações em memória não gerenciada, é possível recuperar um ponteiro para o endereço de memória do objeto protegido pelo `GCHandle`. Dessa forma, estruturas como vetores, os quais são sempre serializados na memória, podem ser transmitidos para programas em C sem a necessidade de copiá-los para uma *stream* não gerenciada, evitando sobrecusto. Uma vez que não é mais necessário manter o objeto fixo, basta liberá-lo através do método `Free()` da classe `GCHandle`.

Existem inúmeras possibilidades de realizar operações em memória não gerenciada no Framework .NET, porém as que são relevantes para o entendimento do presente trabalho foram cobertas nessa subseção.

3.3 Delegados

Delegados (*Delegates*) são uma abstração importante da plataforma .NET, criada para permitir o uso de ponteiros para métodos no .NET (MARSHALL, 2008). Através deles é possível tratar métodos como se fossem variáveis, permitindo que sejam passados como parâmetro para outros métodos, retornados via `return` por métodos e inclusive armazenados em vetores.

Diferente dos ponteiros para funções de C e C++, delegados são *type-safe* e orientados a objetos, ou seja, um delegado deve ser criado e utilizado sempre fazendo o *cast* para o tipo específico do objeto encapsulado por ele, ao contrario de C e C++ que permitem a utilização de `void *`. Além disso, um delegado possui métodos para obter informações e realizar operações sobre o objeto encapsulado por ele, como por exemplo, obter o tipo do objeto (o que é útil para trabalhar com tipos genéricos).

Ao criar um delegado, o usuário passa como parâmetro o método que o delegado deverá executar e o objeto no qual esse método será executado. Tanto o método quanto o objeto não necessitam ser conhecidos em tempo de compilação. Ao ser invocado, o delegado executa o método no objeto alvo. Delegados podem ser utilizados para definir funções de *callback*, eventos, classes genéricas e quaisquer situações nas quais o nome de um método não seja conhecido em tempo de compilação.

No contexto do processamento paralelo, a noção de delegado é de suma importância, uma vez que, o método encapsulado por um delegado pode ser utilizado como ponto de partida para a execução de uma *thread* (Subseção 3.7.3). É possível que uma biblioteca utilize delegados para receber, em tempo de execução, métodos a serem executados em

paralelo. Sua utilização no presente trabalho será discutida no Capítulo 4.

3.4 Serialização

Através da serialização o estado de um objeto é armazenado em uma *stream* de *bytes*, a qual pode ser transmitida, armazenada, e convertida novamente para o objeto recuperando seu estado. Na linguagem de programação C#, é possível definir que os objetos de uma classe são serializáveis por anotação, simplesmente a marcando como `[Serializable]`. Um exemplo de classe serializável pode ser visto no Programa 9.

Programa 9 Exemplo de uma classe serializável.

```
[Serializable]
public class MinhaClasse {
  public int x, y, z;
  public String str;
  private double d;
}
```

No Programa 9 todas as variáveis da classe `MinhaClasse` serão seralizadas automaticamente. Caso o usuário não queira que determinada variável seja serializada é possível utilizar a *serialização seletiva*, marcando a declaração da variável com o atributo `[NonSerialized]`, por exemplo, `[NonSerialized] public String str;`. Dessa forma a variável `str` não será serializada.

Em C# objetos podem ser serializados para *bytes* ou para XML. No contexto do presente trabalho, será utilizada somente a serialização para *bytes*. O formato para qual o objeto será serializado é indiferente o mesmo, sendo definido pela função que o serializa. Para serializar um objeto para um *buffer* de *bytes*, utiliza-se a classe `BinaryFormatter`, instanciando um objeto da mesma e nele invocando o método `Serialize()`, o qual recebe como argumentos o objeto a ser serializado e a *stream* para qual ele será serializado. Essa *stream* pode ser, por exemplo, uma área de memória não gerenciada, a qual pode ser transmitida utilizando-se uma biblioteca MPI para C. O processo de desserialização é exatamente o mesmo, aonde a *stream* é passada para o método `Deserialize()`, o qual retorna o objeto para uma variável de seu tipo.

Esse mecanismo automático de serialização/desserialização elimina a responsabilidade do usuário de definir as rotinas de serialização/desserialização para cada campo de suas classes. Além disso, objetos que armazenam outros objetos terão o grafo completo serializado, com a condição que todos objetos sejam serializáveis. Para isso, durante o processo de serialização do objeto, serão chamadas recursivamente as rotinas de serialização para objetos serializáveis contidos nele (assim como em Java), formando automaticamente o grafo.

3.5 Reflexão

Na linguagem C# Reflexão ou *Reflection* é considerada a capacidade de inspecionar e manipular elementos de um programa durante o tempo de execução. Dentre essas capacidades podemos citar:

- Enumerar os membros de um tipo;
- Instanciar um novo objeto;

- Executar os membros de um objeto;
- Recuperar informações sobre um tipo;
- Recuperar informações sobre uma compilação (nesse contexto diferencia-se programa de compilação por esta não necessitar de um método `Main()`, podendo ser simplesmente uma classe);
- Criar e compilar uma nova compilação.

Esses recursos permitem inferir diversas informações em tempo de execução, além de permitirem a manipulação de objetos desconhecidos em tempo de compilação, o que é muito útil na construção de bibliotecas. Sua utilização no presente trabalho será discutida no Capítulo 4.

`Type` é uma das classes mais importantes do mecanismo de *Reflection*. A classe `Object` deriva de `Type`, logo todos objetos derivam dessa classe uma vez que derivam de `Object`. Através das propriedades da classe `Type` é possível obter diversas informações sobre um objeto, por exemplo, o nome de sua classe. Essas informações são obtidas em strings, booleanos e outros objetos do tipo `Type`. Seus métodos (14 ao total) são utilizados para obter detalhes da a classe, por exemplo, é possível obter um vetor contendo todos os métodos de determinado objeto através de `GetMethods()`. Uma vez descobertos, os métodos podem ser invocados através do método `Invoke()` da classe `MethodInfo`.

Através da classe `Assembly` é possível carregar uma compilação em tempo de execução. Uma vez carregada, é possível recuperar um vetor com os tipos de dados contidos na compilação através do método `GetTypes()`. Tendo esses tipos, é possível recuperar seus métodos, as informações de seus métodos e por fim executá-los. Essa ferramenta traz grande dinamicidade para a linguagem de programação, uma vez que é possível carregar classes desconhecidas, instanciá-las e executar seus métodos em tempo de execução.

Uma vez visto o funcionamento básico do mecanismo de *Reflection*, a próxima seção irá tratar de tipos genéricos, os quais o utilizam.

3.6 Genéricos

Tipos genéricos permitem a definição de estruturas *type-safe* sem a necessidade de associá-las a um tipo específico, comprometer seu desempenho ou produtividade. Isso resulta em códigos com maior qualidade por eliminar a necessidade de duplicar código para tipos específicos. Por exemplo, estruturas básicas como listas e dicionários utilizam genéricos podendo ser utilizadas para armazenar quaisquer tipos de dados.

Em C#, declara-se um tipo genérico utilizando `<T>` onde `T` representa o tipo. O Framework .NET define diversas estruturas para armazenar tipos genéricos, como por exemplo o dicionário (`Dictionary()`), classe que provê o mapeamento entre chaves e valores genéricos. Ela é declarada como `Dictionary<TKey, TValue>`, onde `TKey` e `TValue` podem ser variáveis de quaisquer tipos. Um outro exemplo é a classe `List` a qual representa uma lista de tipos genéricos sendo declarada como `List<T>`, onde `T` representa o tipo genérico. Essa classe permite que sejam realizadas operações de lista sobre os dados como adição, remoção, leitura, escrita, *etc*. Um exemplo de sua utilização pode ser visto no Programa 10.

O usuário também pode definir estruturas e classes genéricas, por exemplo, no Programa 11 é criada uma pilha capaz de armazenar qualquer tipo de dado.

Programa 10 Exemplo de utilização da classe List.

```
public class ListaDeEquipes
{
  public static void Main()
  {
    List<string> equipes = new List<string>();

    equipes.Add("Grêmio");
    equipes.Add("Internacional");
    equipes.Add("Juventude");

    foreach(string equipe in equipes)
    {
      Console.WriteLine(equipe);
    }

    equipes.Remove("Internacional");
    equipes.Clear();
  }
}
```

Programa 11 Exemplo de uma classe genérica.

```
public class Pilha<T>
{
   T[] Itens;
   public void Push(T item)
   {...}
   public T Pop()
   {...}
}
```

Diferente dos *templates* de C++, os genéricos de C# realizam a atribuição de tipo em tempo de execução. Outra característica importante é que é possível utilizar reflexão sobre tipos genéricos, permitindo assim obter todas informações do tipo que está sendo utilizado em tempo de execução. Isso não é possível nos *templates*.

Tipos genéricos podem ser utilizados para realizar rotinas muito mais complexas, por exemplo, é possível criar bibliotecas as quais recebam classes genéricas, as instanciem e inclusive as executem como *thread*. Embora não sejam relacionados diretamente com a programação paralela, sua utilização em bibliotecas paralelas permite que sejam manipulados em tempo de execução tipos desconhecidos em tempo de compilação. Esse assunto será tratado no Capítulo 4 uma vez que tipos genéricos são amplamente utilizados na biblioteca MPI.NET.

3.7 Programação Paralela e Distribuída em .NET

A plataforma .NET possui em sua API diversos mecanismos os quais permitem a criação de programas distribuídos e paralelos. No contexto de programação distribuída existem os mecanismos *Remoting* e WCF . Já para a programação paralela (em memória compartilhada) existem as *threads* e a *Task Parallel Library* (TPL). Ao final dessa seção fica claro que para programação paralela de alto desempenho em memória distribuída a melhor alternativa é a utilização de uma biblioteca MPI.

3.7.1 *Remoting*

Remoting permite o acesso a objetos remotos nas aplicações .NET. A proposta dessa tecnologia é que aplicações distribuídas possam ser programadas facilmente, abstraindo

as comunicações. Dessa forma um servidor mantém objetos os quais podem ser acessados e modificados remotamente por clientes.

Para criar uma classe cujos objetos possam ser acessados remotamente basta que ela herde a classe `MarshalByRefObject`. É necessário criar alguns arquivos de configuração do tipo XML, os quais definem a porta em que o servidor irá receber conexões, a URL dos objetos, o nome da classe a ter objetos remotos e o tipo de instanciação a ser utilizada. O Remoting permite diferentes tipos de instanciação dos objetos remotos, sendo elas:

- *Singleton*: uma mesma instância para todas as chamadas e todos os clientes guardando o estado (o mesmo modelo utilizado por RMI);
- *SingleCall*: uma nova instância para cada chamada de cada cliente a qual não guarda estado e é encerrada ao final da chamada;
- *Client-activated*: cada cliente instancia um objeto e o servidor guarda seu estado.

O coletor de lixo do *Remoting* mantém um tempo de vida para cada objeto remoto, o qual é estendido a cada chamada que o objeto recebe. É possível evitar que objetos remotos sejam removidos registrando junto a ele um *sponsor* (realiza automaticamente o pedido de renovação do objeto) garantindo assim sua permanência.

Quanto ao desempenho, diversos benchmarks realizados em (SCHWARZKOPF et al., 2008) demonstraram um desempenho muito inferior ao Java RMI, o que torna sua utilização inviável para computação de alto desempenho.

3.7.2 *Windows Communication Foundation*

WCF é o serviço de *Web Services* do Framework .NET. Sua proposta é facilitar o desenvolvimento de aplicações distribuídas bem como prover interoperabilidade entre diferentes plataformas.

Um serviço WCF é composto por três partes: *Service Class* (classe que irá expor seus métodos no serviço), *Host Environment* (servidor que irá armazenar o serviço) e um ou mais *EndPoint* (componentes nos quais os clientes irão se conectar para realizar o acesso ao serviço).

Para criar um serviço WCF, primeiramente cria-se uma interface. É necessário fazer a anotação `ServiceContract` sobre o nome da interface e a anotação `OperationContract` sobre os métodos que serão disponibilizados. Em seguida basta criar uma classe que implemente essa interface. Para disponibilizar um serviço WCF através de uma aplicação é necessário criar um programa o qual defina:

- um *host*: responsável por definir a classe a ser utilizada podendo ser feito através da classe `ServiceHost` ou definido em um arquivo XML de configuração;
- um contrato: responsável por definir a interface utilizada pela classe podendo ser feito através do método `ContractDescription.GetContract()` ou em um arquivo de configuração XML;
- e por fim um *endpoint*: possui o contrato, o tipo de protocolo a ser utilizado e a URL do serviço, sendo que pode ser definido através da instância de um objeto da classe `ServiceEndpoint` ou através de um arquivo de configuração XML.

Após definir esses três itens a aplicação deve então adicionar o *endpoint* ao *host* através do método `host.Description.Endpoints.Add(endpoint)` e por fim inicializar o serviço com o método `host.Open()`.

É possível também hospedar o WCF em servidores da Microsoft como o IIS ou o WAS sendo que dessa forma as configurações do serviço são feitas através de arquivos XML no servidor.

O cliente é criado através da ferramenta **svcutil**, a qual faz parte do Framework .NET, passando como parâmetro o endereço onde o serviço está disponível. Essa ferramenta gera um arquivo com as configurações de acesso e uma classe com os métodos remotos. Então basta criar uma aplicação que instancie um objeto dessa classe e o cliente pode utilizar os métodos disponibilizados pelo serviço como se fossem métodos locais. O cliente WCF pode acessar *Web Services* disponibilizados tanto por servidores WCF bem como qualquer outro Web Service, como por exemplo *Web Services* Java, garantindo a interoperabilidade entre diferentes plataformas.

Embora esse mecanismo seja interessante para prover interoperabilidade, não deve ser utilizado para o desenvolvimento de aplicações de alto desempenho, conforme visto em (PURE MPI.NET, 2008).

3.7.3 *Threads*

Em C# é possível realizar programação *multithread* através da classe `Thread`. Para criar uma *thread*, instancia-se um novo objeto do tipo `Thread`, o qual recebe como argumento um delegado, contendo o método que irá executar quando a *thread* for inicializada. Diferentemente de Java, qualquer método de uma classe pode ser utilizado como ponto de partida para uma nova *thread*, não sendo necessário implementar um método específico como, por exemplo, o método `run()` de Java.

Para inicializar a execução de uma *thread* é necessário que seja invocado o método `Thread.Start`. Uma vez inicializada, ela pode ser suspensa, resumida ou abortada. Para aguardar o término de uma *thread* utiliza-se o método `Thread.Join`.

A classe `Thread` possui ao menos uma instância por programa, uma vez que cada linha de execução de um programa é tratada como uma *thread*. É possível obter uma referência para a *thread* corrente através do método `Thread.CurrentThread`.

No entanto, *theads* se limitam ao paralelismo de métodos em memória compartilhada.

