UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

DIEGO FRANCISCO DE GASTAL MORALES

Compilação de Código C/MPI para C/Pthreads

Trabalho de Conclusão de Curso

Prof. Dr. Nicolas Maillard
Orientador

Porto Alegre, julho de 2009

"As the complexity of a system increases, human ability to make precise and relevant (meaningful) statements about its behaviour diminishes until a threshold is reached beyond which the precision and the relevance become mutually exclusive characteristics." — Princípio da Incompatibilidade, Zadeh (1973)

AGRADECIMENTOS

Ao professor Nicolas Maillard, pela orientação e pelo apoio.

Ao meu colega e amigo Luis Fernando Heckler, pelas conversas e pelo incentivo.

E ao meu amor, Isabel Gemelli, pela paciência, pelo carinho, e pelos chocolates e coca-colas para me animar.

SUMÁRIO

LISTA DE FIGURAS

LISTAGENS

RESUMO

A tecnologia de processadores *multicore* está trazendo novas demandas para as áreas de processamento paralelo e de alto desempenho. Bastante já foi pesquisado sobre formas de aumentar o desempenho de implementações da norma MPI – umas das principais ferramentas destas áreas – nesse tipo de hardware, mas este trabalho de conclusão de curso (TCC) inicia a exploração de uma alternativa pouco investigada de compilação de código MPI para código que use *threads*. Esta alternativa pode levar a otimizações impossíveis de serem alcançadas por bibliotecas de tempo de execução. É apresentada aqui a implementação de um protótipo de compilador C/MPI para C/Pthreads, e uma avaliação de seu desempenho, que se mostrou competitivo frente a outras distribuições MPI baseadas em memória compartilhada. Ao final são avaliadas as vantagens do uso da compilação, e são discutidas as possibilidades e dificuldades do seu uso para alcançar otimizações maiores, que podem ser pesquisadas em trabalhos futuros.

Palavras-chave: Programação Paralela, MPI, Threads POSIX, Compiladores, Otimização.

Compiling C/MPI to C/Pthreads Code

ABSTRACT

The multicore processor technology is casting new demands upon the parallel processing and high performance computing fields. Much research has been done on improving the performance of implementations of the MPI standard – one of the most important tools of those areas – on parallel hardware, but this graduate conclusion work starts exploring an under-investigated alternative of compiling MPI code to multi-threaded code. Such alternative may lead to optimizations impossible to achieve by runtime libraries. A prototype of C/MPI to C/Pthreads compiler is presented here, along with some performance evaluation which shows competitive results against other shared-memory based MPI distributions. In the final remarks, the advantages of the compiling approach are discussed, together with the possibilities and difficulties of reaching higher optimizations, which may be the subject of further research.

Keywords: Parallel Programming, MPI, POSIX Threads, Compilers, Optimization.

1 INTRODUÇÃO

Este Trabalho de Conclusão de Curso (TCC) estuda uma alternativa para a otimização de desempenho de programas escritos usando a norma MPI (*Message Passing Interface*) em computadores multiprocessados.

Com o surgimento de processadores *multicore*, arquiteturas paralelas de hardware estão se popularizando e alcançando todos computadores, mesmo os pessoais e os portáteis. Essa nova tecnologia aumenta a demanda de programas paralelos, que possam aproveitar toda sua capacidade, e conseqüentemente aumenta também a demanda por novas ferramentas e técnicas que auxiliem no desenvolvimento destes programas.

Existem várias alternativas para ganhar desempenho com a adição de paralelismo às aplicações. Além do uso direto de *Threads* ou do MPI, existem soluções como o OpenMP[1] e o Cilk++[2], que buscam paralelizar programas escritos sequencialmente através de pequenas indicações fornecidas pelo desenvolvedor. Ou ainda outras abordagens como a do UPC[3], que busca abstrair do programador a execução distribuída de um programa, fornecendo uma ilusão de memória compartilhada entre todo o sistema.

Porém essas alternativas em geral falham em prover o desempenho esperado por trás das abstrações, e tentativas de paralelização automática não conseguem muito mais do que paralelizar laços. Não existe solução universal para a obtenção de desempenho em programas paralelos. Mas existem muito programas MPI escritos para ambientes distribuídos que poderiam aproveitar melhor as arquiteturas de hardware paralelas. Versões de bibliotecas MPI que aproveitam a memória compartilhada já existem há bastante tempo, mas pouco foi explorado a respeito de uma outra possibilidade interessante: a transformação estática do código MPI em uma alternativa de maior desempenho nessas arquiteturas, através de um compilador.

A alternativa estudada neste TCC consiste na utilização de técnicas e ferramentas de compiladores para realizar a transformação de código C usando MPI para um código C que utiliza *threads* ao invés de processos, e que realiza toda comunicação entre as *threads* pelo acesso compartilhado à memória. Com estas técnicas, supõe-se que poderosas alterações possam ser realizadas no código destes programas, e novas oportunidades de otimização possam ser exploradas.

Para testar essa hípotese, um compilador simples para esta tarefa é construído, batizado de **M2PI** (*MPI to Pthreads Interface*). Seu desempenho é testado frente a outras implementações da norma MPI, através de uma série de programas de testes.

[1] http://openmp.org/wp/
[2] http://www.cilk.com
[3] http://upc.lbl.gov/

O restante desta monografia está organizada da seguinte forma: o contexto científico do trabalho é discutido no capítulo 2, expondo os fatores que o motivam, o objetivo traçado, e a revisão bibliográfica e dos trabalhos relacionados. O capítulo 3 apresenta uma discussão sobre a ferramenta utilizada para implementar o compilador, as premissas e simplificações adotadas, e a arquitetura geral do M2PI.

Os capítulos 4 e 5 introduzem conceitos e ferramentas necessários para a compreensão da implementação do trabalho, descrita detalhadamente no capítulo 6. Os resultados obtidos são discutidos no capítulo 7, exibindo gráficos de desempenho, e as descrições da metodologia utilizada e dos programas utilizados para teste.

O relatório é completado pelas considerações finais do capítulo 8. Os apêndices A e B listam o código fonte transformado de dois programas simples, para que o leitor possa visualizar como é alterado um programa processado pelo M2PI.

2 CONTEXTO CIENTÍFICO

2.1 Motivação

Com a popularização de arquiteturas de hardware paralelas, capitaneada pelo surgimento dos processadores *multicore*, cresce cada vez mais a importância da programação concorrente, necessária para o completo aproveitamento dos múltiplos processadores que agora são encontrados até em computadores pessoais e portáteis. Os métodos de realizar esse tipo de programação, as ferramentas atualmente utilizadas, e até mesmo as formalizações teóricas nessa área estão ainda longe de ser consenso, seja na indústria, seja na academia.

Nas arquiteturas paralelas de hardware mais comuns atualmente, que seguem um modelo de memória compartilhada uniformemente acessível, a abordagem predominante de programação paralela é o uso de *threads*. Essa abordagem é geralmente aceita como sendo a de melhor desempenho, muito em razão de que programas usando *threads* podem aproveitar completamente o modelo compartilhado de acesso a memória usado pelo hardware. No entanto, justamente por esse modelo – que traz muito indeterminismo à programação – alguns pesquisadores da área argumentam que a programação com *threads* é demasiadamente difícil, e conseqüentemente muito propensa a erros, mesmo para profissionais altamente capacitados e experientes (ver por exemplo o artigo *The Problem With Threads*, de LEE (2006)).

O modelo de troca de mensagens, em que se baseiam bibliotecas como a MPI, é uma opção mais simples e confiável, mas normalmente tem desempenho inferior ao das soluções que usam *threads* diretamente.

Uma alternativa interessante, que busca o melhor dos dois mundos, é manter o modelo de troca de mensagens para o programador, através da interface de programação, mas automaticamente gerar o código executável de uma forma mais adequada à arquitetura do hardware alvo. Algumas implementações de MPI já buscam algo nesse sentido, fazendo com que o seu ambiente de execução (*runtime*) realize a comunicação entre os processos MPI via memória compartilhada (ao invés de sockets TCP) quando estes processos estão na mesma máquina.

É possível implementar a alternativa acima transformando o código gerado pelo programador, em um processo de compilação. Tal compilação poderia buscar uma série de otimizações que o ambiente de execução do MPI não poderia realizar. Essa possibilidade, e o desafio técnico de implementar tal compilador, são as motivações deste trabalho.

2.2 Objetivo

O objetivo deste trabalho de graduação é ser um estudo preliminar sobre a viabilidade e utilidade da implementação de um compilador que transforme código fonte C/MPI em código C/pthreads (POSIX threads), onde cada processo criado pelo MPI seja uma *thread*, e toda comunicação entre os processos/*threads* rodando na mesma máquina seja feita através da memória compartilhada.

Para isso, será feita a implementação de um protótipo simplificado de tal compilador, restrito a um subconjunto de instruções do padrão MPI, e então será avaliada sua correção e seu desempenho. O escopo também se limita à execução não distribuída de programas MPI (caso contrário seria necessário se mesclar comunicação de rede com a feita através da memória, o que foge do foco deste trabalho).

Como decorrência desse esforço de planejamento e implementação, é esperado para o aluno um grande aprofundamento do conhecimento e da experiência em programação concorrente – em particular no uso de pthreads e MPI – e na construção de compiladores. Além disso, a participação em um projeto de desenvolvimento dessa complexidade será uma experiência extremamente importante.

2.3 Revisão bibliográfica e trabalhos relacionados

Como foi dito na seção 2.1, o projeto e a programação de sistemas paralelos e distribuídos são problemas em aberto para a ciência da computação. Esta situação, somada às muitas facetas e aplicações de técnicas de programação concorrente (desempenho, distribuição, tolerância a falhas, etc), tem motivado um sem número de modelos e ferramentas, criados para atender as necessidades da área.

O uso de *threads* é a opção mais direta para o aproveitamento de computadores com múltiplas CPUs (*Central Processing Units*) e **memória compartilhada**, pois seu modelo coincide exatamente com o modelo da arquitetura de hardware destas máquinas. BUTENHOF explica o funcionamento e a utilização das *POSIX Threads* em seu livro *Programming with POSIX Threads* (1997). DOWNEY (2008) discute uma extensa lista de problemas e padrões de sincronização.

O MPI, padrão de fato da indústria de computação de alto desempenho, adota uma abordagem diferente, a de **troca de mensagens**, mais adequada a sistemas de memória distribuída como os *clusters* onde o MPI é muito utilizado. O padrão é muito bem apresentado em GROPP; LUSK; SKJELLUM (1994) – versão 1.1 do padrão – e GROPP; LUSK; THAKUR (1999) – versão 2.0 –, além do volume duplo de referência por SNIR et al. (1998). QUINN (2004) tem uma abordagem prática, apresentando algoritmos paralelos clássicos, e trazendo uma visão mais geral da área de programação paralela, também confrontando e combinando o MPI com o OpenMP[1].

O OpenMP acresce diretivas de pré-processador às linguagens C, C++ e FORTRAN para que o desenvolvedor indique trechos de código que podem ser paralelizados de forma automática, usando *threads*. É um das muitas linguagens/bibliotecas/*frameworks* de desenvolvimento que fornecem para o programador uma abstração maior sobre a implementação de *threads*.

O projeto Cilk (BLUMOFE et al., 1996), do MIT, é outra extensão da linguagem C, que permite a chamada de funções como *threads* separadas, e a sincronização

[1] http://openmp.org/wp/

delas, de forma simples, elegante e eficiente. Esse projeto derivou um produto comercial, o Cilk++[2] que promete simplificar a programação paralela para a grande massa de programadores.

A biblioteca de C++ *Threading Building Blocks* (TBB)[3] da Intel, é uma alternativa de facilitação do desenvolvimento com *threads*, permitindo a definição de tarefas de alto nível para serem executadas de forma concorrente, e a utilização de *templates* C++ prontos de alguns algoritmos paralelos.

A troca de mensagens é uma das características mais marcantes da linguagem Erlang[4], que tem instruções de envio e recepção de mensagens fazendo parte do cerne da linguagem. O foco da Erlang no entanto está na construção de sistemas distribuídos e confiáveis, com mínimo tempo de parada, e não na construção de sistemas de alto desempenho. O projeto HiPE (*High Performance Erlang*), que foi integrado a distribuição livre do Erlang, desenvolveu um compilador *Just-In-Time* (JIT) para código nativo para o Erlang. Algumas medições mostram um aumento significativo de desempenho(JOHANSSON; PETTERSSON; SAGONAS, 2000). ARMSTRONG (2007) contém um capítulo dedicado a programação de computadores *multicore*, enfatizando a alta escalabilidade que pode ser alcançada por programas feitos em Erlang, caso sejam seguidas algumas boas práticas durante seu desenvolvimento.

Uma **abordagem híbrida** é adotada pelo *Unified Parallel C* (UPC)[5], que fornece uma ilusão de memória compartilhada para o programador, enquanto usa uma combinação de memória compartilhada e troca de mensagens para comunicação entre as *threads* e processos do sistema.

Um dos primeiros trabalhos a abordar a implementação de processos MPI como *threads* foi o TOMPI (DEMAINE, 1997), com o propósito de ser um ambiente de testes de programas paralelos em um único computador (com um ou mais processadores). Outro trabalho semelhante é o TMPI (TANG; SHEN; YANG, 1999), buscando diretamente o aumento de desempenho de aplicações MPI em máquinas de memória compartilhada e *clusters* delas.

O MPICH[6] e o Open-MPI[7] são as distribuições MPI de código aberto com desenvolvimento ativo[8] mais difundidas, e as duas tem módulos de comunicação que utilizam memória compartilhada. Vários fabricantes possuem sua implementação proprietária de MPI, e possivelmente todos tem suporte a esta otimização nos seus produtos.

Tanto TOMPI quanto TMPI adotam uma estratégia de tradução/pré-processamento do código para solucionar o uso de variáveis globais e estáticas no programa original (ver seções 3.3 e 6.3 para mais detalhes sobre este problema), no entanto não é conhecido trabalho que busque tirar outras vantagens de técnicas de **compilação**, como este.

A referência canônica de compiladores é o "Livro do Dragão" (AHO; SETHI; ULLMAN, 1986). PARR (2007) faz uma introdução mais superficial e leve à área enquanto apresenta sua ferramenta de geração de compiladores. A seção 3.1 apresenta

[2]http://www.cilk.com

[3]http://www.threadingbuildingblocks.org/

[4]http://www.erlang.org/

[5]http://upc.lbl.gov/

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

[7]http://www.open-mpi.org/

[8]O desenvolvimento da conhecida distribuição LAM-MPI está em sua maior parte terminado, em favor do Open-MPI, onde seus autores estão focados atualmente.

algumas ferramentas deste tipo, discute suas principais características, e mostra qual foi escolhida para auxiliar a implementação do M2PI.

3 PROJETO DO COMPILADOR M2PI

Este capítulo apresenta algumas premissas e definições que formaram uma base para o desenvolvimento deste trabalho. Foram decisões tomadas na fase inicial, como qual a ferramenta utilizada para criar o compilador (seção 3.1); quais as simplificações necessárias para manter o trabalho realizável dentro do tempo disponível (seção 3.2); e qual a arquitetura geral adotada para a implementação do M2PI (seção 3.3).

3.1 Ferramentas para a construção de compiladores

Como o reconhecimento e interpretação de código fonte é uma tarefa central na implementação deste trabalho, foi feita uma pesquisa sobre ferramentas que possam servir de auxílio. Essas ferramentas são classificadas normalmente sob o rótulo de **Compiladores de Compiladores** (CC), e normalmente se compõem de um gerador de analisadores léxicos – que reconhecem e agrupam sequências de caracteres em entidades significativas chamadas normalmente de *tokens* – e de um gerador de analisadores sintáticos, ou *parsers* que reconhecem e tomam ações sobre um conjunto estruturado de *tokens*.

Normalmente estas ferramentas lêem arquivos de especificação com regras que especificam a gramática – usando notações similares a BNF (*Backus-Naur Form*) –, e os *tokens* – usando expressões regulares –, e permitem que ações sejam associadas a cada regra. Estas ações são nada mais que trechos de código na mesma linguagem na qual o *lexer* ou *parser* é gerado, que vão ser executadas quando um trecho da entrada do analisador casar com a regra associada.

