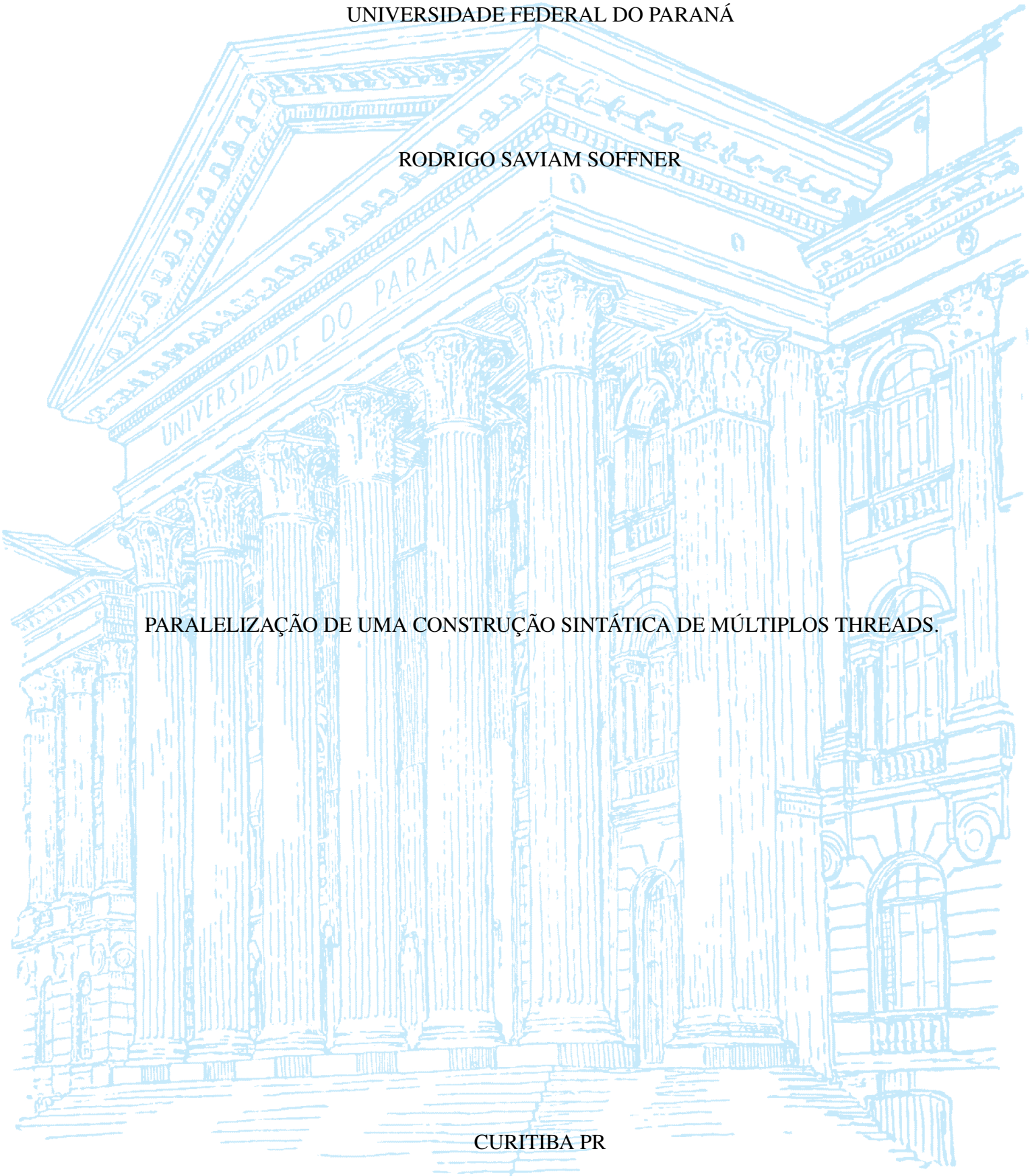
UNIVERSIDADE FEDERAL DO PARANÁ

RODRIGO SAVIAM SOFFNER



PARALELIZAÇÃO DE UMA CONSTRUÇÃO SINTÁTICA DE MÚLTIPLOS THREADS.

CURITIBA PR

2024

RODRIGO SAVIAM SOFFNER

PARALELIZAÇÃO DE UMA CONSTRUÇÃO SINTÁTICA DE MÚLTIPLOS THREADS.

Trabalho apresentado como requisito parcial à conclusão do Curso de Bacharelado em Ciência da Computação, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Bruno Müller Junior.

CURITIBA PR

2024

RESUMO

Este trabalho aborda o estudo do paralelismo no contexto de compiladores, explorando a análise de paralelização e suas implicações em linguagens de programação. Como parte do estudo, foi desenvolvida uma extensão para um compilador Pascal e seu interpretador, adicionando suporte a uma nova construção paralela, o "for paralelo". Essa construção permite a execução simultânea de laços de repetição, possibilitando a análise e implementação de dependências de dados no contexto de execução paralela. Foi necessário realizar a expansão da gramática da linguagem Pascal para acomodar a nova construção, além de aprimorar as capacidades do interpretador com a inclusão de instruções específicas para o gerenciamento de *threads*. Para atingir esses objetivos, foram empregadas as ferramentas *Flex* e *Bison*. Essa nova construção possibilitou o estudo da eficácia do processo de paralelização, considerando tanto as adaptações necessárias no compilador quanto os ajustes na máquina virtual para suportar a execução paralela.

Palavras-chave: Compiladores. Paralelismo. Interpretador.

ABSTRACT

This work explores parallelism in the context of compilers, focusing on the analysis of parallelization and its implications for programming languages. As part of this study, an extension was developed for a Pascal compiler and its interpreter, introducing support for a new parallel construct, the "for parallel." This construct enables the simultaneous execution of loop iterations, allowing the analysis and implementation of data dependencies in parallel execution contexts. The grammar of the Pascal language was expanded to accommodate the new construct, and the interpreter's capabilities were enhanced with the inclusion of specific instructions for managing *threads*. To achieve these goals, the *Flex* and *Bison* tools were employed. This new construct facilitated the study of the efficiency of the parallelization process, addressing both the necessary adaptations in the compiler and the adjustments in the virtual machine to support parallel execution.

Keywords: Compilers. Parallelism. Interpreter.

LISTA DE FIGURAS

LISTA DE ACRÔNIMOS

UFPR	Universidade Federal do Paraná
MEPA	Máquina de Execução para Pascal
MV-MEPA	Máquina Virtua MEPA
LR	Left-to-Right, Rightmost Derivation
LALR(1)	Lookahead Left-to-Right Parsing
CTHR	Criar Thread
STHR	Sair Threads
ETHR	Espera Thread
CRCT	Carregar Constante
AMEM	Alocar Memória
DSVS	Desviar Sempre
DSVF	Desviar Se Falso
CHPR	Chamar Procedimento
ENPR	Entrar no Procedimento
CRVL	Carrega Valor

SUMÁRIO

1 INTRODUÇÃO

O principal objetivo deste trabalho é estudar o processo de paralelização tanto do ponto de vista do compilador quanto das adaptações necessárias em uma máquina virtual para a execução paralela. A implementação deste conceito exige a análise de diversos aspectos, como a gramática para sustentar a construção paralela pelo compilador e a criação de novas instruções na máquina virtual que permitam a gestão de múltiplos fluxos de execução (*threads*).

Este trabalho apresenta uma nova construção na linguagem Pascal denominada "for paralelo". Esta construção permite a execução em paralelo das instruções contidas em um laço de repetição, desde que não haja dependências entre as iterações do laço. A adição dessa construção representa uma inovação na linguagem, permitindo a execução paralela de instruções e contribuindo para a eficiência em aplicações onde esse modelo de execução é viável.

Para viabilizar o desenvolvimento, foi utilizado o compilador Pascal elaborado durante a disciplina de Construção de Compiladores do curso de Ciência da Computação da UFPR. Esse compilador suporta um subconjunto da linguagem Pascal e gera comandos para uma máquina virtual chamada MEPA, que originalmente opera com um único fluxo de execução.

Este trabalho amplia as capacidades do compilador e da MEPA, adicionando o comando `for_paralelo` ao subconjunto da linguagem Pascal e criando novas instruções específicas na MEPA para lidar com múltiplos fluxos de execução. Essas alterações possibilitam a execução de fluxos de execução paralelos para casos onde não existem dependências.

1.1 MOTIVAÇÃO

A disciplina de Construção de Compiladores oferece uma abordagem prática para o entendimento dos conceitos fundamentais relacionados a compiladores. Durante o curso, os alunos desenvolvem um compilador da linguagem Pascal que traduz programas para linguagem intermediária MEPA. No entanto, esse compilador apresenta limitações, como suporte apenas para tipos inteiros e booleanos, ausência de suporte a *arrays* e falta de construções para paralelização.

O desenvolvimento deste projeto iniciou-se nessa disciplina, com o objetivo de superar essas limitações. A inclusão da construção "for paralelo" surge como uma extensão natural do trabalho, explorando os desafios e as oportunidades da integração de paralelismo em compiladores e interpretadores. Ao implementar essa contrução, foi necessário criar novas instruções e realizar os ajustes necessários na máquina virtual MEPA.

Com isso, este trabalho não apenas expande as funcionalidades originais do compilador e interpretador, mas também contribui para o estudo e a aplicação de técnicas de paralelismo.

1.2 OBJETIVOS

Para alcançar os principais objetivos propostos, realizaram-se diversas modificações e extensões no compilador e no interpretador MEPA. A primeira etapa consistiu em adicionar suporte à nova construção "for paralelo" no compilador Pascal, permitindo que a estrutura fosse reconhecida e processada. Incluiu-se a geração de comandos apropriados para a linguagem intermediária MEPA, garantindo que o paralelismo fosse corretamente traduzido para instruções executáveis pelo interpretador. Além disso, foram criadas novas instruções MEPA específicas para a criação e o gerenciamento de *threads*, possibilitando a execução simultânea das iterações do laço paralelo.