3.7.4 *Task Parallel Library*

A biblioteca TPL contida na versão 4 do Framework .NET (a ser lançada), consiste em uma biblioteca voltada para o paralelismo em nível de tarefas. Ela possui um conjunto de métodos voltados a simplificar a construção de programas paralelos, por exemplo, o método `Invoke()` permite que diversos métodos sejam chamados de forma concorrente, como visto no Programa 12. Nesse exemplo os três métodos são executados de forma concorrente por *threads*, porém a criação e inicialização das *threads* ocorre de forma transparente. É importante explicar que o operador `()=>` é chamado de `Action` o qual é um tipo de delegado.

Programa 12 Um exemplo de utilização do método Invoke().

```
Parallel.Invoke(
            ()=> MetodoA(),
            ()=> MetodoB(),
            ()=> MetodoC());
```

A abstração básica na TPL são as tarefas e não as *threads*, e sua relação com *threads* não é direta, podendo uma *thread* executar uma ou mais tarefas. Além dessa estrutura existe também o paralelismo de dados, o qual pode ser realizado através dos métodos `ForEach()` e `For()` da classe `Parallel`. Esse mecanismo estará disponível na próxima versão do Framework .NET a ser lançada em 2010.

3.8 Conclusões sobre o Capítulo

Nesse capítulo foram vistas diversas características da plataforma .NET, em especial sobre a ótica da linguagem de programação C#. Foram explorados os mecanismos básicos da plataforma, os quais são a CLR e MSIL demonstrando sua importância para interoperabilidade e desempenho. Também foi discutida a importância do gerenciamento de memória, tendo seu funcionamento explicado de forma resumida.

Foram demonstrados alguns mecanismos oferecidos pelo Framework voltados para simplificar a programação, dos quais podemos destacar os tipos genéricos, os quais permitem a reusabilidade de código de forma simples e eficiente. A serialização automática também é um mecanismo importante uma vez que permite comunicar ou persistir objetos de maneira simples.

Por fim foram vistos os mecanismos voltados para a computação paralela e distribuída permitindo concluir que os mecanismos voltados para a programação paralela servem somente para memória compartilhada e os mecanismos para programação distribuída não possuem alto desempenho. A plataforma .NET não oferece mecanismos que permitam o desenvolvimento de aplicações paralelas de alto desempenho em memória distribuída. Tendo em vista que MPI é um padrão *de facto*, a biblioteca MPI.NET tem sido desenvolvida para suportar MPI na plataforma .NET. No entanto, ela não possibilita a criação dinâmica de processos. O capítulo seguinte detalha esse problema e a solução que este trabalho propõe.

4 CRIAÇÃO DINÂMICA DE TAREFAS MPI.NET

Este capítulo discorre sobre a criação dinâmica de tarefas na biblioteca MPI.NET. Para estender a norma MPI para outras linguagens de programação foram pesquisadas diversas bibliotecas. Dentre elas, a biblioteca MPI.NET possui algumas características desejadas (API compatível com a linguagem de programação à qual a biblioteca se destina, desempenho aceitável, suporte completo a MPI-1 e suporte a comunicação de objetos). No entanto, ela possui uma lacuna quando se trata de suporte a norma MPI-2.

A norma MPI-2 traz diversas funcionalidades desejadas para um ambiente de programação moderno. Dentre elas podemos destacar a criação dinâmica de processos, a qual simplifica a programação de certas classes de aplicações como por exemplo Divisão & Conquista e Mestre/Escravo, além disso, permite um melhor balanceamento de carga entre as máquinas utilizadas para a computação paralela uma vez que os processos são criados e distribuídos dinamicamente entre as máquinas.

O modelo de programação paralela baseada em tarefas tem se mostrado cada vez mais presente na programação paralela. Esse modelo é suportado pelas principais bibliotecas de programação paralela orientadas a objetos. Tarefa pode ser descrita como uma unidade sequencial abstrata que possui um bloco de instruções (LIMA; MAILLARD, 2008). Uma tarefa pode depender de outras tarefas e criar novas tarefas.

O presente trabalho propõe explorar a criação dinâmica de processos (para os quais são mapeadas tarefas) na biblioteca MPI.NET, avaliando as diversas alternativas e realizando testes de desempenho.

4.1 A Biblioteca MPI.NET

A biblioteca MPI.NET, como discutida brevemente no capítulo 2, é uma biblioteca MPI para a plataforma .NET, a qual segue a API MPI para C++. Ela abstrai as chamadas MPI, diminuindo a quantia de parâmetros necessários, através da utilização de mecanismos presentes na plataforma .NET, em especial a serialização automática, *reflection* e genéricos.

Essa biblioteca é executada sobre as distribuições MPI para linguagem de programação C, as utilizando como base para executar as chamadas MPI e gerenciar os processos. Essa abordagem possui a vantagem de permitir a utilização de distribuições MPI consolidadas, tirando proveito de sua qualidade e desempenho, além de permitir a interoperabilidade entre programas escritos em C# e programas escritos em C, C++ e Fortran.

Embora ela utilize as bibliotecas MPI-C como base, seu modelo de programação é orientado a objetos, onde todas as funções MPI são encapsuladas dentro de classes. Suas principais classes serão discutidas nas próximas subseções.

4.1.1 A Classe Environment

A classe `Environment` é responsável por inicializar e finalizar a biblioteca MPI-C, utilizada como base, além de ser responsável por obter informações da plataforma de execução, como por exemplo, o nome do *host* (função `MPI_Get_processor_name()`) e o tempo (função `MPI_Wtime()`).

Quando o programa MPI estiver contido dentro de um bloco `Environment` a inicialização/finalização da biblioteca MPI ocorrem de forma automática, conforme o Programa 13. Internamente a classe `Environment` converte o parâmetro `args` para o formato de caracteres da linguagem C, e após chama a função `MPI_Init_thread()` (versão da função `MPI_Init()` que permite especificar o tipo de *threading* conforme visto na subsubseção 2.1.1.4). A finalização da biblioteca MPI-C é realizada através de uma chamada automática ao método `Dispose()` (método que deve ser implementado por classes que implementem a interface `IDisposable` e é chamado automaticamente na finalização do bloco), que realiza uma chamada a função MPI_Finalize() da biblioteca MPI-C.

Programa 13 Exemplo de utilização de um bloco Environment.

```
static void Main(string[] args)
{
  using (new MPI.Environment(ref args))
  {
    // CÓDIGO FONTE DO PROGRAMA MPI.
  }
}
```

Além de possuir a capacidade de inicializar/finalizar a biblioteca MPI-C de forma automática, esses procedimentos também podem ser feitos manualmente pelo usuário, assim como na API MPI padrão, conforme o Programa 14. Ao utilizar essa abordagem, o usuário deve chamar o método `Dispose()` para finalizar a biblioteca MPI-C.

Programa 14 Exemplo de inicialização/finalização manuais.

```
static void Main(string[] args)
{
  MPI.Environment environment = new MPI.Environment(ref args);
  // CÓDIGO FONTE DO PROGRAMA MPI.
  environment.Dispose();
}
```

Para definir o nível de *threading* desejado, o usuário deve passá-lo ao construtor da classe `Environment`, através da utilização da enumeração `enum` é um tipo que consiste em um conjunto de constantes nomeadas) `Threading`, conforme visto no Programa 15. Essa enumeração possui todas as variações presentes na norma MPI, e caso seja omitida, por padrão será escolhida `MPI_THREAD_SINGLE`.

Programa 15 Exemplo de utilização da enumeração Threading.

```
static void Main(string[] args)
{
  using (new MPI.Environment(ref args,Threading.Multiple))
  {
    // CÓDIGO FONTE DO PROGRAMA MPI.
  }
}
```

Uma vez visto o gerenciamento do ambiente MPI, as próximas subseções irão tratar das estruturas de comunicação oferecidas pela biblioteca.

4.1.2 A Classe DatatypeCache

Essa classe realiza a verificação dos tipos de dados passados para os métodos de comunicação. Como a comunicação é realizada por uma biblioteca MPI-C, é necessário que a biblioteca MPI.NET descubra o tipo de dado em MPI-C que está sendo comunicado.

Para descobrir o tipo de dado a ser comunicado, essa classe possui um dicionário `Dictionary` (representa uma coleção de chaves e valores) contendo os tipos primitivos de C# e seus respectivos `MPI_Datatype`. Através do método `GetDatatype()` dessa classe a biblioteca descobre se o tipo de dado a ser comunicado é um `MPI_Datatype`. Para isso, esse método recupera o tipo de dado a ser comunicado utilizando o método `GetType()` do .NET. Uma vez recuperado, o tipo de dado é pesquisado no dicionário. Caso seja um `MPI_Datatype`, retorna o tipo de dado MPI equivalente, caso contrário retorna `null`. Uma vez descoberta essa informação, cabe ao método de comunicação decidir como será realizada a comunicação, assunto que será discutido na próxima subseção.

4.1.3 A Classe Communicator

A classe `Communicator` é responsável por encapsular as comunicações na biblioteca MPI.NET. Os métodos de comunicação implementados nessa classe possuem rotinas utilizadas para receber os dados a serem comunicados, convertê-los em dados compatíveis com C, e repassá-los para a biblioteca MPI-C a qual executa a comunicação.

Um exemplo de utilização da classe `Communicator` está na realização de troca de mensagens ponto-a-ponto, ao realizar o envio/recebimento de uma mensagem através dos métodos `Send()/Receive()`, internamente esses métodos chamam o método `GetDatatype()` da classe `MPI_Datatype`, para verificar se o dado a ser comunicado é um tipo MPI primitivo. Caso não seja, é realizado um processo automático de serialização/desserialização para enviá-lo/recebê-lo, sendo que, o dado passa a ser enviado/recebido como um vetor do tipo `MPI_Byte`. Uma vez que existe esse mecanismo de serialização automático, é possível comunicar qualquer tipo de dado serializável de forma transparente utilizando a biblioteca MPI.NET.

O sobrecusto obtido pelo envio de dados não conhecidos pela biblioteca MPI-C é o custo de uma mensagem do tipo `MPI_INT` mais o tempo de serialização dos dados. Essa serialização não é realizada para um vetor qualquer, e sim para uma área de memória não gerenciada pelo coletor de lixo (utilizando a classe `UnmanagedMemoryStream`), permitindo que seja enviado um ponteiro para a biblioteca MPI-C com garantia que os dados não serão movidos na memória até o término da comunicação. Uma vez concluída, a área de memória é desalocada.

Para realizar o envio de vetores de tipos primitivos, basta protegê-los do coletor de lixo através da classe `GCHandle`, passando para a biblioteca MPI-C um ponteiro para o vetor protegido. Tipos primitivos singulares utilizam uma implementação própria da biblioteca MPI.NET escrita em MSIL para recuperar um ponteiro para eles, uma vez que ficam armazenados na pilha.

As comunicações coletivas também são implementadas seguindo esse modelo. Elas possuem diversas simplificações, como por exemplo o método `Gather()` o qual necessita de cinco parâmetros a menos que o mesmo método em C++ uma vez que os outros parâmetros podem ser inferidos. Isso ocorre porque cinco dos sete parâmetros necessá-

rios por esse método em C++ são para descrever tamanho de *buffers* e *tipos* de dados, informações que podem ser facilmente inferidas.

Os métodos de redução também tiram proveito da capacidade de *reflection* da linguagem C# sendo que, por exemplo, o método `Reduce()` necessita da metade dos parâmetros necessários na mesma operação em C++. As operações de redução são encapsuladas em métodos estáticos da classe `Operation`, as quais devem ser passadas como parâmetro para os métodos de redução, por exemplo, `int resultado = comm.Reduce (x, Operation<int>.Add, 0)`.

Uma vez vistos os métodos de comunicação, é importante discorrer sobre a classe responsável por realizar a interface entre a biblioteca MPI.NET e a biblioteca MPI-C, o que será feito na próxima subseção.

4.1.4 A Classe Unsafe

Nessa classe são implementados os cabeçalhos das funções a serem chamadas na biblioteca MPI-C utilizada como base pela biblioteca MPI.NET. Essas funções são chamadas pelos métodos das outras classes, como por exemplo, o construtor da classe `Environment` o qual chama o método `MPI_Init_thread()` da classe `Unsafe`, sendo esse executado diretamente sobre a biblioteca MPI-C.

O usuário não tem acesso aos métodos contidos nessa classe, sendo todos eles acessados pelas classes internas devido à necessidade de tratar os dados para torná-los compatíveis com C. Um exemplo está no método `Send()`, da classe `Communicator`. Esse método recebe apenas três parâmetros na biblioteca MPI.NET, no entanto, a função em C necessita de seis parâmetros conforme visto na Subseção 2.1.2. Cabe ao método interno da biblioteca obter os parâmetros que faltam, processá-los e por fim chamar a função `MPI_Send()` na classe `Unsafe` a qual será executada diretamente sobre a biblioteca MPI-C.

Para que esse acesso à biblioteca MPI-C seja possível é necessário utilizar o suporte a chamadas nativas da plataforma .NET, definindo a biblioteca que irá ser chamada e o cabeçalho de cada função MPI a ser chamada pela biblioteca. Essa classe deve ser compilada com a *flag* **unsafe** do compilador C#, sendo que dentro desse modo é permitida a utilização de ponteiros C, os quais devem ser sempre protegidos do coletor de lixo.

Visto a importância da criação dinâmica de tarefas e as estruturas básicas da biblioteca MPI.NET, a próxima seção discorre sobre a proposta desse trabalho a qual é explorar na biblioteca MPI.NET a utilização do mecanismo de criação dinâmica de tarefas.

4.2 Projeto e Implementação da Criação Dinâmica de Tarefas na Biblioteca MPI.NET

Uma vez que a biblioteca MPI.NET demonstrou bom desempenho combinado a uma API de programação orientada a objetos deixando a desejar no quesito criação dinâmica de processos, essa seção discute o projeto, implementação e testes das funções MPI responsáveis por criar e gerenciar processos dinamicamente na biblioteca MPI.NET, sendo elas: `MPI_Comm_spawn()`, `MPI_Comm_spawn_multiple()` e `MPI_Comm_get_parent()`.

Em um algoritmo paralelo divide-se um problema inicial em partes menores as quais podem ser executadas concorrentemente, de forma a diminuir o tempo de resolução do problema. Podemos definir **tarefa** como sendo uma dessas partes do problema. Em MPI essas tarefas são mapeadas diretamente para processos. Essas tarefas devem ser mapeadas

entre os recursos computacionais disponíveis, o que é feito pelo gerenciador de processos do MPI de forma transparente. Como visto previamente no Capítulo 2 essas tarefas se comunicam através da troca de mensagens.

Devido à linguagem de programação C++ ser orientada a objetos assim como C#, foi utilizada como base para a implementação da criação dinâmica de processos em C# a API MPI-2 para C++, embora a biblioteca chame efetivamente funções MPI-C. A API MPI-2 para C++ encapsula as funções de criação dinâmica de processos dentro da classe `Comm` a qual é equivalente à classe `Communicator` da biblioteca MPI.NET. Portanto as funções de criação dinâmica de processos foram encapsuladas dentro da classe `Communicator`, de forma a seguir o padrão MPI.

