NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTAMENTO DE INFORMÁTICA

LUÍS MIGUEL FRADE FERREIRA MONTEIRO
Licenciado em Ciência e Engenharia Informática

PYTHON vs JULIA vs MATLAB PARA PROGRAMAÇÃO PARALELA

MESTRADO EM ENGENHARIA INFORMÁTICA

Universidade NOVA de Lisboa

Novembro, 2021

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTAMENTO DE INFORMÁTICA

PYTHON vs JULIA vs MATLAB PARA PROGRAMAÇÃO PARALELA

LUÍS MIGUEL FRADE FERREIRA MONTEIRO

Licenciado em Ciência e Engenharia Informática

Orientador: Vítor Manuel Alves Duarte,
Professor Auxiliar, FCT-NOVA

Júri:

Presidente: Miguel Jorge Tavares Pessoa Monteiro,
Professor Auxiliar, FCT-NOVA

Arguente: Luís Manuel da Costa Assunção,
Professor Adjunto, ISEL

Orientador: Vítor Manuel Alves Duarte,
Professor Auxiliar, FCT-NOVA

MESTRADO EM ENGENHARIA INFORMÁTICA

Universidade NOVA de Lisboa
Novembro, 2021

Python vs Julia vs Matlab para programação paralela

Copyright © Luís Miguel Frade Ferreira Monteiro, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa.

A Faculdade de Ciências e Tecnologia e a Universidade Nova de Lisboa têm o direito, perpétuo e sem limites geográficos, de arquivar e publicar esta dissertação através de exemplares impressos reproduzidos em papel ou de forma digital, ou por qualquer outro meio conhecido ou que venha a ser inventado, e de a divulgar através de repositórios científicos e de admitir a sua cópia e distribuição com objetivos educacionais ou de investigação, não comerciais, desde que seja dado crédito ao autor e editor.

Resumo

As linguagens de programação Python, Julia e Matlab são muito populares na computação científica, sendo utilizadas, em todo o mundo, para resolução de problemas de elevada complexidade podendo envolver cálculo numérico, resolução de sistemas de equações, efetuar simulações e otimizações, etc. Deste modo, estas linguagens são relevantes para engenheiros, economistas e cientistas a nível mundial, pelo que a sua utilidade se estende muito para além do meio informático.

O sucesso destas linguagens está ligado, entre outros fatores, à sua facilidade de aprendizagem e utilização, à qualidade (e quantidade) das bibliotecas providenciadas e à possibilidade de utilizar de forma eficiente os recursos disponíveis para resolução de problemas.

Outro aspeto relevante destas linguagens é a eficiência dos mecanismos de paralelismo oferecidos. O recurso à programação paralela tem emergido cada vez mais como um fator imprescindível na programação moderna como forma de melhorar a *performance* dos programas criados. Perante dúvidas levantadas sobre o desempenho destas linguagens relativamente às formas de paralelismo oferecidas, surgiu a necessidade de fazer uma análise e comparação das mesmas. Deste modo, os utilizadores poderão formar opiniões fundamentadas para decidir a que linguagem recorrer.

Para avaliar e comparar o desempenho de programas nas três linguagens recorremos, nesta dissertação, ao uso de *benchmarks* e *microbenchmarks* baseados no algoritmo de otimização DMS (utilizado no contexto em que o trabalho se realiza). Fazemos também uma análise dos benefícios e desvantagens de reescrever o algoritmo, inicialmente implementado em Matlab, para Python e Julia.

Palavras-chave: Paralelismo, Python, Julia, Matlab, Performance, Benchmarks, Microbenchmarks, Algoritmos de Otimização

ABSTRACT

Python, Julia, and Matlab are very popular in computer science worldwide for solving highly complex problems, including numerical calculation, resolution of equations systems, performing simulations and optimization, and so forth. These are essential languages for worldwide engineers, economists, and scientists, therefore used on vaster areas and not only computing.

These three languages' success is due, among other factors, to their programing and learning facility, the quality and quantity of libraries provided, and the possibility of efficiently using the available resources for problem-solving.

Furthermore, these languages have efficient mechanisms for parallel programming. The use of parallel programming became essential in modern computation as a way of increasing programs' performances. Due to doubts raised concerning these languages' performance regarding parallelism, the need to analyze and compare them emerged. This way, users can make reasoned decisions regarding which language to use.

To evaluate and compare the programs' performance in the three languages we use, in this dissertation, benchmarks and microbenchmarks based on the DMS optimization algorithm (in the context of this work). We also analyze the potential benefits and disadvantages of rewriting the optimization algorithm (initially implemented in Matlab) in Python and Julia.

Keywords: Parallelism, Python, Julia, Matlab, Performance, Benchmarks, Microbenchmarks, Optimization Algorithms

Conteúdo

Lista de Figuras xiii

Lista de Tabelas xvii

1. Introdução 1

2. Estudo das Linguagens Python, Matlab e Julia 5

2.1. Introdução às Linguagens 5
2.2. Python 7
2.2.1. Bibliotecas Relacionadas com Paralelismo 7
2.2.2. Modos de Paralelismo 8
2.2.2.1. Módulo Threading 8
2.2.2.2. Módulo Multiprocessing 11
2.2.2.3. Módulo Multiprocessing.sharedmemory 13
2.2.2.4. Módulo Concurrent.futures 13
2.2.2.5. Módulo Queue 14
2.3. Matlab 14
2.3.1. Bibliotecas Relacionadas com Paralelismo 15
2.3.1.1. Parallel Computing Toolbox 15
2.3.1.2. Matlab Parallel Server 16
2.3.2. Modos de Paralelismo 16
2.3.2.1. Ambiente baseado em Threads 16
2.3.2.2. Ambiente baseado em Processos 17
2.3.2.3. Escolha do Ambiente de Paralelismo 18
2.4. Julia 20
2.4.1. Bibliotecas Relacionadas com Paralelismo 20
2.4.2. Modos de Paralelismo 21
2.4.2.1. Multithreading 21
2.4.2.2. Computação Distribuída e Multiprocessing 22

3. Implementações 25

3.1. Introdução a Benchmarks e Microbenchmarks 25
3.2. Algoritmo DMS 29
3.2.1. Algoritmos de Otimização sem Derivadas 29
3.2.2. Estrutura do algoritmo DMS e Funções-Objetivo 31

3.3. Metodologia de Implementação 36
3.4. Implementação DMS Matlab 37
3.5. Implementação DMS Python 39
3.6. Implementação DMS Julia 43
3.7. Implementação Microbenchmarks 47

4. Avaliações **51**

4.1. Estratégia de Avaliação e Objetivos 51
4.2. Apresentação de Resultados 54
4.2.1. Benchmarks 55
4.2.1.1. Styrene 55
4.2.1.1.1. Multiprocessing 55
4.2.1.1.2. Multithreading 56
4.2.1.1.3. Sequential 57
4.2.1.1.4. Execução mais Eficiente de entre todos os Modos 58
4.2.1.2. Conjunto ZDT 59
4.2.1.2.1. Multiprocessing 60
4.2.1.2.2. Multithreading 66
4.2.1.2.3. Sequential 70
4.2.1.2.4. Execução mais Eficiente de entre todos os Modos 70
4.2.1.3. Comparações de performance com implementação base 71
4.2.2. Microbenchmarks 73
4.2.2.1. Multiprocessing 73
4.2.2.2. Multithreading 75
4.2.2.3. Sequential 77
4.2.2.4. Diferenças entre modos DMSInit e DriverInit: 78
4.2.2.4.1. Multiprocessing 78
4.2.2.4.2. Multithreading 79

5. Conclusões **81**

Referências **89**

Apêndices **97**

A. Resultados Experimentais **99**

A.1. Benchmarks 100
A.2. Microbenchmarks 110

B. Comparação dos Resultados Experimentais **117**

B.1. Benchmarks 117
B.1.1. Multiprocessing 117
B.1.2. Multithreading 123

B.1.3. Sequential 129
B.1.4. Todos os Modos 135
B.1.5. Estudo ZDT All Modes..160
B.2. Microbenchmarks..165
B.2.1. Python Multiprocessing Chunksizes....................................165
B.2.2. Comparação Microbenchmarks 166
B.2.3. Comparação DMSInit e DriverInit.....................................176