Adicionalmente, foi introduzido o suporte a *arrays* no compilador e no interpretador MEPA, uma funcionalidade ausente no projeto original da disciplina. A inclusão de *arrays* exigiu a criação de estruturas no compilador para permitir a declaração, manipulação e acesso a elementos dessas coleções. No interpretador MEPA, implementaram-se comandos específicos para operar sobre *arrays*, garantindo o gerenciamento consistente dessas estruturas no ambiente de execução. Com isso, ampliaram-se as capacidades gerais do compilador e do sistema, permitindo o processamento de programas mais complexos.

Dessa forma, o trabalho estende o projeto desenvolvido na disciplina e demonstra como conceitos avançados de paralelismo podem ser incorporados em um ambiente didático. Os resultados obtidos buscam contribuir para a compreensão de técnicas de otimização e paralelismo, ao mesmo tempo que ilustram os desafios e as soluções envolvidas na construção de compiladores e interpretadores.

1.3 ESTRUTURA DO TRABALHO

O trabalho está organizado da seguinte forma. O Capítulo 2 apresenta os termos e conceitos fundamentais utilizados no desenvolvimento deste trabalho, incluindo definições relacionadas a compiladores, linguagens de programação e paralelismo. O Capítulo 3 discorre sobre a descrição conceitual do projeto, abordando os desafios enfrentados, as soluções implementadas e os avanços realizados, como o suporte à estrutura paralela "for paralelo" e a adição de *arrays* ao compilador. O Capítulo 4 apresenta os aspectos práticos da implementação da nova instrução, tais como o ganho de desempenho. Por fim, o Capítulo 5 apresenta as conclusões, discutindo as contribuições deste trabalho, as limitações identificadas e sugestões para trabalhos futuros.

2 REVISÃO DE TERMOS

Este capítulo aborda os fundamentos teóricos deste trabalho, organizados em cinco seções. Na Seção 2.1, é apresentada uma descrição detalhada do compilador desenvolvido na disciplina de Construção de Compiladores, com ênfase nas ferramentas utilizadas, como *Bison* e *Flex*, e nos fundamentos teóricos aplicados. A Seção 2.2 explora os aspectos da MEPA, discutindo sua estrutura e funcionalidade. Na Seção 2.3, são analisados os interpretadores, com destaque para o funcionamento e as adaptações realizadas neste trabalho. A Seção 2.4 aborda o conceito de *threads*, detalhando sua relevância para a execução paralela e sua implementação no interpretador. Por fim, a Seção 2.5 discute as dependências de dados em paralelismo, classificando seus tipos, analisando seu impacto no desempenho de programas paralelos e destacando sua importância na construção "for paralelo". Esses tópicos são fundamentais para compreender as soluções apresentadas e os desafios enfrentados no desenvolvimento deste trabalho.

2.1 O COMPILADOR DESENVOLVIDO

O compilador desenvolvido no contexto da disciplina de Construção de Compiladores foi implementado na linguagem de programação *C++*, utilizando as ferramentas *Bison* e *Flex*. Esse compilador serviu como base para o desenvolvimento deste trabalho, a partir do qual foram realizadas as extensões necessárias para atender aos requisitos e objetivos propostos.

O *Flex* e o *Bison* são ferramentas amplamente utilizadas no desenvolvimento de compiladores, reconhecidas por sua eficiência e flexibilidade. De acordo com Levine (2009), essas ferramentas foram criadas para automatizar e simplificar as tarefas de análise léxica e sintática, componentes fundamentais no processo de tradução de linguagens de programação.

O *Flex* (*Fast Lexical Analyzer*) é um gerador de analisadores léxicos de código aberto que transforma um conjunto de regras, definidas por expressões regulares, em um programa C capaz de identificar os padrões léxicos em um fluxo de entrada (Levine, 2009). Sua principal função é segmentar o código-fonte em *tokens*, sendo as menores unidades significativas da linguagem, como símbolos-reservados, identificadores, números e símbolos. Cada *token* identificado pelo *Flex* é enviado ao analisador sintático para processamento posterior.

O *Bison*, por sua vez, é um gerador de analisadores sintáticos que utiliza a técnica de análise *bottom-up* baseada em *LR parsing*. Ele recebe como entrada uma gramática determinística, especificada em formato *BNF* (*Backus-Naur Form*), e gera um analisador determinísticos do tipo LR[1], utilizando tabelas LALR(1)[2] em C ou *C++* capaz de analisar a estrutura sintática do código-fonte (Levine, 2009).

O *Bison* verifica se a sequência de *tokens* fornecida pelo analisador léxico está conforme a gramática da linguagem, identificando e reportando erros sintáticos quando necessário. Além da análise sintática, é possível incluir regras semânticas na estrutura do *Bison*. No contexto deste trabalho, essas regras foram implementadas em *C++* e integradas ao compilador para realizar tarefas como verificação de tipos e constantes.

O *Bison* é amplamente utilizado para criar analisadores para diversas linguagens, sendo reconhecido por sua eficiência e flexibilidade. Alguns compiladores e linguagens inicialmente

[1] *Left-to-right, Rightmost derivation* é uma técnica de análise sintática que processa a entrada da esquerda para a direita e constrói a derivação de forma *bottom-up*.

[2] *Lookahead Left-to-Right parsing* é uma abordagem compacta que combina estados de tabelas LR para reduzir o uso de memória, mantendo a capacidade de prever produções com um símbolo de antecipação.

adotaram o *Bison*, mas posteriormente migraram para implementações próprias. Por exemplo, o *GCC* utilizou o *Bison* para gerar analisadores de C e *C++* até a versão 3.4, quando substituiu por um analisador descendente recursivo desenvolvido diretamente pelos programadores para a linguagem *C++* (GCC, 2004). Em 2006, a migração foi concluída com a substituição para C e Objective-C na versão 4.1 (GCC, 2006).

A gramática da linguagem Pascal utilizada como base no desenvolvimento foi inspirada nos exemplos apresentados no livro (Kowaltowski, 1983). Esse livro foi o material didático utilizado na disciplina de Construção de Compiladores, apresentando conceitos fundamentais e exemplos práticos que auxiliam na construção de analisadores sintáticos e léxicos. O conteúdo do livro serviu como guia tanto para a implementação da gramática quanto para a estruturação geral do projeto, proporcionando uma base sólida para o desenvolvimento do compilador. Contudo, como o *Bison* utiliza uma arquitetura de análise sintática *bottom-up*, foi necessário realizar uma tradução da gramática original, que seguia o formato *top-down*, para adaptar-se às características exigidas pela ferramenta.

O compilador desenvolvido segue as etapas tradicionais do processo de compilação. Inicia-se com a análise léxica, onde o *Flex* identifica e categoriza os tokens no código-fonte. Em seguida, ocorre a análise sintática, conduzida pelo *Bison*, que verifica se os tokens estão organizados conforme as regras da gramática definida. A partir dessas etapas, o compilador é capaz de gerar código intermediário na linguagem MEPA, seguindo os requisitos do projeto proposto na disciplina.

O compilador desenvolvido também conta com um analisador semântico, responsável por verificar se as instruções e declarações no programa fonte fazem sentido em termos da lógica da linguagem. Enquanto o analisador sintático avalia a conformidade estrutural do programa com a gramática definida, o analisador semântico examina aspectos como o uso correto de tipos, a existência de variáveis antes de seu uso, e a correspondência entre os parâmetros e argumentos nas chamadas de funções ou procedimentos. Essa etapa é crucial para garantir que o programa fonte esteja semanticamente correto ao ser traduzido.

Outro componente essencial do compilador é a tabela de símbolos. Essa estrutura de dados é utilizada para armazenar informações sobre os elementos declarados no código-fonte, como variáveis, procedimentos e funções. Para cada variável declarada, a tabela de símbolos armazena o nível léxico, o deslocamento na memória, o símbolo associado correspondente e o tipo de dado associado, informações fundamentais para a correta alocação e acesso à memória durante a execução do programa. No caso de procedimentos e funções, a tabela também registra os parâmetros associados. Esses parâmetros incluem tanto os nomes e os tipos quanto as convenções de passagem, como valores ou referências. Essas informações permitem a verificação de chamadas de funções e a geração de código correto para esses elementos.

Durante as fases de análise sintática e semântica, o compilador desenvolvido é responsável por traduzir o código-fonte Pascal para linguagem intermediária MEPA. À medida que o analisador sintático identifica estruturas válidas na gramática, instruções equivalentes em MEPA são geradas e gravadas em um arquivo de saída. Paralelamente, o analisador semântico verifica a coerência lógica das instruções, como o uso correto de tipos e variáveis, e também contribui para a geração de código ao garantir que as operações sejam semanticamente válidas. Caso seja detectado um erro durante essas etapas, o processo de tradução é imediatamente interrompido, e uma mensagem de erro específica é exibida, indicando a natureza e a localização do problema no código-fonte. Esse mecanismo assegura que apenas programas corretos sejam traduzidos e fornece informações úteis para a correção de erros por parte do programador.

Como um desafio adicional proposto na disciplina de Construção de Compiladores, foi solicitado aos alunos que implementassem o suporte à definição de novos tipos utilizando a

sintaxe *type* da linguagem Pascal. Essa funcionalidade permite que o programador crie nomes alternativos para tipos existentes, ou mesmo defina tipos personalizados, ampliando a flexibilidade da linguagem. A implementação desse recurso exigiu alterações na gramática do compilador, assim como ajustes na tabela de símbolos, de forma a registrar as associações entre os novos tipos definidos e suas características correspondentes, além de uma nova tabela para armazenar os tipos novos criados.

Em suma, o compilador desenvolvido na disciplina de Construção de Compiladores representou uma oportunidade de aplicação prática dos conceitos fundamentais relacionados à análise léxica, sintática e semântica, além da geração de código MEPA. A implementação de funcionalidades como tabelas de símbolos e rótulos, suporte à definição de novos tipos e adaptação de uma gramática para o formato necessário ao *Bison* evidenciou a complexidade e os desafios envolvidos no desenvolvimento de compiladores. Esse projeto não apenas consolidou o conhecimento teórico apresentado ao longo da disciplina, mas também incentivou a exploração de avanços adicionais, preparando os alunos para enfrentar problemas mais sofisticados na área de linguagens de programação e sistemas de compilação.