Como as chamadas MPI são executadas efetivamente por uma biblioteca MPI-C, através da utilização de chamadas nativas, as interfaces nativas que permitem essas chamadas devem ser criadas na classe `Unsafe`. Essas interfaces traduzem os tipos de dados C# para os tipos de dados C e realizam a chamada de função em C. Para cada um dos parâmetros utilizados pelas interfaces, foi especificado o tipo de dado C# mais compatível em relação ao parâmetro da chamada nativa em C. Essa interface somente especifica os tipos de dados da função C, deixando o processamento desses dados para ser realizado pelos métodos que a chamam.

Como algumas variáveis são passadas como referência para a interface nativa, elas devem ser protegidas do coletor de lixo. Para proteger variáveis do coletor de lixo um objeto do tipo `GCHandle` deve ser criado. Esse objeto deve inicializado como `GCHandleType.Pinned`. Esse método recebe como parâmetro a variável a ser protegida permitindo que seja recuperada a referência para ela e prevenindo que o gerenciador de memória a mova de local na memória. A interface nativa recebe o objeto `GCHandle`, recuperando o endereço da variável como se fosse um ponteiro. Após a função nativa ser chamada, o `GCHandle` é liberado e os dados voltam a ser gerenciados pelo coletor de lixo. Esse procedimento é realizado por todas as variáveis passadas como referência para evitar que sejam movidas na memória enquanto são acessadas pela biblioteca nativa.

4.2.1 Implementação da Primitiva MPI_Comm_spawn

O método `MPI_Comm_spawn()`, responsável por criar **n** processos de determinado programa, foi implementado na biblioteca com o nome `Spawn`, seguindo o padrão de nomenclatura adotado pelos outros métodos já implementados na biblioteca MPI.NET e o padrão de nomes de métodos da linguagem de programação C# os quais devem começar por letra maiúscula. Foram criadas diversas interfaces para esse método, permitindo que o usuário utilize desde uma interface mais simples do que a presente na API MPI para e C e C++ até uma interface tão completa quanto a de C++. Para isso foram escritas cinco sobrecargas de método, sendo elas:

1. `public Communicator Spawn(String command)`: cria uma nova tarefa do programa `command`;

2. `public Communicator Spawn(String command, int maxprocs)`: especifica a quantidade máxima de tarefas a serem criadas em `maxprocs`;

3. `public Communicator Spawn(String command, int maxprocs, int root)`: especifica qual dos processos MPI.NET irá criar a nova tarefa em `root` (nas outras chamadas o padrão é 0);

4. `public Communicator Spawn(String command, String[] argv, int maxprocs)`: Envia argumentos à nova tarefa em `argv`;

5. `public Communicator Spawn(String command, String[] argv, int maxprocs, int root)`: combina o conteúdo de todas outras chamadas.

As sobrecargas do método implementadas na biblioteca MPI.NET retornam o comunicador para o processo pai via `return` assim como na API MPI-C++, porém diferente da API MPI-C a qual retorna o comunicador via ponteiro em um dos seus parâmetros. Como efetivamente a função é executada pela biblioteca MPI-C é necessário que no método `Spawn()` seja criado uma nova variável de comunicador MPI a qual deve ser passada como 'ponteiro' para a interface nativa. Para isso, o parâmetro `*intercomm` é passado como `out MPI_Comm newComm` para a biblioteca nativa, aonde `out` especifica que a variável `newComm` irá ser tratada como ponteiro pela biblioteca nativa. Dessa forma a biblioteca MPI-C irá executar normalmente como se fosse chamada por um programa escrito em C, criando um novo comunicador e o retornando através da variável `newComm`.

Após receber o novo comunicador, é necessário convertê-lo para um comunicador orientado a objetos, o qual é utilizado pelo usuário, uma vez que o usuário não tem acesso ao comunicador MPI nativo na biblioteca MPI.NET. Para isso um novo objeto do tipo `Intercommunicator` da biblioteca MPI.NET é alocado e tem sua variável `Comm`(variável responsável por armazenar o comunicador MPI nativo) sobrescrita pelo comunicador MPI nativo. Dessa forma, as comunicações executadas sobre o `Intercommunicator` serão realizadas internamente sobre a variável `Comm`. Por exemplo, ao realizar o envio de uma mensagem sobre um comunicador MPI.NET, ao chamar a interface nativa será passado como parâmetro a variável `Comm`. Por fim o novo comunicador MPI.NET é retornado via `return` e pode ser utilizado pelo usuário como qualquer outro comunicador MPI.NET, tendo todas as funcionalidades contidas na norma MPI para C++ além das funcionalidades contidas na biblioteca MPI.NET como, por exemplo, a comunicação de objetos.

O parâmetro `array_of_errcodes[]` da interface MPI para C não aparece na 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, não havendo a necessidade de disponibilizar um vetor de códigos de erros para o usuário.

Os parâmetros passados pelo usuário nos métodos `Spawn()` devem passar por modificações. 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* contendo o código ASCII de cada um dos caracteres da string. Esse vetor de *bytes* deve ser protegido do coletor de lixo, uma vez que será passado como referência para o C. A variável `argv` deve passar pelo mesmo procedimento.

4.2.2 Implementação da Primitiva MPI_Comm_spawn_multiple

A função `MPI_Comm_spawn_multiple()` permite que sejam lançados processos de diversos programas MPI em uma única chamada. Sua API para linguagem de programação C é: `int MPI_Comm_spawn_multiple (int count, char *array_of_commands[], char** array_of_argv[], int array_of_maxprocs[], MPI_Info array_of_info[], int root, MPI_Comm comm, MPI_Comm *intercomm, int array_of_errcodes[])`.

As principais diferenças em relação à função `MPI_Comm_spawn()` estão nos argumentos `array_of_commands` e `array_of_maxprocs[]` os quais definem quais programas serão executados e quantos processos de cada programa serão criados.

Sua implementação se deu de forma similar na biblioteca MPI.NET onde o tratamento dos dados sofreu pequenas modificações devido aos novos vetores. Foram implementadas cinco sobrecargas de método similares as do método `Spawn()`, sendo a mais simples `SpawnMultiple(String[] command)` na qual é lançado um processo de cada programa, tendo os outros dados necessários inferidos via *Reflection*.

4.2.3 Implementação da Primitiva MPI_Comm_get_parent

Por fim o método `GetParent` o qual serve para recuperar o comunicador do processo pai nos processos filhos foi implementado. A implementação dele foi mais simples uma vez que ele retorna via referência o comunicador, portanto sua interface nativa foi implementada da seguinte maneira: `MPI_Comm_get_parent(out newComm)` onde a variável `newComm` é criada previamente e protegida do gerenciador de memória.

Como nos outros dois métodos o novo comunicador passou a ser retornado via `return` e a interface da função ficou da seguinte maneira: `public static Communicator GetParent()` sendo exatamente igual a interface do MPI-C++.

4.2.4 Programação com MPI.NET-Spawn

O desenvolvimento de programas que utilizem a criação dinâmica de tarefas na biblioteca MPI.NET segue o mesmo padrão utilizado pelo MPI C++, porém contendo uma API com maiores abstrações e maior expressividade, uma vez que um único método pode enviar tanto tipos primitivos quanto objetos. O modelo também é o mesmo utilizado em MPI-C, porém orientado a objetos.

Ao reescrevermos os Programas 3 e 4 demonstrados no Capítulo 2 em C++, obteremos os Programas 16 e 17.

Programa 16 Programa pai em C++

```
/* PROGRAMA PAI */
int main(int argc, char **argv)
{
  long n;
  char cmd[] = "./filho";
  MPI_Comm intercomm;

  MPI::Init(argc, argv);

  // CRIA UM PROCESSO DINAMICAMENTE DO PROGRAMA FILHO
  MPI::COMM_SELF.Spawn(cmd, MPI_ARGV_NULL, 1, MPI_INFO_NULL, 0);
  intercomm.Send(&n, 1, MPI_LONG, 0, 0);
  intercomm.Recv(&n, 1, MPI_LONG, 0, 1);

  MPI::Finalize();
}
```

Utilizando a implementação da criação dinâmica de tarefas realizada no presente trabalho, esses programas a serem reescritos em C# podem ser vistos nos Programas 18 e 19.

Nesses pequenos programas é possível perceber uma grande diminuição na quantidade de parâmetros necessários para realizar a mesma operação que no programa escrito em C++ ao mesmo tempo que o modelo de programação é mantido. Caso o usuário deseje utilizar mais parâmetros para programas onde sejam necessários criar mais processos

Programa 17 Programa filho em C++

```
/* PROGRAMA FILHO */
int main(int argc, char **argv)
{
  long n;
  MPI_Comm parentcomm;

  MPI::Init(argc, argv);

  // RECUPERA O COMUNICADOR COM O PROCESSO PAI
  parentcomm = MPI::Comm::Get_parent ();
  intercomm.Recv(&n, 1, MPI_LONG, 0, 0);
  intercomm.Send(&n, 1, MPI_LONG, 0, 1);

  MPI::Finalize();
}
```

Programa 18 Programa pai em C#

```
/* PROGRAMA PAI */
public static void Main(string[] args)
{
  long n;
  string cmd[] = "filho.exe";
  using (new MPI.Environment(ref args))
  {
    Communicator intercomm;

    // CRIA UM PROCESSO DINAMICAMENTE DO PROGRAMA FILHO
    intercomm = Communicator.self.Spawn(cmd);
    intercomm.Send(n, 0, 0);
    n = intercomm.Receive<long>(0, 1);
  }
}
```

Programa 19 Programa filho em C#

```
/* PROGRAMA FILHO */
public static void Main(string[] args)
{
  long n;
  Communicator parentcomm;

  using (new MPI.Environment(ref args))
  {
    // RECUPERA O COMUNICADOR COM O PROCESSO PAI
    parentcomm = Communicator.GetParent();
    n = parentcomm.Receive<long>(0, 0);
    parentcomm.Send(n, 0, 1);
  }
}
```

dinamicamente ou passar argumentos para os processos basta utilizar as sobrecargas de método.

4.2.5 Avaliação de desempenho

Em uma primeira avaliação de desempenho foi testado o tempo de criar processos MPI.NET dinamicamente utilizando a biblioteca proposta em relação ao tempo necessário para criar processos C++ dinamicamente utilizando a biblioteca MPI-C++. Para isso foi criado e executado um benchmark chamado Spawn-n, o qual cria **n** tarefas dinamicamente através de uma única chamada ao método `Spawn()`. Os resultados do teste podem ser vistos na Figura 4.1. A plataforma de execução para a realização do teste foi um cluster com a seguinte configuração:

- **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

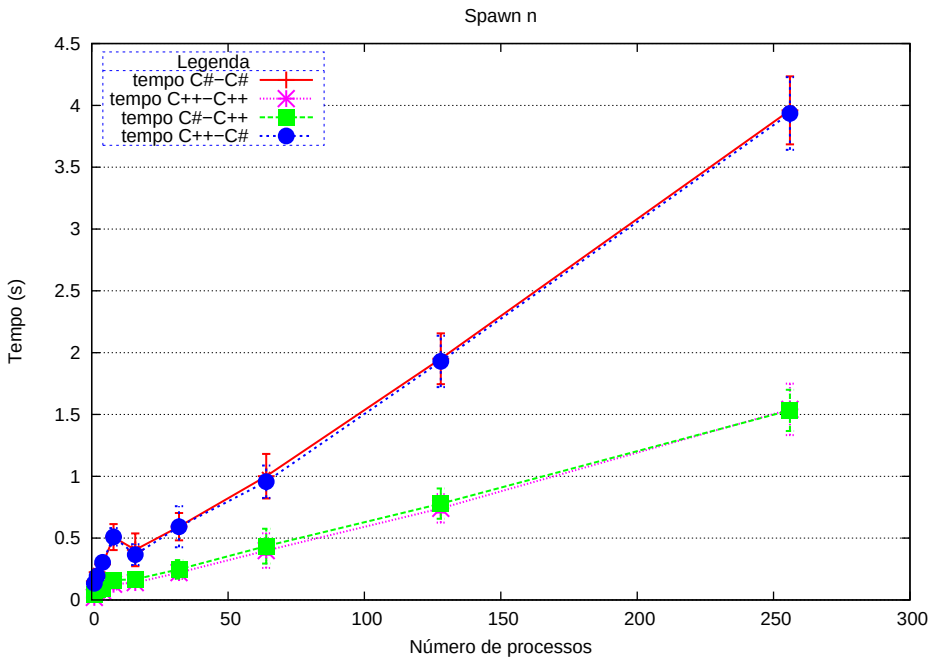

Figura 4.1: Execução do programa Spawn-n. Esse gráfico demonstra o tempo necessário para criar n processos MPI-C# e MPI-C++. O gráfico também demonstra o tempo necessário para criar n processos MPI-C# à partir de um programa C++ e vice-versa.

Através da Figura 4.1 é possível perceber que o tempo de criar um novo processo em C# é muito alto. Isso se deve ao fato do .NET inicializar uma nova máquina virtual para cada processo, o que causa um grande sobrecusto. O gráfico também permite visualizar

que entre a chamada ao método `Spawn()` no programa do usuário em C#, o processamento dos argumentos, e a chamada do método na biblioteca nativa não existe sobrecusto significativo, uma vez que a biblioteca MPI-C++ possui praticamente o mesmo desempenho lançando processos C# que a biblioteca MPI.NET, e o mesmo vale para a biblioteca MPI.NET lançando processos C++. Com essa comparação é possível verificar o verdadeiro sobrecusto introduzido pela biblioteca sobre a chamada nativa. Por fim, o gráfico demonstra que o tempo da biblioteca MPI nativa inicializar os processos é majoritário, enfatizando que o problema de desempenho está na inicialização do novo processo C#.

Uma vez detectado esse problema de desempenho, mostrou-se necessária a pesquisa de soluções as quais otimizem a inicialização dos processos MPI.NET, pois esse sobrecusto pode causar grande impacto nos programas Divisão & Conquista.

4.3 Conclusões sobre o Capítulo

Esse capítulo discorreu sobre a criação dinâmica de tarefas, bem como alguns modelos de programação que tiram proveito desse mecanismo. A implementação da biblioteca MPI.NET foi explorada, demonstrando como foram implementados suas principais classes.

Em seguida foi discutida a proposta desse trabalho e demonstrada sua implementação na biblioteca MPI.NET. Foi discutida a programação com criação dinâmica de tarefas na biblioteca MPI.NET, onde foi demonstrado em um exemplo simples que o padrão de programação MPI é mantido. Também foi realizado um teste de desempenho preliminar a fim de verificar o tempo de criação dos processos na biblioteca MPI.NET em relação ao tempo de criação dos processos em MPI-C++.

Uma vez que o desempenho da biblioteca MPI.NET com criação dinâmica de processos se mostrou insatisfatório, o próximo capítulo trata da proposta e implementação de um algoritmo voltado para otimizar o desempenho da biblioteca.