LISTA DE FIGURAS

FIGURA 1 - THREAD-BASED ENVIRONMENT [16] 17
FIGURA 2 - PROCESS-BASED ENVIRONMENT [16] 17
FIGURA 3 - ESCOLHA DE AMBIENTES DE PARALELISMO [16] 19
FIGURA 4 - BASE DE ALGORITMO DE BUSCA DIRECIONADA [35] 30
FIGURA 5 - BENCHMARKS STYRENE MULTIPROCESSING 55
FIGURA 6 - BENCHMARKS STYRENE MULTITHREADING 56
FIGURA 7 - BENCHMARKS STYRENE SEQUENTIAL 57
FIGURA 8 - BENCHMARKS STYRENE DMSINIT T1-T5 58
FIGURA 9 - BENCHMARKS ZDT1 MULTIPROCESSING 60
FIGURA 10 - BENCHMARKS ZDT2 MULTIPROCESSING 61
FIGURA 11 - BENCHMARKS ZDT3 MULTIPROCESSING 62
FIGURA 12 - BENCHMARKS ZDT4 MULTIPROCESSING 63
FIGURA 13 - BENCHMARKS ZDT6 MULTIPROCESSING 64
FIGURA 14 - BENCHMARKS ZDT1 MULTITHREADING 66
FIGURA 15 - BENCHMARKS ZDT2 MULTITHREADING 67
FIGURA 16 - BENCHMARKS ZDT3 MULTITHREADING 67
FIGURA 17 - BENCHMARKS ZDT4 MULTITHREADING 68
FIGURA 18 - BENCHMARKS ZDT6 MULTITHREADING 69
FIGURA 19 - BENCHMARKS ZDT1 E ZDT6 SEQUENTIAL 70
FIGURA 20 - MICROBENCHMARKS MULTIPROCESSING DMSINIT T1-T5 73
FIGURA 21 - MICROBENCHMARKS MULTIPROCESSING DRIVERINIT T1-T5 74
FIGURA 22 - MICROBENCHMARKS MULTITHREADING DMSINIT T1-T5 75
FIGURA 23 - MICROBENCHMARKS MULTITHREADING DRIVERINIT T1-T5 76
FIGURA 24 - MICROBENCHMARKS SEQUENTIAL T1-T5 77
FIGURA 25 - MICROBENCHMARKS MULTIPROCESSING - DMSINIT VS. DRIVERINIT - 20 COLS. 78
FIGURA 26 - MICROBENCHMARKS MULTITHREADING - DMSINIT VS. DRIVERINIT 79
FIGURA 27 - BENCHMARKS STYRENE MULTIPROCESSING 118

FIGURA 28 - BENCHMARKS ZDT1 MULTIPROCESSING....119
FIGURA 29 - BENCHMARKS ZDT2 MULTIPROCESSING....120
FIGURA 30 - BENCHMARKS ZDT3 MULTIPROCESSING....121
FIGURA 31 - BENCHMARKS ZDT4 MULTIPROCESSING....122
FIGURA 32 - BENCHMARKS ZDT6 MULTIPROCESSING....123
FIGURA 33 - BENCHMARKS STYRENE MULTITHREADING....124
FIGURA 34 - BENCHMARKS ZDT1 MULTITHREADING....125
FIGURA 35 - BENCHMARKS ZDT2 MULTITHREADING....126
FIGURA 36 - BENCHMARKS ZDT3 MULTITHREADING....127
FIGURA 37 - BENCHMARKS ZDT4 MULTITHREADING....128
FIGURA 38 - BENCHMARKS ZDT6 MULTITHREADING....129
FIGURA 39 - BENCHMARKS STYRENE SEQUENTIAL....130
FIGURA 40 - BENCHMARKS ZDT1 SEQUENTIAL....131
FIGURA 41 - BENCHMARKS ZDT2 SEQUENTIAL....132
FIGURA 42 - BENCHMARKS ZDT3 SEQUENTIAL....133
FIGURA 43 - BENCHMARKS ZDT4 SEQUENTIAL....134
FIGURA 44 - BENCHMARKS ZDT6 SEQUENTIAL....135
FIGURA 45 - BENCHMARKS STYRENE DMSINIT T1-T5....136
FIGURA 46 - BENCHMARKS STYRENE DMSINIT T2-T5....137
FIGURA 47 - BENCHMARKS STYRENE DRIVERINIT T1-T5....138
FIGURA 48 - BENCHMARKS STYRENE DRIVERINIT T2-T5....139
FIGURA 49 - BENCHMARKS ZDT1 DMSINIT T1-T5....140
FIGURA 50 - BENCHMARKS ZDT1 DMSINIT T2-T5....141
FIGURA 51 - BENCHMARKS ZDT1 DRIVERINIT T1-T5....142
FIGURA 52 - BENCHMARKS ZDT1 DRIVERINIT T2-T5....143
FIGURA 53 - BENCHMARKS ZDT2 DMSINIT T1-T5....144
FIGURA 54 - BENCHMARKS ZDT2 DMSINIT T2-T5....145
FIGURA 55 - BENCHMARKS ZDT2 DRIVERINIT T1-T5....146
FIGURA 56 - BENCHMARKS ZDT2 DRIVERINIT T2-T5....147
FIGURA 57 - BENCHMARKS ZDT3 DMSINIT T1-T5....148
FIGURA 58 - BENCHMARKS ZDT3 DMSINIT T2-T5....149
FIGURA 59 - BENCHMARKS ZDT3 DRIVERINIT T1-T5....150
FIGURA 60 - BENCHMARKS ZDT3 DRIVERINIT T2-T5....151
FIGURA 61 - BENCHMARKS ZDT4 DMSINIT T1-T5....152
FIGURA 62 - BENCHMARKS ZDT4 DMSINIT T2-T5....153

FIGURA 63 - BENCHMARKS ZDT4 DRIVERINIT T1-T5 154
FIGURA 64 - BENCHMARKS ZDT4 DRIVERINIT T2-T5 155
FIGURA 65 - BENCHMARKS ZDT6 DMSINIT T1-T5 156
FIGURA 66 - BENCHMARKS ZDT6 DMSINIT T2-T5 157
FIGURA 67 - BENCHMARKS ZDT6 DRIVERINIT T1-T5 158
FIGURA 68 - BENCHMARKS ZDT6 DRIVERINIT T2-T5 159
FIGURA 69 - MICROBENCHMARKS PYTHON MULTIPROCESSING - CHUNKSIZES 165
FIGURA 70 - MICROBENCHMARKS MULTIPROCESSING DMSINIT T1-T5 166
FIGURA 71 - MICROBENCHMARKS MULTIPROCESSING DMSINIT T2-T5 167
FIGURA 72 - MICROBENCHMARKS MULTIPROCESSING DRIVERINIT T1-T5 168
FIGURA 73 - MICROBENCHMARKS MULTIPROCESSING DRIVERINIT T2-T5 169
FIGURA 74 - MICROBENCHMARKS MULTITHREADING DMSINIT T1-T5 170
FIGURA 75 - MICROBENCHMARKS MULTITHREADING DMSINIT T2-T5 171
FIGURA 76 - MICROBENCHMARKS MULTITHREADING DRIVERINIT T1-T5 172
FIGURA 77 - MICROBENCHMARKS MULTITHREADING DRIVERINIT T2-T5 173
FIGURA 78 - MICROBENCHMARKS SEQUENTIAL T1-T5 174
FIGURA 79 - MICROBENCHMARKS SEQUENTIAL T2-T5 175
FIGURA 80 - MICROBENCHMARKS MULTIPROCESSING - DMSINIT VS. DRIVERINIT - 20 COLS. 177
FIGURA 81 - MICROBENCHMARKS MULTIPROCESSING - DMSINIT VS. DRIVERINIT - 100 COLS. 178
FIGURA 82 - MICROBENCHMARKS MULTIPROCESSING - DMSINIT VS. DRIVERINIT - 500 COLS. 179
FIGURA 83 - MICROBENCHMARKS MULTITHREADING - DMSINIT VS. DRIVERINIT 180