2.2 MEPA (MÁQUINA DE EXECUÇÃO PARA PASCAL)

A Máquina de Execução para Pascal (MEPA), desenvolvida por Kowaltowski (1983), é uma máquina baseada no modelo de pilha, projetada para execução de programas escritos na linguagem Pascal, codificadas na linguagem MEPA. Esse modelo executa programas por meio de uma sequência de instruções que operam diretamente em uma pilha de execução.

A MEPA utiliza a pilha como principal mecanismo de armazenamento temporário. Cada instrução da MEPA é projetada para manipular diretamente os valores na pilha, seja para realizar cálculos, alocar memória ou controlar o fluxo do programa. Por exemplo, operações como soma e multiplicação retiram os operandos do topo da pilha, realizam o cálculo e armazenam o resultado de volta na pilha. Essa característica elimina a necessidade de registradores explícitos.

Além disso, a MEPA adota um formato de instruções de tamanho fixo e um conjunto de operações básicas, incluindo entrada e saída de dados, manipulação de variáveis locais e globais, e controle de fluxo. O controle de execução é gerenciado por um contador de programa (*program counter*), que aponta para a próxima instrução a ser executada.

Um aspecto fundamental da MEPA é o suporte a níveis léxicos, utilizados para representar escopos de variáveis em programas Pascal. Para acessar essas variáveis, a MEPA utiliza a pilha de registradores base juntamente com o nível léxico, que aponta para o início do quadro de ativação atual na pilha. Isso permite localizar variáveis locais e em escopos externos por meio de deslocamentos em relação ao registrador base. Cada quadro de ativação na pilha armazena o estado da execução, incluindo o registrador base anterior, o endereço de retorno e as variáveis locais, facilitando a chamada e o retorno de funções e a manutenção de escopos aninhados.

Na Seção 2.2.1 será apresentada a linguagem intermediária MEPA[3].

A MEPA foi escolhida na disciplina de Construção de Compiladores por sua simplicidade e alinhamento com os objetivos educacionais do curso, proporcionando uma introdução prática aos conceitos de linguagens intermediárias, pilhas de execução e controle de fluxo. No entanto, a versão original da MEPA desenvolvida por Kowaltowski (1983) não incluía suporte para operações de manipulação de *arrays*. Na Seção 2.2.2, discutiremos as instruções criadas para adicionar esse suporte.

[3]Observe que o termo MEPA é utilizado tanto para se referir à máquina virtual quanto à linguagem intermediária. Para evitar confusões, neste texto utilizaremos o termo *MV-MEPA* para designar a máquina virtual e MEPA para a linguagem intermediária.

2.2.1 Linguagem MEPA

A linguagem MEPA é uma linguagem intermediária projetada para ser o alvo de compiladores que traduzem programas escritos na linguagem Pascal. Sua principal função é servir como uma ponte entre o código-fonte Pascal e a execução final na máquina virtual MEPA, simplificando o processo de interpretação e execução dos programas.

Cada instrução da MEPA é projetada para realizar tarefas específicas, como alocação de memória, controle de fluxo, operações aritméticas e manipulação de dados. Esse modelo simplificado permite que os programas em Pascal sejam traduzidos para uma sequência de instruções que podem ser facilmente interpretadas e executadas.

Entre as instruções principais da linguagem MEPA, destacam-se aquelas destinadas ao controle de fluxo, como *DSVS* e *DSVF*, que permitem a implementação de laços e decisões condicionais. Além disso, a MEPA oferece suporte à manipulação de variáveis e constantes através de comandos como *AMEM* (aloca memória) e *CRCT* (carrega constante na pilha). A interação com procedimentos e funções é realizada por meio de instruções como *CHPR* e *ENPR*, que garantem o controle adequado dos níveis léxicos e do fluxo de execução.

2.2.2 Extensões à MEPA

Para atender às necessidades deste trabalho, a MEPA foi ampliada com instruções adicionais que possibilitam a manipulação de matrizes. As instruções adicionadas foram:

- **CONT:** Carrega um elemento de um *array* para o topo da pilha, utilizando o endereço presente no topo da pilha para localizar o elemento.
- **ARMM:** Armazena um valor em um elemento de um *array*, utilizando o valor no topo da pilha como dado a ser armazenado e o valor imediatamente abaixo como endereço de destino.

Essa extensão foi essencial para viabilizar as funcionalidades desenvolvidas neste trabalho, considerando a estreita relação entre o paralelismo e a manipulação de vetores.

Na Seção 3.1 entraremos em mais detalhes sobre a implementação de matrizes na MV-MEPA e o esquema de tradução para essas instruções.

2.3 INTERPRETADORES

Um interpretador é uma ferramenta que executa diretamente o código-fonte ou uma representação intermediária de um programa, sem a necessidade de compilá-lo para código de máquina. Diferentemente de um compilador, que traduz todo o programa de uma vez antes da execução, um interpretador processa o programa linha a linha ou instrução por instrução, executando-as normalmente em uma máquina virtual. Essa abordagem permite maior flexibilidade e facilita a depuração, mas geralmente resulta em desempenho inferior em comparação com programas previamente compilados.

No contexto da MEPA, o interpretador opera sobre uma máquina virtual, simulando a execução de programas a partir de uma linguagem intermediária. Ele processa cada instrução MEPA sequencialmente, manipulando a pilha de execução para realizar operações como armazenamento, recuperação e modificação de dados, conforme especificado pelo programa. Da mesma forma, instruções de controle de fluxo são interpretadas para alterar a sequência de execução com base nas condições definidas, garantindo o comportamento esperado do programa.

Os interpretadores são amplamente utilizados em linguagens de script, como *Python* e *JavaScript*, e em contextos educacionais, como no caso da MEPA, por sua simplicidade e clareza na simulação de máquinas abstratas. No ambiente acadêmico, a MV-MEPA foi fornecida na disciplina de Construção de Compiladores como uma ferramenta auxiliar para a execução de programas traduzidos pelo compilador desenvolvido pelos alunos. Isso permitiu que os estudantes focassem na construção do compilador enquanto utilizavam o interpretador para validar a geração do código intermediário.

Neste trabalho, a MV-MEPA foi estendida para suportar novas funcionalidades. Entre as principais modificações está a implementação de comandos para paralelismo, que gerenciam a criação e a sincronização de *threads*, e o suporte à manipulação de arrays, permitindo a execução de programas com estruturas de dados mais complexas. Essas extensões foram fundamentais para a execução da nova construção "for paralelo", permitindo que operações independentes fossem distribuídas e processadas simultaneamente na máquina virtual.

2.4 THREADS

Threads são unidades independentes de execução dentro de um processo. Cada *thread* possui seu próprio fluxo de controle, mas compartilha o mesmo espaço de memória com as demais *threads* do processo, podendo também ter suas próprias variáveis locais e pilha de execução. Essa característica torna as *threads* uma ferramenta poderosa para a programação paralela, pois permite que diferentes partes de um programa sejam executadas simultaneamente, otimizando o uso de recursos computacionais, como núcleos de um processador (Butenhof, 1997).

As *threads* são amplamente utilizadas para melhorar o desempenho de aplicações que realizam tarefas intensivas em processamento. Em sistemas multiprocessadores ou com múltiplos núcleos, o uso de *threads* permite dividir o trabalho entre os núcleos disponíveis, reduzindo o tempo total de execução. No entanto, essa abordagem exige mecanismos de sincronização adequados para evitar problemas como condições de corrida, onde duas ou mais *threads* tentam acessar e modificar simultaneamente os mesmos dados.

Para implementar o suporte ao paralelismo no MV-MEPA, foram utilizadas *threads*, para permitir a execução simultânea de múltiplos fluxos. Foi utilizada a biblioteca *POSIX Threads* (*pthreads*) da linguagem C, que é um padrão adotado para programação com *threads* em sistemas operacionais compatíveis com POSIX. A biblioteca *pthreads* fornece um conjunto de funções para criação, sincronização e gerenciamento de *threads*, facilitando o desenvolvimento das instruções de paralelismo para a MV-MEPA.

2.5 DEPENDÊNCIAS DE DADOS EM PARALELISMO

No contexto do paralelismo, a análise de dependências de dados é uma etapa fundamental para determinar se as instruções ou operações de um programa podem ser executadas simultaneamente. Uma dependência de dados ocorre quando uma instrução depende do resultado de outra para ser corretamente executada. Essas dependências afetam diretamente a capacidade de dividir as tarefas entre diferentes unidades de processamento, uma vez que a execução paralela exige, idealmente, que as operações sejam independentes.

A análise de dependências é amplamente discutida na literatura como uma das principais barreiras e, simultaneamente, ferramentas para a exploração eficiente do paralelismo. Por exemplo, Hennessy e Patterson (2012) abordam o papel das dependências na execução de instruções e destacam como elas podem impactar a eficiência de arquiteturas paralelas.

As dependências de dados são geralmente classificadas em três tipos principais:

- **Dependência de fluxo:** Ocorre quando uma operação precisa ler o resultado produzido por uma instrução anterior. Por exemplo:

```
A = B + C;
D = A + E;
```

  As duas instruções não podem ser executadas em paralelo, uma vez que, a segunda instrução depende do valor calculado por `A` na primeira.

- **Anti-dependência:** Surge quando uma instrução tenta escrever em uma variável antes que outra instrução tenha terminado de lê-la. Por exemplo:

```
D = A + E;
A = F + G;
```

  Neste exemplo, as duas instruções não podem ser executadas em paralelo, pois a segunda pode atualizar o valor de `A` antes que a segunda instrução consiga acessá-lo corretamente. Uma solução para esse problema é renomear a variável `A`, substituindo-a, por exemplo, por `A1` a partir da segunda linha. Essa técnica, abordada por Aho et al. (2006), é utilizada para resolver conflitos de dependência e viabilizar a execução paralela.