A dupla mais tradicional e conhecida de *lexer* e *parser* é a *Lex & Yacc* (MASON; BROWN, 1990), hoje em dia normalmente representados pelas implementações compatíveis do projeto GNU[1], respectivamente o Flex e o Bison. O Yacc suporta gramáticas LALR(1), gerando um parser *bottom-up* em código C. Flex e Bison suportam também a linguagem C++.

O *JavaCC*[2] é a opção mais popular para a plataforma java. Ele gera analisadores léxicos e *parsers top-down* LL(k) escritos em Java, e inclui ferramentas para geração de árvores de sintaxe, e geração automática de documentação a partir dos arquivos de definição da gramática.

Algumas das vantagens destacadas pelos seus autores são suas ótimas facilidades de depuração do processo de reconhecimento, mensagens de erro e diagnóstico muito claras, o suporte completo a Unicode nas gramáticas e nos *parsers* gerados, e a

[1] http://www.gnu.org
[2] https://javacc.dev.java.net/

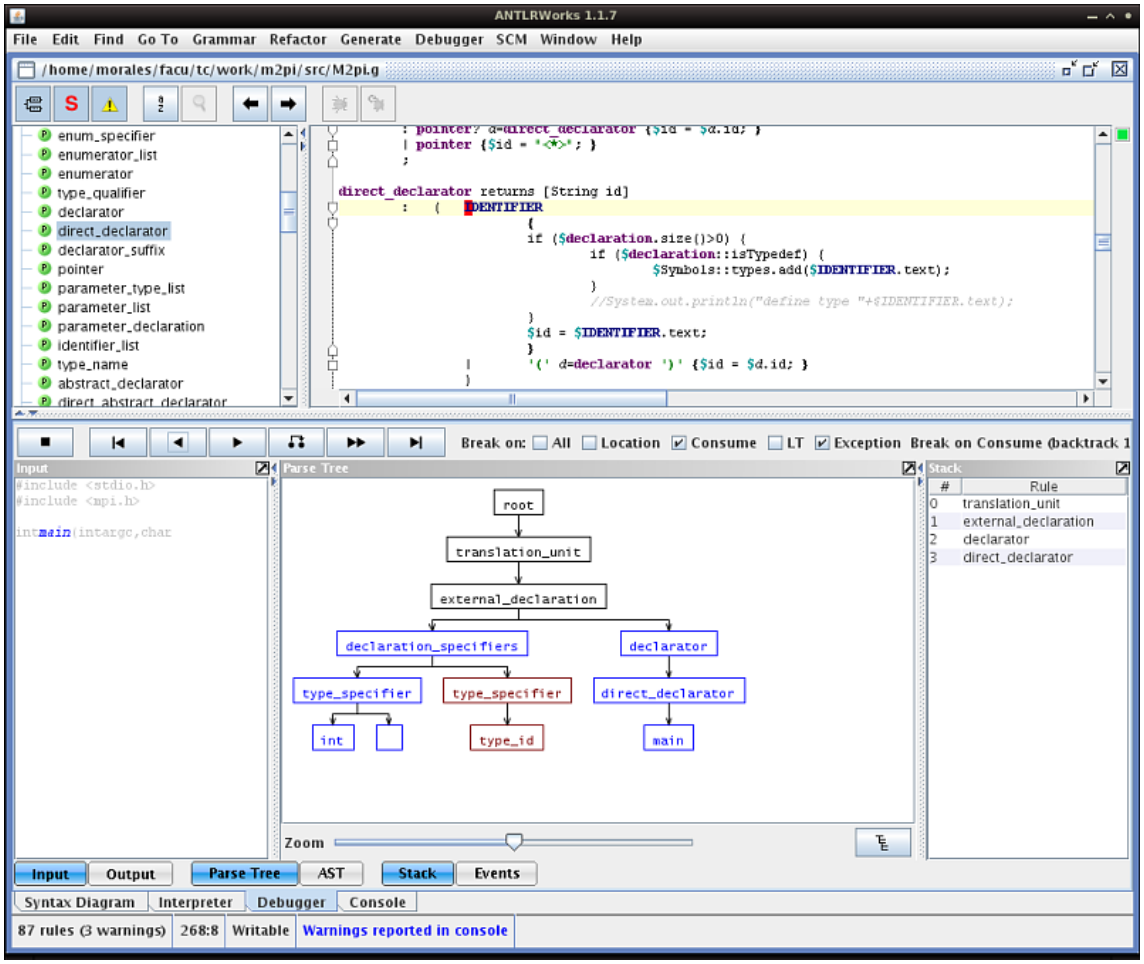

Figura 3.1: ANTLRWorks durante uma sessão de depuração do M2PI

portabilidade associada a plataforma Java.

O *ANTLR*[3] (*ANother Tool for Language Recognition*), é uma outra opção popular na comunidade Java, mas suportando também (com variados níveis de completude) C, C++, Python, C#, Objective C, entre outras. Ele gera *parsers top-down* com um algoritmo LL(*), uma extensão ao algoritmo LL(k), que permite um nível de *lookahead* arbitrário (PARR, 2007).

O ANTLR foi o escolhido, pois apresenta vantagens similares às do JavaCC, e algumas outras que são bastante úteis para este trabalho. Vantagens como o ANTLRWorks, uma IDE (*Integrated Development Environment*) que possui um interpretador e um depurador embutidos (ver figura 3.1), permitindo uma análise detalhada do processo de reconhecimento; e como o seu mecanismo de modelos, que separa a lógica de geração do texto do seu conteúdo, e pode facilitar a geração de programas que fazem tradução de código ou geram algum tipo de texto estruturado, como é o caso deste trabalho.

Para mais informações sobre o funcionamento de *lexers* e *parsers*, algoritmos *bottom-up* e *top-down* e as classes de linguagens LALR e LL, ver (AHO; SETHI; ULLMAN, 1986).

[3] http://www.antlr.org/

3.2 Premissas e simplificações

Para que os objetivos traçados possam ser atingidos em tempo compatível com a carga horária prevista para o TCC, diversas simplificações são necessárias. Em particular, é assumido sobre o programa MPI original[4]:

- A quantidade total de memória utilizada (incluindo todos os processos/*threads* do programa) cabe no espaço de endereçamento disponível para um processo.
- Todos os arquivos abertos simultaneamente podem ser abertos da mesma forma por um processo apenas.
- Não utiliza bibliotecas ou chamadas de sistema que não sejam *thread-safe*.
- Não faz uso de *threads*.

Além dessas premissas, os seguintes itens são importantes para a simplificação da implementação do M2PI:

- O M2PI não valida completamente o código C que interpreta, deixando que a validação completa seja feita só pelo GCC sobre o programa traduzido. A maior desvantagem disso é a dificuldade extra na depuração de erros de compilação, pois a numeração das linhas e até o próprio código envolvido no relatório de erros pode ser diferente do original passado pelo usuário ao M2PI.
- Portabilidade não é uma preocupação primordial deste trabalho. Ele está sendo desenvolvido em uma plataforma Intel 32, com ambiente Ubuntu GNU / Linux de 32 bits.

Também é abordado apenas um pequeno subconjunto das funções definidas pelo padrão MPI 1.1, como é detalhado abaixo na seção 3.2. Por fim, a prova formal da equivalência entre o programa original e o traduzido não faz parte do escopo do trabalho.

Definição do subconjunto MPI a ser tratado

Foram escolhidas para este projeto apenas sete funções do MPI, listadas abaixo.

- Funções Básicas: `MPI_Init, MPI_Finalize, MPI_Comm_size, MPI_Comm_rank.`
- Funções de comunicação ponto-a-ponto: `MPI_Send, MPI_Recv.`
- Funções de comunicação coletiva: `MPI_Bcast.`

As seis primeiras são apontadas por GROPP; LUSK; SKJELLUM (1994) como sendo suficientes para a implementação de programas completos de troca de mensagens. O resto do padrão define operações de mais alto nível, que podem ser construídas usando estas seis funções. A sétima função escolhida (`MPI_Bcast`) é um exemplo destas operações de alto nível, uma pequena amostra das funções de comunicação coletiva presentes no MPI.

[4]Estes quatro itens abaixo são as mesmas premissas de TANG; SHEN; YANG (1999).

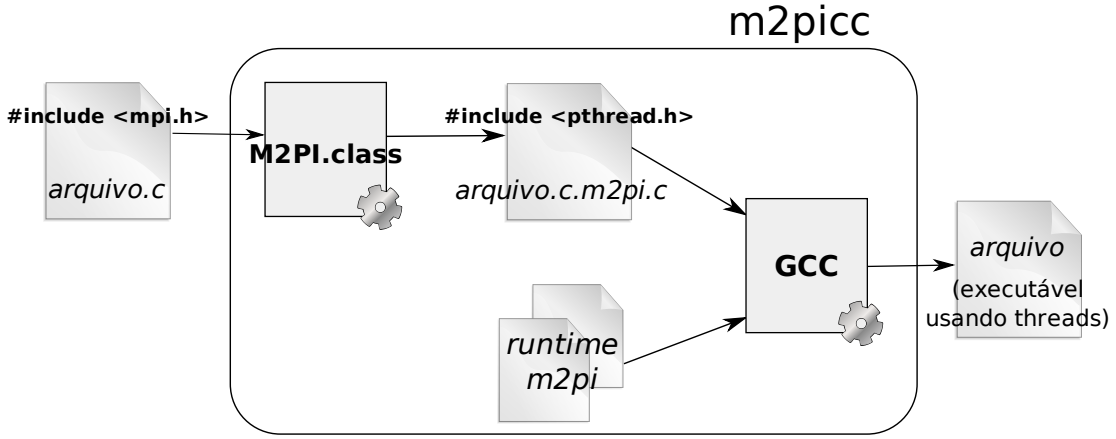

Figura 3.2: Visão geral da compilação de um programa C/MPI usando o M2PI

3.3 Arquitetura do compilador

Para efetuar a tradução de programas C/MPI para C/*Pthreads*, o M2PI precisa executar três tarefas distintas, a saber:

Transformação das variáveis globais e estáticas: Para preservar, em um conjunto de *threads*, a semântica de um conjunto de processos MPI que acessam variáveis globais ou locais estáticas, é preciso transformar essa variáveis em dados que de alguma forma sejam particulares a cada *thread*. TANG; SHEN; YANG (1999) apresentam este problema e suas possíveis soluções.

Gerenciamento das *threads* e da infraestrutura de comunicação: Geração de código para criar as N *threads* (N só é conhecido no momento da execução), as estruturas de comunicadores e grupos, entre outras necessárias para a comunicação entre as *threads*.

Tradução das funções MPI: A implementação das funções MPI escolhidas (seção 3.2) usando os recursos construídos na tarefa mencionada acima. Cada função é um subproblema diferente. É neste ponto que um ganho pode ser atingido pela abordagem de compilação usada neste trabalho.

Após estas tarefas serem completadas, o código fonte resultante (com extensão m2pi.c[5]) deve ser ligado com a biblioteca *runtime* do M2PI para gerar um executável, equivalente ao programa original, mas que usa *threads* ao invés de processos. A figura 3.2 ilustra esse processo. Um *script* chamado **m2picc** encapsula estes diferentes passos, fornecendo um processo de compilação análogo ao que seria feito para o programa MPI.

O compilador M2PI é construído em Java usando o ANTLR. A biblioteca *runtime* é feita em C usando POSIX Threads, e o utilitário mpicc é apenas um *script* bash simples que invoca o compilador M2PI e compila o código C resultante ligando este com a biblioteca. A seção 6 trata dos detalhes da implementação do compilador e da biblioteca. As seções 4.1, 4.2.1, 4.2.2 e 5 servem de suporte ao entendimento

[5]O código transformado permanece no sistema de arquivos com esta extensão, possibilitando a sua inspeção manual.

dos detalhes da implementação, apresentando ferramentas e conceitos que foram utilizados nela.

4 INTERFACES DE PROGRAMAÇÃO PARALELA

Este capítulo apresenta o que pode ser tido como o formato de entrada e o formato de saída do M2PI.

A entrada do compilador M2PI é um programa C utilizando MPI, uma interface de programação paralela baseada em troca de mensagens. Os principais conceitos e funções MPI necessários para o acompanhamento da implementação deste trabalho são apresentados na seção 4.1.

A saída do compilador M2PI é um programa C utilizando *threads*. A seção 4.2 versa sobre elas, apresentando a tradicional biblioteca Posix Threads, e também o recurso *Thread Local Storage*, que foi utilizado neste trabalho para preservar a semântica do acesso de variáveis globais e locais estáticas em um programa que usa *threads*.

4.1 MPI: A linguagem de entrada do M2PI

O MPI (*Message Passing Interface*) é uma norma que define uma API (*Application Programming Interface*) para a construção de programas paralelos usando troca de mensagens. É o padrão de fato da indústria de computação de alto desempenho, onde sistemas de memória distribuída baseados em grandes agrupamentos de computadores poderosos são o caso mais comum, e por isso suporta os mais variados tipo de redes e mecanismos de interconexão de computadores. Esta seção apresenta uma breve introdução às interfaces definidas no MPI, e aos conceitos e terminologia empregados na norma. Para aprofundar o conhecimento sobre ela, podem ser consultados GROPP; LUSK; SKJELLUM (1994), GROPP; LUSK; THAKUR (1999), e SNIR et al. (1998).

Apesar do MPI ser uma especificação muito grande, GROPP; LUSK; SKJELLUM (1994) aponta que quase qualquer programa MPI pode ser implementado com apenas seis funções básicas. O resto da norma fornece na maior parte combinações e atalhos para idiomas freqüentes que podem construídos de forma mais complexa com estas primitivas básicas. Estas seis funções são apresentadas nos tópicos 4.1.1, 4.1.2 e 4.1.3 a seguir. A seção 4.1.4 apresenta uma sétima função, um recurso extra associado às comunicações coletivas suportadas pelo MPI.

4.1.1 Inicialização e finalização

As funções `MPI_Init` e `MPI_Finalize` fazem a inicialização e finalização da biblioteca MPI. Suas assinaturas são as seguintes:

```
int MPI_Init(int *argc, char ***argv)
```

```
int MPI_Finalize()
```

Muito poucas funções do MPI podem ser chamadas fora do trecho de código delimitado por estas duas funções. A norma não especifica como deve ser o ambiente antes e depois da execução delas, nem ao menos quantos processos existirão nesses momentos. Por isso é recomendado que se faça o mínimo possível fora delas, e programas MPI normalmente começam por `MPI_Init` e terminam por `MPI_Finalize` (a menos de um `return` colocado após esta última).

4.1.2 Ranks, grupos, e comunicadores

Os processos no MPI se dividem em grupos, dentro dos quais cada um tem um número de identificação, o *rank*. A cada grupo estão associados um ou mais comunicadores, que organizam os processos em uma topologia virtual. O comunicador é a referência utilizada nas funções de comunicação, indicando quais processos estão relacionados com a operação, e em que ordem eles tomam parte nela, quando necessário.

Ao inicializar um programa MPI, será criado um comunicador global, chamado `MPI_COMM_WORLD`, contendo todos os processos que foram disparados.