Lista de Tabelas

Tabela 1 - Styrene Multiprocessing - Speedup (vs. Matlab Sequential)................72
Tabela 2 - Parallel Implementations- Performance Ratio....................................72
Tabela 3 - Resultados Benchmarks Python Multiprocessing (1/2).......................100
Tabela 4 - Resultados Benchmarks Python Multiprocessing (2/2).......................101
Tabela 5 - Resultados Benchmarks Python Multithreading...............................102
Tabela 6 - Resultados Benchmarks Python Sequential 103
Tabela 7 - Resultados Benchmarks Julia Multiprocessing - DriverInit..............103
Tabela 8 - Resultados Benchmarks Julia Multiprocessing - DMSInit..................104
Tabela 9 - Resultados Benchmarks Julia Multithreading 105
Tabela 10 - Resultados Benchmarks Julia Sequential 105
Tabela 11 - Resultados Benchmarks Matlab Multiprocessing - DriverInit.......106
Tabela 12 - Resultados Benchmarks Matlab Multiprocessing - DMSInit...........107
Tabela 13 - Resultados Benchmarks Matlab Multithreading - DriverInit 108
Tabela 14 - Resultados Benchmarks Matlab Multithreading - DMSInit 109
Tabela 15 - Resultados Benchmarks Matlab Sequential 109
Tabela 16 - Resultados Microbenchmarks Python Multiprocessing (1/2)..........110
Tabela 17 - Resultados Microbenchmarks Python Multiprocessing (2/2)..........111
Tabela 18 - Resultados Microbenchmarks Python Multithreading..................112
Tabela 19 - Resultados Microbenchmarks Python Sequential 112
Tabela 20 - Resultados Microbenchmarks Julia Multiprocessing 113
Tabela 21 - Resultados Microbenchmarks Julia Multithreading........................114
Tabela 22 - Resultados Microbenchmarks Julia Sequential 114
Tabela 23 - Resultados Microbenchmarks Matlab Multiprocessing..................115
Tabela 24 - Resultados Microbenchmarks Matlab Multithreading 116
Tabela 25 - Resultados Microbenchmarks Matlab Sequential............................116
Tabela 26 - Resultados Benchmarks Multiprocessing: Styrene T1-T5 e T2-T5...118
Tabela 27 - Resultados Benchmarks Multiprocessing: ZDT1 T1-T5 e T2-T5........119
Tabela 28 - Resultados Benchmarks Multiprocessing: ZDT2 T1-T5 e T2-T5........120

TABELA 29 - RESULTADOS BENCHMARKS MULTIPROCESSING: ZDT3 T1-T5 E T2-T5........121
TABELA 30 - RESULTADOS BENCHMARKS MULTIPROCESSING: ZDT4 T1-T5 E T2-T5........122
TABELA 31 - RESULTADOS BENCHMARKS MULTIPROCESSING: ZDT6 T1-T5 E T2-T5........123
TABELA 32 - RESULTADOS BENCHMARKS MULTITHREADING: STYRENE T1-T5 E T2-T5....124
TABELA 33 - RESULTADOS BENCHMARKS MULTITHREADING: ZDT1 T1-T5 E T2-T5 125
TABELA 34 - RESULTADOS BENCHMARKS MULTITHREADING: ZDT2 T1-T5 E T2-T5 126
TABELA 35 - RESULTADOS BENCHMARKS MULTITHREADING: ZDT3 T1-T5 E T2-T5 127
TABELA 36 - RESULTADOS BENCHMARKS MULTITHREADING: ZDT4 T1-T5 E T2-T5 128
TABELA 37 - RESULTADOS BENCHMARKS MULTITHREADING: ZDT6 T1-T5 E T2-T5 129
TABELA 38 - RESULTADOS BENCHMARKS SEQUENTIAL: STYRENE....................................130
TABELA 39 - RESULTADOS BENCHMARKS SEQUENTIAL: ZDT1..131
TABELA 40 - RESULTADOS BENCHMARKS SEQUENTIAL: ZDT2..132
TABELA 41 - RESULTADOS BENCHMARKS SEQUENTIAL: ZDT3..133
TABELA 42 - RESULTADOS BENCHMARKS SEQUENTIAL: ZDT4..134
TABELA 43 - RESULTADOS BENCHMARKS SEQUENTIAL: ZDT6..135
TABELA 44 - RESULTADOS BENCHMARKS STYRENE DMSINIT T1-T5 136
TABELA 45 - RESULTADOS BENCHMARKS STYRENE DMSINIT T2-T5 137
TABELA 46 - RESULTADOS BENCHMARKS STYRENE DRIVERINIT T1-T5.............................138
TABELA 47 - RESULTADOS BENCHMARKS STYRENE DRIVERINIT T2-T5.............................139
TABELA 48 - RESULTADOS BENCHMARKS ZDT1 DMSINIT T1-T5 140
TABELA 49 - RESULTADOS BENCHMARKS ZDT1 DMSINIT T2-T5 141
TABELA 50 - RESULTADOS BENCHMARKS ZDT1 DRIVERINIT T1-T5 142
TABELA 51 - RESULTADOS BENCHMARKS ZDT1 DRIVERINIT T2-T5 143
TABELA 52 - RESULTADOS BENCHMARKS ZDT2 DMSINIT T1-T5 144
TABELA 53 - RESULTADOS BENCHMARKS ZDT2 DMSINIT T2-T5 145
TABELA 54 - RESULTADOS BENCHMARKS ZDT2 DRIVERINIT T1-T5 146
TABELA 55 - RESULTADOS BENCHMARKS ZDT2 DRIVERINIT T2-T5 147
TABELA 56 - RESULTADOS BENCHMARKS ZDT3 DMSINIT T1-T5 148
TABELA 57 - RESULTADOS BENCHMARKS ZDT3 DMSINIT T2-T5 149
TABELA 58 - RESULTADOS BENCHMARKS ZDT3 DRIVERINIT T1-T5 150
TABELA 59 - RESULTADOS BENCHMARKS ZDT3 DRIVERINIT T2-T5 151
TABELA 60 - RESULTADOS BENCHMARKS ZDT4 DMSINIT T1-T5 152
TABELA 61 - RESULTADOS BENCHMARKS ZDT4 DMSINIT T2-T5 153
TABELA 62 - RESULTADOS BENCHMARKS ZDT4 DRIVERINIT T1-T5 154
TABELA 63 - RESULTADOS BENCHMARKS ZDT4 DRIVERINIT T2-T5 155

TABELA 64 - RESULTADOS BENCHMARKS ZDT6 DMSINIT T1-T5 156
TABELA 65 - RESULTADOS BENCHMARKS ZDT6 DMSINIT T2-T5 157
TABELA 66 - RESULTADOS BENCHMARKS ZDT6 DRIVERINIT T1-T5 158
TABELA 67 - RESULTADOS BENCHMARKS ZDT6 DRIVERINIT T2-T5 159
TABELA 68 - PERFORMANCE COMPARISON - ALL MODES 160
TABELA 69 - RESULTADOS MICROBENCHMARKS: PYTHON MULTIPROCESSING - CHUNKSIZES..165
TABELA 70 - RESULTADOS MICROBENCHMARKS: MULTIPROCESSING DMSINIT T1-T5 166
TABELA 71 - RESULTADOS MICROBENCHMARKS: MULTIPROCESSING DMSINIT T2-T5 167
TABELA 72 - RESULTADOS MICROBENCHMARKS: MULTIPROCESSING DRIVERINIT T1-T5..168
TABELA 73 - RESULTADOS MICROBENCHMARKS: MULTIPROCESSING DRIVERINIT T2-T5..169
TABELA 74 - RESULTADOS MICROBENCHMARKS: MULTITHREADING DMSINIT T1-T5......170
TABELA 75 - RESULTADOS MICROBENCHMARKS: MULTITHREADING DMSINIT T2-T5......171
TABELA 76 - RESULTADOS MICROBENCHMARKS: MULTITHREADING DRIVERINIT T1-T5 ..172
TABELA 77 - RESULTADOS MICROBENCHMARKS: MULTITHREADING DRIVERINIT T2-T5 ..173
TABELA 78 - RESULTADOS MICROBENCHMARKS: SEQUENTIAL T1-T5...............................174
TABELA 79 - RESULTADOS MICROBENCHMARKS: SEQUENTIAL T2-T5...............................175
TABELA 80 - RESULTADOS MICROBENCHMARKS: MULTIPROCESSING - DMSINIT VS. DRIVERINIT - 20 COLS. 177
TABELA 81 - RESULTADOS MICROBENCHMARKS: MULTIPROCESSING - DMSINIT VS. DRIVERINIT - 100 COLS. 178
TABELA 82 - RESULTADOS MICROBENCHMARKS: MULTIPROCESSING - DMSINIT VS. DRIVERINIT - 500 COLS. 179
TABELA 83 - RESULTADOS MICROBENCHMARKS: MULTITHREADING - DMSINIT VS. DRIVERINIT...180

Capítulo 1

Introdução

As linguagens Python, Julia e Matlab são exemplos de linguagens de programação muito populares entre cientistas e engenheiros de diferentes áreas, mesmo fora do âmbito da Informática, para resolver diversos tipos de problemas. Estes problemas podem envolver cálculo numérico de grande complexidade, simulações de fenómenos, procura e otimização de soluções, entre outros. Python, Julia e Matlab são também reconhecidos pelas diversas funcionalidades que as suas bibliotecas oferecem, bem como pela facilidade de programação na resolução de problemas relacionados com *Big Data* e aprendizagem automática. Este tipo de problemas requerem uma elevada quantidade de recursos computacionais, sendo que o uso de métodos de paralelismo é fundamental para melhorar a sua eficiência. Este é um dos fatores que faz com que o estudo dos mecanismos de paralelismos oferecidos pelas linguagens seja muito relevante.