- **Dependência de saída:** Acontece quando duas instruções tentam escrever na mesma variável, mas a ordem das operações é importante para o resultado final. Por exemplo:

```
A = B + C;
A = D + E;
```

  Neste exemplo, ambas as instruções atualizam `A`, sendo crucial que a segunda seja executada apenas após a primeira, para assegurar que as instruções subsequentes utilizem o valor correto de `A`. Embora neste caso simples a primeira instrução possa ser descartada, em cenários mais complexos, como em laços, essa análise torna-se menos evidente. Para esses casos, uma solução é o uso de mecanismos que contornam condições de corrida como *mutex*.

A análise de dependências é crucial em laços de repetição, como o `for`, onde várias iterações podem ser executadas em paralelo. No entanto, a existência de dependências entre as iterações pode impedir uma paralelização direta e segura. O exemplo a seguir ilustra um cenário que inclui dependências de fluxo e anti-dependência, tornando a execução paralela problemática sem ajustes adicionais:

```
for
    A[i] := A[i-1] + A[i+1];
fimfor
```

Nesse caso, cada iteração do laço depende do valor calculado na iteração anterior e do valor que havia antes de executar a iteração futura. Esse padrão de dependência exige técnicas avançadas e segundo Aho et al. (2006), compreender as dependências de dados é essencial para explorar o paralelismo em arquiteturas computacionais, pois permite identificar oportunidades para otimizações e evitar conflitos que poderiam comprometer os resultados.

No contexto deste trabalho, a construção "for paralelo" foi projetada para operar exclusivamente com iterações independentes, ou seja, iterações onde não existem dependências de dados entre si. Um exemplo dessa situação é ilustrado a seguir:

```
for
    A[i] := B[i] + C[i];
fimfor
```

Essa abordagem simplificada foi adotada para garantir a corretude do programa gerado pelo compilador e executado na MV-MEPA, eliminando a necessidade de mecanismos adicionais para lidar com dependências de dados.

Apesar da simplificação, a identificação de dependências em laços é um tema de pesquisa ativo, envolvendo técnicas como análise estática, transformação de código e algoritmos especializados. Como destacado por Aho et al. (2006), a análise estática desempenha um papel crucial na detecção de possíveis conflitos e na aplicação de transformações que maximizem a eficiência da execução paralela, mantendo a corretude dos resultados.

3 DESCRIÇÃO DAS IMPLEMENTAÇÕES

Este capítulo apresenta as implementações realizadas neste trabalho, detalhando as extensões no compilador e no interpretador para suportar as novas funcionalidades propostas. A Seção 3.1 aborda a adição do suporte a matrizes, descrevendo os ajustes realizados para integrar essa funcionalidade à linguagem Pascal e à arquitetura MEPA.

A Seção 3.2 apresenta o desenvolvimento técnico da construção "for paralelo", abrangendo seu funcionamento, as adaptações necessárias e os fundamentos teóricos que embasaram sua concepção e implementação.

A Seção 3.3 discute o sistema de gerenciamento de memória desenvolvido na MV-MEPA para suportar a execução de *threads*, garantindo o isolamento apropriado e o acesso ao escopo de memória global.

3.1 IMPLEMENTAÇÃO DE MATRIZES

Uma das extensões realizadas neste trabalho foi a implementação do suporte a matrizes, uma funcionalidade ausente no compilador e interpretador base desenvolvidos na disciplina. Essa adição permite que o compilador processe programas que utilizem arrays, ampliando significativamente as capacidades da linguagem.

A sintaxe para declarar matrizes foi projetada para manter-se próxima à do Pascal, mas com elementos que remetem à simplicidade da linguagem C. Por exemplo, a declaração a seguir define três arrays unidimensionais com 100 elementos do tipo `integer`:

```
1 var A, B, C: integer[100];
```

Essa implementação suporta a declaração de matrizes unidimensionais, possibilitando operações sobre índices para manipulação de dados. Para isso, foram criadas instruções específicas no interpretador MEPA para lidar com matrizes. Essas instruções serão detalhadas na Seção 3.1.1, onde discutimos as modificações realizadas no MEPA para atender às novas funcionalidades propostas por este trabalho.

Essa extensão foi essencial para permitir a realização das análises de paralelismo propostas neste trabalho, possibilitando a manipulação eficiente de dados estruturados em matrizes. Com isso, a linguagem foi adaptada para atender às necessidades específicas do estudo, ampliando sua funcionalidade para suportar operações paralelas em estruturas de dados mais complexas.

3.1.1 Funcionamento na Máquina Virtual

Para o suporte a matrizes, foram adicionadas instruções que permitem a manipulação eficiente de índices e elementos armazenados. As instruções implementadas foram:

- **CONT:** Esta instrução é responsável por carregar na pilha o valor de um elemento localizado em um endereço de memória específico, geralmente utilizado em operações envolvendo vetores. O valor contido no topo da pilha é interpretado como o endereço do elemento a ser acessado, e a instrução substitui esse valor pelo conteúdo armazenado no endereço indicado. Por exemplo, em uma operação como `... := A[i+j]`, `CONT` acessa o índice calculado `i+j` do vetor `A` e carrega o valor correspondente no topo da pilha.

- **ARMM:** Essa instrução realiza a operação de armazenamento em uma posição específica da matriz. Para isso, consome os dois valores do topo da pilha: o penúltimo valor é utilizado como o endereço de memória onde o dado será armazenado, enquanto o valor do topo é o dado que será gravado nesse endereço.

  Por exemplo, a atribuição `A[i] := A[j]` seria traduzida da seguinte maneira:

```
CRVL i        % Carrega o valor do índice i na pilha
CRCT A        % Carrega a base do vetor A na pilha
SOMA          % Calcula o endereço do elemento A[i]
CRVL j        % Carrega o índice j na pilha
CRCT A        % Carrega a base do vetor A na pilha
SOMA          % Calcula o endereço do elemento A[j]
CONT          % Carrega o valor armazenado no endereço A[j]
ARMM          % Armazena o valor carregado no endereço A[i]
```

Essas instruções foram essenciais para habilitar a funcionalidade de matrizes no MEPA, garantindo suporte para operações básicas de leitura e escrita, alinhadas à sintaxe implementada no compilador.

3.2 CONSTRUÇÃO FOR PARALELO

Nesta seção, é descrita a implementação da construção "for paralelo". Essa construção foi fundamentada em uma pesquisa comparativa com funcionalidades semelhantes em outras linguagens de programação. Durante a análise, identificou-se que a abordagem mais adequada para este trabalho é a construção `Parallel.For` da linguagem C#, que será apresentada na subseção 3.2.1.

Na subseção 3.2.2, entraremos em detalhes sobre a construção "for paralelo" desenvolvida para este trabalho, abordando seu funcionamento, as escolhas realizadas durante o desenvolvimento e os ajustes necessários no compilador e no interpretador para sua integração.

3.2.1 Parallel.For no C#

A construção `Parallel.For` no C# facilita a execução paralela de laços ao distribuir automaticamente as iterações entre várias threads, maximizando a utilização dos recursos do sistema. Essa abordagem elimina a necessidade de o programador criar e gerenciar threads manualmente, simplificando significativamente a programação paralela (Microsoft, 2024).

Essa construção permite personalização de comportamentos, como o uso de partições customizadas e controle granular sobre as iterações por meio de funções de inicialização e finalização de estado local. O desenvolvedor define os limites do laço e fornece o corpo da iteração como uma expressão lambda ou método, enquanto o `Parallel.For` gerencia a divisão de trabalho e a sincronização interna.

Além disso, o `Parallel.For` ajusta automaticamente o nível de paralelismo com base na quantidade de threads disponíveis no hardware, garantindo uma utilização eficiente dos recursos do sistema. No entanto, é possível incluir um parâmetro para definir a quantidade máxima de threads a ser utilizada, permitindo que o desenvolvedor controle explicitamente o nível de paralelismo em cenários específicos.

A construção `Parallel.For` foi utilizada como referência para o desenvolvimento do "for paralelo", adaptando seus conceitos ao contexto da linguagem Pascal e ao funcionamento do compilador e interpretador deste trabalho.

3.2.2 For Paralelo

O "for paralelo", desenvolvido para a linguagem Pascal, é uma construção que permite a execução simultânea das iterações de um laço, explorando o paralelismo por meio da criação de múltiplas *threads*. A organização e o funcionamento dessa construção foram projetados considerando as limitações e características específicas da linguagem Pascal, bem como as necessidades de integração com o compilador e o interpretador MEPA.

Essa construção foi projetada para ser utilizada em operações onde não há dependências entre as iterações do laço, o que possibilita que cada iteração seja atribuída a uma *thread* diferente, assegurando a corretude do programa.

Diferentemente da construção `Parallel.For` do C#, que ajusta dinamicamente a quantidade de *threads* em tempo de execução com base nos recursos disponíveis do hardware, o "for paralelo" requer que o programador defina explicitamente o número de *threads* a ser utilizado no laço. Essa abordagem fixa o nível de paralelismo durante a compilação, simplificando a tradução e execução do código, mas demanda que o programador avalie previamente os recursos do sistema para determinar a quantidade ideal de *threads*.

Na Seção 3.2.2.1 descreve o funcionamento da construção, abordando seu comportamento e execução. A Seção 3.2.2.2, é apresentada a gramática que define a construção, descrevendo as regras sintáticas necessárias para sua implementação. Em seguida, a Seção 3.2.2.3 explica as instruções introduzidas na MV-MEPA para viabilizar o paralelismo. Por último, a Seção 3.2.2.4 detalha o esquema de tradução que converte o código-fonte Pascal para as instruções MEPA.

3.2.2.1 Funcionamento

A construção "for paralelo" foi desenvolvida com o objetivo de permitir a execução simultânea de iterações de laços, distribuindo-as de forma eficiente entre múltiplas *threads*. Para ilustrar o funcionamento dessa construção, consideramos o código de exemplo apresentado a seguir:

```
1  program forParallel(input, output);
2  var A, B, C: integer[100];
3        i: integer;
4  begin
5        create_threads th:= 4 do
6        var desloc: integer;
7        for_parallel ii:= threadId to 100 step th
8        begin
9              A[ii] := B[ii] + C[ii];
10       end;
11 end.
```

O programa apresentado ilustra o funcionamento da construção "for paralelo", cuja execução é iniciada pela instrução `create_threads`. Essa instrução é responsável por configurar o ambiente necessário para a execução do bloco `for_parallel`, fornecendo as variáveis implícitas essenciais para o funcionamento do paralelismo. Essas variáveis são descritas a seguir:

- **Identificador da Thread:** Representa um identificador único para cada *thread*, variando de 0 até o número total de *threads* menos 1 (`0..n-1`), permitindo distinguir cada *thread* no conjunto criado.