5 OTIMIZAÇÃO DE DESEMPENHO

Este capítulo trata das propostas para otimizar o desempenho da criação dinâmica de processos na biblioteca MPI.NET. Uma vez detectado que o sobrecusto no desempenho se concentra na inicialização dos processos, mostrou-se necessário otimizá-la. Para isso foram estudadas três soluções:

- Pré compilar os programas C#;
- Modificar a máquina virtual (Mono) do .NET;
- Lançar *threads* e processos de forma alternada.

A primeira solução é a mais simples de ser aplicada, uma vez que ao chamar o compilador C# basta passar como parâmetro a *flag* `aot`. Ao identificar essa *flag* o compilador irá gerar um arquivo do tipo `.exe.so` o qual irá conter o código nativo do programa. No entanto, nem todos métodos são compilados e nos testes realizados não houve diferença de desempenho. Isso se dá porque a máquina virtual continua tendo que ser inicializada para executar os programas o que possui tempo majoritário.

Uma outra solução de compilação é a utilização da *flag* `full-aot`, a qual compila o programa por inteiro. Dessa forma, ele passa a ser executado sem a necessidade de utilizar a máquina virtual. Porém, esse método não pode ser utilizado em programas que utilizem código dinâmico, o que é amplamente utilizado pela biblioteca MPI.NET, inviabilizando a utilização dessa *flag*.

A segunda solução é a mais sofisticada, uma vez que seria necessário modificar o código fonte da máquina virtual (Mono) que é *opensource*. A máquina virtual poderia ser modificada a fim de permitir que processos do mesmo programa não criassem uma nova máquina, e sim executassem sob a forma de *threads*, uma vez que não é possível executar mais de um processo por máquina virtual. No entanto, essa solução não é viável pois seria necessário muito trabalho técnico que estaria sujeito a não ser reaproveitado em novas versões da máquina virtual, a qual durante a realização do presente trabalho passou por 11 versões.

A última solução permite uma aproximação da segunda solução, porém, implementada dentro da biblioteca MPI.NET. O trabalho (LIMA; MAILLARD, 2008) propõe a utilização de *threads* no MPI-C++ para otimizar o desempenho, tendo obtido excelentes resultados conforme visto na Figura 5.1, a qual demonstra o tempo de execução do cálculo do 20º número da sequência de Fibonacci, gerando 13.530 tarefas. No gráfico é possível observar que quanto menor for o número de *cores* melhor é o desempenho obtido pela utilização de *threads* em relação ao desempenho de processos, isso se dá por dois motivos: primeiro porque criar *threads* é mais rápido que criar processos e segundo porque

as comunicações são feitas em memória compartilhada ao invés da rede. Ao aumentar o número de *cores* o desempenho com a utilização de *threads* se mantém estável e o desempenho com a utilização de processos melhora uma vez que são utilizados mais recursos computacionais. No entanto, o desempenho com processos não atinge o desempenho obtido com *threads*.

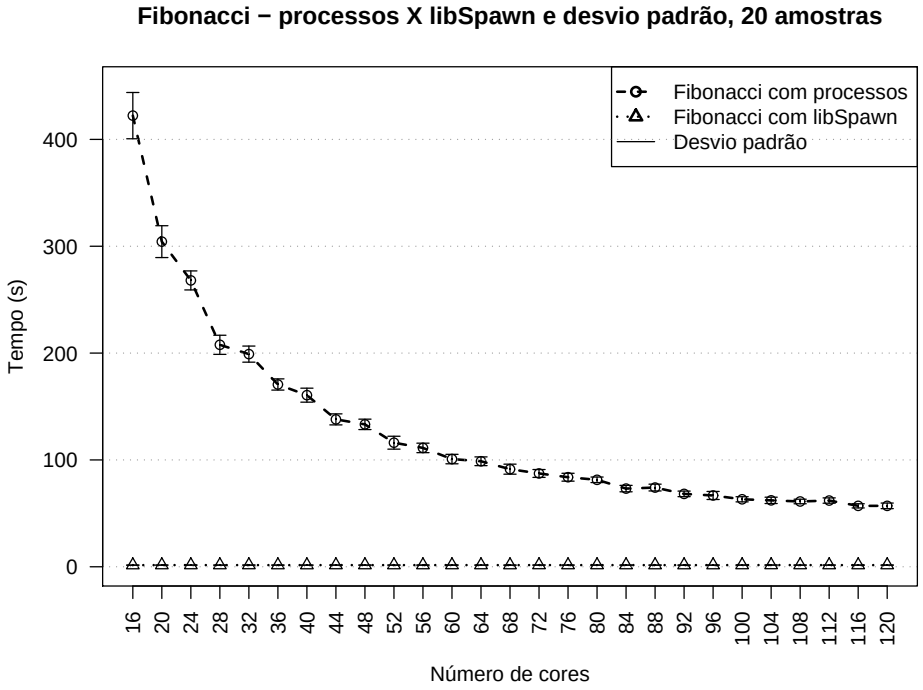

Figura 5.1: Tempos de execução para o cálculo do 20º número da sequência de Fibonacci variando o número de cores em um cluster com 4 cores por nós (LIMA; MAILLARD, 2008)

5.1 Algoritmo Proposto

O algoritmo proposto no presente trabalho, tem como objetivo otimizar o desempenho do `Spawn()` na biblioteca MPI.NET. Seu principal foco é otimizar o tempo de inicialização das tarefas. Ele realiza a criação intercalada de *threads* e processos da seguinte forma:

$$\texttt{Spawn()} = \begin{cases} \text{Cria Thread(s)}, & \textit{se} \text{ o programa é um processo;} \\ \text{Cria Processo(s)}, & \textit{se} \text{ o programa é uma thread.} \end{cases}$$

Dessa forma o número de *threads* e processos no ambiente de execução é balanceado, permitindo que seja tirado proveito do menor custo de comunicação entre *threads* e do menor tempo de criação das mesmas, e, ao mesmo tempo, o ambiente computacional será carregado de forma homogênea pelo escalonador da distribuição MPI, o qual é responsável por alocar os recursos. Um diagrama ilustrando o funcionamento do mecanismo proposto pode ser visto na Figura 5.2.

Ele funciona de maneira diferente ao algoritmo proposto por (LIMA; MAILLARD, 2008), uma vez que esse foi criado para realizar o controle de granularidade automático

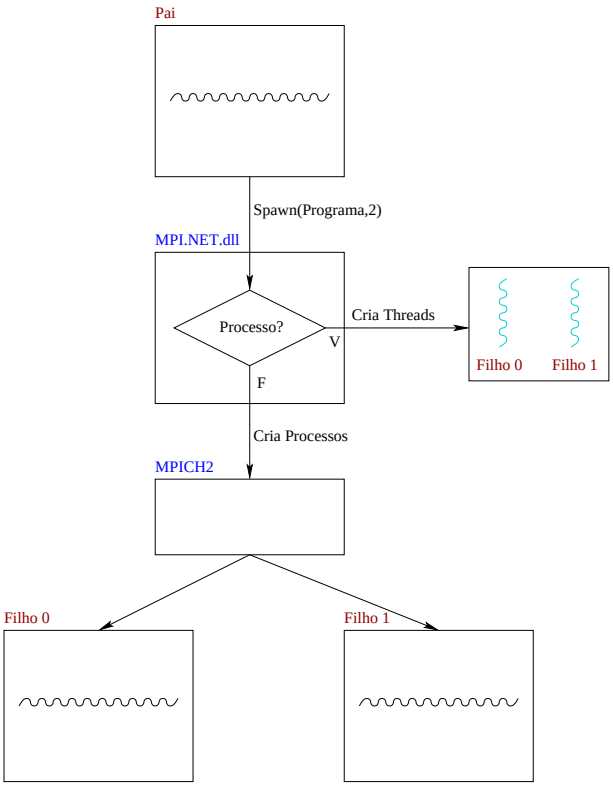

Figura 5.2: Diagrama representando o funcionamento do método Spawn com *threads* e processos intercalados na biblioteca MPI.NET.

de aplicações MPI através de *threads*, agrupando tarefas em uma mesma máquina até que ela esteja com toda sua capacidade de processamento utilizada. O algoritmo controla a criação de *threads*/processos de acordo com os recursos da plataforma de execução, monitorando ativamente a carga de trabalho do sistema para decidir se na execução de um `Spawn()` serão criadas *threads* ou processos.

5.2 Implementação do Algoritmo na Biblioteca MPI.NET

O algoritmo foi implementado dentro da biblioteca MPI.NET, com o objetivo ser transparente para o programador. Sua implementação pode ser dividida em três etapas: criação das *threads*, construção do comunicador e construção das comunicações. Essas etapas serão descritas nas próximas sub-subseções.

5.2.1 Criação das Threads

Para decidir se uma chamada ao método `Spawn()` irá criar *threads* ou processos basta testar o comunicador no qual o método foi chamado a fim de detectar se é um comunicador de *threads* ou de processos (essa diferença será explicada em 5.2.1). Esse procedimento é transparente para o usuário, o qual chama o método normalmente.

Caso o comunicador seja um comunicador de processos então as tarefas a serem criadas pelo `Spawn()` serão criadas como *threads*, para isso utiliza-se o mecanismo de *Reflection* para obter algumas informações:

1. Recupera-se o tipo (`Type`) da classe que possui o ponto de entrada do programa

(`Main()`);

2. Através do objeto `Type` recuperado, recupera-se o método `Main()` através do método `GetMethod()`, passando como parâmetro para ele uma *query* solicitando o método `Main()`, seja ele público ou privado, o qual é armazenado em um objeto do tipo `MethodInfo`;

Uma vez recuperados o método `Main()` e o `Type`, instancia-se um objeto do tipo recuperado através da classe `Activator` (responsável por criar objetos). Em seguida é construído um vetor de *threads* do tamanho da quantia de *threads* a serem criadas. Por fim é realizado um laço o qual cria e inicializa as *threads*, utilizando um *delegate* para invocar seu método `Main()`, o qual recebe como parâmetro os argumentos passados pelo usuário ou uma *string* vazia para o caso do usuário não ter passado argumentos. A lista de *threads* é armazenada na classe `Environment` a qual é utilizada na finalização da biblioteca.

Caso o comunicador seja um comunicador de processos, então é realizada uma chamada nativa ao método `Spawn()`, o qual irá retornar um comunicador MPI nativo sendo tratado como discutido na Subseção 4.2.1, podendo ser utilizado normalmente pelo processo pai e pelos processos filhos.

5.2.2 Construção do Comunicador

Existem duas abordagens possíveis para a implementação do comunicador entre as *threads*:

1. Utilizar comunicadores MPI nativos;

2. Criar um comunicador próprio em memória compartilhada.

Na primeira abordagem poderíamos clonar o comunicador `self` do processo pai, e utilizar *tags* para diferenciar as origens e destinos das mensagens. Para essa abordagem (a qual chegou a ser prototipada) seria necessário calcular *tags* únicas para cada mensagem levando em consideração a origem, destino e a própria *tag* passadas pelo usuário. O problema com essa abordagem está no grande custo de clonar o comunicador `self` do processo pai, o que diminui consideravelmente o ganho de desempenho na criação das *threads*, sendo que todas comunicações em memória compartilhada teriam que passar pela biblioteca MPI nativa além de dificultar a implementação de comunicações coletivas.

A segunda abordagem se mostrou mais natural para comunicação em memória compartilhada, embora exija a reescrita de todos métodos de comunicação MPI, é mais rápido diretamente no nível da biblioteca MPI.NET utilizando memória compartilhada. Essa abordagem foi adotada para o desenvolvimento do novo comunicador.

O comunicador de *threads* utiliza a classe `Communicator` da biblioteca MPI.NET, a qual teve que ser modificada para suportar o novo tipo de comunicação. Os métodos de comunicação foram alterados e algumas variáveis foram adicionadas. O comunicador de *threads* é criado antes das *threads* serem criadas, sendo retornado pelo método `Spawn()` para o processo pai e recuperado pelos filhos através do método `GetParent()`.

Dentre as variáveis adicionadas está a `isThread`, um booleano inicializado por padrão como *false* o qual é responsável por definir se o comunicador é de *threads* ou *processos*. Quando é criado um comunicador de *threads*, esse booleano é modificado, permitindo que o novo comunicador seja distinguido.

O comunicador `self` das *threads* é o mesmo comunicador `self` nativo do processo pai. Dessa forma é possível que as *threads* chamem nativamente o método `Spawn()`

nesse comunicador. Para que isso seja possível, é necessário inicializar a biblioteca MPI sinalizando por argumento que diversas *threads* irão acessar funções MPI ao mesmo tempo, conforme visto na Sub-subseção 2.1.1.4. Portanto quando uma *thread* realiza uma chamada ao `Spawn()`, é criado um novo comunicador MPI nativo o qual ela irá utilizar para comunicar com os processos filhos.

O *rank* MPI das *threads* é armazenado no atributo `Name` delas, sendo adicionado antes da inicialização. Para armazenar as mensagens trocadas no comunicador de *threads* foram criadas algumas listas: mensagens para o pai, mensagens para os filhos. Conforme visto na Sub-subseção 2.1.2.1, uma vez que o comunicador `parentcomm` deve ser estritamente utilizado para as comunicações entre pai e filhos, as mensagens enviadas pelos filhos para o pai são armazenadas na lista de mensagens do pai, e, as mensagens enviadas do pai para os filhos são armazenadas na lista de mensagens dos filhos.

O comunicador `world` das *threads* é uma cópia do comunicador de *threads*, porém, contendo uma *flag* booleana que informa que será utilizado para troca de mensagens somente entre elas. Essa cópia é realizada na classe `Environment` a qual é responsável por inicializar os comunicadores. Nesse comunicador todas as mensagens são armazenadas na lista de mensagens dos filhos. Os *ranks* dos filhos se mantém os mesmos nesse comunicador.

Uma vez que chamadas nativas são executadas sobre o comunicador `self` do pai, é imprescindível que a função `MPI_Finalize()` não seja chamada antes dos filhos terem encerrado suas execuções, o que poderia ocorrer caso o pai terminasse antes dos filhos. Para evitar que isso ocorra, na classe `Environment` do pai é mantida a lista com todas as *threads* criadas por ele, uma vez invocado o método `Dispose()` no pai, ele ficará trancado até que todas as *threads* na lista finalizem sua execução, para somente depois chamar a função `MPI_Finalize()`.

5.2.3 Comunicações

Uma vez que o comunicador de *threads* realiza as comunicações em memória compartilhada foi necessário reescrever os métodos de comunicação. Nem todas comunicações MPI foram prototipadas, embora seja possível prototipá-las de maneira mais simples do que em memória distribuída. Foram prototipadas as comunicações ponto-a-ponto bloqueantes.

Antes de falarmos nas comunicações é importante explicar como as mensagens em memória compartilhada são implementadas. Elas são representadas por objetos da classe `Message`, a qual armazena as informações de origem, destino, *tag* e tipo da mensagem além de armazenar os dados a serem comunicados. Os dados são armazenados em um objeto do tipo `Object` o qual tem a capacidade de armazenar qualquer objeto ou tipo primitivo.