Devido à elevada popularidade e uso frequente destas linguagens para os objetivos descritos, surgiram questões acerca do seu desempenho na resolução de problemas de maior complexidade e dimensão. O desempenho obtido neste tipo de problemas, em particular problemas de otimização, está muitas vezes relacionado com a utilização de mecanismos de computação paralela. Esta Dissertação irá estudar este tipo de situações.

As três linguagens suportam diversos mecanismos de programação paralela, sendo estes diferentes na sua forma de utilização, nas bibliotecas que os suportam, na *performance* relativa a diferentes objetivos e na forma como os respetivos *pools* de *workers* são criados.

Esta Dissertação vem no seguimento do projeto BoostDFO [1], realizado no CMA/Departamento de Matemática em parceria com o Departamento de Informática da FCT/UNL e relativo à otimização de funções sem derivadas. O nosso objetivo é utilizar o algoritmo de otimização DMS, proveniente do projeto BoostDFO e escrito na linguagem Matlab, como base para testar a *performance* das linguagens Python, Julia e Matlab. A ideia para a realização deste trabalho originou no facto de os algoritmos de otimização implementados em Matlab, no projeto BoostDFO, demorarem por vezes diversas horas a executar, o que levantou a questão sobre se esses algoritmos não deveriam ser reescritos em outras linguagens de modo a estudar potenciais melhorias de desempenho provenientes das diferentes eficiências dos métodos de programação paralela. Esta questão é o foco principal desta Dissertação.

Na sequência dos objetivos desta Dissertação, será feita a avaliação e comparação de desempenho das implementações do algoritmo de otimização DMS nas linguagens Python, Julia e Matlab relativamente à sua *performance* utilizando múltiplas *threads* (paralelismo *multithreading*) e múltiplos processos (paralelismo *multiprocessing*). Este estudo permitirá retirar conclusões acerca da eficiência dos diferentes modos de paralelismo neste contexto e, deste modo, será um estudo muito relevante para os utilizadores das linguagens. Para avaliar o desempenho das linguagens nas características em estudo iremos recorrer a diferentes baterias de testes (*benchmarks*) bem como a testes de *performance* sobre as operações e primitivas relativas aos métodos de paralelismo oferecidos (*microbenchmarks*).

Relativamente à forma como a informação está organizada no Relatório:

- O capítulo "Estudo das Linguagens Python, Matlab e Julia" contém informação relativa ao estudo de cada uma das linguagens. Inclui uma descrição geral das linguagens, alguns dos seus aspetos mais relevantes e informação sobre as principais bibliotecas relacionadas com programação paralela. Este estudo analisa também diversos métodos de paralelismo disponíveis em cada linguagem, não estando limitado apenas aos métodos utilizados nas implementações feitas nesta Dissertação;

- O capítulo "Implementações" começa com uma introdução aos conceitos de *benchmarks* e *microbenchmarks*, assim como uma breve descrição de alguns *benchmarks* populares estudados como inspiração para as implementações feitas

nesta Dissertação. Em seguida analisamos o algoritmo de otimização DMS, indicando os diversos ficheiros que o constituem e as funções-objetivo implementadas e utilizadas pelo algoritmo durante a sua execução. Após isso é feita uma descrição da metodologia de implementação do algoritmo nas diferentes linguagens e são indicados os detalhes e comparações mais relevantes observados nas diferentes implementações, bem como as formas de paralelismo utilizadas nas execuções *multithreading* e *multiprocessing*. Por fim, são detalhados os ficheiros relativos aos *microbenchmarks* implementados, com o objetivo de estudar de forma isolada os mecanismos de paralelismo utilizados na implementação do algoritmo DMS em cada linguagem;

- O capítulo "Avaliações" descreve as estratégias de avaliação e organização de dados utilizadas na análise dos *benchmarks* e *microbenchmarks,* as condições em que a extração de resultados foi efetuada e os principais objetivos dos estudos realizados. Em seguida são apresentadas as análises, comparações e conclusões relativas a todos os modos de execução, para todas as funções-objetivo utilizadas e em todas as linguagens;

- O capítulo "Conclusões" consiste numa análise final da Dissertação, num resumo dos principais objetivos alcançados, na exposição dos principais resultados obtidos e na descrição de algumas sugestões relativas a trabalhos futuros que possam ser realizados na sequência das descobertas feitas durante a elaboração desta Dissertação.

CAPÍTULO 2

ESTUDO DAS LINGUAGENS PYTHON, MATLAB E JULIA

2.1. Introdução às Linguagens

As linguagens Python, Julia e Matlab têm bastante relevância em diferentes áreas da ciência, economia e engenharia [2][32][33]. A sua utilização estende-se para além do meio informático, sendo muito frequente no domínio da computação científica, resolução de problemas envolvendo cálculo numérico, processamento de dados e simulação de fenómenos físicos, entre outros. É comum estas linguagens serem ensinadas e utilizadas nas universidades, nas mais variadas áreas da ciência e engenharia, assim como em economia e em outros contextos.

Das três linguagens em estudo, Julia e Python são linguagens *open-source* (ou seja, o seu código está disponível de forma gratuita e têm licença para eventual edição e redistribuição, entre outras características [3]), contrariamente à linguagem Matlab que é *closed-source* e em que algumas das suas funcionalidades (*toolboxes*) são vendidas separadamente do ambiente e compilador base. Deste modo, sendo que Julia e Python disponibilizam as suas ferramentas de forma gratuita, a facilidade de acesso ajuda à transparência e reprodutibilidade do código (ou seja, a possibilidade de um *third-party* conseguir obter um resultado igual ao de uma experiência publicada, baseando-se na informação fornecida, quando a experiência é replicada nas mesmas condições [4]). Também existe uma implementação *open-source* que procura ser compatível com Matlab, o Octave.

Python e Matlab são linguagens interpretadas e Julia é uma linguagem compilada [38]. No entanto, Julia utiliza compilação *just-in-time* (JIT) o que permite compilar o código em *runtime* ao invés de compilar o código antes da execução (o que acontece, por exemplo, no caso das linguagens compiladas C e Fortran). Isto faz com que Julia tenha semelhanças a uma linguagem interpretada, mas usufrua de benefícios das linguagens compiladas, como a rapidez de execução [36].

Julia é, também, a linguagem que foi criada mais recentemente de entre as três, e alguns artigos [2] afirmam que é a linguagem mais eficiente relativamente à execução de algoritmos que utilizem *solvers* numéricos (isto é, esquemas numéricos para resolução de equações diferenciais [5]).

Relativamente à linguagem Python, esta beneficia de um elevado número de bibliotecas e módulos, disponibilizados pela equipa que desenvolve a linguagem ou implementados e oferecidos pela comunidade de utilizadores Python, o que facilita a obtenção das ferramentas necessárias para vários tipos de problemas.

No que respeita aos benefícios e características da linguagem Matlab, esta usufrui de extensa documentação e de várias *built-in toolboxes*, não sendo necessário recorrer a bibliotecas externas para atingir os objetivos pretendidos, na maioria das situações.

Alguns artigos [6] afirmam que, apesar de as linguagens em estudo serem bastante úteis para a resolução de problemas com complexidade elevada, historicamente, apresentam problemas de *performance* e carecem do suporte necessário para a obtenção de programação paralela eficiente. Este tipo de estudos reforça a necessidade das análises e comparações de *performance* que serão efetuadas nesta Dissertação.