- **Número de Threads:** Especifica a quantidade total de *threads* utilizadas para executar o bloco paralelo, definido por um número.
- **Índice:** Variável responsável por controlar o progresso do laço em cada *thread*, inicializada com o valor de uma expressão inteira.
- **Valor Limite do Loop:** Define o limite superior para as iterações do laço, especificado por uma expressão inteira.
- **Step:** Determina o incremento aplicado ao índice em cada iteração, sendo também definido por uma expressão inteira.

Os símbolos das variáveis de índice e número de *threads* são definidos pelo programador, sendo especificados diretamente na construção do "for paralelo". No exemplo apresentado, o índice é representado pela variável `ii`, enquanto a quantidade de *threads* é atribuída à variável `th`.Além disso, a variável de identificador da *thread* está sempre associada ao símbolo `threadId`.

A variável `th`, que armazena a quantidade total de *threads*, é utilizada no `step` do laço para garantir que as iterações sejam corretamente distribuídas entre as *threads*. Adicionalmente, a variável especial `threadId`, associada a cada *thread*, define o valor inicial da variável de índice `ii`. Essa organização assegura que cada *thread* processe um subconjunto específico de iterações de maneira não sobreposta.

No contexto do código apresentado, a construção opera da seguinte forma:

- ***Thread* 1:** A *thread* possui o valor `threadId` igual a 0, e seus índices terão os seguintes valores durante as iterações: `ii = 0, 4, 8, ...`.
- ***Thread* 2:** A *thread* possui o valor `threadId` igual a 1, e seus índices terão os seguintes valores durante as iterações: `ii = 1, 5, 9, ...`.

Para as demais *threads*, o comportamento é similar, com os índices iniciando em `threadId` e avançando conforme o valor de `step`.

Esse comportamento é garantido pela definição do valor de `step`, configurado como o número total de *threads* (`th`). Isso assegura que a distribuição do trabalho seja realizada de forma equilibrada entre as *threads*, além de manter o processamento adequado e a corretude das operações paralelas.

As variáveis utilizadas no laço paralelo possuem escopo bem definido. No exemplo, a variável `desloc`, declarada no bloco de variáveis locais (`var`), é exclusiva a cada *thread*, eliminando conflitos entre as operações realizadas simultaneamente. Além disso, variáveis inerentes à construção, como `threadId`, `ii` e `th`, também estão disponíveis dentro do escopo do laço e são isoladas entre as *threads*, garantindo que cada *thread* opere de forma independente.

A execução do laço paralelo é realizada até que todas as *threads* completem suas respectivas iterações. Durante esse processo, a *thread* principal do programa é suspensa e retomada somente após a conclusão de todas as *threads* criadas pelo "for paralelo". Essa abordagem não apenas maximiza o aproveitamento dos recursos computacionais do sistema, mas também reduz significativamente o tempo de execução do laço em comparação à execução sequencial, mantendo a sincronização e a corretude do programa.

3.2.2.2 *Gramática*

A gramática desenvolvida para a construção "for paralelo" descreve a estrutura necessária para implementar e organizar o paralelismo de forma funcional, atendendo aos objetivos deste trabalho. Essa gramática define as regras sintáticas que possibilitam a criação e execução de blocos paralelos utilizando múltiplas *threads*. A Figura 3.1 apresenta a gramática formal em BNF (*Backus-Naur Form*), que orienta a implementação dessa funcionalidade no compilador.

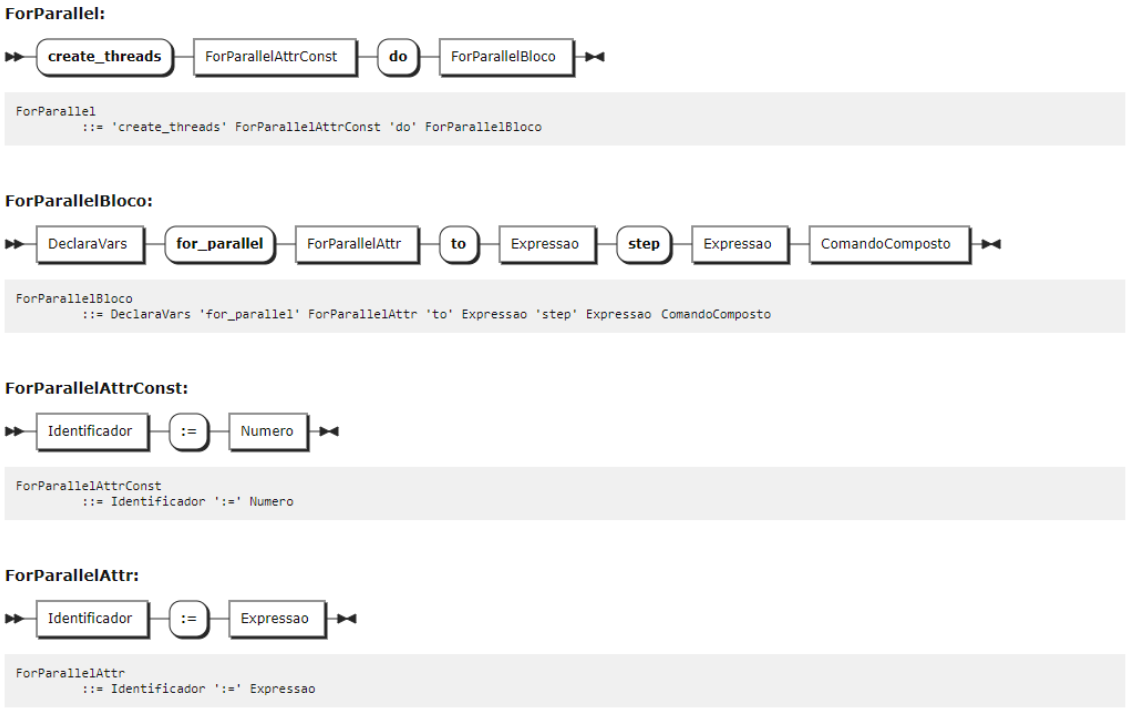


Figura 3.1: BNF da construção "for paralelo".

Conforme ilustrado na Figura 3.1, a construção inicia-se com a definição do número de *threads*, utilizando a palavra reservada `create_threads`, seguida por uma atribuição. Essa atribuição associa um identificador ao valor informado, criando um novo símbolo que estará disponível no contexto do "for paralelo". É importante destacar que o número de *threads* deve ser uma constante, pois seu valor é necessário para especificar o número de *threads* a ser criado.

Após a definição das *threads*, um bloco de declaração de variáveis locais pode ser especificado. Esse bloco, similar ao utilizado em programas principais ou em funções e procedimentos, permite que cada *thread* possua suas próprias variáveis locais. As variáveis declaradas nesse escopo são isoladas entre as *threads*, eliminando conflitos e garantindo a execução paralela sem interferências.

A estrutura do laço paralelo é definida pela palavra reservada `for_parallel`. Esse laço introduz uma variável de índice que é atribuída a um valor inicial e incrementada conforme especificado pela palavra reservada `step`. O limite superior do laço é indicado pela palavra reservada `to`, seguida por uma expressão que delimita o alcance das iterações. Essas regras, apresentadas na Figura 3.1, asseguram que o laço seja configurado adequadamente para distribuir as iterações entre as *threads*.

Cada *thread* recebe um identificador especial, `threadId`, que varia de `0` a `create_threads - 1`. Esse identificador é configurado na MV-MEPA durante a criação das *threads* e pode ser utilizado dentro do bloco de execução para personalizar o comportamento de cada *thread*, garantindo maior flexibilidade na programação.

É importante ressaltar que, de forma similar às variáveis definidas no bloco de declaração de variáveis, as variáveis intrínsecas à construção "for paralelo" — como o número de *threads*, o índice, `threadId`, `step` e o limite superior do laço — também estão disponíveis no escopo do laço. Essas variáveis são isoladas entre as *threads*, garantindo que cada uma opere de maneira independente e sem interferências.

O corpo do laço paralelo é delimitado pelas palavras reservadas `begin` e `end`, formando um bloco de execução (`comando_composto`). Esse bloco encapsula as instruções que serão executadas paralelamente pelas *threads*.

3.2.2.3 Funcionamento na MV-MEPA

A execução da construção "for paralelo" na MV-MEPA exigiu a introdução de instruções específicas para gerenciar e executar *threads*, assegurando a correta divisão de tarefas e a sincronização entre as *threads* criadas. Essas instruções foram projetadas para suportar o paralelismo e atender às necessidades de programas que utilizam essa construção.

As instruções adicionadas para o suporte ao paralelismo são:

- **CTHR *m*, *n*, *p* (Cria *Thread*):** Responsável pela criação de uma nova *thread* com identificador único (*m*) e início da execução a partir de um rótulo especificado (*p*). Durante sua execução, a MV-MEPA cria as estruturas de dados necessárias, inicializa as `pthreads` e define dinamicamente o valor do `threadId` (*m*). O parâmetro *n* corresponde ao nível léxico, permitindo que a *thread* acesse corretamente variáveis locais e globais. A atribuição dinâmica do `threadId` pela MV-MEPA é crucial, pois o código gerado pelo compilador é idêntico para todas as *threads*, tornando impossível para o compilador atribuir identificadores exclusivos.

- **STHR (Para *Thread*):** Marca o encerramento de uma *thread* criada no contexto do "for paralelo". Essa instrução garante que todas as *threads* finalizem suas operações de forma controlada, liberando os recursos alocados, incluindo o encerramento da `pthread` correspondente.