Ao executar o envio de uma mensagem o método `Send()` testa se o comunicador é um comunicador de *threads*, caso seja, então é chamado o método responsável por realizar o envio de mensagens em memória compartilhada: o método `SendThread()`. Esse método cria um objeto da classe `Message` contendo as informações da mensagem. Ele verifica se quem está enviando a mensagem é o pai ou algum filho. Caso seja o pai, a mensagem passa a ser armazenada na lista de mensagens dos filhos. Caso seja algum filho, a mensagem é armazenada na lista de mensagens do pai. Essas listas são armazenadas em memória compartilhada pelas *threads* e pelo processo pai.

Ao receber uma mensagem utilizando o método `Receive()`, é realizado o teste se o comunicador é de *threads*, caso seja, o método `ReceiveThread()` é invocado. Esse

método primeiramente identifica se é o processo pai que está recebendo a mensagem. Caso seja, varre a lista de mensagens do pai até encontrar uma mensagem equivalente a que espera receber. Ao encontrar a mensagem, ela é removida da fila e retornada para o pai. O mesmo vale para os filhos, porém, a lista dos filhos é utilizada.

Embora as comunicações tenham sido limitadas a comunicações entre pai e filhos no protótipo implementado, ainda assim são suficientes para permitir a paralelização de diversos algoritmos. Qualquer algoritmo que possa ser implementado utilizando Divisão & Conquista, por exemplo, pode ser implementado. Esse modelo é uma técnica na qual divide-se um problema grande em subpartes menores as quais possam ser resolvidas separadamente. Essa divisão é aplicada de forma recursiva para cada subparte até que o subproblema possa ser resolvido de forma simples e imediata. As soluções parciais obtidas são combinadas até obter-se a solução do problema inicial. Um exemplo de ordenação utilizando Divisão & Conquista pode ser visto na Figura 5.3.

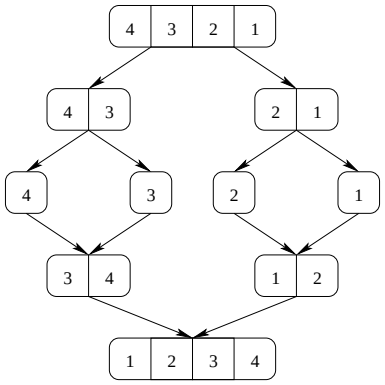

Figura 5.3: Ordenação utilizando Divisão & Conquista

Essa técnica provê algoritmos eficientes na programação sequencial (CORMEN et al., 2001) podendo ser estendida para a programação paralela afim de obter algoritmos paralelos eficientes. Alguns exemplos de aplicações que utilizam Divisão & Conquista podem ser vistos a seguir:

- **Ordenação**: Algoritmos como *mergesort* e *quicksort* dividem um vetor inicial criando novas tarefas para cada divisão do vetor até atingir a condição de parada. Após os resultados são combinados gerando o vetor ordenado;

- **Ray tracing**: cada linha de uma imagem pode ser gerada em paralelo, logo cada tarefa pode gerar parte da imagem. Ao final, combinam-se as partes gerando a imagem desejada;

- **Problema do Caixeiro Viajante (TSP)**: Uma árvore com todos os caminhos iniciais possíveis é gerada. Em seguida, a cada bifurcação, novas tarefas são geradas para explorar os caminhos que surgem. Ao completar um caminho compara-se o valor obtido nele com o valor do menor caminho conhecido. Caso seja menor, avisa as demais tarefas que foi encontrado um caminho com menor custo;

- **Busca de elementos em árvore**: A cada bifurcação da árvore novas tarefas são lançadas para explorar cada uma das sub-árvores encontradas. Ao encontrar o elemento procurado, todas as demais tarefas são sinalizadas.

Além dos problemas descritos podemos citar outros como construção de modelos 3D por octree, operações em sequências de números, teste de primalidade de Eratóstenes, *etc*. Isso demonstra a importância desse modelo de programação, o qual se adapta muito bem à programação paralela pois as tarefas podem ser mapeadas diretamente para processos.

5.3 Escalonamento e Resultados de Desempenho

O escalonador utilizado para alocar os recursos é o **smpd** da distribuição MPICH-2. Esse escalonador utiliza um algoritmo de *Round Robin* global para distribuir os processos entre os processadores. Ele possui uma lista com os nós disponíveis no ambiente de execução, onde, ao invocar um programa MPI, ele começará a alocar os processos a partir do primeiro nó da lista, e conforme são executadas chamadas ao método `Spawn()`, os outros nós são alocados. Um exemplo de uma possível distribuição de processos pode ser visto na Figura 5.4, a qual representa um ambiente computacional com 8 nós, onde cada processo realiza uma chamada ao método `Spawn()` para lançar dois novos processos. Dessa forma, o ambiente de execução irá ter ao menos um processo por nó em três chamadas recursivas ao método `Spawn()`, possuindo um balanceamento de carga com desnível máximo de 1 processo (não levando em consideração que processos mais antigos terminem antes dos mais novos).

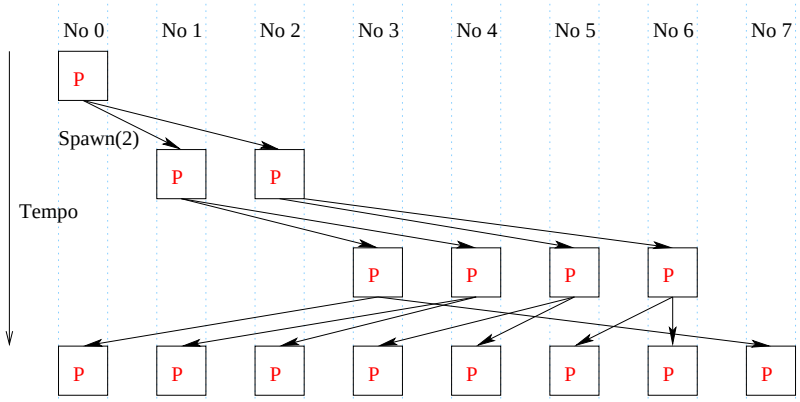

Figura 5.4: Utilização do escalonador smpd em um ambiente computacional com 8 nós, onde quadrados representam processos e setas representam a criação dos mesmos.

Uma vez que o algoritmo proposto cria *threads* intercaladas com processos, ao redesenharmos a Figura 5.4 para a utilização com o escalonador proposto, obteremos um ambiente computacional onde (considerando que processos são criados em rodadas síncronas) nas mesmas três rodadas teremos 5 nós alocados, conforme visto na Figura 5.5.

Essa distribuição de *threads*/processos embora tenha alocado menos recursos, irá executar mais rápido em cenários atuais onde os nós de um cluster possuem processadores *multi-core*. Além de ter menor custo para criar as *threads*, as comunicações serão mais rápidas por serem executadas em memória compartilhada e, dependendo da arquitetura de execução e do sistema operacional, cada *thread* irá executar em um *core* distinto.

Considerando um modelo de aplicação que parte de uma tarefa inicial γ a qual cria δ tarefas ao chamar o método `Spawn()` e cada sub-tarefa cria δ tarefas ao chamar o método `Spawn()`, de forma homogênea, até que determinada condição seja satisfeita, formando uma árvore T de altura h, é possível calcular o número de *threads*/processos existentes na

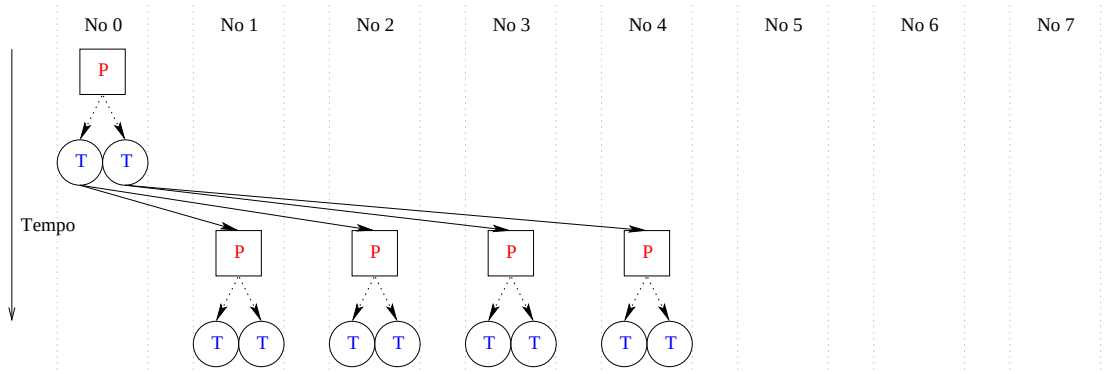

Figura 5.5: Utilização do escalonador smpd em um ambiente computacional com 8 nós, onde quadrados representam processos, círculos representam *threads*, setas pontilhadas representam a criação de *threads* e setas contínuas representam a criação de processos.

árvore através das seguintes fórmulas:

- $\sum_{i=0}^{(h-2)/2} \delta^{(2i+1)}$ para calcular o número total de *threads*;
- $\sum_{i=0}^{(h-1)/2} \delta^{(2i)}$ para calcular o número total de processos;
- $\sum_{i=0}^{(h-1)} \delta^{i}$ para calcular o número total de tarefas.

O número total de tarefas folha pode ser calculado utilizando-se a fórmula $\delta^{(h-1)}$, sendo que o desnível máximo na quantia de tarefas folha entre dois nós é o desnível máximo d garantido pelo escalonador da distribuição MPI acrescido de $\delta * d$, uma vez que *threads* são criadas sempre localmente e processos são gerenciados pelo escalonador, garantindo que será mantida a qualidade do escalonamento da distribuição MPI utilizada.

Ao redesenharmos o gráfico demonstrado na Figura 4.1, com os resultados obtidos na criação de *threads*, é possível ver claramente o ganho de desempenho obtido em relação a criação de processos, conforme demonstrado na Figura 5.6. O tempo de criar um processo C# utilizando o `Spawn()` é de 0.113s contra 0.008s para criar uma *thread* e 0.0126s para criar um processo C++. É possível ver claramente nesses números a vantagem da utilização de *threads* para reduzir o tempo de criação das tarefas.

5.4 Conclusões sobre o Capítulo

Uma vez que o desempenho obtido pela implementação do mecanismo de criação dinâmica de tarefas na biblioteca MPI.NET foi insatisfatório, foram exploradas alternativas para melhorar seu desempenho. Dentre as alternativas a que se mostrou mais interessante foi mesclar *threads* com processos, uma vez que o tempo de criação de *threads* é significativamente menor do que o tempo de criação de processos, além de permitir o uso de memória compartilhada otimizando as comunicações. Nos novos testes de desempenho

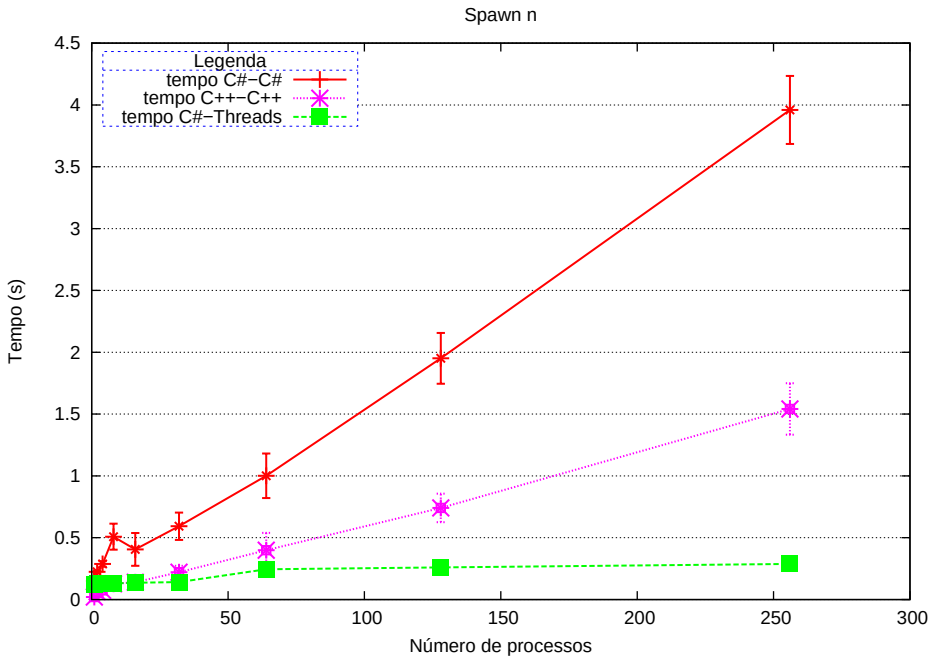

Figura 5.6: Execução do programa Spawn-n com utilização de *threads*.

foi possível perceber um ganho significativo no tempo de criar *threads* em relação ao tempo de criar processos.

Uma vez visto que compensa criar *threads* ao invés de processos, o próximo capítulo avalia se o algoritmo proposto, o qual cria *threads* e processos de forma intercalada, provê ganhos reais de desempenho em benchmarks sintéticos.

6 AVALIAÇÃO EXPERIMENTAL

Este capítulo trata da avaliação de desempenho realizada sobre a implementação da criação dinâmica de processos na biblioteca MPI.NET.

6.1 Benchmarks

Para medir o desempenho da biblioteca proposta foram utilizados *benchmarks* sintéticos. Essa seção disserta sobre os *benchmarks* utilizados para testar o desempenho da biblioteca proposta. Todas as aplicações foram desenvolvidas em versões semelhantes nas linguagens de programação C++ e C#. Não foram utilizadas características específicas de uma linguagem exceto as incluídas pela biblioteca MPI.NET nas comunicações e a gerencia automática de memória do C# a qual teve de ser feita manualmente quando necessária em C++. Dessa forma, os algoritmos realizam trabalho similar nas duas linguagens, permitindo comparar de forma precisa o desempenho das duas bibliotecas.

6.1.1 N-ésimo Número da Sequência de Fibonacci

O cálculo do n-ésimo número da sequência de Fibonacci trata de uma sequência numérica onde o primeiro número é 0, o segundo é 1 e cada subsequente é igual a soma dos 2 anteriores. Sua implementação no presente trabalho, seguiu a implementação proposta pelo ambiente de programação paralela Cilk (BLUMOFE et al., 1995), porém, adaptada para a biblioteca MPI, conforme o Programa 20. O algoritmo testa se a entrada é menor que 2. Caso seja retorna a entrada, caso contrário, são criadas duas novas tarefas, uma responsável por calcular F_{n-1} e outra responsável por calcular F_{n-2}. A tarefa que as criou aguarda seus resultados para combiná-los. Isso ocorre de forma recursiva até que seja atingida a condição de parada.

O programa apresentado, é uma simplificação da implementação do *benchmark* sintético, no qual foram inseridas duas versões do programa, uma sequencial e outra paralela. Por questões de desempenho, o programa é executado em paralelo até que seja atingido um caso trivial. Uma vez atingido, executa-se sequencialmente até obter o resultado. Esse programa cria uma quantidade moderada de tarefas, as quais executam uma quantidade moderada de trabalho sequencial uma vez atingido o caso trivial (ou *threshold*).