Entre as seis funções básicas mencionadas neste capítulo, duas delas inspecionam comunicadores para descobrir quantos processos estão agrupados no comunicador, e qual o seu *rank* entre eles. São respectivamente as funções `MPI_Comm_size` e `MPI_Comm_rank`, cujas assinaturas são exibidas abaixo:

```
int MPI_Comm_size(MPI_Comm comm, int *size)
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

As duas funções recebem um comunicador `comm`, e um ponteiro para um inteiro onde a informação requisitada será armazenada. O valor retornado por elas, como por todas funções MPI (à exceção de duas que tratam da contagem de tempo) é uma sinalização de sucesso ou insucesso na execução da operação.

4.1.3 Comunicação ponto-a-ponto

As últimas duas das seis funções básicas são a `MPI_Send` e `MPI_Recv`, que permitem a comunicação entre os processos. Esta comunicação nada mais é do que a transferência de uma série de bytes de um processo para outro processo. Suas assinaturas são as seguintes:

```
int MPI_Send(void* buf, int count, MPI_Datatype datatype,
     int dest, int tag, MPI_Comm comm)

int MPI_Recv( void *buf, int count, MPI_Datatype datatype,
     int source, int tag, MPI_Comm comm, MPI_Status *status )
```

O significado de cada parâmetro é explicado na lista abaixo:

buf: Indica a área de memória que é a origem ou destino dos bytes.

count: Quantidade de elementos transferidos na mensagem.

datatype: Tipo dos dados transferidos na mensagem, normalmente indicado por constantes `MPI_INT`, `MPI_FLOAT`, `MPI_CHAR`, etc.

source/dest: Os *ranks* de origem/destino da mensagem, sendo que a origem pode ser especificada como a constante `MPI_ANY_SOURCE`, indicando que mensagens de qualquer destino devem ser recebidas por essa chamada.

tag: Chamadas `MPI_Recv` podem especificar que desejam receber apenas mensagens marcadas com um número específico, o `tag`, atuando este como um filtro.

comm: O comunicador utilizado para transferência da mensagem.

status: A estrutura apontada por este parâmetro recebe informações sobre a mensagem recebida, como por exemplo sua origem, importante informação quando `MPI_ANY_SOURCE` é utilizado.

Além disso, vale notar que a operação de recebimento é bloqueante, e a norma MPI institui que o `MPI_Send` pode bloquear até que seja feito o recebimento da mensagem. O MPI suporta outros modos de comunicação, incluindo um não-bloqueante através de funções `MPI_ISend` e `MPI_IRecv`, mas estes modos adicionais não são abordados neste trabalho.

4.1.4 Comunicação coletiva

Entre as muitas facilidades de comunicação providas pela norma MPI, uma delas é um conjunto de funções de comunicação coletiva. A mais básica delas é a função de nome `MPI_Bcast`, que faz a difusão de dados. Sua assinatura é apresentada abaixo:

```
int MPI_Bcast (void *buf, int count, MPI_Datatype datatype,
     int root, MPI_Comm comm )
```

O significado dos parâmetros é similar ao das chamadas de comunicação ponto-a-ponto, mas esta função indica adicionalmente a raiz da difusão, o *rank* especificado pelo parâmetro `root`. O `MPI_Bcast` é chamado por todos os processos que fazem parte de um grupo comunicador. Após o retorno da chamada, o *buffer* do processo raiz foi copiado para o *buffer* de cada outro processo do grupo.

Existem várias outras funções de comunicação coletiva, que fazem outras formas de distribuição dos dados, fazem a redução de resultados usando várias operações aritméticas, entre outros recursos. Mas a `MPI_Bcast` é a única abordada neste trabalho, sendo implementada pelo M2PI como uma funcionalidade extra, além das outras seis funções básicas discutidas anteriormente.

4.2 Threads: A linguagem de saída do M2PI

4.2.1 Posix Threads

POSIX (*Portable Operating System Interface*) é o nome de uma coleção de padrões relacionados, especificados pelo IEEE[1] (Institute of Electrical and Electronics Engineers), que definem diversas interfaces de programação e utilização de sistemas operacionais. Um destes padrões é o **POSIX Threads**, ou **Pthreads**, que define

[1] http://www.ieee.org

uma API para o desenvolvimento de aplicações com múltiplas *threads*. Esta seção apresenta uma visão geral sobre as interfaces definidas no padrão e sobre os conceitos subjacentes. Uma abordagem completa sobre o Pthreads é dada em BUTENHOF (1997).

Uma *thread* é um contexto de execução; um fluxo, ou "linha" (a tradução literal de *thread*), de execução dentro de um processo em um sistema operacional (SO). Um processo com várias "linhas" de execução distintas e concorrentes é chamado de um processo *multithreaded*, podendo assim executar diferentes tarefas de forma asíncrona e independente. Enquanto uma tarefa fica bloqueada, outras continuam sua execução, evitando que o processo fique "parado". Além disso, todas *threads* compartilham a memória alocada para o processo, o que fornece um meio de comunicação extremamente rápido e direto.

Adicionalmente, dependendo da forma de implementação do sistema de *threads*, elas permitem que as várias unidades de processamento de uma máquina multiprocessada sejam aproveitadas por um mesmo processo. Para isso é necessário que as *threads* sejam implementadas em *nível de sistema*, com suporte do hardware e do sistema operacional (o *kernel* enxerga e manipula cada *thread* individualmente). Mas *threads* podem ser também gerenciadas por código executando completamente no espaço de usuário do SO (ditas então de *nível de usuário*), sem nenhum conhecimento delas no *kernel*. Modelos híbridos (usuário/sistema) também são utilizados.

Neste trabalho consideramos uma implementação em nível de sistema, não híbrida, a mais comumente usada nas plataformas Linux atuais. Informações sobre as diferenças, vantagens e desvantagens destas opções podem ser encontradas em TOSCANI; OLIVEIRA; CARISSIMI (2003), TANENBAUM (2001), ou BUTENHOF (1997).

A norma POSIX, como trata da interface de programação, é independente da opção de implementação do sistema de *threads*. Ela define as assinaturas e a semântica de uma séries de funções para criação, destruição, manutenção, espera e comunicação entre *threads*. Todas destas que foram utilizadas na implementação deste trabalho de conclusão são apresentadas a seguir, nas seções 4.2.1.1, *Criação e junção*; e 4.2.1.2, *Sincronização*. A seção 4.2.1.3 dá uma breve visão sobre o conceito de *Thread Specific Data* (TSD), uma forma de recuperar a semântica de variáveis globais em um programa feito com Pthreads.

4.2.1.1 Criação e junção

Na biblioteca Pthreads, uma *thread* é criada usando a chamada `pthread_create`. Essa função recebe um ponteiro para uma outra função que irá definir o código da *thread*, e um outro ponteiro para uma variável que será passada como parâmetro para a função apontada. A assinatura completa segue abaixo:

```
int pthread_create(pthread_t  *  thread,
  pthread_attr_t * attr,
  void * (*start_routine)(void *), void * arg);
```

A variável apontada de tipo `pthread_t` serve de identificador da *thread*. Como todos tipos definidos pelo padrão, ele é um tipo opaco, ou seja, não deve ser assumido nada sobre o seu formato, se é um inteiro, ou uma estrutura, etc. O parâmetro `attr` pode alterar uma série de características da *thread*, como por exemplo seu escopo: de sistema ou de usuário, em implementações que suportam os dois tipos. Para a

maioria dos casos, no entanto, o valor `NULL` pode ser passado como `attr`, criando uma *thread* "padrão".

Quando termina a *thread* principal do processo (aquela que executa a função `main`), todo o processo é terminado, sem esperar o término de cada *thread* que foi disparada. Por isso, é prática comum fazer a função `main` aguardá-las, usando a função `pthread_join`. Essa função, com a assinatura logo abaixo, faz a junção da *thread* passada como argumento à *thread* chamadora, aguardando o seu final, liberando os recursos utilizados por ela, e opcionalmente guardando seu valor de retorno.

```
int pthread_join(pthread_t th, void **thread_return);
```

A listagem 6.5 mostra um código gerado pelo M2PI usando as funções `pthread_create` e `pthread_join`. Estas são as funções mais básicas da Pthreads, e programas concorrentes podem ser escritos usando somente elas. Normalmente porém, em programas de alguma complexidade, dados compartilhados são acessados pelas *threads*, ou elas precisam executar uma série distribuída de passos em uma ordem coerente. Nestes casos, é preciso **sincronizar** as *threads*.

4.2.1.2 Sincronização

Normalmente um programa de certa complexidade possui algumas estruturas de dados compostas que, quando alteradas, passam temporariamente por um estado inconsistente. Por exemplo, uma lista duplamente encadeada, ao ter um de seus elementos removidos, necessariamente passará por um estado onde os elos de um dos lados deste elemento já foram ajustados e do outro lado não, fazendo com que a lista seja diferente dependendo do sentido em que é percorrida.

Quando múltiplos fluxos de execução estão acessando dados compartilhados, este tipo de inconsistência precisa ser escondido de todos os fluxos menos daquele que está efetuando a alteração. O código que executa a alteração constitui uma **seção crítica** (SC), um trecho de código que deve ser executado por no máximo uma *thread* ao mesmo tempo. Então as *threads* precisam coordenar, ou em outras palavras, sincronizar, sua execução de uma SC.

São apresentados agora três mecanismos diferentes de sincronização, o **mutex**, as **variáveis de condição**, e as **barreiras**[2].

Mutex

Um dos principais recursos da Pthreads é um mecanismo de exclusão mútua, o *mutex*. Um *mutex* está trancado ou destrancado[3]. Quando uma *thread* deseja entrar em uma seção crítica, ela tranca o *mutex* com a função `pthread_mutex_lock`, função esta que só retorna quando o *mutex* tiver sido "capturado" para esta *thread*. Se o *mutex* estiver trancado por outra *thread*, a requisitante ficará presa aguardando que a atual dona do *mutex* o destranque com a função `pthread_mutex_unlock`.

[2]Cabe lembrar aqui que **semáforos** são uma outra alternativa muito popular de sincronização em programas *multithreaded*, mas são definidos por uma norma diferente da família POSIX. BUTENHOF argumenta que o uso do *mutex* e de variáveis de condição é preferível na maioria dos casos, por sua maior simplicidade.

[3]Por vezes serão usados neste relatório os termos "alocado" e "desalocado", principalmente quando se tratar de um recurso que é protegido por um *mutex*, e quando o contexto não permitir confusão com o sentido de "alocação de memória".

Antes de ser usado, o *mutex* – identificado por um tipo abstrato `pthread_mutex_t` – deve ser inicializado pela função `pthread_mutex_init`, e quando não for mais necessário, deve ser destruído pela `pthread_mutex_destroy`, liberando recursos do sistema. Alternativamente o *mutex* pode ser inicializado com uma constante `PTHREAD_MUTEX_INITIALIZER`. Seguem as assinaturas destas funções:

```
int pthread_mutex_init(pthread_mutex_t *mutex,
  const pthread_mutexattr_t *mutexattr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Variáveis de condição

Além do *mutex* para exclusão mútua, o POSIX *threads* disponibiliza um recurso de comunicação, através das variáveis de condição (VC). Uma VC está relacionada a um predicado, como "a lista está vazia", ou "a lista está cheia". Uma *thread* pode escolher aguardar até que o predicado se torne verdadeiro, para isso chamando a função `pthread_cond_wait`. Quando uma *thread* altera dados de forma a satisfazer a condição, ela sinaliza esse fato, "acordando" uma (com a `pthread_cond_signal`), ou todas (com a `pthread_cond_broadcast`) *threads* que estão esperando no `pthread_cond_wait` por aquela VC. As assinaturas destas funções e das de inicialização e destruição de variáveis de condição são listadas abaixo:

```
int pthread_cond_init(pthread_cond_t *cond,
  pthread_condattr_t *cond_attr);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond,
  pthread_mutex_t *mutex);
int pthread_cond_destroy(pthread_cond_t *cond);
```

Como pode-se notar pelas assinaturas, uma variável de condição está sempre associada a um *mutex*, que garantirá o acesso exclusivo ao objeto da condição pelas *threads* envolvidas. As funções de sinalização não alteram o *mutex*, sendo responsabilidade do programador cuidar de destrancá-lo antes ou depois da chamada de sinalização. Mas as funções de espera, como a `pthread_cond_wait`, sempre devem ser chamadas com o *mutex* relacionado trancado pela *thread* chamadora. A função destrancará o *mutex*, e quando ela retornar, após a condição ser satisfeita e sinalizada, ele estará novamente trancado.

É importante não assumir que o predicado seja verdadeiro após o retorno do *wait*, testando a condição antes prosseguir, pois esta pode já não ser mais verdadeira quando a *thread* conseguir trancar o *mutex* (BUTENHOF (1997), na página 80, explica três situações onde isso é possível: predicados aproximados, *wakeups* interceptados, e *wakeups* espúrios). A função de recebimento de mensagens do M2PI por exemplo, toma essa precaução. No cenário descrito na figura 6.1 da seção 6.4.2, ela insere um descritor de recebimento de dados no canal de comunicação do M2PI, e então aguarda a sinalização de que o descritor já foi processado pela *thread* que faz o envio do dados. Isso é feito com o seguinte código:

```
hqPush(M2PI_CHANNELS[src][dst].recv, handler);
while (handler->done == 0)
    pthread_cond_wait(&handler->cond_done,
            &M2PI_CHANNELS[src][dst].mutex);
```

A variável `done` do descritor `handler` é o predicado da condição, e é assinalada como verdadeira pela função que faz o envio dos dados, antes desta sinalizar a VC. A função de recebimento aguarda a condição dentro de um laço que testa o valor de `done`, acordando realmente apenas quando o predicado for verdadeiro.

Barreiras

Uma outra primitiva de sincronização muito útil, fornecida por uma extensão do padrão Pthreads original, é a *barreira*. Uma barreira provê um "ponto de encontro" para um grupo de *threads*. A barreira deve ser inicializada uma vez com um contador de valor $N > 0$, pela função `pthread_barrier_init`. Cada *thread* interessada na barreira chama a função `pthread_barrier_wait`, que decrementa o contador e depois espera até que o contador chegue a zero, para só retornar. Ou seja, todas *threads* que chegam a instrução de *wait*, são liberadas ao mesmo tempo, só após a chegada da enésima *thread*. As funções tem as seguintes assinaturas:

```
int pthread_barrier_init(pthread_barrier_t *restrict barrier,
    const pthread_barrierattr_t *restrict attr, unsigned count);
int pthread_barrier_wait(pthread_barrier_t *barrier);
int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

Barreiras são utilizadas no M2PI para fazer uma implementação muito simples da função de *broadcast* do MPI, apresentada na seção 6.5.

4.2.1.3 Thread-Specific Data

Thread-Specific Data (TSD) é um recurso das POSIX Threads que permite a declaração de variáveis que tenham um mesmo nome em todas as *threads*, mas que tenham valores diferentes para cada uma delas. Seu objetivo é facilitar a utilização de dados que, em um programa que não usa *threads*, seriam normalmente alocados estaticamente na sua área de memória, como variáveis globais ou variáveis locais declaradas com o qualificador `static`. E é exatamente para traduzir esse tipo de variável que o TSD é interessante para a implementação do M2PI.

A desvantagem deste recurso, no entanto, é que a sintaxe de leitura e escrita dessas variáveis TSD é completamente diferente da usual, exigindo chamadas de funções para realizar essas operações, além de chamadas de função para criação e destruição de uma "chave" de acesso à variável. Como o M2PI é um tradutor automático, esta inconveniência não seria grande empecilho para sua utilização.

Não obstante, existe outro recurso equivalente, com sintaxe mais transparente, chamado **Thread Local Storage** (TLS), disponibilizado pelo compilador C utilizado pelo M2PI. Por sua simplicidade, foi decidido usar o TLS em favor do TSD Posix neste trabalho. A seção 4.2.2 a seguir apresenta maiores detalhes sobre seu funcionamento.

4.2.2 Thread Local Storage

Thread Local Storage (TLS) é um mecanismo suportado pelo *GNU Compiler Collection* (GCC), que permite a definição de variáveis globais e estáticas de forma que exista uma instância de cada variável para cada *thread*, e todo o trabalho de criação, acesso, e destruição dessas variáveis é feito pelo compilador e pelas bibliotecas do sistema operacional.

O TLS é uma alternativa ao *Thread Specific Data*, que simplifica a utilização da funcionalidade que os dois disponibilizam. Tudo que é preciso fazer é declarar a variável em questão com o modificador `__thread`. A partir de então, o uso dela é absolutamente idêntico ao de uma variável comum, como pode ser visto na listagem 6.11 mostrada na seção 6.3. É interessante notar que o operador C que retorna o endereço de uma variável – o & – quando aplicado a uma variável TLS, tem seu resultado computado em tempo de execução, e tal endereço pode ser utilizado por qualquer outra *thread*. Mas quando uma *thread* se encerra, todos ponteiros para variáveis TLS desta *thread* se tornam inválidos.

O manual do GCC dá mais detalhes sobre a aplicabilidade e as limitações do *Thread Local Storage*[4].

[4]http://gcc.gnu.org/onlinedocs/gcc-4.3.0/gcc/Thread_002dLocal.html#Thread_002dLocal

5 ANTLR: COMPILADOR DE COMPILADORES

O ANTLR (*ANother Tool for Language Recognition*) é um compilador de compiladores, que gera código para analisadores sintáticos e léxicos a partir da definição de uma gramática aumentada. A gramática é dita "aumentada", pois ações são associadas com as regras de geração da gramática, sendo executadas quando o *parser* gerado encontra construções correspondentes no texto de entrada. As ações são especificadas como trechos de código feitos na linguagem alvo, a linguagem na qual o código dos analisadores é gerado. E são as ações que transformam o texto analisado, seja em um outro programa escrito na mesma linguagem de entrada, seja em um programa de uma linguagem completamente diferente. Assim, a gramática não apenas valida a pertinência da entrada a uma linguagem, mas também compila uma nova saída a partir dela.

Os *parsers* gerados pelo ANTLR usam um algoritmo *top-down* LL(*), uma extensão do algoritmo LL(k) que permite um nível de *lookahead* arbitrário. Uma descrição do algoritmo LL(*) e dos recursos do ANTLR é dada de forma detalhada em PARR (2007). A referência canônica sobre a construção de compiladores é AHO; SETHI; ULLMAN (1986).

Este capítulo visa apresentar alguns dos recursos do ANTLR importantes para a implementação deste trabalho. A seção 5.1 apresenta a sintaxe da definição das regras da gramática; a seção 5.2 discute as maneiras pelas quais as regras podem alterar o texto analisado e gerar um novo texto estruturado; e por fim a utilização dos escopos do ANTLR é descrita na seção 5.3.

5.1 Regras

As regras no ANTLR se assemelham a regras de uma definição formal de gramáticas. Eis um exemplo de regra, que define instruções de desvio incondicional na gramática da linguagem C:

```
jump_statement
    : 'goto' IDENTIFIER ';'
    | 'continue' ';'
    | 'break' ';'
    | 'return' ';'
    | 'return' expression ';'
    ;
```

Esta regra define as construções derivadas a partir de `jump_statement`, que por este fato – derivar outras construções – é chamado de um **não-terminal**. As alter-

nativas de construções são separadas por uma barra vertical (|), e podem referenciar outros não-terminais, como é o caso da última alternativa do exemplo, que referencia `expression`. Não-terminais no ANTLR são sempre iniciados por uma letra minúscula.

As palavras `goto`, `continue`, `break`, `return`, `;`, e `IDENTIFIER` são **terminais** (ou *tokens*), que estão associados diretamente a palavras ou símbolos presentes no texto sendo analisado. Terminais podem ser especificados como palavras literais, entre aspas, ou por expressões regulares (ERs), identificadas por nomes que começam por uma letra maiúscula. O terminal `IDENTIFIER` é definido pela expressão abaixo (`LETTER` é um fragmento de uma ER, usado apenas para a reutilização de subexpressões, não podendo ser utilizado em regras de derivação de não-terminais).

```
IDENTIFIER
    :   LETTER (LETTER|'0'..'9')*

fragment
LETTER
    :   '$'
    |   'A'..'Z'
    |   'a'..'z'
    |   '_'
    ;
```

As **ações** podem ser especificadas nas regras colocando o código associado entre chaves. Por exemplo, se fosse desejado listar todas as instruções `goto` de um arquivo fonte, a regra `jump_statement` exibida anteriormente poderia ser alterada da seguinte forma:

```
jump_statement
  : 'goto' id=IDENTIFIER ';' {System.out.println('GOTO: ' + $id.text);}
  | 'continue' ';'
  ...
```

Como se vê neste último exemplo, um terminal como o `IDENTIFIER` possui um valor (atributo `.text`), contendo o texto que foi casado pela ER. Também é possível atribuir um outro nome pelo qual o *token* pode ser referenciado, como é feito com o nome `id` acima.

5.2 Reescrita do fluxo de *tokens* e modelos

O ANTLR disponibiliza duas maneiras de gerar o texto resultante de uma compilação, além da alternativa trivial de usar as ações para imprimir cada linha da saída a partir de uma função como a `System.out.println` do Java. São a opção de reescrita do fluxo de *tokens* gerados pelo analisador léxico, e o uso de modelos (*templates*). A primeira é recomendada para alterações mais simples, e a segunda pode ser usada para casos complexos.

A reescrita é ativada colocando-se a opção `rewrite = true` no cabeçalho da gramática utilizada. Quando isso é feito, o objeto `input` – que contém o fluxo de *tokens* fornecidos pelo analisador léxico – passa a ser uma instância da classe

`TokenRewriteStream`, que fornece métodos para alterá-lo, permitindo a inserção, deleção e substituição de *tokens*. Essas operações de alteração são primeiramente armazenadas, e realizadas somente quando é emitido o texto de saída ao final da compilação, mantendo o *buffer* de *tokens* inalterado durante a análise. O uso desse recurso é muito simples, como é demonstrado nas listagens 6.7 e 6.9 na seção 6.3.

Os modelos funcionam como um tipo especializado de ação, permitindo a geração de texto estruturado diretamente a partir dos terminais e não-terminais que compõem as alternativas de uma regra, além de poder utilizar resultados produzidos por outras ações executadas junto com ele. Cada modelo possui um texto que o define, e esse texto pode conter lacunas que são preenchidas quando o modelo é "chamado" (similarmente a uma chamada de função). As definições dos modelos são especificadas em um arquivo separado, informado ao compilador em tempo de execução. Portanto é possível trocar o conjunto de modelos, alterando a saída gerada, sem modificar o compilador.

O uso de modelos na implementação do M2PI é apresentado nas seções 6.1 e 6.2. Para utilizar um exemplo mais simples neste momento, supõe-se uma gramática cujo objetivo seja reformatar definições de função segundo uma nova norma de estilo de programação (a função deste exemplo é apenas mostrar os elementos sintáticos do uso de modelos em um caso simples). Nesse caso, uma regra como a seguinte poderia ser utilizada:

```
function_definition
  : t=type id=IDENTIFIER '(' al=arg_list ')' b=block
    -> modelo_func(type={$t.text},
   name={$id.text},
   args={$al.text},
   block={$b.text})
  ;
```

O operador `->` indica a aplicação de um modelo, para o qual podem ser passados trechos de texto como parâmetros. No exemplo acima são passados todos os elementos reconhecidos pela regra. Um exemplo de definição do modelo que poderia ser utilizada neste caso segue abaixo:

```
modelo_func(type,name,args,block) ::= <<
<type>
<name>(<args>)
/**** começo da função <name> ****/
<block>
/**** final da função <name> ****/
>>
```

A definição do modelo é delimitada por `<<` e `>>`, e os parâmetros podem ser referenciados pelo seu nome entre `<` e `>`. Qualquer uso destes caracteres no texto literal do modelo deve ser escapado com uma barra invertida.

É possível escolher entre aplicar ou não um modelo (ou aplicar modelos diferentes), com base em **predicados semânticos**. Estes predicados ativam ou desativam alternativas de um regra, e podem também ser usados em outros casos em que não se apliquem modelos. É um recurso útil para especificar ações que devem ser tomadas

em casos especiais, com base não na estrutura do texto analisado, mas no seu valor. Se fosse agora desejado alterar o estilo apenas das funções definidas para o M2PI por exemplo, poderia ser utilizada a regra a seguir.

```
function_definition
  : t=type id=IDENTIFIER {$id.text.beginsWith("M2PI_")}?
      '(' al=arg_list ')' b=block
    -> modelo_func(type={$t.text},
   name={$id.text},
   args={$al.text},
   block={$b.text})
  | type IDENTIFIER '(' arg_list ')' block
  ;
```

A primeira alternativa é selecionada apenas quando o nome da função começa com "M2PI_", aplicando então o modelo. Caso contrário, o texto de entrada permanece inalterado. É importante atentar que a alternativa mais específica deve vir antes da mais geral, pois elas são testadas em ordem, e a primeira que tiver o predicado verdadeiro será selecionada.

5.3 Escopos

Mapeando diretamente um conceito chave em linguagens de programação, o ANTLR implementa escopos como uma de suas palavras-chave (`scope`). Um `scope` define uma série de atributos e pode ser associado a uma ou mais regras. Cada vez que uma destas regras for "executada", uma nova instância desse escopo será empilhada no início, e desempilhada ao final, automaticamente.

A listagem 6.8, na seção 6.3, mostra a utilização deste recurso na implementação do M2PI.

6 IMPLEMENTAÇÃO

Este capítulo apresenta a implementação do M2PI de forma bastante detalhada. Para um bom entedimento das seções por vir, é preciso alguma familiaridade com os conceitos e ferramentas apresentados nos capítulos sobre MPI (4.1), Threads (4.2.1), e sobre o ANTLR (5).

Como já foi mencionado, a implementação do M2PI se divide em duas partes principais: o compilador, e a biblioteca de tempo de execução. O compilador é feito em Java usando o ANTLR, e aproveitando uma gramática C que é distribuída junto com ele, como exemplo. Esta gramática é uma tradução da tradicional gramática ANSI C para o Yacc distribuída por Jutta Degener na Internet[1], e apesar de não suportar a sintaxe completa usada por compiladores C reais, como o da Intel ou do projeto GNU (que possuem muitas extensões ao padrão ANSI), é suficiente para os fins deste trabalho. As três primeiras seções abaixo tratam das funções realizadas pelo compilador do M2PI.

Tanto o código gerado pelo compilador M2PI, quanto a biblioteca de tempo de execução (*runtime*) são feitos na linguagem C, usando a biblioteca de *threads* POSIX. A biblioteca *runtime* implementa funções análogas as do padrão MPI, se restringindo ao subconjunto definido na seção 3.2. E com a exceção das funções de inicialização e finalização, todas as funções MPI deste subconjunto são implementadas integralmente pelo código da biblioteca, sem auxílio de código gerado diretamente pelo compilador, que nesses casos apenas faz uma substituição no prefixo do nome das funções, de "MPI" para "M2PI"[2]. À exceção das seções 6.1 e 6.3, todas a seções a seguir tratam da implementação de funções na biblioteca *runtime*.

O processo de desenvolvimento deste projeto foi em geral orientado pelo teste da compilação e execução de programas MPI que exigem, cada um deles, um subconjunto maior do padrão, motivando o acréscimo de novos recursos ao M2PI. Este capítulo se organiza da mesma forma, seguindo a ordem em que o desenvolvimento foi realizado, e frequentemente apresentando trechos dos programas de teste para auxiliar a explicação dos recursos implementados.

6.1 Reconhecimento das chamadas de funções MPI

Como ações especiais precisam ser tomadas pelo compilador M2PI ao encontrar chamadas às funções do MPI, as regras da gramática C utilizada foram alteradas de

[1] Disponível em http://www.lysator.liu.se/c/ANSI-C-grammar-y.html.

[2] Isso pode parecer estranho, dado que a concepção deste trabalho enfatiza o uso de técnicas de compiladores para realizar a tradução. Uma reflexão sobre este aspecto da implementação é feita na seção 6.6 e também nas considerações finais.

forma a especializar o reconhecimento destas funções, tratando-as como se fossem palavras chave da linguagem.

Isso foi feito encontrando a regra da gramática que reconhece chamadas de função, e colocando uma alternativa adicional, um não-terminal que deriva nas chamadas específicas do MPI, como é mostrado na listagem 6.1. A regra `postfix_expression` deriva, entre outras coisas, chamadas de função, e possui uma alternativa que deriva no não-terminal `mpi_call`, cujas alternativas derivam por sua vez algumas das chamadas MPI. Essas alternativas repassam os argumentos das funções MPI aos modelos do ANTLR que geram o código de saída. Esses argumentos são reconhecidos de forma posicional, cada um deles é um não-terminal `assignment_expression` (a lista de parâmetros em uma chamada de função é definida como uma série de `assignment_expression`s, separados por vírgula). Na listagem abaixo, as longas listas de parâmetros das funções de comunicação são abreviadas como ".. args .." para facilitar a leitura.

Listagem 6.1: Reconhecimento das chamadas de função MPI

```
postfix_expression
    : mpi_call
    | primary_expression ( ... | '(' ')'
        | '(' argument_expression_list ')'
        | '++'
        | ... )*
    ;

mpi_call
    : 'MPI_Comm_size' '(' a=assignment_expression ',' b=assignment_expression ')'
          -> m2pi_comm_size(comm={$a.text},address={$b.text})
    | 'MPI_Comm_rank' '(' a=assignment_expression ',' b=assignment_expression ')'
          -> m2pi_comm_rank(comm={$a.text},address={$b.text})
    | 'MPI_Send' '(' .. args .. ')' -> m2pi_send(.. args ..)
    | 'MPI_Recv' '(' .. args .. ')' -> m2pi_receive(.. args ..)
    | 'MPI_Bcast' '(' .. args .. ')' -> m2pi_bcast(.. args ..)
    ;
```

Os modelos nestes casos fazem muito pouco, apenas traduzem o nome da chamada de função e, nas três funções de comunicação, agrupam os argumentos de tipo e número de elementos a serem transmitidos em um parâmetro só, a quantidade de memória a ser transmitida. O modelo `m2pi_bcast`, por exemplo, é definido assim:

```
m2pi_bcast(buf, count, type, root, comm) ::=
    "M2PI_Bcast(<buf>, ((<count>) * sizeof(<type>)), <root>, <comm>)"
```

Como mostra a listagem 6.2, as funções `MPI_Init` e `MPI_Finalize` são um caso especial. Elas não são associadas a modelos, pois o código de inicialização e finalização do M2PI é gerado em outro ponto da gramática. Além disso, elas são tratadas como instruções (*statements*), e não como expressões, pois é assumido neste trabalho que elas serão usadas assim, sem retorno de valor. Essa funções delimitam o que o M2PI considera como código que será executado por cada *thread*, e por isso são feitas algumas anotações sobre as posições dos *tokens* encontrados durante o reconhecimento delas. Essas posições serão utilizadas posteriormente, como é explicado na seção 6.2.1.

Listagem 6.2: Reconhecimento das chamadas `MPI_Init` e `MPI_Recv`

```
statement
    : compound_statement
    | iteration_statement
    | jump_statement
        | ...
    | mpi_statement
    ;

mpi_statement
    : b='MPI_Init' '(' argument_expression_list ')' e=';' {
        mpi_init_begin = b.getTokenIndex();
        mpi_init_end = e.getTokenIndex();
        }
    | b='MPI_Finalize' '(' ')' ';' {mpi_finalize_begin = b.getTokenIndex();}
    ;
```

6.2 Primeiro programa: Hello World em MPI

A listagem 6.3 mostra um programa *Hello World* em MPI. A transformação deste código foi o primeiro alvo a ser alcançado na implementação deste trabalho, pois é o código MPI mais simples possível, mas mesmo assim já exige as funções mais básicas do compilador M2PI e de sua biblioteca de execução, a saber: a criação das *threads* e extração do código que será executado por elas; o comunicador global (`MPI_COMM_WORLD`); e a atribuição de números (*ranks*) às *threads*.

Listagem 6.3: Hello World usando MPI

```
1  int main(int argc, char* argv[]) {
2      int p, r ;
3      MPI_Init(&argc, &argv);
4      MPI_Comm_size(MPI_COMM_WORLD, &p);
5      MPI_Comm_rank(MPI_COMM_WORLD, &r);
6      printf("Oi pessoal, do processo %d dentro de %d!\n", r, p) ;
7      MPI_Finalize();
8      return 0 ;
9  }
```

As subseções abaixo apresentam a implementação destas funções básicas. O apêndice A exibe o código transformado completo do Hello-World MPI.

6.2.1 *Threads* em lugar de processos

Para disparar *threads* ao invés de processos, o código C original deve ser analisado, de forma a identificar e separar o trecho de código que cada *thread* deve executar (linhas 4-6 no programa da listagem 6.3), e inserir no seu lugar o código que dispara as N *threads* (uma informação conhecida só em tempo de execução), entre outras tarefas de inicialização e finalização.

Este trabalho assume que todo o código da função `main` original reside entre as chamadas `MPI_Init` e `MPI_Finalize`, a exceção das declarações de variáveis. Assim, o código que vem antes do `MPI_Init` é aproveitado pelo M2PI, pois normalmente conterá declarações de variáveis, mas qualquer código que venha depois da `MPI_Finalize` é completamente descartado. Esta decisão é uma simplificação

bastante grande, mas está apoiada sobre práticas comuns de programação MPI – onde se observa que os programas normalmente se encaixam nesta estrutura – e também no texto que define o padrão. O texto não especifica completamente o que deve acontecer fora do limite definido por estas chamadas (GROPP; LUSK; SKJELLUM, 1994), deixando esta decisão para a implementação[3].

A identificação e separação do código a ser executado por cada *thread* é feita pelo *parser* C modificado usado com o ANTLR, na regra que dita a sintaxe de definição de funções. A listagem 6.4 mostra essa regra, levemente simplificada para facilitar a compreensão[4].

Listagem 6.4: Indentificação do código das threads