- **ETHR *k* (Espera *Thread*):** Pausa a execução da *thread* principal até que uma *thread* específica, identificada pelo *k*, conclua suas operações. Essa instrução atua como um mecanismo de sincronização, assegurando que a *thread* principal só retome sua execução após a finalização da *thread* alvo. Após a conclusão, os recursos utilizados pela *thread* são liberados, garantindo eficiência no uso da memória e prevenindo desperdícios de recursos.

O uso dessas instruções pela MV-MEPA permite o controle do paralelismo durante a execução do "for paralelo". A instrução `CTHR`, além de criar e iniciar as *threads*, associa dinamicamente o valor de `threadId` a cada uma, tornando possível identificar e distribuir corretamente as iterações do laço entre as *threads*. Isso garante que cada *thread* processe exclusivamente as iterações designadas a ela, conforme definido pela construção.

A instrução `STHR` assegura que todas as *threads* sejam encerradas adequadamente ao final do bloco paralelo, prevenindo a persistência de *threads* desnecessárias que poderiam gerar problemas de concorrência ou consumir recursos indevidamente.

Já a instrução `ETHR` desempenha um papel fundamental na sincronização da execução. Ela pausa a *thread* principal até a conclusão de uma *thread* específica, garantindo a integridade e a ordem de execução do programa. Após o término da *thread*, a *thread* principal é desbloqueada e os

recursos alocados à *thread* concluída são liberados, otimizando o uso da memória e assegurando a corretude do programa.

Para o funcionamento das *threads* adicionais na MV-MEPA, criadas pelas instruções `CTHR`, foi necessário implementar um ambiente de execução dedicado para esses fluxos paralelos. Cada *thread* cria uma nova instância da máquina virtual, possuindo um isolamento e independência para os diferentes fluxos de execução. Essas instâncias possuem pilhas de memória e registradores base próprios, além de ponteiros individuais para a instrução atual e o topo da pilha, permitindo que as execuções ocorram de forma totalmente independente e paralela.

3.2.2.4 Esquema de Tradução

O esquema de tradução da construção "for paralelo" detalha como o compilador transforma o código-fonte Pascal em instruções MEPA, permitindo sua execução pela MV-MEPA. Nesta seção, utiliza-se o fluxo representado na Figura 3.2 e a gramática em BNF apresentada na Figura 3.3, com as traduções, para ilustrar as etapas de tradução do código apresentado na Seção 3.2.2.1.

A Figura 3.2 ilustra a separação prática do que ocorre durante a execução do "for paralelo". No processo principal, são realizadas quatro instruções `CTHR`, cada uma responsável por criar uma *thread*. Cada *thread* executa a seguinte sequência de ações:

- **Declaração de Variáveis:** Realiza a alocação de memória para as variáveis locais da *thread*.
- **Armazenamento de Variáveis da Construção:** Calcula e armazena os valores das variáveis necessárias para a execução da construção paralela, elas sendo:
  - Número de *threads*;
  - Identificador da *thread*;
  - Índice;
  - Valor limite do laço;
  - Passo (*step*).
- **Execução do Laço:** Executa o bloco de comandos definido dentro da estrutura de laço.
- **Finalização da *Thread*:** Ao concluir o laço, executa a instrução `STHR`, sinalizando a finalização da *thread*.

Após a finalização de todas as *threads*, o processo principal retoma sua execução normal. Essa sincronização é realizada por meio de quatro instruções `ETHR`, cada uma responsável por aguardar a conclusão da execução de uma *thread*.

Os comandos do processo principal são responsáveis por gerenciar tanto a criação quanto a sincronização das *threads* durante a execução de um laço paralelo. A criação das *threads* é realizada por meio da instrução `CTHR`. E para a sincronização, o fluxo de execução é direcionado à instrução `ETHR`, localizada após as instruções das *threads*.

No bloco de execução das *threads*, realiza-se a tradução das instruções que cada *thread* criada irá executar. Como o código é idêntico para todas as *threads*, ele deve ser suficientemente genérico para garantir a execução correta em cada instância. Primeiramente, são alocadas as cinco variáveis utilizadas pela estrutura "for paralelo", incluindo o armazenamento do número de *threads* na variável `th`.

Após isso, realiza-se a alocação de memória para o bloco de declaração de variáveis e o armazenamento das demais variáveis relacionadas ao laço "for paralelo": o índice, o limite superior do laço e o valor de incremento (*step*). Com essas preparações concluídas, o início do laço é traduzido com instruções para comparar o valor do índice com o limite superior e controlar o fluxo, dependendo do resultado da comparação. Na Figura 3.3, é possível identificar as instruções necessárias para realizar essas operações.

Dentro do laço, são traduzidas as instruções correspondentes ao bloco de comandos do usuário. Por fim, o término do laço realiza a soma do valor de *step* ao índice. Após a execução do laço, a instrução `STHR` é inserida para indicar que a *thread* deve encerrar sua execução.

3.3 GERENCIAMENTO DE MEMÓRIA PARA *THREADS* NO INTERPRETADOR

Um dos grandes desafios ao implementar o suporte ao paralelismo foi garantir a coexistência de memória global, memória local e memória de cálculo para as *threads*, assegurando que cada *thread* mantivesse seu próprio contexto de execução enquanto compartilhasse a mesma visão da memória global. Como ilustrado no exemplo presente na Seção 3.2.2.1, durante a soma dos vetores `B` e `C`, com armazenamento em `A`, os vetores estão no escopo global do programa, acessíveis por todas as *threads*. Por outro lado, variáveis de controle, como o índice e a variável local `desloc`, são locais ao escopo do "for paralelo" e devem ser independentes para cada *thread*.

Para atender a esses requisitos, foi implementado um sistema de gerenciamento de memória dentro da MV-MEPA, garantindo que cada *thread* possuísse seu próprio espaço de

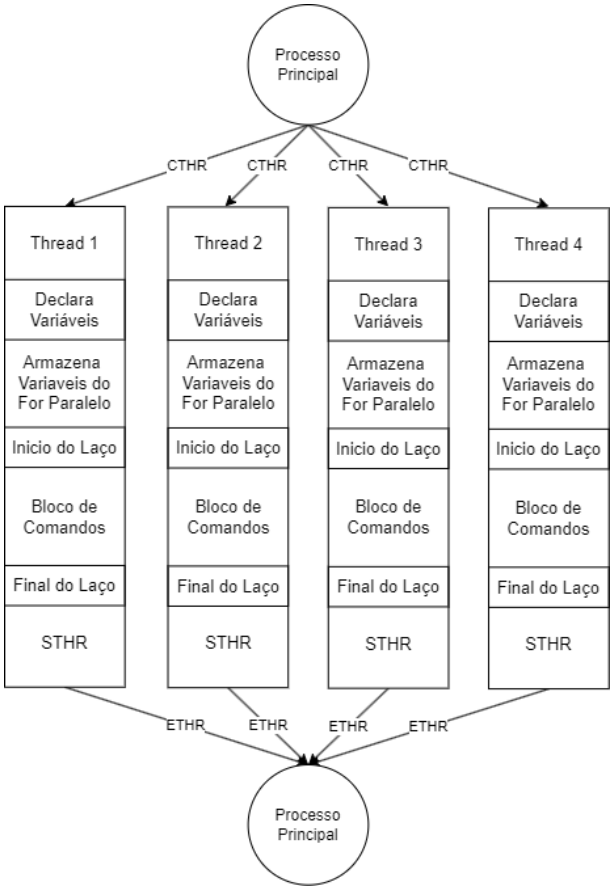


Figura 3.2: Fluxo da construção "for paralelo".

memória local e de cálculo, enquanto compartilhasse o espaço de memória global associado ao processo principal.

Cada *thread* criada recebe uma pilha de memória individual, usada para armazenar variáveis locais e dados temporários necessários durante sua execução. Essa divisão assegura que as operações de uma *thread* não interfiram diretamente nos dados locais de outra.

Além dessas pilhas, cada *thread* também possui uma pilha **D**, destinada ao armazenamento das informações dos níveis léxicos, como endereços de retorno e variáveis de controle. Essas informações são copiadas da pilha **D** do processo principal no momento em que a *thread* é criada. Essa pilha **D** é fundamental para viabilizar a execução de chamadas dentro do bloco de execução paralelo, mantendo a consistência entre os níveis léxicos das *threads* e a estrutura do programa.

O bloco de execução do "for paralelo" opera de maneira similar a um procedimento ou função no MEPA. Durante a execução do bloco, o nível léxico é aumentado, permitindo que a *thread* acesse variáveis e contextos locais de forma segura. Essa característica também assegura que o escopo das variáveis e os controles léxicos sejam respeitados.

O gerenciamento de acesso à memória é realizado de forma a determinar se o acesso ocorre no processo principal ou em uma das *threads* criadas. Caso o acesso seja feito pelo

ForParallel:

```
ForParallel
        ::= 'create_threads'
            ForParallelAttrConst
            'do' -> CTHR 0, 1, R00          # Cria a thread com identificador 0.
                 -> CTHR 1, 1, R00          # Cria a thread com identificador 1.
                 -> CTHR 2, 1, R00          # Cria a thread com identificador 2.
                 -> CTHR 3, 1, R00          # Cria a thread com identificador 3.
                 -> DSVS R03                # Desvia para o código após as threads.
            ForParallelBloco -> DMEM 6      # Desaloca a memória das variáveis do For Paralelo e locais.
                             -> STHR        # Finaliza o processamento da thread.
                             -> R03: NADA   # Rótulo para o código após as instruções da thread.
                             -> ETHR 0      # Aguarda a finalização da thread 0.
                             -> ETHR 1      # Aguarda a finalização da thread 1.
                             -> ETHR 2      # Aguarda a finalização da thread 2.
                             -> ETHR 3      # Aguarda a finalização da thread 3.
```