6.1.2 N-Queens

O algoritmo N-Queens realiza a colocação de n rainhas em um tabuleiro de xadrez, de tamanho $n \times n$, de forma que uma rainha não possa atacar outra em uma jogada. O algoritmo utilizado para resolver esse problema realiza a colocação ordenada e exaustiva de rainhas linha a linha até que cenários válidos sejam encontrados. Enquanto um deter-

Programa 20 Algoritmo paralelo para o cálculo do n-ésimo número da sequência de Fibonacci.

```
int main()
{
 Receive(n);
 if(n>40)
  n = fibonacciPar(n);
 else
  n = fibonacciSeq(n);
 Send(n,0);
}

int fibonacciSeq
{
 if (n<2)
  return n;
 return fibonacciSeq(n-1) + fibonacciSeq(n-2);
}

int fibonacciPar(int n)
{
  if (n < 2)
  {
    Send(n, 0);
  }
  else
  {
    Spawn("fibonacci", 2);
    Send(n-1, 0);
    Send(n-2, 1)
    return (Receive(0)+Receive(1));
  }
}
```

minado cenário contém rainhas em posições nas quais possam atacar ou ser atacadas, o algoritmo retorna ao último cenário válido e continua calculando novas posições.

Na versão paralela do algoritmo, uma tarefa inicial explora `n` cenários iniciais do tabuleiro e os envia para `n` tarefas. Essas tarefas geram outras tarefas recursivamente para resolver as possíveis posições de rainha na próxima linha. Esse procedimento é executado recursivamente até que todas as rainhas estejam posicionadas. Esse problema gera quantidades consideravelmente grandes de tarefas, sendo para `n=10` geradas 32.979 tarefas, as quais realizam poucas comunicações, e executam poucas operações, permitindo testar exaustivamente o mecanismo de criação dinâmica de tarefas proposto.

6.1.3 Mergesort

O mergesort, é um algoritmo de ordenação do tipo Divisão & Conquista. O funcionamento do algoritmo pode ser sintetizado da seguinte maneira:

1. Se o vetor possui tamanho maior que 1, é dividido em dois, repetindo essa operação recursivamente até que essa condição seja atingida;

2. Se o vetor possui tamanho menor ou igual a 1, então ele já está ordenado e é retornado;

3. Combina-se cada metade do vetor, combinando-as, até que seja obtido o vetor original ordenado.

A versão paralela implementada para executar os testes de desempenho, divide a entrada recursivamente, assim como o algoritmo sequencial, criando novas tarefas para cada

parte da entrada. Por motivos de desempenho, é estabelecido um limite de tamanho inferior para o vetor, onde vetores menores ou do tamanho do limite inferior passam a ser ordenados pela versão sequencial do algoritmo. Por fim, as tarefas combinam os subvetores, retornando-os ordenados para seus pais até formarem o vetor final ordenado. O funcionamento do programa é representado pela Figura 6.1

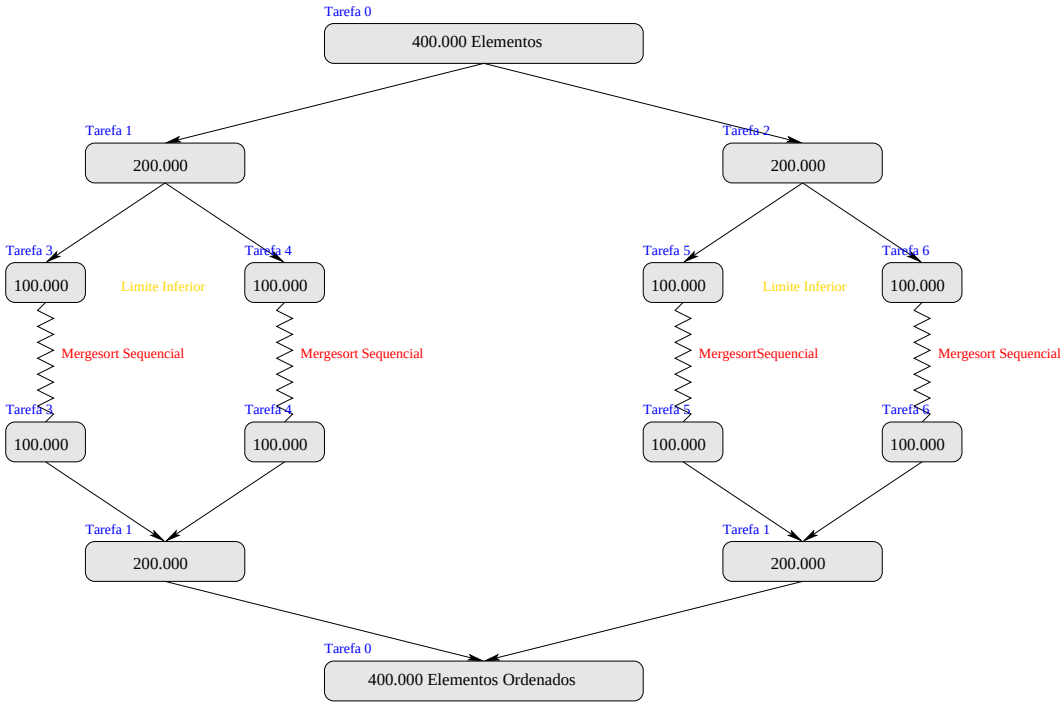

Figura 6.1: Representação do mergesort paralelo.

Para um vetor com 51.200.000 elementos são criadas 1.022 tarefas, com o grão sequencial utilizado de 100.000. Cada tarefa realiza uma quantidade pequena de trabalho e comunica muitos dados, permitindo testar exaustivamente a criação dinâmica de tarefas e a comunicação em memória compartilhada.

6.1.4 Fractal de Mandelbrot

O fractal de Mandelbrot consiste de um conjunto de pontos no plano complexo, os quais formam um fractal. Ele é calculado através de um algoritmo (iterativo) o qual realiza a iteração entre números complexos.

Embora esse não seja um problema resolvido tipicamente via Divisão & Conquista, nada impede que seja paralelizado dessa forma. O algoritmo foi programado de forma paralela da seguinte maneira, onde x representa o número de colunas na imagem:

$$Mandelbrot = \begin{cases} MandelbrotSequencial, & \text{se } x < 100; \\ Spawn(Mandelbrot, 4), & \text{se } x >= 100. \end{cases}$$

Dessa forma, o fractal é sub-dividido entre 4 tarefas recursivamente até atingir subpartes que possuam menos que 100 colunas.

6.2 Resultados da Avaliação Experimental

Essa seção discute os resultados de desempenho obtidos na execução dos *benchmarks* sintéticos. A quantidade de vezes que cada teste foi executado variou para os diferentes

testes a fim de obter-se desvio padrão pequeno o suficiente para que os resultados sejam confiáveis. A plataforma de execução para a realização dos testes de desempenho foi um cluster com a seguinte configuração:

- **Nós alocados**: 8;
- **Processadores por Nó**: 1 Intel Xeon E5310;
- **Cores por processador**: 4;
- **Threads por core** 2;
- **Memória por Nó**: 16GB;
- **Rede**: Gigabit Ethernet.
- **MPI**: MPICH2 1.2.1;
- **Máquina virtual .NET**: Mono 2.4.3.

6.2.1 N-ésimo Número da Sequência de Fibonacci

O primeiro teste executado foi o cálculo do n-ésimo número da sequência de Fibonacci. Primeiramente foram feitas execuções da versão sequencial do programa para medir o sobrecusto introduzido pelo uso do C# em relação ao C++, uma vez que a linguagem executa sobre máquina virtual. Os resultados obtidos podem ser vistos na Figura 6.2. Através da figura é possível visualizar que o tempo de execução do programa sequencial em C# é bem similar ao tempo de execução do programa em C++, não possuindo sobrecusto significativo, sendo de 5% para a execução com n=50 e 2% para n=40.

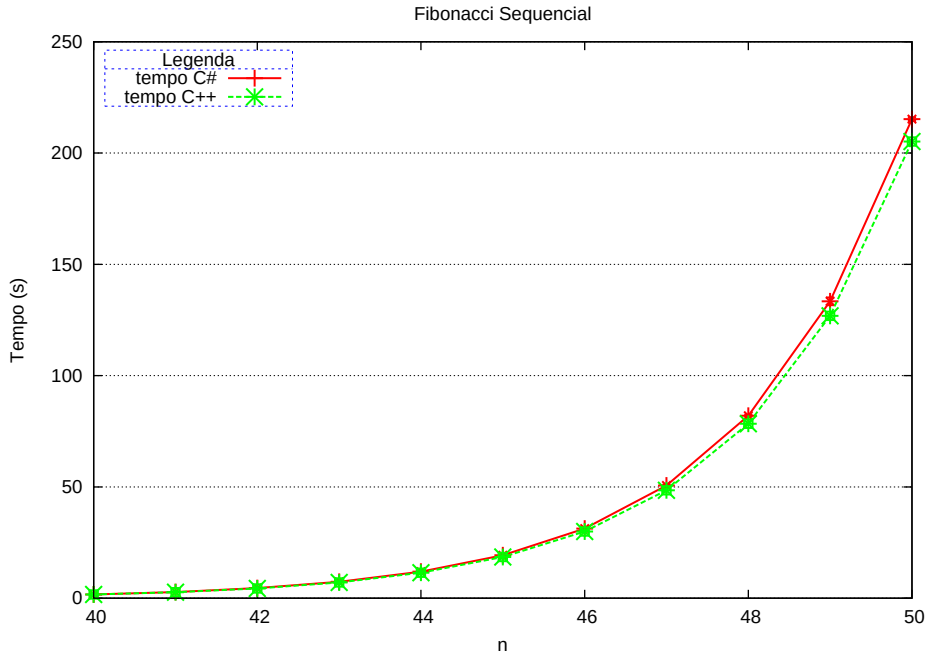

Figura 6.2: Tempo de execução do fibonacci sequencial.

Tendo a diferença de tempo de execução do Fibonacci sequencial entre o C# e o C++ foram feitas execuções paralelas do Fibonacci para verificar o comportamento da

biblioteca com a criação dinâmica de tarefas. O tamanho do grão (ponto a partir do qual se executa sequencialmente) utilizado foi 40 devido ao tempo de execução sequencial desse valor ser diversas vezes maior que o tempo de execução de um `Spawn()`.

A Figura 6.3 ilustra os resultados do Fibonacci paralelo em C# e em C++. Eles demonstram que devido ao tempo de criar um processo C# possuir um sobrecusto significativo em relação a criação de um processo C++, combinado, com a natureza dessa aplicação, a qual cria muitos processos, o desempenho obtido para a biblioteca MPI.NET utilizando somente processos foi inferior em relação ao desempenho de C++.

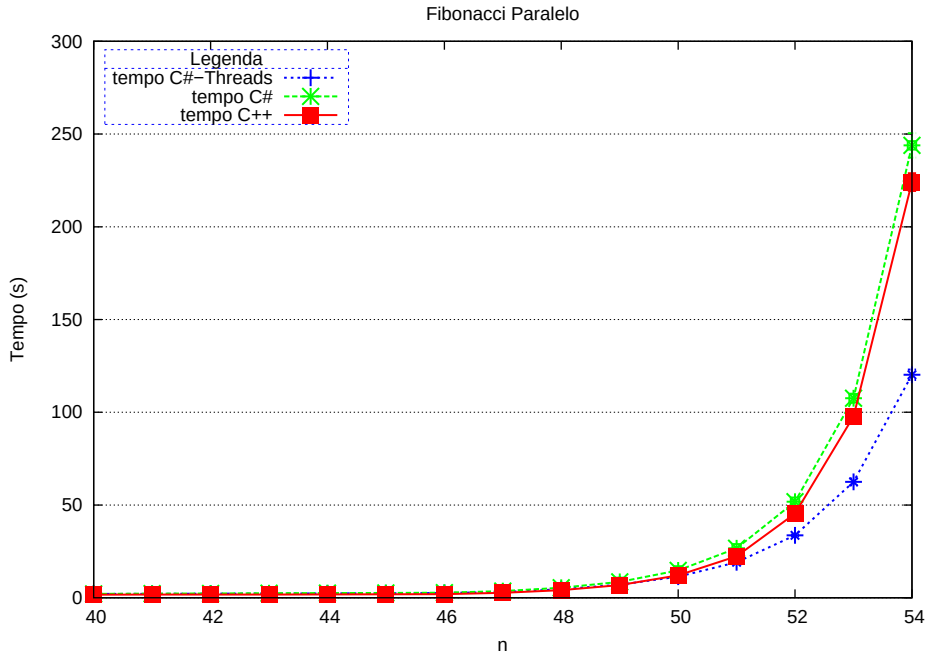

Figura 6.3: Tempo de execução do fibonacci paralelo.

A Figura 6.3 também permite perceber que conforme a entrada aumenta a versão C# com *threads* supera o desempenho da biblioteca MPI-C++ nativa. Esse aumento de desempenho pode ser notado a partir da entrada 50, a qual gera 286 tarefas. O C# com *threads* é em torno de 10% mais rápido para entrada 50, chegando a ser 84% mais rápida para entrada 54 na qual são criadas 1972 tarefas dinamicamente. Por fim, é possível verificar que a versão em C++ é 9% mais rápida que C# com somente processos para n=54.

Ao utilizar *threads* intercaladas com processos o desempenho obtido em C# superou significativamente o desempenho de C++ para as entradas maiores que 50 e superou sempre o desempenho obtido com C# utilizando somente processos. Esses dados podem ser visualizados em maiores detalhes na Figura 6.4. Também é possível verificar que até a entrada de tamanho 49, na qual são criados dinamicamente 176 tarefas, a versão em C++ foi mais rápida do que ambas versões em C#.