```
function_definition :
   declaration_specifiers ? { input.LT(1).getText().equals("main") }? declarator
  { main_block_begin = input.LT(1).getTokenIndex(); }
  compound_statement
  {
    run_decl = input.toString(main_block_begin + 1, mpi_init_begin − 1);
    run_body = input.toString(mpi_init_end + 1, mpi_finalize_begin − 1);
  } −> m2pi_main(run_decl={run_decl}, run_body={run_body})
|  declaration_specifiers ? declarator compound_statement
;
```

As duas alternativas da regra possuem a mesma estrutura sintática, mas a primeira alternativa é especializada através de um predicado semântico, sendo ativada somente quando a função tem o nome de "main". Nesse caso, a primeira ação tomada marca o início do bloco da função `main`, logo antes do não-terminal `compound_statement`. A segunda ação tomada, após a interpretação do `compound_statement`, define dois blocos de texto, o `run_decl` (declarações de variáveis e mais tudo que está antes do `MPI_Init`) e o `run_body` (tudo entre o `MPI_Init` e o `MPI_Finalize`), e os passa para o modelo `m2pi_main`, que gera o novo corpo principal do programa. Os valores de `mpi_init_begin`, `mpi_init_end`, e `mpi_finalize_begin` são definidos nas regras que casam essas chamadas MPI, como mostrado na seção 6.1.

O código gerado pelo modelo `m2pi_main` inclui uma nova função `main` para o programa transformado (exibida na listagem 6.5). Essa nova função de início é igual para todo programa processado pelo M2PI, e realiza as seguintes tarefas:

1. Inicialização de estruturas necessárias, tais como comunicadores, filas de comunicação, etc., realizadas pelo `M2PI_Init`.

2. Leitura do número de *threads* que devem ser criadas a partir do parâmetro "-np" passado na linha de comando, alterando o argc e o argv para remover esse parâmetro já lido (também feito pelo `M2PI_Init`).

3. Criação das *threads*. A função que contém o código da *thread* é a `m2pi_run`, a qual é repassada uma estrutura chamada `M2PI_ThreadSpec`, com informações necessárias a sua execução, como o *rank* da *thread* e os parâmetros `argc` e `argv` que ela deve conhecer.

[3] A *manpage* da função MPI_Init da distribuição MPICH, por exemplo, recomenda que o mínimo seja feito antes da sua chamada, e o mesmo depois do MPI_Finalize.

[4] A simplificação se limita à remoção das linhas que tratam do estilo alternativo de definição de funções em C, conhecido como o "estilo K&R" (Kernighan & Ritchie).

4. Junção (`pthread_join`) de todas as *threads* criadas, aguardando sua finalização.

Listagem 6.5: Função main de um programa transformado pelo M2PI

```
int main(int argc, char* argv[]) {
    int m2pi_np, m2pi_i;

    /* MPI_Init() */
    argc = M2PI_Init(argc, argv);
    m2pi_np = MPI_COMM_WORLD->group->size;

    pthread_t m2pi_threads[m2pi_np];
    M2PI_ThreadSpec m2pi_threadspecs[m2pi_np];

    for (m2pi_i = 0; m2pi_i < m2pi_np; m2pi_i++) {
        m2pi_threadspecs[m2pi_i].myrank = m2pi_i;
        m2pi_threadspecs[m2pi_i].argc = argc;
        m2pi_threadspecs[m2pi_i].argv = argv;
        pthread_create(&m2pi_threads[m2pi_i], NULL,
            m2pi_run, (void *) &(m2pi_threadspecs[m2pi_i]));
    }

    /* MPI_Finalize() */
    for (m2pi_i = 0; m2pi_i < m2pi_np; m2pi_i++) {
        pthread_join(m2pi_threads[m2pi_i], NULL);
    }
    M2PI_Finalize();
    return 0 ;
}
```

O código da função `m2pi_run` apenas armazena o seu argumento em uma variável TLS (ver seção 4.2.2) e chama a função `m2pi_main`, que contém o código transformado da função `main` original (o código destas funções pode ser visto no apêndice A).

6.2.2 *Ranks* e o comunicador global

Além das funções de inicialização e finalização, as únicas funções do MPI usadas no exemplo do Hello-World são a `MPI_Comm_size` e `MPI_Comm_rank`. O M2PI não tem suporte para a criação de comunicadores ou grupos, então existe apenas o comunicador global (`MPI_COMM_WORLD`), do qual participam todas as *threads*.

A estrutura de um comunicador do M2PI é mostrada na listagem 6.6. Ela contém o tamanho do comunicador, mais um ponteiro genérico e duas variáveis descritoras de barreiras, usadas pela função `M2PI_Bcast`, descrita na seção 6.5. Normalmente essa estrutura deveria ainda conter ao menos uma lista ou vetor de *ranks*, indicando quais *threads*/processos fazem parte deste comunicador, mas como existe apenas o comunicador global, isso não é necessário.

Listagem 6.6: Estrutura de um comunicador no M2PI

```
typedef struct M2PI_Comm_STR {
    int size;
    /* int *ranks;    nao e' necessario */
    void *bcast_buf;
    pthread_barrier_t bcast_buf_set;
```

```
    pthread_barrier_t  bcast_copy_done;
} *M2PI_Comm;
```

O comunicador global é uma variável global, inicializada pela função `M2PI_Init`, que descobre qual deve ser o seu tamanho. A função `M2PI_Comm_size` simplesmente acessa o atributo `size` do comunicador passado. Já a função `M2PI_Comm_rank` acessa a variável TLS `ThreadSpec`, que contém o número da *thread*.

6.3 Tratando variáveis globais e locais estáticas

Ao usar *threads* ao invés de processos, o M2PI preserva a semântica das variáveis globais e locais estáticas através do mecanismo *Thread Local Storage* (apresentado na seção 4.2.2). Com o TLS, basta inserir o modificador `__thread` na declaração da variável global/estática, que está passará a ter uma instância separada para cada *thread*, e o compilador cuidará de organizar o acesso a elas.

A inserção deste modificador é muito simples usando um recurso do ANTLR que permite a reescrita do fluxo de *tokens*. Quando é ativado este recurso, o objeto `input`, que contém o fluxo de *tokens* fornecidos pelo analisador léxico, passa a ser uma instância da classe `TokenRewriteStream`, que fornece métodos para alteração do fluxo.

A listagem 6.7 mostra o trecho da gramática que reconhece a declaração de variáveis, onde a ação tomada ao encontrar o especificador de tipo (char, int, float, etc.) insere a palavra "__thread" antes dele, se o escopo atual for global. `$Symbols` é um escopo do ANTLR (ver seção 5.3) para o qual foi definido um atributo `isGlobalScope`. Instâncias deste escopo vão sendo empilhadas e desempilhadas pelo *parser* gerado, a medida que as regras vão sendo processadas.

Listagem 6.7: Inserção da palavra-chave __thread na declaração de variáveis globais

```
declaration_specifiers :
  (   storage_class_specifier   | a=type_specifier {
            if  ($Symbols::isGlobalScope)
              ((TokenRewriteStream)input).insertBefore($a.start, " __thread ");
    } |  type_qualifier
  )+
  ;
```

A listagem 6.8 mostra os trechos da gramática responsáveis pelo "rastreamento" do escopo global, incluindo a declaração do escopo `$Symbols` e sua utilização. Instâncias deste escopo são criadas em dois pontos da grámatica: na regra inicial (a `translation_unit`), onde `isGlobalScope` é assinalado como verdadeiro; e na regra que casa definições de função, onde `isGlobalScope` é assinalado como falso.

Listagem 6.8: Rastreamento do escopo global na gramática do M2PI

```
scope Symbols {
    boolean isGlobalScope;
}

translation_unit
scope Symbols;
@init {
    $Symbols::isGlobalScope = true;
}
```

```
: external_declaration+
;

function_definition
scope Symbols;
@init {
    $Symbols::isGlobalScope = false;
}
: ... ;
```

Até aqui foi apresentada a solução de uso do TLS para ajuste das váriaveis globais. A listagem 6.9 mostra o trecho da gramática que trata do outro caso, o das variáveis locais estáticas. Esse trecho reconhece a palavra-chave `static`, e insere o `__thread` logo após ela, apenas quando o escopo não for global (para não repetir o que já é feito no tratamento das variáveis globais, apresentado anteriormente).

Listagem 6.9: Inserção da palavra-chave __thread na declaração de variáveis estáticas locais

```
storage_class_specifier
    : 'extern'
    | a='static' {
        // Coloca o __thread do TLS nas variaveis locais static
        if (!$Symbols::isGlobalScope)
            ((TokenRewriteStream)input).insertAfter($a, " __thread ");
    }
    | 'auto'
    | 'register'
    ;
```

Para testar a inserção do TLS pelo M2PI, foi definido um programa muito simples, que incrementa três contadores um mesmo número fixo e grande de vezes. Um dos contadores é local à função `main`, outro é global, e o terceiro é estático local dentro de uma outra função chamada `inc`. A listagem 6.10 mostra um trecho deste programa. O código da função `main`, que não é mostrada, apenas incrementa os contadores local, global, e estático local (chamando a função `inc`) `N` vezes, e então os imprime.

Se o TLS não fosse utilizado, o valor dos contadores não seria o mesmo ao final da execução deste programa. Não só pelo fato de que todas as *threads* estariam acessando as mesmas posições de memória dos contadores estático e global, mas também porque estariam fazendo isso sem nenhuma sincronização. Como os três incrementos não são uma unidade atômica, – nem sequer o incremento de uma única posição de memória é uma operação atômica para o processador – os resultados seriam completamente inconsistentes[5], como os mostrados na saída abaixo (exemplo com três *threads*).

```
Rank 0: local 1000000, global 1262837, static 1283975
Rank 1: local 1000000, global 1930452, static 1962467
Rank 2: local 1000000, global 2662394, static 2694409
```

A listagem 6.11 mostra o mesmo trecho após a transformação com o M2PI. O apêndice B mostra o código completo do programa de teste transformado.

[5]Caso os incrementos fossem atômicos, ou então caso as *threads* rodassem em ordem, uma após a outra, os valores dos contadores global e estático seriam iguais e múltiplos de N.

Listagem 6.10: Trecho de programa usando variáveis globais e estáticas

```
int N = 1000000; /* contar ate' N */
int inc(void) {
    int dummy;    /* so' para ver que nao sera' alterado com __thread */
    static int sCounter = 0;
    return ++sCounter;
}
int gCounter;       /* contador global */
```

Listagem 6.11: Variáveis globais e estáticas ajustadas para usar TLS

```
__thread int N = 1000000; /* contar ate' N */
int inc(void) {
    int dummy;    /* so' para ver que nao sera' alterado com __thread */
    static __thread int sCounter = 0;
    return ++sCounter;
}
__thread int gCounter;       /* contador global */
```

Cabe observar aqui que a colocação de uma declaração de variável antes da função `inc`, e outra depois, foi proposital, para verificar que o rastreamento do escopo está correto. Ou seja: global até a declaração da função, local dentro da definição desta (a variável `dummy` não é alterada), e novamente global após o término da definição da função.

O ajuste das variáveis globais e locais estáticas apresentado nesta seção, com o uso do recurso TLS, é bastante simples, e poderia ser implementado por um programa que fizesse procura e substituição de texto, possivelmente usando expressões regulares, sem grandes dificuldades (apenas tendo que definir quando o escopo é global ou não). Mas a utilização do ANTLR, modificando a gramática C usada para reconhecer o código fonte, tornou a solução bastante trivial, elegante, e robusta.

6.4 Comunicação ponto-a-ponto: MPI_Send e MPI_Receive

Esta seção apresenta a implementação das funções de envio e recebimento na biblioteca de execução do M2PI, que permitiram a implementação de dois programas para testes de trocas de mensagens, um pingue-pongue, e outro mestre-escravo. Sobre estes programas já foi possível realizar algumas medições de desempenho, que são apresentadas no capítulo 7.

Em comparação às funções `MPI_Send` e `MPI_Recv` do MPI, suas análogas neste trabalho servem-se de algumas simplificações, como a ausência do suporte a *tags*, e também a impossibilidade de fazer um recebimento de mensagem de qualquer origem (o `MPI_ANY_SOURCE`): sempre deve ser especificado um *rank* de origem/destino específico nas chamadas destas funções.

O comportamento definido para estas operações é bloqueante para o recebimento, e não-bloqueante para o envio. Foi considerado que, já que apenas um par de funções de envio e recebimento seria implementado neste trabalho, este seria o comportamento mais intuitivo para as duas funções.

As seções a seguir apresentam as estruturas desenhadas para suportar a comunicação entre as *threads*, e a utilização destas estruturas em dois casos distintos de comunicação: um quando o envio é disparado antes do recebimento, e o outro quando o recebimento é disparado antes do envio.

6.4.1 Infra-estrutura de comunicação

As estruturas utilizadas para fazer a comunicação no M2PI foram inspiradas nas estruturas usadas no TMPI, ainda que de forma muito simplificada. Na execução de um programa com N *threads*, são criados N x N **canais** de comunicação $C[i, j]$, onde i é o número da *thread* que envia a mensagem, e j o da que recebe.

Cada canal é composto de duas filas, uma de envio e outra de recebimento. Essas filas armazenam descritores (*handlers*) que contém alguns metadados sobre a comunicação sendo realizada. Na fila de envio, apenas a *thread* emissora insere descritores, e apenas a *thread* receptora os retira. Na fila de recebimento é o inverso. Dessa forma, evita-se que uma *thread* tente retirar um descritor que ela mesmo inseriu, sem a necessidade de diferenciar descritores de envio e recebimento, ou identificá-los de outra forma.

O descritor é assim composto:

- Um ponteiro para o *buffer* que deve fornecer ou receber os dados a serem copiados (que é um dos parâmetros das funções de comunicação).
- Uma variável de condição, que é usada quando uma *thread* fica bloqueada na operação de recebimento, para avisá-la de que pode prosseguir sua execução.
- Uma variável de valor verdadeiro/falso associada à variável de condição descrita acima, que indica se a *thread* sinalizada deve de fato prosseguir sua execução.

Quando uma das *threads* precisar acessar uma das filas de um canal, ela faz o *lock* do canal inteiro (aloca o canal), usando um *mutex* que é um atributo do canal. Isso simplifica a interação entre o emissor e receptor enquanto estes manipulam as filas, como exposto na seção seguinte. Uma otimização que faça esta alocação ser mais granular é com certeza interessante, mas tal possibilidade foi deixada para um desenvolvimento futuro do trabalho.

6.4.2 Envio e recebimento de mensagens

A comunicação ponto-a-ponto no M2PI é implementada por funções análogas às originais do MPI, porém com uma lista de parâmetros simplificada, como podemos ver pelas suas assinaturas abaixo:

```
int M2PI_Send(void *buf, int sz, int dst);
int M2PI_Recv(void *buf, int sz, int src);
```

Portanto, cada uma delas especifica uma aŕea de memória que é a origem/destino dos dados, o tamanho dos dados que devem ser copiados, e o número da *thread* de origem ou destino. As duas retornam um valor inteiro que indica se a operação foi bem sucedida, ou um código de erro. O código que as implementa trata basicamente de dois cenários de comunicação distintos, dependentes da ordem em que o envio e o recebimento são executados pelo processador.

Quando o `M2PI_Receive` é executado antes do `M2PI_Send`, como a operação de recebimento é bloqueante, esta pode aguardar a instrução de envio, e a cópia dos dados pode ser feita diretamente do *buffer* de origem para o de destino. Esta situação é mostrada na figura 6.1, que apresenta de forma resumida a comunicação

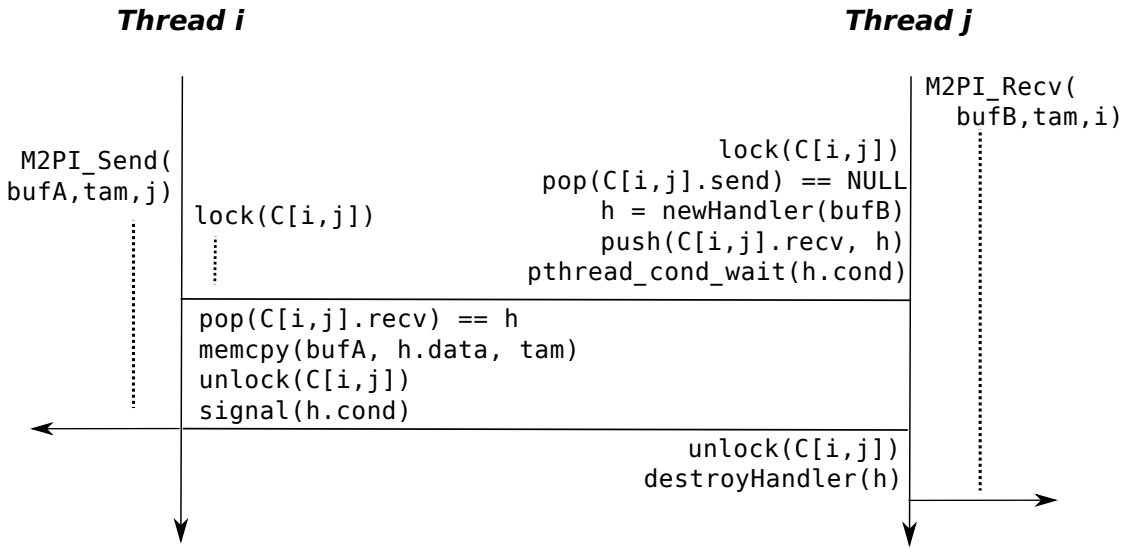

Figura 6.1: Recebimento iniciado antes do envio

entre duas *threads* i e j. Para facilitar a compreensão e esclarecer a notação utilizada na figura (uma forma resumida e simplificada do código C executado pela rotina), são detalhadas aqui as operações executadas:

1. A *thread* j – a receptora – aloca o canal $C[i, j]$.
2. Ela verifica se o emissor (i) já postou um descritor de cópia na fila de envio do canal, o que neste caso é falso (`==NULL`).
3. j cria um descritor h apontando para o seu *buffer* de recepção, e o insere na fila de recepção adequada ($C[i, j].recv$).
4. A *thread* j então aguarda que o emissor faça a cópia dos dados, esperando que a variável de condição do descritor que ela criou seja sinalizada. A chamada à função `pthread_cond_wait` desaloca o *mutex* associado ao canal.
5. A *thread* i agora pode alocar o canal $C[i, j]$, e remover o mesmo descritor h da fila $C[i, j].recv$.
6. Ela copia os dados para o espaço apontado no descritor (`h.data`), desaloca o canal, e sinaliza a variável de condição `h.cond`. A execução do `MPI_Send` termina aqui, retornando uma constante que indica sucesso da operação.
7. A *thread* j acorda, desaloca o canal – que foi automaticamente realocado para ela, mas que não é necessário –, destrói o descritor h e retorna também a constante de sucesso na operação.

O outro cenário de comunicação se dá quando o procedimento de envio é iniciado antes do procedimento de recebimento. Este caso, bastante similar ao anterior, é exposto na figura 6.2, usando a mesma notação, por isso poupa-se o detalhamento passo-a-passo. A diferença aqui, é que a a operação de envio não é bloqueante, e portanto não pode aguardar pela função de recebimento. Isso exige a cópia dos dados

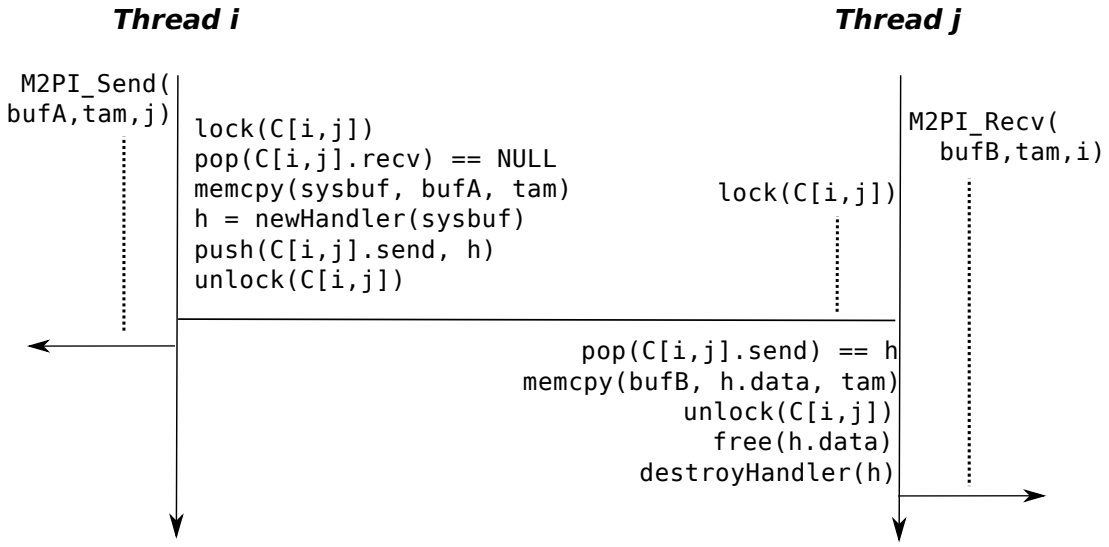

Figura 6.2: Envio iniciado antes do recebimento

a serem enviados para um *buffer* de sistema (denominado `sysbuf` na figura), sendo esta cópia apontada no descritor inserido na fila $C[i, j]$. Assim a função `M2PI_Send` pode retornar e o programa pode sobreescrever os dados originais.

Além disso, neste segundo cenário não é necessário o uso da variável de condição, pois quando a função de recebimento consegue alocar o canal, os dados já estão prontos para a transferência, que ela então realiza. Por fim, além de destruir o descritor após seu uso, a *thread* receptora libera o espaço que foi alocado para o `sysbuf`, já que este não é mais necessário.

É claro que a maneira com que o *buffer* de sistema está sendo implementado aqui é ingênua, fazendo uma chamada de alocação de memória cada vez que um *buffer* de sistema é necessário, e depois as correspondentes desalocações. Um esquema de alocação mais adequado tomaria um bloco maior de memória inicialmente, e cada *thread* utilizaria pedaços deste bloco conforme a necessidade, reusando os pedaços após a transferência ser completada. Porém essa solução incorreria em uma série de problemas que, na forma atual, são tratados pelas rotinas do próprio sistema operacional, tais como o controle da locação dos pedaços, o tratamento da fragmentação, e ainda a sincronização do acesso ao *buffer*. Foi decidido então deixar esta otimização para um possível desenvolvimento futuro.

6.5 Comunicação coletiva: MPI_Bcast

Como exemplo de comunicação coletiva, foi adicionada a biblioteca de execução do M2PI uma função análoga à `MPI_Bcast`, a mais simples das funções desta categoria no MPI (ver seção 4.1.4). A assinatura da `M2PI_Bcast` é mostrada abaixo:

```
int M2PI_Bcast(void* buf, int sz, int root, M2PI_Comm comm)
```

A semântica dessa função é a de um ponto de sincronização entre todos os processos que fazem parte de um comunicador, pois todos devem fazer a chamada à função com os mesmos parâmetros `root` e `comm`, e todos devem esperar que todas as

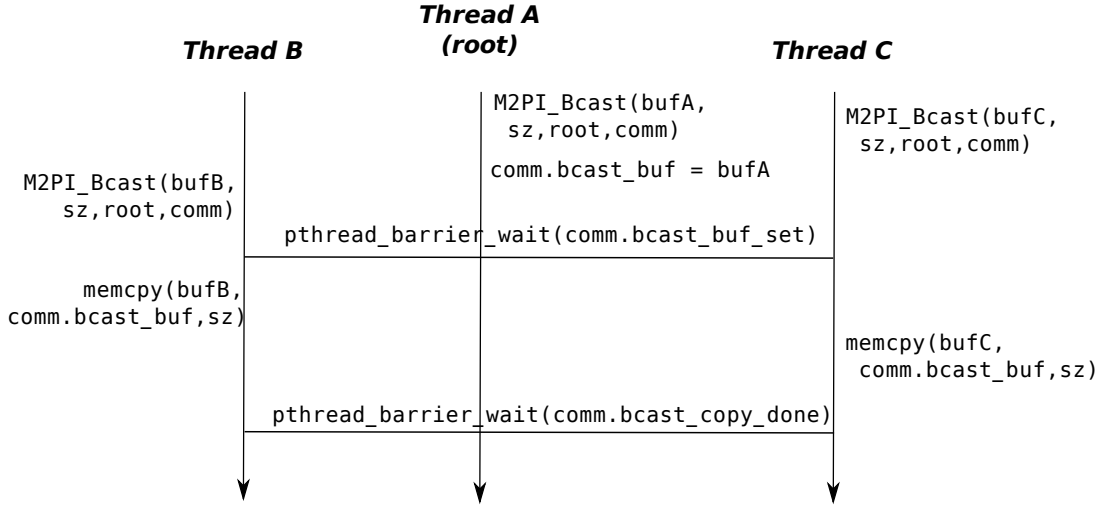

Figura 6.3: Função MPI_Bcast sendo executada por três *threads*

cópias terminem antes de prosseguir. Isso torna sua implementação muito simples utilizando *barreiras*, como mostra a figura 6.3.

São apenas duas barreiras, uma antes de realizar a cópia, para que a *thread* `root` copie o ponteiro do seu *buffer* para a variável `bcast_buf` do comunicador, e outra após a cópia, que serve para garantir que nenhuma outra operação de *broadcast* inicie antes de todas *threads* terminarem a operação atual.

O programa utilizado para verificar o funcionamento desta implementação faz a contagem de números primos menores que N utilizando o algoritmo "Crivo de Eratóstenes". O capítulo 7 apresenta o algoritmo e mostra o resultado sobre o desempenho deste programa com o M2PI.

6.6 Reflexão sobre a implementação

É possível dizer que pouco foi implementado para aproveitar a compilação do código e para justificar o uso de um ferramental complexo e poderoso como o ANTLR. Certamente, para a restruturação do código apresentada na seção 6.2, e os ajustes nas variáveis com alocação estática (seção 6.3), essa abordagem é mais poderosa, trazendo simplicidade e permitindo a expansão do que foi feito para situações mais complexas. Mas não foi só por estas vantagens que a abordagem da compilação foi escolhida para este trabalho. Ela foi escolhida com o objetivo de obter para ganhos maiores, com otimizações baseadas em detalhes que só a análise do código fonte em tempo de compilação pudesse revelar.

No entanto, as oportunidades para alcançar essas otimizações parecem todas apontar para tarefas complexas demais para o escopo deste trabalho. O capítulo de considerações finais apresenta algumas alternativas que podem ser estudadas para alcancar estes benefícios.

7 AVALIAÇÃO DE DESEMPENHO

O desempenho do protótipo de compilador construído neste trabalho foi avaliado pela mensuração de tempos de execução de um conjunto de programas MPI de teste, comparando o M2PI ao MPICH[1] e ao TMPI[2], duas outras implementações do MPI que foram apresentadas na seção 2.3. Foram observadas as alterações do tempo de execução conforme a variação da quantidade e tamanho das mensagens, e do número de *threads*/processos em execução.

Os resultados se mostraram bastante interessantes. O M2PI apresenta os melhores tempos em alguns casos, e seu desempenho é em geral bastante similar ao do TMPI, que tem uma arquitetura semelhante. A comparação com o MPICH se divide em duas situações, pois foram utilizadas duas versões dessa distribuição: uma configurada para utilizar comunicação através de memória compartilhada (mas ainda assim instanciando processos e não *threads*), que teve um desempenho muito bom, com vantagens expressivas em alguns casos; e outra usando comunicação via *sockets*, com um desempenho em geral bastante inferior aos demais. Fica claro que a implementação do MPICH usando *sockets* sofre bastante em comparação às outras testadas, e por isso ela foi testada apenas em um dos programas de teste, o Ping-Pong. Os outros testes focaram apenas as implementações que usam memória compartilhada.

O restante deste capítulo apresenta a metodologia empregada e os ambientes de testes utilizados, na seção 7.1; os programas testados, na seção 7.2; e os resultados obtidos na seção 7.3.

Em prol da clareza do texto, neste capítulo o termo **processo** é usado no sentido de "unidade de execução" do MPI, se referindo tanto a um processo no sistema operacional, quanto a uma *thread* de sistema, quando for conveniente se referir aos dois casos de forma generalizada.

7.1 Ambiente e metodologia de testes

Dois ambientes distintos, descritos a seguir, foram utilizados para a execução dos testes. Nos gráficos e no restante deste relatório, os sistemas serão referenciados pela identificação linux32-2c e linux64-8c.

linux32-2c: Um computador pessoal, com 2 gigabytes de memória RAM, e um processador dual Intel Pentium Dual Core E2180, com frequência de 2 Ghz e

[1] http://www.mcs.anl.gov/research/projects/mpich2/
[2] http://www.cs.ucsb.edu/projects/tmpi/

memória cache nível 2 de 1 MB. O sistema operacional é um Ubuntu GNU/-Linux 8.04 32 bits, com kernel 2.6.24, e glibc versão 2.7.

O compilador C utilizado foi o do projeto GNU (GCC), versão 4.2.4. A versão do MPICH foi a 1.2.7, instalada a partir dos pacotes binários distribuídos com o SO, nas versões com comunicação via *sockets* e via memória compartilhada.

linux64-8c: Um computador servidor, da marca Dell, com 16 gigabytes de memória RAM, e dois processadores *quad-core* Intel Xeon E5310, com frequência de 1.60 Ghz, e memória cache nível 2 de 8 MB (2x4) por processador. O sistema operacional é openSUSE 10.3 64 bits, com kernel 2.6.18 e glibc 2.6.1.

O MPICH utilizado neste ambiente foi o distribuído pela Intel, versão 2-1.0.6, compilado manualmente e configurado para comunicação via memória compartilhada.

O compilador C utilizado para a compilação do testes com o MPICH foi o Intel ICC versão 10.1. Para a compilação dos testes com o M2PI e o TMPI, foi utilizado o GCC versão 4.2.1.

Os programas de teste foram compilados em cada um dos ambientes, sem utilizar quaisquer parâmetros de otimização. O mesmo vale para a compilação do TMPI, e da biblioteca de execução do M2PI. No entanto, não foi possível compilar o TMPI na máquina linux64-8c, por problemas de versão do compilador C e dependências relacionadas. Assim foi utilizada nesse ambiente a mesma versão do TMPI compilada para a linux32-2c, e os programas de teste foram compilados em modo de 32 bits (já que não é suportado ligar código de 64 bits a bibliotecas de 32 bits).

Cada tomada de tempo mostrada nos gráficos deste capítulo é a média dos tempos de cinco execuções consecutivas dos programas de teste. Destas execuções também são coletados os tempos mínimo e máximo, que são exibidos nos gráficos como barras verticais – mas na maior parte dos casos os tempos de execução mostram-se bastante estáveis, fazendo com que estas barras se assemelhem a pontos sobre as linhas dos gráficos. Os tempos foram tomados usando o utilitário `time` dos sistemas operacionais onde foram realizados os testes, com precisão na ordem dos milisegundos.

Além disso, foi desenvolvida uma mini-biblioteca de funções auxiliares para a execução dos testes, que são disparados por *scripts* que usam estas funções. Elas disparam as execuções, tomam os tempos, calculam os valores necessários e armazenam os resultados. Dessa forma a realização dos testes ficou mais prática, e os resultados mais confiáveis.

7.2 Programas testados

Os programas testados foram o Ping-Pong, o Mestre-Escravo, e o Crivo de Eratóstenes (*Sieve of Eratosthenes*), que por vezes será referenciado pela abreviatura CE. Estes são exemplos comuns de programação MPI, e apesar de serem programas simples, fornecem uma base para a avaliação de desempenho do M2PI.

No **Ping-Pong** dois processos MPI trocam mensagens como no jogo de mesa, um processo manda uma mensagem e o outro a responde, com o mesmo conteúdo. Versões usadas para testar a correção da implementação comparam o conteúdo retornado para garantir que é o mesmo que foi enviado. Na versão usada para as

tomadas de tempo apresentadas aqui, esse teste foi removido, para medir apenas uma sequência contínua e ininterrupta de transmissão de mensagens.

A mensagem transmitida é um vetor de números de ponto flutuante de precisão dupla (8 bytes), preenchido com dados aleatórios, que são calculados antes de começarem as transmissões. O tamanho do vetor é parametrizável, bem como o número de rodadas (ping + pong) que devem ser executadas. Os resultados são apresentados no gráfico 7.1, comentados na seção seguinte.

O **Mestre-Escravo** mantém um processo gerente (o mestre) que distribui tarefas a processos operários (os escravos). As tarefas são números de ponto flutuante de precisão simples (4 bytes), lidos de um arquivo que contém uma série destes, gerados aleatoriamente em uma faixa de 0 a 500. O escravo recebe um vetor destes números, calcula a raiz quadrada de cada item, soma os resultados, responde essa soma ao mestre, e aguarda uma nova tarefa.

Normalmente este programa seria implementado de forma que, ao receber a resposta de qualquer escravo, não importando seu *rank*, o mestre repasse uma nova tarefa a ele, até todas tarefas serem completadas. Como o M2PI não tem suporte ao recebimento de uma origem qualquer (o `MPI_ANY_SOURCE` da norma MPI), uma adaptação teve de ser feita nesse algoritmo: o mestre passa uma tarefa para cada escravo, e então espera a resposta de cada um deles, na mesma ordem. Quando não restam mais tarefas suficientes para ocupar todos os escravos, o mestre informa o término para os escravos e todos processos se encerram. Antes de encerrar, o mestre imprime a soma de todas as respostas recebidas, ou seja, a soma das raízes quadradas de todos os números utilizados como tarefas. Esse resultado é conferido para verificar que o M2PI calcula o mesmo resultado que as outras implementações testadas.

O arquivo de entrada com os números e o tamanho do vetor usado como mensagem são parâmetros do programa, e definem a quantidade e o tamanho das tarefas. O número de escravos é dado pela quantidade de processos disparados, descontado o mestre. Os resultados são apresentados nos gráficos 7.5 e 7.2, comentados na seção seguinte.

O algoritmo **Crivo de Eratóstenes** calcula os números primos menores que um dado valor n, descobrindo e marcando todos números compostos entre 2 e n. O pseudo-código do algoritmo, em uma versão puramente sequencial, é o seguinte:

1. Crie uma lista de números naturais de 2 a n, nenhum deles marcado.
2. Defina k como 2, o primeiro número não marcado da lista.
3. Repita até $k^2 > n$:
   (a) Marque todos os múltiplos de k entre k^2 e n
   (b) Encontre o primeiro número desmarcado maior que k, e redefina k como este novo valor.
4. Todos os números desmarcados são primos.

O programa de teste CE executa este algoritmo e então conta quantos números primos foram encontrados, imprimindo o valor na saída. A implementação utilizada é baseada na que é apresentada no capítulo 5 de QUINN (2004), que dá mais detalhes sobre o algoritmo e sua implementação como programa paralelo.

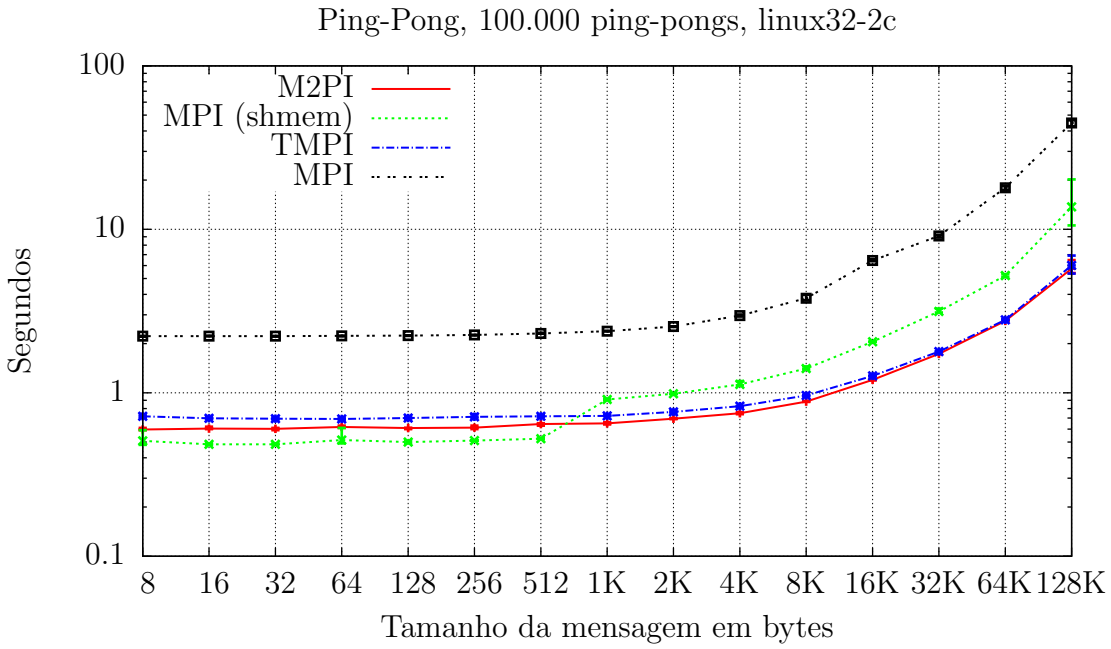

Figura 7.1: Ping-Pong variando o tamanho das mensagens

Essa implementação é também uma arquitetura mestre-escravo, mas desta vez o mestre também realiza parte do trabalho. A lista criada no primeiro passo do pseudo-código é dividida em p listas de $\lfloor p/n \rfloor$ ou $\lceil p/n \rceil$ elementos, uma para cada um dos p processos, que marca os múltiplos de k na sua lista. Além disso, é assumido que todos elementos que poderão assumir o valor de k estão na lista do primeiro processo, que é então o único responsável por encontrar o próximo valor de k e difundí-lo para todos (usando o `MPI_Bcast`, por exemplo). Essa hipótese é verdadeira para valores grandes de n, quando $\lfloor n/p \rfloor > \sqrt{n}$, pois k não pode ser maior que $\sqrt{n}$ (todos os valores de n usados nos testes se encaixam nesse caso). Quando o laço termina, cada processo conta os números primos em sua lista e os escravos informam seus valores ao processo mestre, que os soma junto com o seu contador[3].

O único parâmetro recebido pelo programa de teste do Crivo de Eratóstenes é o valor de n. Os resultados são apresentados nos gráficos 7.3 e 7.4, comentados na seção seguinte.

7.3 Resultados

Esta seção apresenta e comenta os gráficos obtidos a partir das tomadas de tempo realizadas com os programas de teste apresentados na seção anterior. Cada gráfico é comentado individualmente, e ao final é feita uma breve reflexão sobre os resultados. Na legenda dos gráficos, a versão do MPICH que utiliza memória compartilhada é identificada como `MPI (shmem)`, enquanto a versão que utilizada *sockets* é identificada apenas como `MPI`.

A figura 7.1 mostra o desempenho do ping-pong executando 100 mil rodadas (2 mensagens: o ping e o pong), variando o tamanho da mensagem de 8 bytes a 64

[3]Essa operação normalmente seria feita no MPI com uma **redução** (`MPI_Reduce`), mas como o M2PI não implementa esta função, ela feita manualmente com `MPI_Send` e `MPI_Recv`.

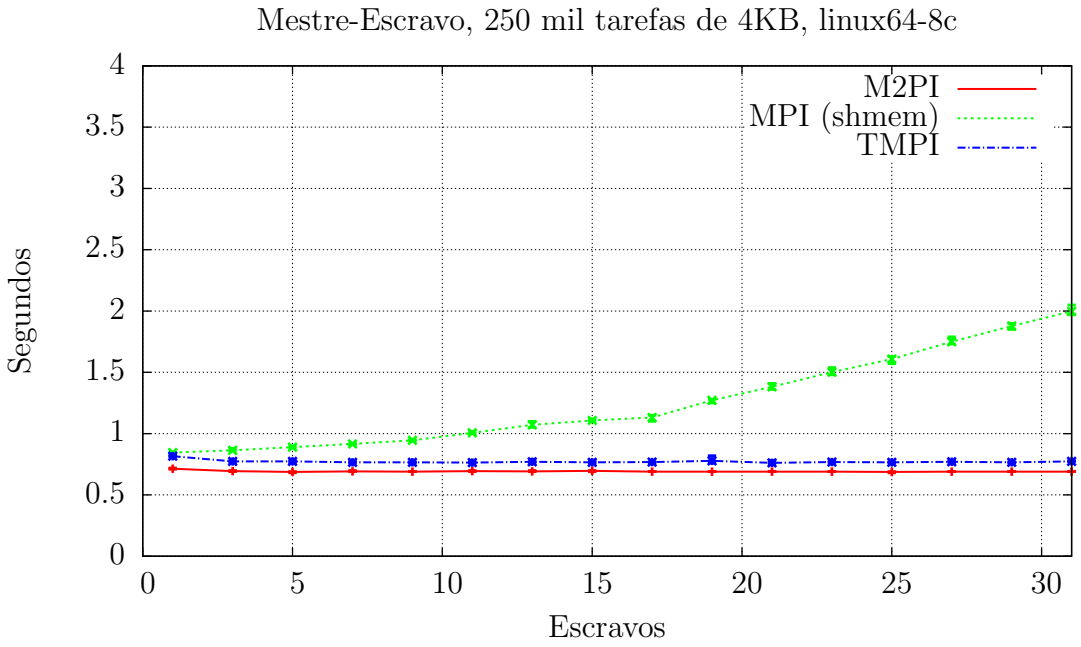

Figura 7.2: Mestre-escravo variando o número de processos

Kbytes. Os dois eixos são apresentados em escala logarítmica. O início do gráfico, até aproximadamente mensagens de 1KB, é quase totalmente dominado pela latência, que é notadamente maior para o MPI em todo o gráfico. Neste ponto nota-se também uma queda de desempenho significativa do MPICH. A menor latência fica com o M2PI e o TMPI, que têm desempenho praticamente idêntico.

A figura 7.2 mostra o desempenho do mestre-escravo executando 250 mil tarefas de 4 KBytes, variando o número de escravos, de 1 a 31 (contando o mestre, 2 a 32 processos/*threads*). A implementação do MPICH, baseada em processos, mostra uma queda de desempenho a medida que o número de processos aumenta. As implementações baseadas em *threads* mantêm-se estáveis, com uma pequena vantagem para o M2PI.

A figura 7.3 mostra o desempenho do Crivo de Eratóstenes com 8 processos contando os números primos menores que 2^n com n variando entre 20 (1.048.576) e 28 (268.435.456). O dois eixos são exibidos em escala logarítmica. Neste caso o M2PI se sai bastante melhor que o TMPI. O MPICH apresenta um tempo praticamente constante até n se tornar suficientemente grande, possivelmente devido ao seu tempo de inicialização maior. A direção da curva indica que o desempenho do MPICH deve ser superior aos outros para valores maiores de n maiores que 28.

A figura 7.4 mostra o desempenho do CE contando os números primos menores que 100 milhões, variando o número de processos (o TMPI não aparece neste gráfico, pois sua execução neste teste particular apresentou problema de falha de segmentação). O M2PI apresenta um comportamento bastante esperado, tendo ganho de desempenho até a utilização dos oito *cores* do sistema, e depois perdendo um pouco a medida que o número de *threads* supera o de unidades de processamento. Além disso, como a variação do número de processos MPI muda a distribuição dos números primos entre eles – podendo concentrar mais primos em um único processo – explica-se a oscilação dos tempos de execução vista ao longo do gráfico.

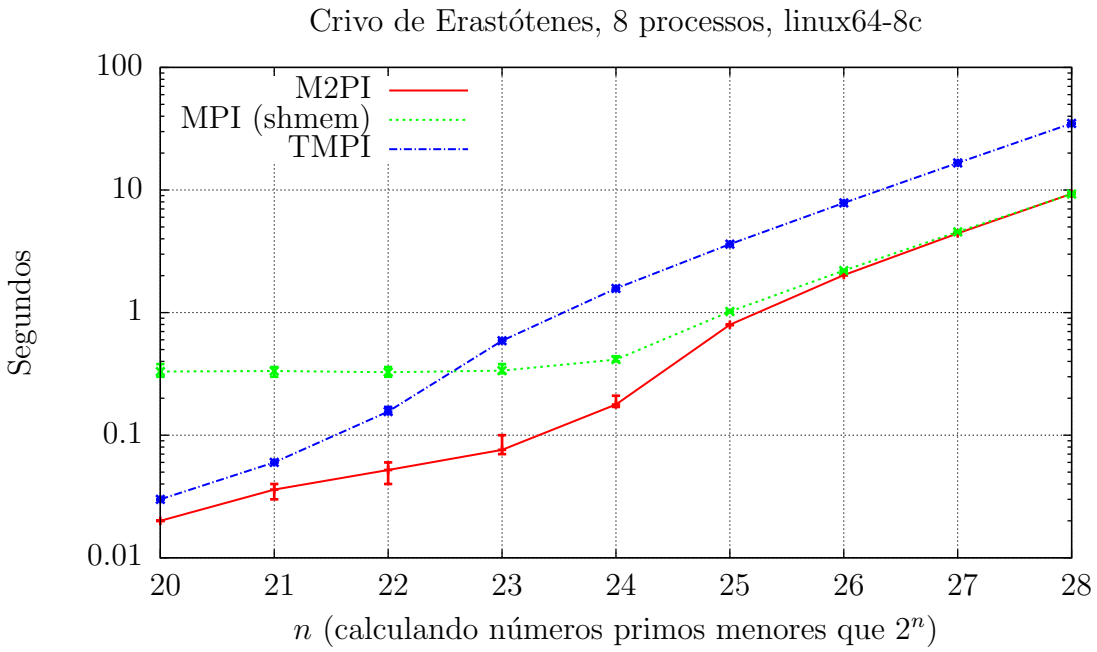

Figura 7.3: CE variando o número de mensagens

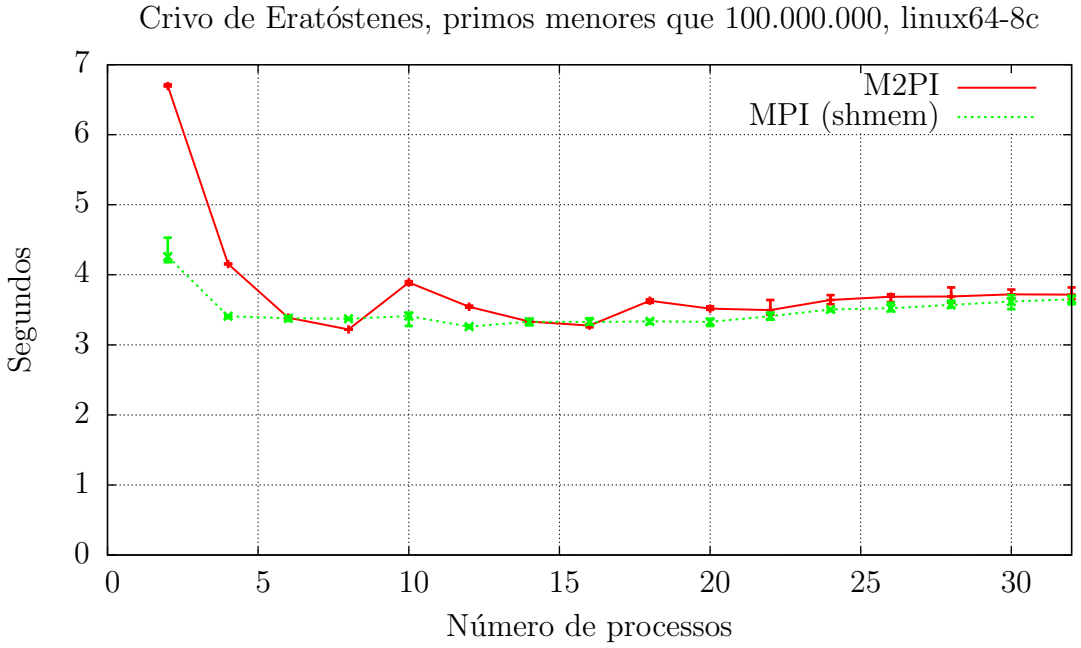

Figura 7.4: CE variando o número de processos

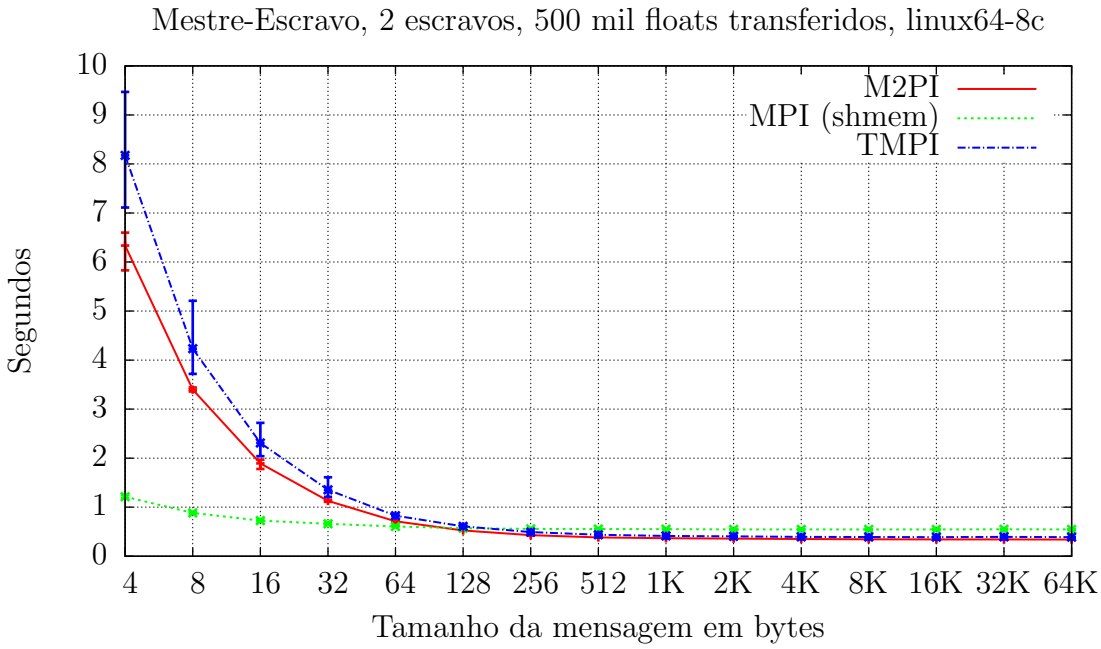

Figura 7.5: Mestre-escravo variando o tamanho da mensagem

Já o comportamento do MPICH neste teste não é exatamente o esperado: ele alcança rapidamente um desempenho muito próximo do seu ideal, usando só quatro processos, e mantém uma boa regularidade nos tempos de execução também. Além disso o desempenho não se deteriora tanto com o aumento do número de processos, como acontece no gráfico 7.2 comentado anteriormente. Um fator que deve explicar ao menos o primeiro ponto (a rápida saturação do desempenho), é que o compilador da Intel utilizado no teste com o MPICH faz por padrão algumas otimizações que o GCC faz apenas quando instruído para tal, como o desenrolamento de laços.

A figura 7.5 mostra o desempenho do mestre-escravo transferindo 500 mil números de ponto flutuante, em mensagens de tamanho crescente, de 8 bytes a 64 KBytes. Como é esperado, o desempenho melhora com o aumento do tamanho da mensagem, que privilegia a taxa de transferência da memória e diminui o custo de processamento de cada operação de envio e recebimento. Mais uma vez o desempenho do M2PI é bastante similar ao do TMPI, sendo um pouco melhor para tamanhos de mensagens muito pequenos. O que chama a atenção no gráfico, no entanto, é o excelente desempenho MPICH, mesmo para as mensagens muito pequenas. Isso se deve novamente à otimização do laço feita pelo ICC, que deve agrupar os dados e transferí-los em unidades de tamanho maior.

Reflexão sobre os resultados

Apesar da utilização de compiladores diferentes e das otimizações realizadas pelo compilador da Intel dificultarem a análise dos resultados, ainda assim o M2PI mostrou um desempenho competitivo com as duas outras implementações testadas. Este fato é bastante notável, dado que o MPICH é atualmente a mais importante implementação do MPI, e que o TMPI é um fruto de um trabalho bem mais longo, com vários artigos publicados a seu respeito.

Uma das conclusões feitas em TANG; SHEN; YANG (1999) sobre a utilização

de *threads* para a implementação dos "processos" MPI é de que essa alternativa apresenta vantagens expressivas em ambientes não dedicados, isto é, ambientes em que dois ou mais processos do programa MPI competem pela mesma unidade de processamento. Esta conclusão parece ser corroborada pelo gráfico 7.2, apesar de que essa perda de desempenho talvez seja mais relacionada ao custo adicional da inicialização dos processos do que a um sobrecusto na comunicação propriamente dita.

Em trabalhos futuros que continuem o desenvolvimento iniciado neste, seria interessante realizar testes com programas mais complexos, que forneçam cargas de trabalho mais realísticas do que as induzidas pelos programas mais simples que foram utilizados aqui.

8 CONSIDERAÇÕES FINAIS

O presente trabalho de conclusão buscou explorar a viabilidade da construção de um compilador que traduza programas escritos para utilizar a interface MPI em programas que usem *threads* POSIX, fazendo então toda sua comunicação por meio de memória compartilhada, considerando sua execução em máquinas multiprocessadas e *multicore*.

Muitos outros trabalhos já foram realizados estudando a comunicação através da memória e a utilização de *threads* ao invés de processos. O diferencial deste trabalho foi buscar as oportunidades oferecidas pela análise estática do código, e pelo uso de técnicas e ferramentas de construção de compiladores, que facilitem a transformação do código-fonte e – principalmente – que permitam otimizações que seriam impossíveis ou impraticáveis de serem realizadas em tempo de execução do programa.

A implementação de um protótipo de tal compilador foi realizada – recebendo o nome de M2PI – e demonstrou várias vantagens na utilização de tal abordagem na transformação de um programa C/MPI para C/Pthreads.

O uso de *threads* no lugar de processos, por exemplo, que sozinha já é uma otimização interessante, implica nas necessidades de reorganizar a estrutura do código fonte, e de restaurar a semântica de utilização das variáveis globais e das variáveis locais estáticas. Essas transformações seriam impossíveis caso fosse feita uma análise puramente léxica do código, mas é alcançada de forma razoavelmente simples por um *parser*, a ferramenta ideal para este trabalho.

A utilização da gramática da linguagem na qual foi escrito o programa que se deseja otimizar, permite que alterações no código fonte sejam dirigidas pela análise sintática deste. Isso reduz falsos positivos e falsos negativos no reconhecimento de oportunidades de otimização do código gerado.

Além disso, é possível durante a análise inspecionar e alterar aspectos semânticos do código fonte. Um pequeno exemplo disso foi a alteração realizada sobre um par de parâmetros passados às funções de comunicação do MPI, que indicam o número de elementos e o tipo de cada elemento, e que foram transformados num único parâmetro, indicando diretamente a quantidade de memória que deve ser transferida.

No entanto, utilizar o conhecimento do código-fonte para aumentar o desempenho do programa – através de otimizações feitas sobre a **comunicação** entre as *threads* – mostrou-se uma tarefa bastante desafiadora. As oportunidades vislumbradas tem viabilidade incerta, ou implementação muito complexa para o escopo deste trabalho, carecendo de pesquisa adicional.

Uma dessas oportunidades seria realizar a transferência de dados entre um par de funções de envio e recebimento de forma direta, sem a utilização de uma infra-

estrutura de canais como a que foi montada para o M2PI, sem a necessidade de descritores para a transferência. Porém essa alternativa implicaria na necessidade de fazer o pareamento entre estas instruções – qual `MPI_Recv` recebe qual `MPI_Send` – o que, além de exigir uma análise complexa da estrutura do código fonte, poderia ainda depender de fatores dinâmicos, baseados em desvios condicionais e outras instruções de controle.

Outra otimização que poderia ser muito interessante seria evitar a cópia de memória quando o dado transferido fosse constante, não sendo modificado por nenhuma das partes envolvidas, após o envio. O programador poderia ajudar o reconhecimento de casos como este, usando por exemplo a palavra-chave `const` da linguagem C. Ainda assim, esta é uma situação muito complexa de ser analisada, pois além do *buffer* utilizado pela função `MPI_Recv` para receber os dados não poder ser caracterizado como uma constante, o C possui ponteiros, e o MPI os utiliza intensamente. Não é possível saber quantos e quais ponteiros apontam para determinada posição de memória, não havendo garantia portanto de que uma porção de memória não será alterada.

Uma investigação mais profunda destas e outras alternativas de otimização pode ser objeto de novos trabalhos de pesquisa, que poderão usar o M2PI como uma base para a implementação e o teste destes novos desenvolvimentos.

REFERÊNCIAS

AHO, A. V.; SETHI, R.; ULLMAN, J. D. **Compilers, Principles, Techniques, and Tools**. [S.l.]: Addison-Wesley, 1986.

ARMSTRONG, J. **Programming Erlang**: software for a concurrent world. [S.l.]: Pragmatic Programmers, 2007.

BLUMOFE, R. et al. Cilk: an efficient multithreaded runtime system. **Journal of Parallel and Distributed Computing**, [S.l.], v.37, n.1, p.55–69, 1996.

BUTENHOF, D. R. **Programming with Posix Threads**. [S.l.]: Addison-Wesley, 1997.

DEMAINE, E. D. **A Threads-Only MPI Implementation for the Development of Parallel Programs**. 1997.

DOWNEY, A. B. **The Little Book of Semaphores**. [S.l.]: Green Tea Press, 2008.

GROPP, W.; LUSK, E.; SKJELLUM, A. **Using MPI**: portable parallel programming with the Message Passing Interface. Cambridge, Massachusetts, USA: MIT Press, 1994.

GROPP, W.; LUSK, E.; THAKUR, R. **Using MPI-2**: advanced features of the Message Passing Interface. Cambridge, Massachusetts, USA: MIT Press, 1999.

JOHANSSON, E.; PETTERSSON, M.; SAGONAS, K. A high performance Erlang system. In: PPDP '00: PROCEEDINGS OF THE 2ND ACM SIGPLAN INTERNATIONAL CONFERENCE ON PRINCIPLES AND PRACTICE OF DECLARATIVE PROGRAMMING, 2000, New York, NY, USA. **Anais...** ACM, 2000. p.32–43.

LEE, E. A. The Problem with Threads. **IEEE Computer**, [S.l.], v.39, n.5, p.33–42, May 2006.

MASON, T.; BROWN, D. `lex & yacc`. [S.l.]: O'Reilly & Associates, Inc., 1990. xviii + 216p.

PARR, T. **The Definitive ANTLR Reference**: building domain-specific languages. [S.l.]: Pragmatic Programmers, 2007.

QUINN, M. J. **Parallel Programming in C with MPI and OpenMP**. [S.l.]: McGraw-Hill, 2004.

SNIR, M.; OTTO, S.; HUSS-LEDERMAN, S.; WALKER, D.; DONGARRA, J. **MPI**: the complete reference vol. 1 and 2. [S.l.]: MIT Press, 1998.

TANENBAUM, A. S. **Modern operating systems**. 2.ed. [S.l.]: Prentice-Hall, 2001. xxiv + 951p.

TANG, H.; SHEN, K.; YANG, T. Compile/Run-Time Support for Threaded MPI Execution on Multiprogrammed Shared Memory Machines. In: PPOPP, 1999. **Anais...** [S.l.: s.n.], 1999. p.107–118.

TOSCANI, S. S.; OLIVEIRA, R. S. d.; CARISSIMI, A. d. S. **Sistemas Operacionais e Programação Concorrente**. [S.l.]: Instituto de Informática da UFRGS, Sagra Luzzatto, 2003. (Série Livros Didáticos, n. 14).

APÊNDICE A LISTAGEM COMPLETA DO HELLO-WORLD MPI COMPILADO PELO M2PI

Listagem A.1: Hello-World MPI compilado pelo M2PI