ForParallelBloco:

```
ForParallelBloco
        ::= -> R00: NADA                    # Rótulo de início das threads.
            -> AMEM 5                       # Aloca memória para variáveis do "For Paralelo": Número de Threads,
                                              Identificador da Thread, Índice, Valor Limite do Loop e Step.
            -> CRCT 4                       # Carrega o Número de Threads especificado em create_threads.
            -> ARMZ th                      # Armazena o valor de Número de Threads.
            DeclaraVars -> AMEM 1           # Aloca memória para variáveis locais.
            'for_parallel'
            ForParallelAttr
            'to'
            Expressao -> CRCT 100           # Calcula o valor da expressão do limite do laço.
                      -> ARMZ to            # Armazena o limite na variável Valor Limite.
            'step'
            Expressao -> CRVL th            # Calcula a expressão.
                      -> ARMZ step          # Armazena o valor de Step.
            -> R01: NADA                    # Início do laço.
            -> CRVL ii                      # Carrega o valor do índice.
            -> CRVL to                      # Carrega o valor limite do laço.
            -> CMME                         # Compara se o índice é menor que o limite do laço.
            -> DSVF R02                     # Desvia para o final do laço se falso.
            ComandoComposto -> CRVL ii      # Carrega o valor do índice.
                            -> CRVL step    # Carrega o valor do step.
                            -> SOMA         # Soma os dois valores.
                            -> ARMZ ii      # Armazena o valor resultante como novo índice.
                            -> DSVS R01     # Desvia para o início do laço.
                            -> R02: NADA    # Rótulo para o final do laço.
```

ForParallelAttr:

```
ForParallelAttr
        ::= Identificador
            ':='
            Expressao -> CRVL threadId      # Carrega o identificador da thread.
                      -> ARMZ ii            # Armazena no índice.
```

Figura 3.3: BNF com as instruções MEPA geradas na construção "for paralelo".

processo principal, ele é direcionado diretamente para a pilha global. Para as *threads*, é realizado um cálculo que utiliza um deslocamento específico para diferenciar a memória global da memória local da *thread*. Qualquer endereço abaixo do limite de deslocamento é interpretado como pertencente à memória global, enquanto endereços acima desse limite são resolvidos na memória local da *thread*.

4 RESULTADOS E ANÁLISE

Este capítulo apresenta a análise do "for paralelo" em um cenário real, demonstrando sua aplicação prática e os impactos na execução paralela. A Seção 4.1 descreve o teste realizado, incluindo os resultados obtidos. Em seguida, a Seção 4.2 discute os resultados, analisando o desempenho e as implicações do uso da construção "for paralelo" no contexto do paralelismo.

4.1 TESTE E RESULTADOS

Para avaliar o funcionamento e o desempenho da construção "for paralelo", realizamos testes utilizando um programa Pascal que manipula um vetor de 10.000.000 posições. Foram comparados os tempos de execução para dois cenários: o uso de uma única *thread* sem a construção desenvolvida e com diferentes quantidades de *threads* (2, 4, 8 e 16) utilizando a construção "for paralelo". Cada caso foi executado 100 vezes, e o tempo médio de execução foi calculado para análise.

```
1  program forParallel(input, output);
2  var B, C, A: integer[10000000];
3        i: integer;
4  begin
5        create_threads th:= 4 do
6        for_parallel ii:= threadId to 10000000 step th
7        begin
8              A[ii] := 0;
9              B[ii] := ii + 1;
10             C[ii] := ii + 1;
11       end;
12
13       create_threads th:= 4 do
14       for_parallel ii:= threadId to 10000000 step th
15       begin
16             A[ii] := B[ii] + C[ii];
17       end;
18
19       i := 0;
20       while i < 10000000 do
21       begin
22             write(A[i]);
23             i := i + 1;
24       end;
25 end.
```

O programa apresentado foi utilizado como base para os testes. Ele realiza duas operações paralelas sobre os vetores `A`, `B` e `C`. Primeiro, os vetores são inicializados, com os valores de `B` e `C` sendo definidos como índices incrementados de 1. Em seguida, é realizada a soma dos valores correspondentes de `B` e `C`, armazenando o resultado em `A`.

É importante destacar que os testes foram realizados com o código apresentado, excluindo o laço de impressão. Essa decisão teve como objetivo eliminar o custo sequencial associado a essas instruções, permitindo que a análise se concentrasse exclusivamente no desempenho da parte paralelizável do código e no impacto da construção "for paralelo".

É importante destacar que os testes foram realizados com o código apresentado, excluindo o laço de impressão. Essa decisão teve como objetivo eliminar o custo sequencial associado a

essas instruções, permitindo que a análise se concentrasse exclusivamente no desempenho da parte paralelizável do código e no impacto da construção "for paralelo".

Os programas foram testados para verificar sua funcionalidade. Todos os casos produziram os mesmos resultados, confirmando que a construção "for paralelo" preserva a corretude dos programas ao dividir e paralelizar as iterações.

O código apresentado corresponde ao caso de teste com *create_threads* igual a 4. Adicionalmente, os testes foram realizados para 2, 8 e 16 *threads*, assim como para uma versão sem o uso da construção "for paralelo", onde foi empregado um *loop while*, com a intenção de simular a execução sequencial padrão da MV-MEPA.

Os testes foram realizados em uma máquina equipada com um processador AMD Ryzen 7 3800X, com 8 núcleos físicos e *hyperthreading*, totalizando 16 *threads* de processamento disponíveis.

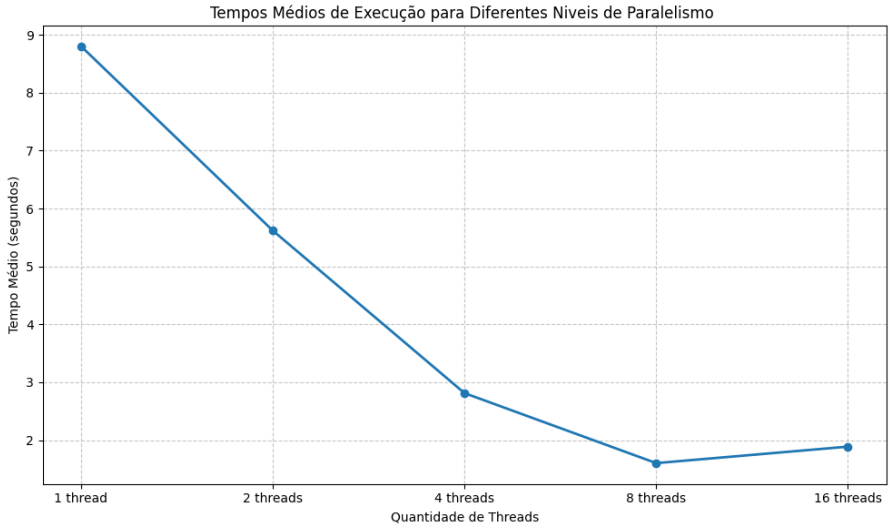


Figura 4.1: Resultados do tempo médio de execução para diferentes quantidades de *threads*.

A Figura 4.1 apresenta os resultados obtidos, destacando a redução de tempo de execução à medida que o número de *threads* aumenta. Os detalhes da análise dos resultados são discutidos na Seção 4.2.

4.2 ANÁLISE DOS RESULTADOS

Os resultados obtidos a partir dos testes mostram que o desempenho da construção "for paralelo" melhorou à medida que o nível de paralelismo aumentou, alcançando o melhor tempo de execução na configuração com 8 *threads*. Esse comportamento é evidenciado na Figura 4.1, que apresenta os tempos médios de execução para diferentes configurações. Contudo, uma perda de desempenho foi observada na configuração com 16 *threads*, indicando que, nesse caso, o custo de gerenciamento de *threads* superou os ganhos esperados. Essa limitação decorre da divisão dos recursos dos núcleos, resultando em maior competição por recursos como largura de banda da memória e unidades de execução, o que reduz a eficiência.

Foram calculados os desvios padrões dos tempos de execução. Para a configuração com 1 *thread*, a média foi de 8.80 segundos, com um desvio padrão de 0.10 segundos. De forma semelhante, para as configurações com 2, 4, 8 e 16 *threads*, os tempos médios foram 5.62, 2.81, 1.60 e 1.89 segundos, com desvios padrões de 0.14, 0.08, 0.05 e 0.07 segundos, respectivamente.

Além da análise dos tempos de execução, o *speedup* entre configurações consecutivas de *threads* (1→2, 2→4, 4→8 e 8→16) foi calculado e é apresentado na Figura 4.2. O *speedup* avalia o ganho de desempenho ao aumentar o paralelismo, comparando o tempo de execução de uma configuração com menos *threads* em relação a outra com mais *threads*. Os resultados indicam um aumento significativo de desempenho ao passar de 1 para 4 *threads*, com ganhos mais modestos nas configurações subsequentes. No entanto, o *speedup* menor que 1 observado entre 8 e 16 *threads* demonstra que a execução com 16 *threads* foi mais lenta do que com 8 *threads*, refletindo os efeitos da sobrecarga no gerenciamento de *threads* e possíveis conflitos de recursos no *hardware*.

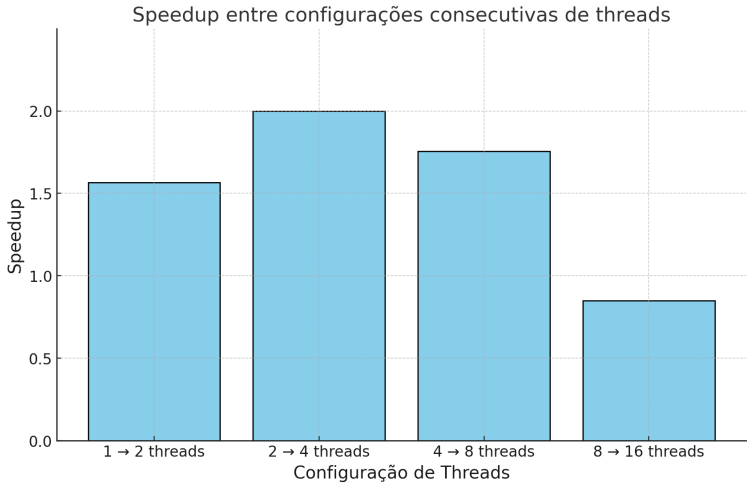


Figura 4.2: Speedup entre configurações consecutivas de *threads*.