Uma característica comum às três linguagens, de grande relevância no contexto deste trabalho, é a de todas suportarem mecanismos de programação paralela com múltiplas *threads* (*multithreading*) e com múltiplos processos (*multiprocessing*).

2.2. Python

Python [2] [6] [7] é uma linguagem de programação *open-source* que ganhou bastante popularidade na comunidade científica (sobretudo a partir da versão 2.7), devido à facilidade de programação e à disponibilidade de várias bibliotecas *performance-oriented* como NumPy [67], SciPy [68] e TensorFlow [69]. As versões estáveis mais recentes (até ao momento da realização deste relatório) são a 3.10, lançada a 4 de Outubro de 2021 e a 3.9.8, lançada a 5 de Novembro de 2021 [37].

A linguagem Python é interpretada (isto é, tira partido de um interpretador, poupando tempo considerável durante o desenvolvimento de programas, pois não requer compilação ou *linking* [8]) e dinâmica (ou seja, é possível a alteração de código e estruturas lógicas em *runtime*, contrariamente às linguagens estáticas [9]), permitindo também a sua utilização como linguagem de *scripting*. Python é considerada, também, uma linguagem de alta produtividade devido à sua sintaxe clara e concisa e ao elevado número de bibliotecas de que dispõe.

Relativamente a programação paralela, Python suporta, quer na *Standard Library* quer em diversas outras bibliotecas, mecanismos para efetuar paralelismo utilizando múltiplas *threads* (*multithreading*) e múltiplos processos (*multiprocessing*).

2.2.1. Bibliotecas Relacionadas com Paralelismo

A linguagem Python é famosa pelo elevado número de bibliotecas de que dispõe, tendo, no final do ano 2020, mais de 137 000 bibliotecas disponíveis para uso [11]. Cada uma destas bibliotecas tem funcionalidades e vantagens diferentes, oferecendo uma variada e extensa lista de opções de acordo com as preferências do utilizador e os problemas que pretende abordar.

Existem diversas bibliotecas, módulos e *frameworks* que oferecem métodos de paralelismo [10], como é o caso de *MPI for Python* [31], Dispy [52] e Ray [53]. No contexto deste trabalho, iremos focar-nos nos mecanismos de paralelismo disponibilizados pela *Python Standard Library*, uma vez que esta é a biblioteca utilizada na fase de implementação desta Dissertação.

2.2.2. Modos de Paralelismo

A *Python Standard Library* [12] disponibiliza diversos módulos relacionados com situações de paralelismo e concorrência, sendo que, no contexto deste trabalho, nos iremos focar primariamente nos módulos *threading* e *multiprocessing*, fazendo também referência a algumas funcionalidades dos módulos *multiprocessing-sharedmemory*, *concurrent.futures* e *queue* (que estão relacionados com os principais módulos).

Em seguida, faremos uma breve descrição de cada um destes módulos, bem como das características mais relevantes no contexto de programação paralela.

2.2.2.1. Módulo Threading

O módulo *threading* [54] disponibiliza interfaces de alto-nível, atuando numa camada de nível superior ao módulo *_thread* [70], que corresponde a um módulo de interfaces de baixo-nível (de interface com o Sistema de Operação).

Neste módulo, são definidas funções de controlo de *Threads*, bem como objetos e classes do mesmo tipo. É interessante referir que este módulo foi parcialmente baseado no modelo de *threading* da linguagem Java, tendo, no entanto, uma implementação diferente relativamente a alguns dos seus processos e objetos, entre outras diferenças.

Algumas classes e objetos relevantes deste módulo são:

- **Thread Local Data**

  Thread Local Data corresponde a informação que é específica e exclusiva a cada *thread*. Uma nova instância local pode ser criada através do uso da classe *threading.local*, e os valores armazenados nessa instância são privados à *thread* em questão. *Threads* diferentes podem ter valores diferentes, para cada uma das instâncias existentes.

- **Objetos Thread**

  A atividade de um objeto *Thread* é inicializada ao chamar o método start(). Este invoca, por sua vez, o método run() numa nova *thread* de controlo. A partir deste momento, a *thread* é considerada como estando *alive*. A *thread* deixa de estar *alive* quando o seu método run() é terminado.

Threads podem invocar o método join() de outras *threads*. Quando isto acontece, a *thread* que invocou o método fica bloqueada (em espera), até a *thread* indicada terminar.

É relevante referir, também, o conceito de *daemon threads*. Este tipo de *threads* [55] executa em *background* e os seus recursos são automaticamente libertados quando a sua execução termina. Um programa Python termina quando apenas existem *daemon threads* em execução. A *main thread* (*thread* inicial do programa Python) nunca é considerada como *daemon thread*.

- **Objetos Lock**

  Este tipo de *locks*, em Python, corresponde às primitivas de sincronização de nível mais baixo que estão disponíveis, sendo implementadas pelo módulo *_thread*. Este tipo de objetos é partilhado por todas as *threads* do programa, permitindo a sua coordenação.

  Cada um destes objetos tem dois estados, "*locked*" e "*unlocked*", sendo que no momento da sua criação, o seu estado inicial corresponde ao estado *unlocked*. Estes objetos têm apenas dois métodos, acquire() e release(). Estes métodos permitem adquirir e libertar um *lock* de forma atómica. Se uma *thread* tentar adquirir um *lock* que se encontre no estado "*locked*", ficará bloqueada até que o *lock* se encontre disponível. A utilização do método release() quando o *lock* se encontra no estado "*unlocked*" resulta no lançamento de uma excepção.

  Quando existe mais do que uma *thread* bloqueada à espera de um *lock*, apenas uma das *threads* poderá adquiri-lo quando este ficar disponível. Não é especificado o processo de seleção da *thread* que obtém o *lock*, variando com as implementações em uso.

- **Objetos RLocks**

  Uma *lock* reentrante (*Rlock*) é uma primitiva de sincronização que pode ser adquirida múltiplas vezes pela mesma *thread* e ter diversos níveis de recursão. A principal diferença entre este objeto e o objeto *Lock* descrito anteriormente passa por a *thread* que adquire o *lock* poder usar novamente o método acquire(). O *lock* adquirido será apenas libertado quando a *thread*

utilizar o método release() em número equivalente a cada um dos níveis de recursão.

- **Objetos de Condição**

  As variáveis de condição estão sempre relacionadas com algum tipo de *lock*. Podem ser passadas explicitamente, ou ser criadas por predefinição.

  O estilo de programação típico relacionado com variáveis de condição recorre a *locks* para sincronizar o acesso a um estado partilhado por diversas *threads*. As *threads* interessadas num determinado estado devem chamar o método wait() repetidamente até observarem o estado pretendido. Este método apenas pode ser chamado quando a *thread* adquiriu previamente o *lock*, momento em que o liberta e bloqueia, ficando em modo de espera até que outra *thread* a notifique da mudança de estado pretendida através dos métodos notify() ou notify_all(), altura em que a thread readquire a *lock* e retorna.

  Do mesmo modo, *threads* podem chamar os métodos notify() ou notify_all() quando alteram o estado partilhado, de modo a notificar *threads* que estejam à espera de observar essa alteração de estado.

- **Objetos Semáforo**

  Semáforos são das primitivas de sincronização mais antigas na história da ciência da computação [30].

  Semáforos são os objetos responsáveis pela gestão de um contador interno que é decrementado de cada vez que o método acquire() é invocado, e incrementado de cada vez que é usado o método release(). O contador nunca gera valores negativos, sendo que se uma *thread* invoca o método acquire() num momento em que o contador está a zero, essa *thread* fica bloqueada até que outra *thread* utilize o método release() e, consequentemente, incremente o valor do contador.

- **Objetos Evento**

  Eventos são um dos métodos mais simples para comunicação e sincronização entre *threads*: de forma resumida, uma *thread* anuncia um evento e as outras *threads* envolvidas esperam até que esse evento ocorra.

Um Evento gere uma *flag* que tem os valores *true* e *false*. Essa *flag* fica com valor *true* através da utilização do método set(), e retorna ao valor *false* através o método clear(). O método wait() deste objeto bloqueia a *thread* que o invoca até que o valor da *flag* seja *true*. A *flag* é inicializada com valor *false*.

- **Objetos Barreira**

  Barreiras são primitivas de sincronização simples que devem ser utilizadas quando se pretende que um grupo de *threads* espere umas pelas outras. Cada uma das *threads* tenta passar a barreira invocando o método wait(), mas apenas conseguem prosseguir quando todas as *threads* do conjunto tiverem invocado esse método. Apenas nesse momento todas as *threads* são libertadas e autorizadas a continuar a sua execução.