```
#include <stdio.h>
#include <m2pi.h>


int m2pi_main(int argc, char* argv[]) {

   int p, r ;


   M2PI_Comm_size(MPI_COMM_WORLD, &p);
   M2PI_Comm_rank(MPI_COMM_WORLD, &r);
   printf("Oi pessoal, do processo %d dentro de %d!\n", r, p) ;


return 0;
};

void *m2pi_run(void *arg) {
    /* saves spec in TLS, and go for new main */
    M2PI_MY_THREAD_SPEC = *((M2PI_ThreadSpec *) arg);

    m2pi_main(M2PI_MY_THREAD_SPEC.argc, M2PI_MY_THREAD_SPEC.argv);

    return NULL;
}

int main(int argc, char* argv[]) {
    int m2pi_np, m2pi_i;

    /* MPI_Init() */
    argc = M2PI_Init(argc, argv);
    m2pi_np = MPI_COMM_WORLD->group->size;

    pthread_t m2pi_threads[m2pi_np];
    M2PI_ThreadSpec m2pi_threadspecs[m2pi_np];

    for (m2pi_i = 0; m2pi_i < m2pi_np; m2pi_i++) {
        m2pi_threadspecs[m2pi_i].myrank = m2pi_i;
        /* All threads get the real and same argc/argv */
        m2pi_threadspecs[m2pi_i].argc = argc;
        m2pi_threadspecs[m2pi_i].argv = argv;
```

```
        pthread_create(&m2pi_threads[m2pi_i], NULL,
            m2pi_run, (void *) &(m2pi_threadspecs[m2pi_i]));
    }

    /* MPI_Finalize() */
    for (m2pi_i = 0; m2pi_i < m2pi_np; m2pi_i++) {
        pthread_join(m2pi_threads[m2pi_i], NULL);
    }
    M2PI_Finalize();
    return 0 ;
}
```

APÊNDICE B LISTAGEM COMPLETA DO PROGRAMA DE TESTE DE VARIÁVEIS GLOBAIS E LOCAIS ESTÁTICAS COMPILADO PELO M2PI

Listagem B.1: Programa dos contadores local, global e local estático, compilado pelo M2PI

```
#include <m2pi.h>
#include <stdio.h>

 __thread int N = 1000000; /* contar ate N */
int inc(void) {
    int dummy;    /* so para ver que nao sera alterado com __thread */
    static __thread int sCounter = 0;
    return ++sCounter;
}
 __thread int gCounter;       /* contador global */


int m2pi_main(int argc, char* argv[]) {

    int i, r, lCounter = 0, sCounter = 0; /* contadores local, estatico */



    M2PI_Comm_rank(MPI_COMM_WORLD, &r);
    for (i = 0; i < N; i++) {
        /* incrementa os 3 contadores */
        gCounter++; lCounter++; sCounter = inc();
    }
    printf("Processo %d: local %d, global %d, static %d\n",
            r, lCounter, gCounter, sCounter);



return 0;
};

void *m2pi_run(void *arg) {
    /* saves spec in TLS, and go for new main */
    M2PI_MY_THREAD_SPEC = *((M2PI_ThreadSpec *) arg);

    m2pi_main(M2PI_MY_THREAD_SPEC.argc, M2PI_MY_THREAD_SPEC.argv);

    return NULL;
```

```
}

int main(int argc, char* argv[]) {
    int m2pi_np, m2pi_i;

    /* MPI_Init() */
    argc = M2PI_Init(argc, argv);
    m2pi_np = MPI_COMM_WORLD->group->size;

    pthread_t m2pi_threads[m2pi_np];
    M2PI_ThreadSpec m2pi_threadspecs[m2pi_np];

    for (m2pi_i = 0; m2pi_i < m2pi_np; m2pi_i++) {
        m2pi_threadspecs[m2pi_i].myrank = m2pi_i;
        /* All threads get the real and same argc/argv */
        m2pi_threadspecs[m2pi_i].argc = argc;
        m2pi_threadspecs[m2pi_i].argv = argv;
        pthread_create(&m2pi_threads[m2pi_i], NULL,
            m2pi_run, (void *) &(m2pi_threadspecs[m2pi_i]));
    }

    /* MPI_Finalize() */
    for (m2pi_i = 0; m2pi_i < m2pi_np; m2pi_i++) {
        pthread_join(m2pi_threads[m2pi_i], NULL);
    }
    M2PI_Finalize();
    return 0 ;
}
```