Além da análise dos tempos de execução, o *speedup* em relação à execução sequencial (1→2, 1→4, 1→8 e 1→16) foi calculado e é apresentado na Figura 4.2. O *speedup* avalia o ganho de desempenho comparando o tempo de execução sequencial com o tempo de execução paralelo para diferentes configurações de *threads*. Os resultados demonstram que o aumento do número de *threads* proporciona ganhos significativos de desempenho, especialmente ao passar de 1 para 8 *threads*. No entanto, o *speedup* inferior ao esperado ao utilizar 16 *threads* reflete a sobrecarga no gerenciamento de *threads* e os limites impostos pelo *hardware*, resultando em um desempenho inferior ao obtido com 8 *threads*.

O *speedup* em relação à execução sequencial (1→2, 1→4, 1→8 e 1→16) foi calculado e é apresentado na Figura 4.3. Ele avalia o ganho de desempenho ao aumentar o número de *threads* em relação à execução sequencial. Os resultados mostram um aumento não linear em relação à quantidade de *threads* usadas. Esse comportamento pode ser explicado pela Lei de Amdahl, que estabelece que o *speedup* máximo de um sistema paralelo é limitado pela fração do programa que não pode ser paralelizada. Mesmo que partes significativas do programa sejam executadas de forma paralela, o desempenho é influenciado pelas porções sequenciais e pela sobrecarga adicional introduzida pela comunicação e sincronização entre *threads* (Amdahl, 1967). No caso analisado, os ganhos expressivos ao aumentar de 1 para 8 *threads* evidenciam a eficácia do paralelismo em partes paralelizáveis do código, enquanto o declínio observado ao utilizar 16 *threads* reflete os limites práticos do paralelismo conforme previsto pela Lei de Amdahl.

Os resultados confirmam que a construção "for paralelo" proporciona ganhos significativos de desempenho em cenários onde o número de *threads* é compatível com os recursos disponíveis no *hardware*. Em um cenário ideal, seria esperado um aumento linear de performance proporcional ao número de *threads*. Entretanto, fatores como a fração do programa que não pode

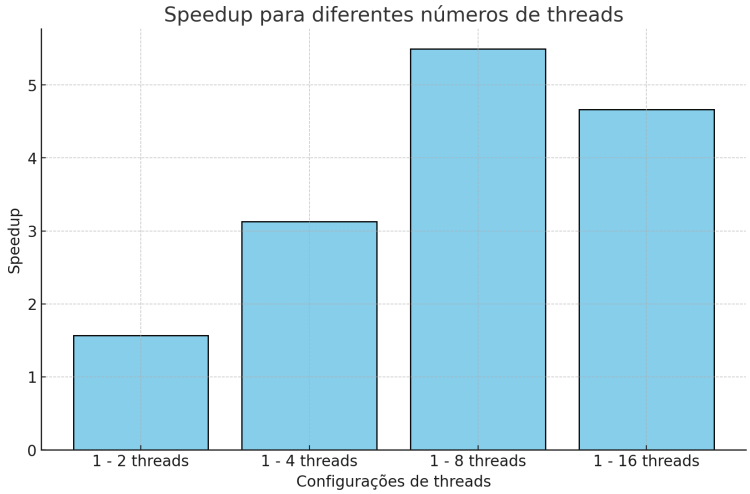


Figura 4.3: Speedup em relação à execução sequencial para diferentes configurações de *threads*.

ser paralelizada e os custos de comunicação e sincronização entre as *threads* explicam por que o desempenho real diverge de um modelo linear.

No geral, a análise dos resultados reforça a eficácia da construção "for paralelo" como uma ferramenta para explorar o paralelismo, destacando tanto os benefícios quanto as limitações impostas pelo *hardware* e pelas características do programa testado.

5 CONCLUSÃO

Este capítulo apresenta as conclusões do trabalho. Na Seção 5.1, são discutidos os tópicos abordados ao longo deste trabalho. A Seção 5.2 apresenta as conclusões gerais sobre o desempenho e os desafios enfrentados, além de sugestões para trabalhos futuros. Por fim, a Seção 5.3 aborda os aprendizados adquiridos durante o desenvolvimento do projeto.

Os códigos-fonte desenvolvidos estão disponíveis em repositórios no GitHub. O interpretador MEPA com suporte a *threads* pode ser acessado em (Soffner, 2024a), enquanto o compilador Pascal com suporte à construção *for paralelo* está disponível em (Soffner, 2024b).

5.1 TÓPICOS DE APRENDIZADOS

Ao longo deste trabalho, foram apresentados os fundamentos teóricos e práticos que sustentaram o desenvolvimento e a análise da construção "for paralelo" e suas extensões no compilador e interpretador MEPA.

No Capítulo 2, foram apresentados os conceitos teóricos essenciais que sustentaram este trabalho, incluindo definições sobre compiladores, interpretadores e paralelismo. Elementos como o funcionamento de compiladores, a estrutura da MV-MEPA e a utilização de *threads* para execução paralela, assim como as limitações impostas pelas dependências de dados, formaram a base para o desenvolvimento das funcionalidades propostas.

No Capítulo 3, foram discutidos os aspectos conceituais e técnicos da implementação, com destaque para os desafios enfrentados e as soluções desenvolvidas. Entre os tópicos abordados, estão a extensão da gramática para suportar a construção "for paralelo", a criação de novas instruções na MV-MEPA e a inclusão do suporte a matrizes, elementos que ampliaram significativamente as capacidades do compilador e do interpretador.

Por fim, no Capítulo 4, foi realizada uma análise experimental detalhada, onde os resultados dos testes mostraram os ganhos de desempenho proporcionados pela construção "for paralelo", bem como suas limitações em cenários com alta quantidade de *threads*. Foi analisado o impacto da sobrecarga no gerenciamento de *threads* e a importância de considerar as limitações do *hardware*.

5.2 CONCLUSÃO E TRABALHOS FUTUROS

Os resultados obtidos demonstraram que o desempenho do "for paralelo" seguiu um comportamento não linear, devido à sobrecarga associada ao gerenciamento de *threads*, como criação, sincronização e finalização. Os testes confirmaram que a construção "for paralelo" é uma solução viável e funcional para explorar o paralelismo em linguagens de programação.

Este trabalho desenvolveu um ambiente propício para estudos relacionados à paralelização com dependências, permitindo a realização de testes em construções paralelas dentro de um cenário controlado e bem definido. Esse ambiente facilitou a análise detalhada do impacto de dependências e da sobrecarga associada ao gerenciamento de *threads*. Essa adequação deve-se ao uso de uma máquina virtual simplificada e à possibilidade de realizar análises tanto no nível do compilador quanto diretamente na máquina virtual, proporcionando maior controle e flexibilidade para a exploração do paralelismo.

Como proposta para trabalhos futuros, sugere-se a implementação de mecanismos no compilador para a detecção automática de dependências dentro dos laços "for paralelo". Essa

funcionalidade poderia incluir a análise estática de dependências e a aplicação de estratégias para resolvê-las, como renomeação de variáveis, inserção de sincronizações ou até mesmo a transformação do código para adaptar o laço às limitações impostas pelas dependências identificadas. Essas melhorias tornariam a construção "for paralelo" mais robusta e aplicável a um conjunto mais amplo de cenários, ampliando o alcance e a eficiência do paralelismo em programas com maior complexidade.

5.3 APRENDIZADOS OBTIDOS DURANTE O DESENVOLVIMENTO

O desenvolvimento deste trabalho proporcionou uma compreensão aprofundada sobre os desafios técnicos envolvidos na implementação de paralelismo em compiladores e interpretadores. A utilização de ferramentas como *Bison* e *Flex* destacou a importância de técnicas robustas para análise léxica e sintática, enquanto a implementação de instruções específicas no interpretador MEPA evidenciou as complexidades de integrar novos conceitos a sistemas existentes. Além disso, a análise dos resultados experimentais permitiu refletir sobre os limites práticos do paralelismo e suas implicações no desempenho de sistemas paralelos. Esses aprendizados serão valiosos para projetos futuros na área de compiladores e arquiteturas paralelas.

Ademais, o desenvolvimento da extensão da gramática para incluir novas construções apresentou-se como um desafio técnico significativo. A necessidade de adaptar a gramática da linguagem Pascal e garantir a geração de código correto e sua execução adequada na MV-MEPA exigiu uma compreensão detalhada tanto das etapas de compilação quanto do funcionamento interno da máquina virtual. A própria construção "for paralelo" tornou-se um desafio por demandar a criação de uma gramática suficientemente expressiva que atendesse a todos os requisitos necessários para seu funcionamento, garantindo a integração adequada entre os diferentes componentes da gramática.

REFERÊNCIAS

Aho, A. V., Lam, M. S., Sethi, R. e Ullman, J. D. (2006). *Compiladores: Princípios, Técnicas e Ferramentas*. Pearson, 2nd edition.

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, 30:483–485.

Butenhof, D. R. (1997). *Programming with POSIX Threads*. Addison-Wesley.

GCC (2004). Changes in gcc 3.4. `https://gcc.gnu.org/gcc-3.4/changes.html`. Acessado em 16/11/2024.

GCC (2006). Changes in gcc 4.1. `https://gcc.gnu.org/gcc-4.1/changes.html`. Acessado em 16/11/2024.

Hennessy, J. L. e Patterson, D. A. (2012). *Instruction-Level Parallelism and Its Exploitation*, páginas 150–156. Morgan Kaufmann Publishers, 5th edition.

Kowaltowski, T. (1983). *Implementação de Linguagens de Programação*. Guanabara Dois.

Levine, J. R. (2009). *Flex & Bison*. O'Reilly Media.

Microsoft (2024). Parallel.for method. `https://learn.microsoft.com/pt-br/dotnet/api/system.threading.tasks.parallel.for`. Acessado em 5/12/2024.

Soffner, R. S. (2024a). MEPA Interpreter With Threads. `https://github.com/Casadororo/MEPA-Interpreter-With-Threads`. Acessado em 22/12/2024.

Soffner, R. S. (2024b). Pascal Compiler With Parallel For. `https://github.com/Casadororo/Pascal-Compiler-With-Parallel-For`. Acessado em 22/12/2024.