2.2.2.2. Módulo Multiprocessing

O módulo *multiprocessing* [56] oferece mecanismos para lidar com concorrência local e remota e permite ao programador tirar o melhor partido dos diversos processos da máquina. O módulo está disponível quer em Unix, quer em Windows.

Os módulos *Multiprocessing.sharedmemory*, *Queue* e *concurrent.futures* estão também relacionados com multiprocessamento e serão referidos em secções seguintes.

No módulo *multiprocessing* os processos são inicializados através da criação de objetos *process* e pela utilização do método **start()**.

O módulo *multiprocessing* oferece três maneiras distintas de inicializar um processo (*spawn*, *fork*, *forkserver*), dependendo da plataforma:

- Com *spawn*, o processo-pai é responsável por criar o novo processo, sendo que este apenas herda os recursos necessários para correr o seu método run(). Este modo de criação de processos está disponível nos ambientes Unix e Windows, mas é mais lento quando comparado com os outros dois modos referidos;
- Usando o método *fork*, o processo-pai utiliza uma função (os.fork()) para fazer *fork* do interpretador de Python. Deste modo, o processo-

filho, quando é inicializado, é idêntico ao processo pai e herda todos os seus recursos. Este método está apenas disponível em ambiente Unix;

- Com *forkserver*, um processo servidor é inicializado. A partir desse ponto, sempre que é necessária a criação de um novo processo, o processo pai faz um pedido ao servidor para que este faça *fork* do novo processo. Este método está apenas disponível em ambiente Unix.

- **Troca de Objetos entre Processos**

Este módulo suporta *Queues* e *Pipes* para comunicação e troca de objetos entre processos.

Queues são *thread-safe* e *process-safe* (referidas mais em detalhe na secção "Módulo Queue"). Os *pipes* retornam um par de objetos de conexão (objetos que permitem enviar e receber objetos ou strings e que podem ser vistos como *sockets* conectadas e orientadas para mensagens) que representam as duas extremidades do *pipe*. Cada objeto de conexão tem os métodos send() e recv(). É importante ter em consideração que a informação no *pipe* pode ficar corrompida se dois processos (ou *threads*) tentarem simultaneamente escrever (ou ler) da mesma extremidade do *pipe*.

- **Sincronização entre Processos**

Relativamente à sincronização entre processos, este módulo possui primitivas de sincronização equivalentes às do módulo *threading* (*Locks*, Semáforos, Barreiras, entre outros).

- **Estados partilhados entre Processos**

O uso de estados partilhados é particularmente desencorajado quando se trabalha com múltiplos processos [57]. No entanto, quando necessário, este módulo oferece algumas maneiras de o fazer. A primeira é através do uso de memória partilhada (referido na secção 2.2.2.3), recorrendo ao uso dos objetos *Value* e *Array*. A segunda forma é através de um processo servidor em que um objeto gestor (*Manager*) controla o processo, contendo os objetos a partilhar, e permite que outros processos interajam com os objetos

através de *proxies* cujas operações desencadeiam pedidos remotos ao respetivo servidor (*proxies* têm referências para objetos partilhados que se encontram em diferentes processos, sendo que podem existir diferentes *proxies* a referir o mesmo objeto). Esta segunda opção é mais flexível do que o uso de memória partilhada devido a suportar tipos arbitrários de objetos.

- **Utilização de um Pool de Workers**

  Um *pool* de processos ou *threads* é representado pela classe *Pool* e permite que *tasks* sejam delegadas para os processos de diferentes formas. É importante notar que os métodos do *pool* devem apenas ser utilizados pelo processo que o criou.

2.2.2.3. Módulo Multiprocessing.sharedmemory

Este módulo [58] é responsável pela alocação e gestão de memória partilhada destinada a ser utilizada por um ou múltiplos processos de uma máquina.

O tipo de memória partilhada oferecida por este módulo permite que processos distintos possam escrever e ler de uma região partilhada da memória volátil. Esta característica é bastante útil uma vez que, convencionalmente, os processos estão limitados a aceder apenas aos seus espaços de memória. Assim, a memória partilhada permite evitar a cópia de dados e o envio de mensagens entre processos contendo a informação a partilhar. O processo de partilha de informação diretamente via memória proporciona também melhorias de *performance* significativas em comparação com partilha de informação através de discos ou outras formas de comunicação que requerem a serialização e cópia da informação a partilhar.

Para auxiliar na gestão dos ciclos de vida da memória partilhada entre processos distintos, este módulo utiliza o *SharedMemoryManager*.

2.2.2.4. Módulo Concurrent.futures

Este módulo [59] oferece uma interface de alto-nível para executar assincronamente tarefas concorrentes.

A classe abstrata Executor oferece métodos para executar chamadas assíncronas. Não deve ser usado de forma direta, sendo recomendado o uso de uma das suas subclasses concretas (*ThreadPoolExecutor* e *ProcessPoolExecutor*):

- ThreadPoolExecutor: Utiliza um *pool* de *threads* para executar chamadas assíncronas. É necessário ter em atenção que podem ocorrer situações de *deadlock* quando um *callable* associado a um objeto *future* necessita de esperar pelos resultados de outro objeto f*uture;*
- ProcessPoolExecutor: Utiliza um *pool* de processos para executar chamadas assíncronas. É usado em conjunto com o módulo *multiprocessing* de forma a contornar o *Global Interpreter Lock* (*GIL*).

- **Objetos Future**

  A classe Future encapsula a execução assíncrona de um *callable*. As instâncias *future* são criadas através do método Executor.submit() e permitem controlar a execução das tarefas, recolhendo, mais tarde, o seu resultado.

2.2.2.5. Módulo Queue

O módulo Queue [60] implementa objetos *queue* que permitem a troca de informação entre múltiplos produtores e consumidores. É especialmente relevante nos casos em que informação deve ser transmitida de forma segura entre múltiplas *threads*.

Este módulo implementa três tipos de *queues* (*Queue*, *LifoQueue*, *PriorityQueue*) que diferem na ordem pela qual a informação é devolvida. Na *Queue*, a primeira *task* adicionada é a primeira a ser devolvida. Na *LifoQueue* a última entrada adicionada é a primeira a ser retornada. Na *PriorityQueue* as entradas são ordenadas por prioridade, sendo desenvolvida primeiro a de valor mais baixo. Estes três tipos de *queues* utilizam *locks* internamente com o objetivo de bloquear temporariamente *threads* concorrentes, quando necessário.

2.3. Matlab

Matlab é a linguagem de programação da MathWorks. É caracterizada por ser *closed-source*, dinâmica, beneficiar de extensa documentação e ter diversas *built-in toolboxes*. É uma linguagem com "*batteries included*", isto é, oferece um ambiente de edição e execução de código, capacidades gráficas e um catálogo extenso de rotinas numéricas. Esta linguagem dispõe de um elevado número de funcio-

nalidades diferentes e alguns artigos afirmam que, atualmente, Matlab é a linguagem de programação mais utilizada por economistas a nível mundial [2]. Esta linguagem é também considerada dominante em diversas áreas da ciência para realização de computações numéricas [31].

Matlab oferece suporte para paralelismo *multiprocessing* e *multithreading*, sendo que as funcionalidades disponíveis para ambientes *multithreaded* são mais limitadas que as disponíveis em ambiente *multiprocessing*.

2.3.1. Bibliotecas Relacionadas com Paralelismo

Os principais produtos Matlab relacionados com paralelismo são a *Parallel Computing Toolbox* e o *Matlab Parallel Server*. Estes serviços têm objetivos distintos dentro do contexto de programação paralela e oferecem diferentes vantagens e funcionalidades, mas o seu uso está frequentemente relacionado.

De referir que existem mais produtos em Matlab, como a *Image Processing Toolbox*, que incluem funcionalidades nas quais o paralelismo *multithreading* é integrado de base.

2.3.1.1. Parallel Computing Toolbox

A *Parallel Computing Toolbox* [13][14] permite resolver problemas computacionais com elevadas quantidades de informação utilizando processadores *multicore*, GPUs e *clusters* computacionais. Mecanismos de alto-nível como *for-loops* paralelos, certos tipos especiais de *arrays* e algoritmos de paralelismo para operações numéricas permitem paralelizar aplicações Matlab sem recorrer à programação em CUDA, OpenMP ou Pthreads.