6.2.2 N-Queens

O segundo *benchmark* sintético a ser executado foi o programa n-queens, o qual cria muitas tarefas de grão pequeno e comunica poucos dados. O desempenho obtido nesse teste não foi satisfatório, uma vez que mesmo com a utilização de *threads* intercaladas com processos o C# perde muito desempenho em relação ao C++, conforme visto na

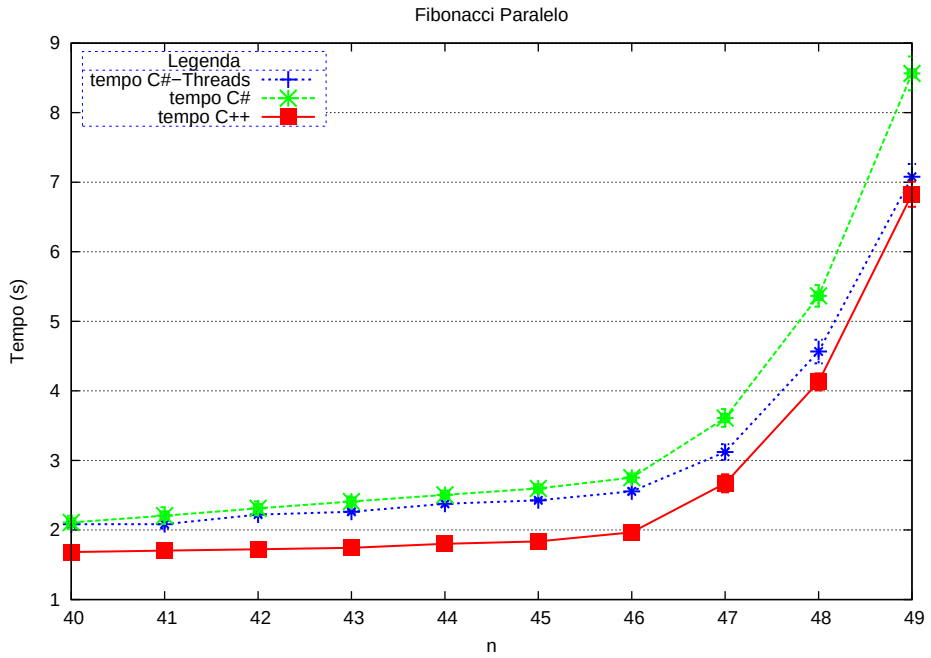

Figura 6.4: Tempo de execução do fibonacci paralelo com entradas menores.

Figura 6.5.

Ao detalharmos os resultados de desempenho obtidos pelas entradas menores, é possível observar que até `n=6` o desempenho do C# com *threads* foi superior ao C++, porém, como a quantidade de tarefas cresce exponencialmente, à partir de `n=7` o C# não consegue alcançar o desempenho de C++. Isso se dá porque cada tarefa cria somente uma sub-tarefa por vez, diminuindo significativamente o ganho de desempenho obtido, pois o maior ganho de desempenho é obtido quando são criadas diversas *threads* em uma única chamada ao método `Spawn()`, além de agrupar mais tarefas em memória compartilhada, otimizando as comunicações. Os resultados podem ser vistos claramente na Figura 6.6.

Mesmo não tendo alcançado o desempenho de C++ para `n=7`, a versão C# com *threads* superou em 128% o desempenho da versão C# que não utiliza *threads* enfatizando os ganhos de desempenho do algoritmo proposto.

6.2.3 Ordenação (mergesort)

O terceiro *benchmark* sintético executado foi o algoritmo de ordenação mergesort. Para esse teste foram utilizadas entradas variando de 100.000 até 51.200.000, dobrando o valor da entrada a cada execução. A entrada foi organizada de forma decrescente, sendo essa a forma onde o algoritmo trabalha mais para ordená-la. O grão sequencial utilizado foi 100.000 elementos no vetor. Os resultados dos testes podem ser vistos na Figura 6.7.

Nesse algoritmo, o tempo majoritário é tempo de comunicação, uma vez que vetores com tamanho de até 25.600.000 inteiros são comunicados entre as tarefas. Dessa forma é possível avaliar o mecanismo de comunicação em memória compartilhada, além de testar o desempenho da criação dinâmica de tarefas. Na Figura 6.7 é possível visualizar um ganho de 80% da versão com *threads* da biblioteca em relação à versão sem *threads* e um ganho de 100% em relação ao mesmo programa em C++ para a entrada de tamanho 51.200.000.

Na Figura 6.8 é possível perceber que a versão do programa em C++ inicia com me-

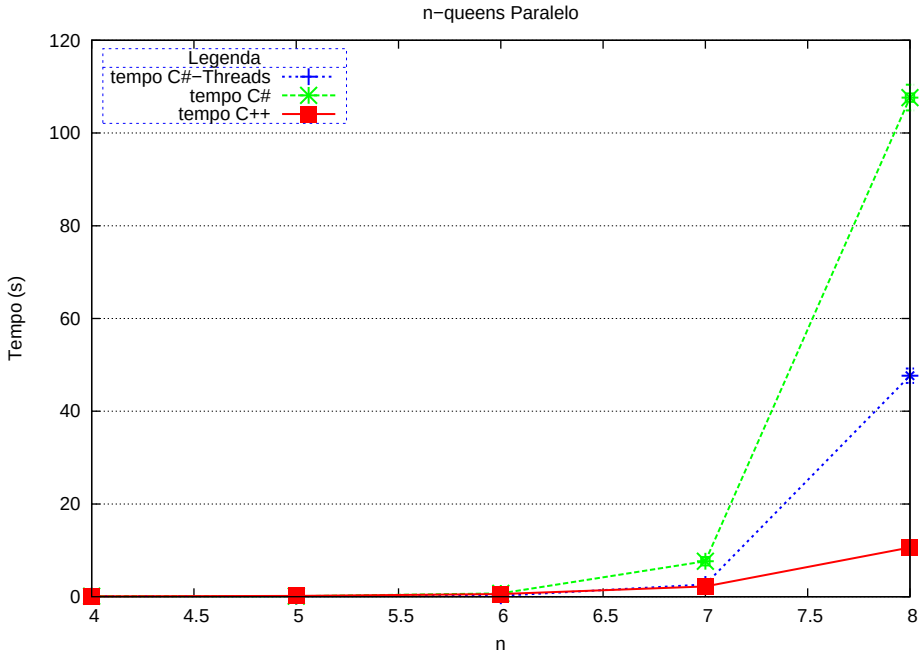

Figura 6.5: Tempo de execução do n-queens paralelo.

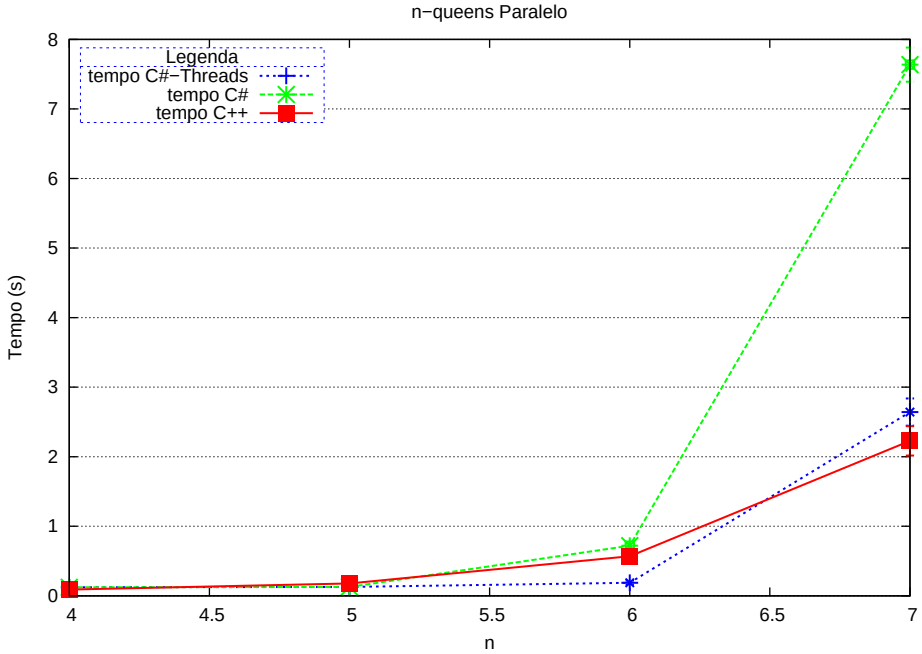

Figura 6.6: Tempo de execução do n-queens paralelo com resultados para n variando de 4 a 7 em detalhe.

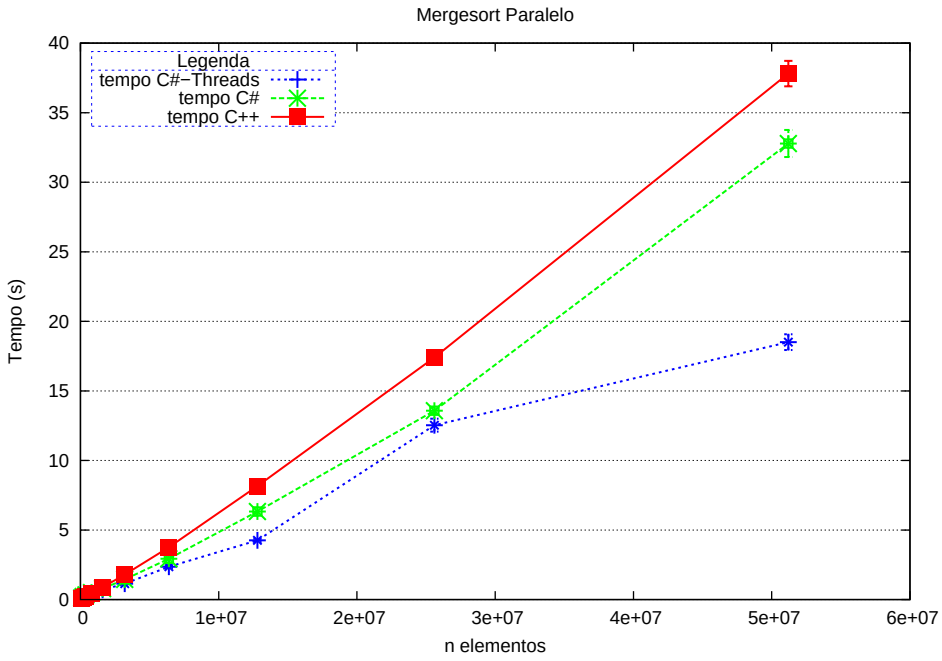

Figura 6.7: Tempo de execução do mergesort paralelo.

lhor desempenho que ambas versões em C#, porém, quando a entrada ultrapassa 800.000 elementos seu desempenho passa a ser pior que ambas versões em C#. Uma possível causa para essa vantagem de desempenho do C# em relação ao C++ é o tempo de comunicação de mensagens grandes utilizando a biblioteca MPI.NET conforme discutido no Capítulo 2. Na Figura 6.8 também percebe-se que o desempenho obtido em relação à versão C# *threads* se mantém melhor, mesmo para os casos com entradas menores onde poucas tarefas são criadas.

6.2.4 Fractal de Mandelbrot

A execução do *benchmark* sintético que gera o fractal de Mandelbrot gerou fractais com a quantidade de colunas variando de 100 até 51.200. Os resultados de desempenho obtidos nas execuções do teste podem ser vistos na Figura 6.9.

Assim como nos outros *benchmarks*, a versão da biblioteca MPI.NET que utiliza *threads* manteve desempenho sempre melhor em relação a outra versão, superando o desempenho de C++ após determinada entrada. Seu desempenho foi 35% melhor que o desempenho da versão sem *threads* da biblioteca e 32% melhor do que a versão do programa escrita em C++ para a entrada com 51.200 colunas.

6.3 Conclusões sobre o Capítulo

Nesse capítulo foram descritos os *benchmarks* sintéticos utilizados para testar o desempenho da biblioteca proposta. Foi explicado brevemente como eles foram programados e quais características que cada *benchmark* testa. Além disso foi realizada uma avaliação do desempenho das duas versões da biblioteca em comparação com MPI nativo para C++. Foram executados quatro *benchmarks* sintéticos sendo que em três deles o desempenho da biblioteca proposta superou o desempenho de C++, e, em todos testes o algoritmo de criação dinâmica de *threads* intercaladas com processos se mostrou mais

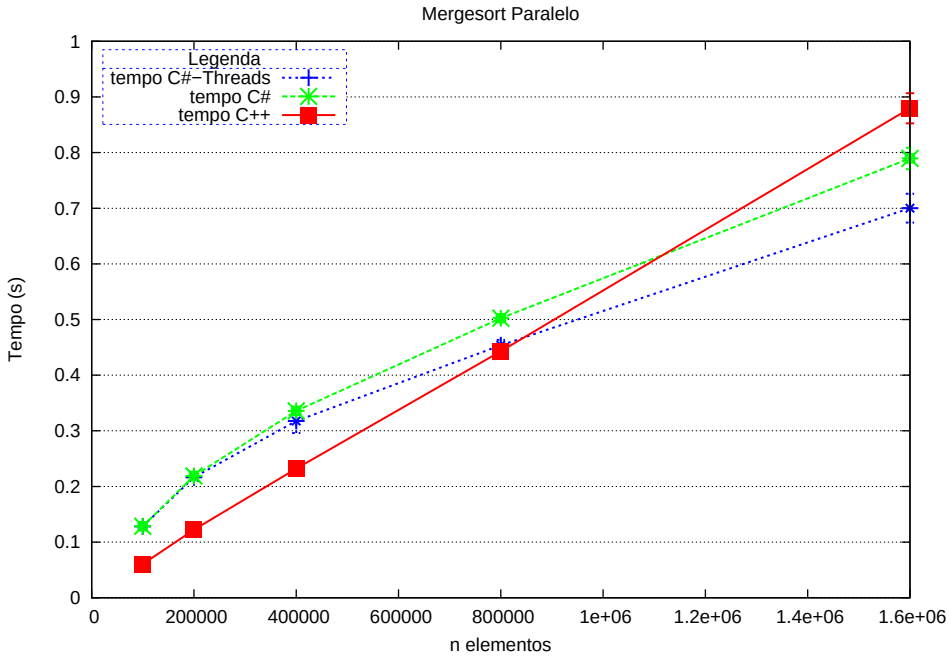

Figura 6.8: Tempo de execução do mergesort paralelo com valores variando de 100.000 até 1.600.000.

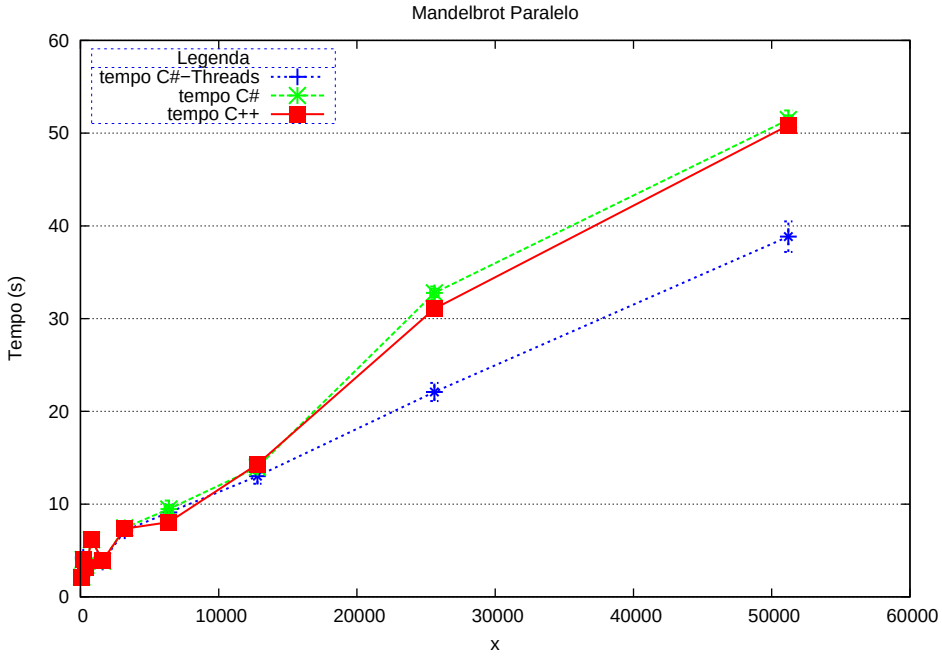

Figura 6.9: Tempo de execução do mandelbrot paralelo gerando imagens variando de 100 até 51.200 colunas

eficiente.