Esta *toolbox* também permite paralelizar funções em outras *toolboxes*. É ainda possível utilizar este produto em conjunto com *Simulink* de modo a executar múltiplas simulações de um modelo em paralelo. Estes programas e modelos podem correr tanto em modo interativo como em modo *batch*.

É possível utilizar esta *toolbox* em conjunto com o *Matlab Parallel Server* de modo a executar aplicações em *clusters* ou *clouds* sem serem necessárias alterações ao seu código inicial.

Outros benefícios do uso da *Parallel Computing Toolbox* são:

- Acelerar a execução de programas através da utilização de ferramentas de computação paralela como *parfor* e *parfeval;*
- Fazer *scaling* de computações usando ferramentas de processamento de *Big Data* como *distributed, tall, datastore* e *mapreduce;*
- Acelerar cálculos no GPU do computador através do uso da ferramenta *gpuArray*;
- Realização de cálculos em *clusters* e *clouds* através da ferramenta *batch*.

2.3.1.2. Matlab Parallel Server

O *Matlab Parallel Server* [15] é usado em conjunto com o *Parallel Computing Toolbox* de forma a permitir que aplicações possam executar em *cluster* e *clouds* sem alterações ao seu código, assim como executar cálculos matriciais de dimensão elevada, de modo a não ter de recorrer apenas à memória de uma única máquina. Para além desta funcionalidade, *Matlab Parallel Server* também suporta *batch jobs* e computações paralelas interativas.

O processo relacionado com o licenciamento no *cluster* é tratado na totalidade pelo *Matlab Parallel Server*. Também não é necessário fornecer licenças Matlab para o *cluster*, uma vez que a licença do utilizador está dinamicamente ligada ao mesmo. Esta licença inclui *features* que suportam *scaling* ilimitado.

2.3.2. Modos de Paralelismo

Utilizando o *Parallel Computing Toolbox* é possível recorrer a ambientes de programação [16] baseados em *threads* (*multithreading*) e em processos (*multiprocessing)* para executar código com paralelismo. Cada um destes ambientes tem vantagens diferentes, mas é importante ter em atenção que, no caso dos ambientes baseados em *threads*, estes apenas suportam um pequeno conjunto das funções e mecanismos disponibilizados para ambientes baseados em processos.

2.3.2.1. Ambiente baseado em Threads

Neste tipo de ambientes, as *features* de paralelismo são executadas em *thread workers*. As *threads* coexistem dentro do mesmo processo e podem partilhar memória, como é exemplificado na figura 1.

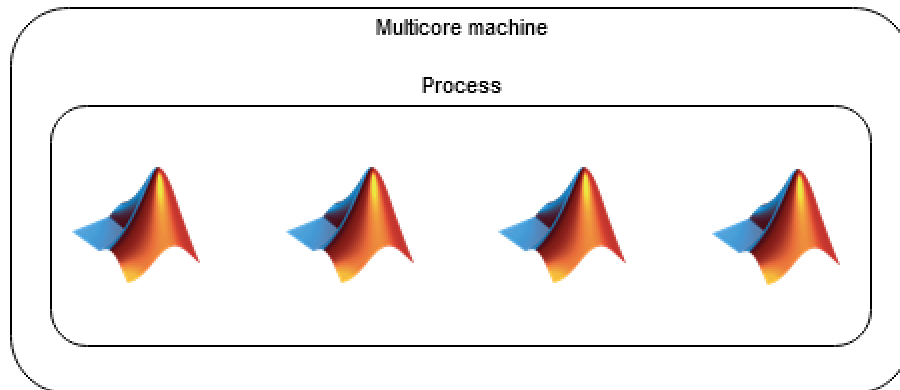


Figura 1 - Thread-based environment [16]

Os ambientes baseados em *threads* têm algumas vantagens relativamente aos ambientes baseados em processos:

- São mais eficientes no que diz respeito a memória, uma vez que podem aceder a informação numérica sem ter necessidade de a copiar, devido aos *thread workers* poderem partilhar memória.
- A comunicação entre *threads* consome menos tempo do que a comunicação entre processos. O *overhead* de agendamento de *tasks* e de comunicações entre *workers* é menor.

2.3.2.2. Ambiente baseado em Processos

Nos ambientes baseados em processos, as *features* de paralelismo são executadas em *process workers*. Contrariamente às *threads*, os processos são independentes entre si. Este ambiente é exemplificado na figura 2.

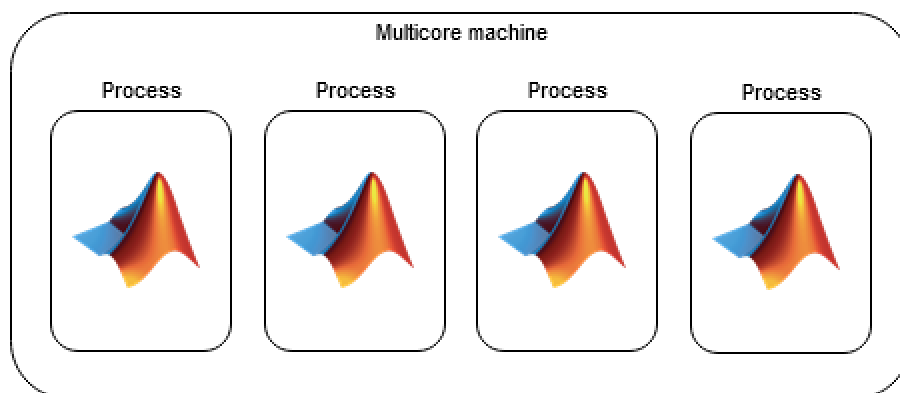


Figura 2 - Process-based environment [16]

Algumas vantagens deste tipo de ambientes relativamente aos ambientes baseados em *threads*, são:

- Suportam todas as *features* da linguagem Matlab;

- São mais robustos na eventualidade de *crashes*. Se um *process worker* tem um *crash*, mas não utiliza *spmd* ou *arrays* distribuídos, os outros processos podem continuar a correr sem problemas associados (pois são independentes);
- Possibilidade de usar *features* de *clusters* como o *batch*;

2.3.2.3. Escolha do Ambiente de Paralelismo

A figura 3 representa um esquema para determinar o tipo de ambiente de programação paralela que o utilizador deve escolher:

- ambiente baseado em *threads*, a correr em máquina local;
- ambiente baseado em processos, a correr em máquina local;
- ambiente baseado em processos, a correr em cluster remoto.

A escolha do ambiente mais indicado é influenciada por diversos fatores: se o utilizador pretende, ou não, executar o código em múltiplas máquinas; se o utilizador deseja executar o programa sem efetuar alterações ao mesmo; se o programa a executar implica a transferência de quantidades elevadas de informação.

Nestas condições, é recomendado:

- Utilizar o ambiente baseado em *threads* nos casos em que se pretende reduzir, tanto quanto possível, a memória utilizada, fazer um *scheduling* mais rápido, diminuir os custos de transferência de informação e tirar partido dos vários *cores* disponíveis. A criação do *pool* de *thread workers* é feita utilizando o comando *parpool*('threads');
- Utilizar o ambiente baseado em processos, em máquina local, nos casos em que a divisão em *threads* ou a partilha de dados possa necessitar de alterações ao código, mas a divisão em processos permita tirar melhor partido dos *cores* disponíveis, e também quando se pretende realizar prototipagem antes de fazer *scaling* do código para *clusters* ou *clouds.* A criação do *pool* de *process workers* é feita utilizando o comando *parpool*('local');