No cálculo do n-ésimo número da sequência de Fibonacci o ganho de desempenho com a utilização de *threads* não ocorre para as execuções com um valor pequeno em `n` devido ao programa não tirar grande proveito das comunicações em memória compartilhada, uma vez que é transmitido somente um inteiro por comunicação. Porém, conforme cresce a quantidade de tarefas criadas dinamicamente, é possível perceber um ganho expressivo de desempenho em relação a C++ devido à criação de *threads*. Por outro lado, se não forem utilizadas *threads* o desempenho do C# se mantém sempre abaixo do C++, chegando a ser 5% mais lento para `n=54`. Esse *benchmark* permitiu perceber a eficiência obtida pela criação de *threads* intercaladas com processos.

No algoritmo n-queens foi possível obter desempenho superior ao de C++ até a entrada `n=6`, porém, como a quantidade de tarefas cresce exponencialmente, as tarefas são criadas uma a uma e as comunicações são pequenas à partir de `n=7` o C# não consegue alcançar o desempenho de C++.

Na ordenação mergesort as duas versões da biblioteca MPI.NET mantiveram desempenho melhor que C++ para entradas grandes, sendo que a versão com *threads* se manteve sempre mais eficiente.

Por fim, na execução do *benchmark* sintético que gera o fractal de Mandelbrot o desempenho da biblioteca com *threads* se mostrou superior em relação a biblioteca sem *threads* sempre e a C++ para entradas grandes.

Os *benchmarks* sintéticos permitem concluir que para programas que criem diversas tarefas, com granularidade não muito pequena, e com ou sem comunicações grandes a biblioteca proposta tende a oferecer desempenho competitivo em relação a C++ em sua versão com *threads*.

7 CONCLUSÃO

Este trabalho explorou a utilização da criação dinâmica de tarefas MPI na biblioteca MPI.NET. Foi realizada uma prototipação inicial, a qual possui uma API mais simples do que a de C++. O baixo desempenho obtido inicialmente levou à proposta de um algoritmo para otimizar a criação dinâmica de processos intercalando *threads* com processos, obtendo maior desempenho. A biblioteca proposta se mostrou eficiente na maioria dos testes executados.

7.1 Contribuições

A principal contribuição desse trabalho foi mostrar que é viável, em termos de desempenho, utilizar a criação dinâmica de tarefas MPI em linguagens de programação que executem sobre máquinas virtuais. Para isso ser concretizado, foi proposto e implementado um algoritmo o qual cria *threads* e processos de maneira intercalada, permitindo obter maior desempenho no tempo de criação das tarefas bem como nas comunicações. Outras contribuições desse trabalho são:

- Mostrar como a programação orientada a objetos simplifica o desenvolvimento de aplicações paralelas;
- Discutir as vantagens de se utilizar linguagens de programação que possuam níveis maiores de abstração do que C++;
- Agrupar e discutir diversos projetos que propõem melhoras à API de programação MPI para C++ bem como APIs MPI para diversas linguagens de programação orientadas a objetos;
- Avaliar a biblioteca implementada com *benchmarks* sintéticos frequentemente utilizados para avaliar ambientes de programação com dinamismo;
- Propor um algoritmo alternativo para criação de tarefas MPI intercaladas com *threads*, o qual permite obter maior desempenho não somente em linguagens que executem sobre máquinas virtuais, mas também sobre outras linguagens que criem tarefas dinamicamente (como C, C++);
- Implementar comunicações MPI em memória compartilhada as quais funcionam de forma transparente para o programador.

7.2 Publicações

Um artigo e dois resumos foram publicados durante esse trabalho, sendo o artigo publicado na revista Scientia em 2009 (AFONSO; MAILLARD, 2009a). Os resumos foram publicados em eventos regionais sendo eles: ERAD 2009 (AFONSO; MAILLARD, 2009b) ERAD 2010 (AFONSO; MAILLARD, 2010).

Estão em vista duas submissões de artigos: uma sobre a primeira parte do presente trabalho a ser submetida para a XXXVI Conferência Latino-americana de Informática (CLEI). Uma sobre o trabalho completo para o *22nd International Symposium on Computer Architecture and High Performance Computing* (SBAC-PAD).

7.3 Trabalhos Futuros

Os trabalhos futuros prevêem a exploração de novos algoritmos para criar tarefas dinamicamente de forma mais eficiente, explorando de maneira mais abrangente a criação de *threads* em sistemas *multi-core*. Teriam de ser exploradas novas proporções na relação *threads*/processos, levando em consideração a quantidade de *cores* das máquinas.

Está previsto também o estudo da utilização da biblioteca MPI.NET em plataformas com hardware e sistema operacional heretogêneos. Isso pode ser feito através da utilização da biblioteca MPICH2 em conjunto com a utilização de tecnologias como *Network Information Service* (NIS), *Windows Services for Linux* e uma rede com sistema de arquivos *Network File System* (NFS) entre as máquinas heterogêneas. Uma outra alternativa mais sofisticada porém auto-contida seria gerenciar as comunicações entre diferentes sistemas operacionais a nível do .NET e as comunicações entre máquinas que executem o mesmo sistema operacional através de MPI-C.

Outra ideia seria a criação de uma funcionalidade que permita realizar o *spawn* de métodos, ocultando inclusive as comunicações. Isso pode ser feito a nível de biblioteca. Ao realizar a chamada ao método *spawn* passando um método como parâmetro (e seus parâmetros), a biblioteca pode capturar utilizando reflection esses parâmetros, criar um novo processo MPI remoto, e repassar para ele o nome do método a ser executado bem como comunicar seus parâmetros. Por sua vez o processo remoto executaria o método, devolvendo seu *return* via mensagem para o processo que o criou.

Além disso, novas avaliações de desempenho devem ser realizadas, abrangendo outras categorias de aplicações e plataformas de execução, bem como realizar a execução de aplicações reais, permitindo a avaliação do comportamento da biblioteca em cenários de produção.

Por fim, implementar o algoritmo de *threads*/processos em MPI-C++ para avaliar o ganho de desempenho do algoritmo nessa linguagem de programação.

REFERÊNCIAS

AFONSO, F.; MAILLARD, N. Implementação da criação dinâmica de tarefas na biblioteca MPI.NET. **Scientia (Unisinos)**, [S.l.], 2009.

AFONSO, F.; MAILLARD, N. Criação Dinâmica de Tarefas MPI.NET em Ambiente Heterogêneo. In: IX ESCOLA REGIONAL DE ALTO DESEMPENHO, ERAD 2009, 2009, Caxias do Sul, BRA. **Anais...** SBC, 2009. p.111–112.

AFONSO, F.; MAILLARD, N. Avaliação de Desempenho da Criação Dinâmica de Processos MPI.NET. In: X ESCOLA REGIONAL DE ALTO DESEMPENHO, ERAD 2010, 2010, Passo Fundo, BRA. **Anais...** SBC, 2010.

BAKER, M.; CARPENTER, B.; FOX, G.; KO, S. H.; LIM, S. MPIJAVA: an object-oriented java interface to mpi. In: IPPS/SPDP'99 WORKSHOPS HELD IN CONJUNCTION WITH THE 13TH INTERNATIONAL PARALLEL PROCESSING SYMPOSIUM AND 10TH SYMPOSIUM ON PARALLEL AND DISTRIBUTED PROCESSING, 11., 1999, London, UK. **Proceedings...** Springer-Verlag, 1999. p.748–762.

BLUMOFE, R. D.; JOERG, C. F.; KUSZMAUL, B. C.; LEISERSON, C. E.; RANDALL, K. H.; ZHOU, Y. Cilk: An Efficient Multithreaded Runtime System. In: THE FIFTH ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, PPOPP'95, 1995, Santa Barbara, USA. **Proceedings...** ACM Press, 1995. p.207–216.

BUDD, T. A. **An Introduction to Object-Oriented Programming**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. **Introduction to Algorithms, Second Edition**. [S.l.]: The MIT Press and McGraw-Hill Book Company, 2001.

COULAUD, O.; DILLON, E. Para++: a high level c++ interface for message passing. **J. Parallel Distrib. Comput.**, [S.l.], v.51, n.1, p.46–62, 1998.

DALCIN, L.; PAZ, R.; STORTI, M. MPI for Python. **Journal of Parallel and Distributed Computing**, [S.l.], v.65, n.9, p.1108–1115, September 2005.

DAREMA, F.; GEORGE, D. A.; NORTON, V. A.; PFISTER, G. F. A single-program-multiple-data computational model for EPEX/FORTRAN. **Parallel Computing**, [S.l.], v.7, n.1, p.11–24, 1988.

DRUMMOND, L. A.; GALIANO, V.; MIGALLóN, V.; PENADéS, J. PyACTS: a python based interface to acts tools and parallel scientific applications. **Int. J. Parallel Program.**, Norwell, MA, USA, v.37, n.1, p.58–77, 2009.

ECKEL, B. **Thinking in Java**. [S.l.]: Prentice Hall PTR, 2006. 1150p.

ASSOCIATES, O. . (Ed.). **Java in a Nutshell**. [S.l.]: Cambridge, 1998.

FLYNN, M. Some Computer Organizations and Their Effectiveness. **IEEE Trans. Comput.**, [S.l.], v.C-21, p.948+, 1972.

FORUM, M. P. I. **Message Passing Interface (MPI) Forum**. 1994.

FORUM, M. P. I. **MPI-2**: Extensions to the Message-Passing Interface. Knoxville, USA: University of Tennessee, 1997. (CDA-9115428).

FORUM, M. P. I. **Message Passing Interface (MPI) Forum**. 2009.

GREGOR, D.; LUMSDAINE, A. 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, 2008, New York, NY, USA. **Anais...** ACM, 2008. p.133–142.

GREGOR, D.; TROYER, M. **BOOST.MPI**. Disponível em http://www.boost.org/doc/libs/1_37_0/doc/html/mpi.html#mpi.intro. Acesso em 10 de nov. de 2008.

GROPP, W.; LUSK, E.; DOSS, N.; SKJELLUM, A. A high-performance, portable implementation of the MPI message passing interface standard. **Parallel Comput.**, Amsterdam, The Netherlands, The Netherlands, v.22, n.6, p.789–828, 1996.

GROPP, W.; LUSK, E.; THAKUR, R. **Using MPI-2**: advanced features of the message-passing interface. Cambridge, MA, USA: MIT Press, 1999.

GRUNDMANN, T.; RITT, M.; ROSENSTIEL, W. TPO++: an object-oriented message-passing library in c++. In: ICPP '00: PROCEEDINGS OF THE PROCEEDINGS OF THE 2000 INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, 2000, Washington, DC, USA. **Anais...** IEEE Computer Society, 2000. p.43.

GUPTA, S. **A Performance Comparison of Windows Communication Foundation (WCF) with Existing Distributed Communication Technologies**. Disponível em http://msdn.microsoft.com/en-us/library/bb310550.aspx. Acesso em 30 de nov. de 2008.

KALE, L. V.; KRISHNAN, S. CHARM++: a portable concurrent object oriented system based on c++. **SIGPLAN Not.**, New York, NY, USA, v.28, n.10, p.91–108, 1993.

KAMBADUR, P.; GREGOR, D.; LUMSDAINE, A.; DHARURKAR, A. Modernizing the C++ Interface to MPI. In: EUROPEAN PVM/MPI USERS' GROUP MEETING, 13., 2006, Bonn, Germany. **Proceedings...** Springer, 2006. p.266–274. (LNCS).

LIANG, S. **Java Native Interface**: programmer's guide and reference. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

LIMA, J. V. F.; MAILLARD, N. Controle de Granularidade com threads em Programas MPI Dinâmicos. In: IX SIMPóSIO EM SISTEMAS COMPUTACIONAIS - WSCAD-SSC 2008, 2008, Campo Grande, Brasil. **Anais...** [S.l.: s.n.], 2008.

MARSHALL, D. **Programming Microsoft Visual C# 2008**: the language. [S.l.]: Microsoft Press, 2008.

MCCANDLESS, B. C.; SQUYRES, J. M.; LUMSDAINE, A. Object-Oriented MPI (OOMPI): a class library for the message passing interface. In: MPIDC '96: PROCEEDINGS OF THE SECOND MPI DEVELOPERS CONFERENCE, 1996, Washington, DC, USA. **Anais...** IEEE Computer Society, 1996. p.87.

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

MORIN, S.; KOREN, I.; KRISHNA, C. M. JMPI: implementing the message passing standard in java. In: IPDPS '02: PROCEEDINGS OF THE 16TH INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM, 2002, Washington, DC, USA. **Anais...** IEEE Computer Society, 2002. p.191.

NAVAUX, P. O. A.; ROSE, C. A. F. D. **Arquiteturas Paralelas**. 1.ed. [S.l.]: Editora Sagra Luzzato, 2003. n.15. (Série Livros Didáticos).

NESTER, C.; PHILIPPSEN, M.; HAUMACHER, B. A more efficient RMI for Java. In: JAVA '99: PROCEEDINGS OF THE ACM 1999 CONFERENCE ON JAVA GRANDE, 1999, New York, NY, USA. **Anais...** ACM, 1999. p.152–159.

PURE Mpi.NET. Disponível em http://www.purempi.net/. Acesso em 14 de nov. de 2008.

RAMEY, R. **Boost Serialization Library**. Disponível em <http://www.boost.org/doc/libs/1_37_0/libs/serialization/doc/index.html>. Acesso em 10 de nov. de 2008.

SCHWARZKOPF, R.; MATHES, M.; HEINZL, S.; FREISLEBEN, B.; DOHMANN, H. Java RMI versus .NET Remoting Architectural Comparison and Performance Evaluation. In: ICN '08: PROCEEDINGS OF THE SEVENTH INTERNATIONAL CONFERENCE ON NETWORKING (ICN 2008), 2008, Washington, DC, USA. **Anais...** IEEE Computer Society, 2008. p.398–407.

SEBESTA, R. W. **Concepts of Programming Languages**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1998.

SNELL, Q. O.; MIKLER, A. R.; GUSTAFSON, J. L. Netpipe: a network protocol independent performace evaluator. In: IN PROCEEDINGS OF THE IASTED INTERNATIONAL CONFERENCE ON INTELLIGENT INFORMATION MANAGEMENT AND SYSTEMS, 1996. **Anais...** [S.l.: s.n.], 1996.

WENSHENG, T. W. Y. H. Y. PJMPI: pure java implementation of mpi. In: HIGH PERFORMANCE COMPUTING IN THE ASIA-PACIFIC REGION, 2000. **Anais...** [S.l.: s.n.], 2000. v.1, p.14–17.