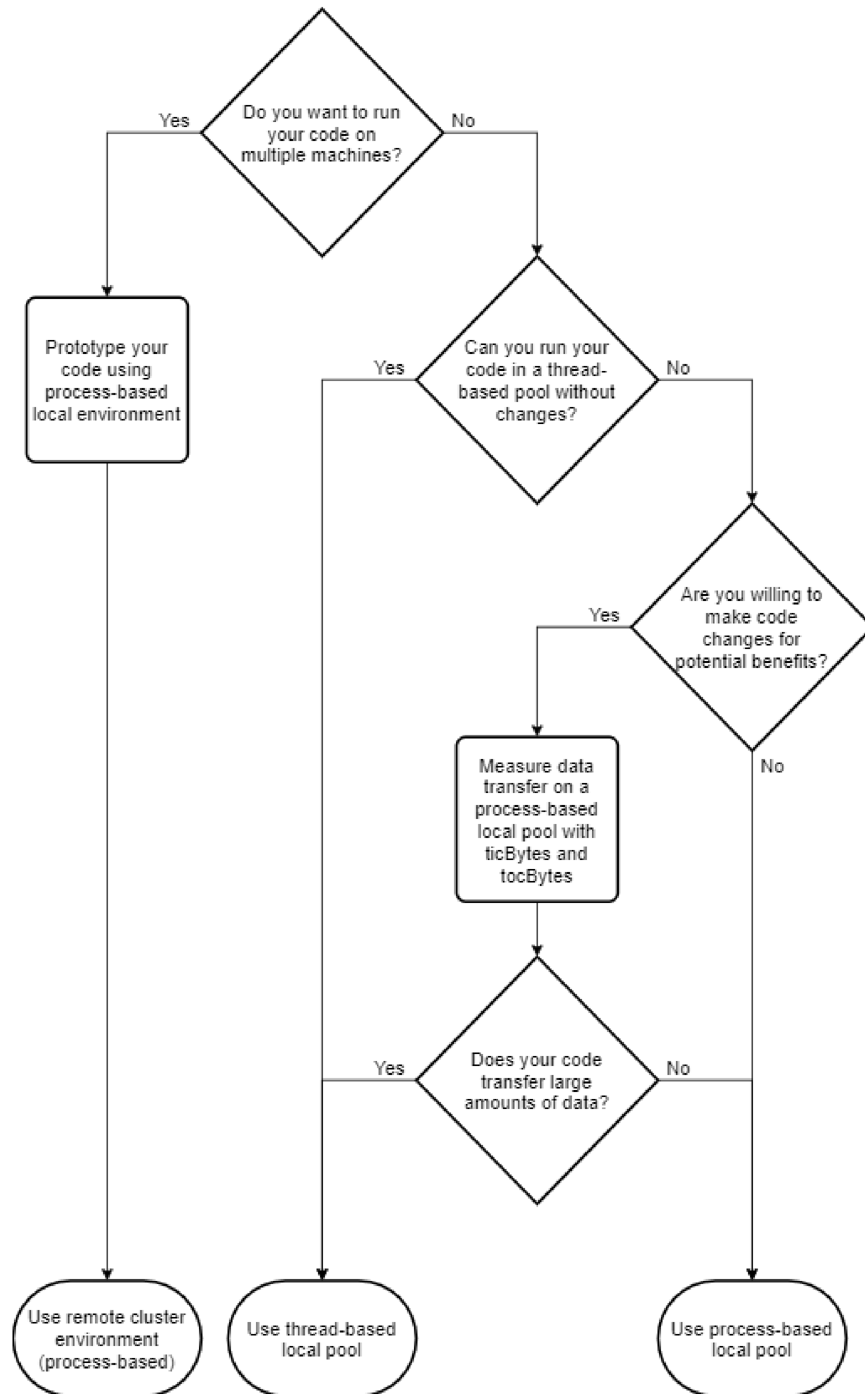


Figura 3 - Escolha de ambientes de paralelismo [16]

- Utilizar ambientes baseados em processos, a correr em *cluster* remoto, para efeitos de *scaling* de computações para *clusters* quando as necessidades de computação ou volume de dados vão além das capacidades de uma única máquina. A criação do *pool* de *process workers* é feita utilizando o comando *parpool*(‘*MyCluste*r’), em que *MyCluster* corresponde ao nome de um *cluster profile*.

Para usar as *features* dos *clusters* (como, por exemplo, a função *batch*) é primeiro necessário criar um objeto *cluster*, utilizando para isso o comando *parcluster*.

2.4. Julia

A linguagem Julia [6] é *open-source*, dinâmica e é a linguagem criada mais recentemente de entre as três em estudo nesta Dissertação. Esta linguagem combina características de programação imperativa, funcional e orientada para objetos. A sua sintaxe é semelhante à de Matlab, característica que foi implementada para facilitar a transição de utilizadores Matlab para esta linguagem.

Julia foi concebido com o objetivo de ser uma linguagem eficiente e de fácil utilização, procurando oferecer alguns dos benefícios de linguagens estáticas como C e C++, e de linguagens dinâmicas e desenhadas para resolução de problemas de computação intensiva, como Python e Matlab. Esta linguagem permite também a chamada de rotinas de linguagens como C e Fortran, sem ser necessária a implementação de código de ligação entre as mesmas.

Julia utiliza compilação *just-in-time* (*JIT*). Deste modo, como todo o código é compilado para código-máquina em *runtime*, esta linguagem tem o potencial de correr com a mesma velocidade que linguagens como C ou Fortran.

Julia oferece mecanismos de programação paralela utilizando múltiplas *threads* (*multithreading*) e múltiplos processos (*multiprocessing)* através dos seus módulos de computação distribuída e *multithreading*.

2.4.1. Bibliotecas Relacionadas com Paralelismo

A linguagem Julia tem a maioria dos mecanismos de paralelismo incorporados nos seus módulos internos, contrariamente a várias linguagens que necessitam de elevado nível de suporte de bibliotecas externas para lidar com situações de paralelismo.

Esta linguagem utiliza *threading* implícito para operações de cálculo numérico, à semelhança de Matlab, e a sua biblioteca *standard* já contempla suporte para paralelismo *multiprocessing* e *multithreading*.

A versão estável mais recente do Julia, até ao momento da escrita deste relatório, é a versão 1.6.4, disponibilizada a 19 de Novembro de 2021.

2.4.2. Modos de Paralelismo

A linguagem Julia suporta duas categorias principais relativas a programação paralela [18]:

1. *Multithreading*
2. Computação Distribuída

As funcionalidades *multithreading* permitem que *tasks* corram simultaneamente em múltiplas *threads* ou CPU *cores* com partilha de memória.

A computação distribuída permite correr múltiplos processos em espaços de memória separados e, potencialmente, em diferentes máquinas. Esta funcionalidade é providenciada pela *Distributed Standard Library*.

2.4.2.1. Multithreading

A execução de programas Julia inicia, por predefinição, com uma única *thread*. O número de *threads* [20] em execução pode ser consultado através da função Threads.nthreads(). O número de *threads* disponíveis pode ser controlado através da utilização da opção *–threads* na linha de comandos, ou usando a variável ambiente JULIA_NUM_THREADS. É possível determinar a *thread* em que nos encontramos através do comando Threads.threadid().

- **Data-Race Freedom**

  A responsabilidade de assegurar que um programa é *data-race free* é do programador. Uma maneira de assegurar esta condição é através do uso de *locks* no acesso a informação que seja observável por múltiplas *threads*. É importante notar que Julia também não garante *memory-safety* na presença de condições de *data-race*.

  Julia também suporta o acesso e modificação de valores de forma atómica, ou seja, de forma *thread-safe*, e deste modo evita *race-conditions*. Para esse efeito, o valor (de um tipo primitivo) deve estar encapsulado por Thread.Atomic.

- **Macro @threads**

  Julia suporta *loops* paralelos através do uso da anotação @threads antes da declaração do ciclo-*for,* de modo a indicar que o *loop* é uma região *multithreaded* que deve ser executado pelas *threads* disponíveis.

2.4.2.2. Computação Distribuída e Multiprocessing

A linguagem Julia oferece um ambiente *multiprocessing* baseado na troca de mensagens, o que permite a programas executarem em múltiplos processos com memória separada.

A comunicação distribuída [21] em Julia é baseada em referências remotas e chamadas remotas. Uma referência remota é um objeto que pode ser usado por qualquer processo para referir outro objeto guardado num processo distinto. Uma chamada remota é um pedido de um processo para chamar uma determinada função, com certos parâmetros, via uma referência remota.

Referências remotas podem ser de dois tipos: *Future* e *RemoteChannels*. Uma chamada remota retorna um objeto *Future* e esse retorno é imediato. O processo que fez a chamada pode continuar para as operações seguintes enquanto a chamada remota é executada em outro local. Opcionalmente, podemos utilizar a função wait() para aguardar pelo resultado da chamada remota antes de continuar a execução, e pode-se obter o valor completo do resultado usando a função *fetch*.

- **Iniciar e Gerir Processos Worker**

  O processo original tem sempre o identificador 1, sendo que os restantes processos utilizados nas operações paralelas são denominados *workers*. O processo com identificador 1 é apenas considerado *worker* quando é o único processo existente.

  Para especificar o número de processos *worker* que podem ser usados numa máquina local pode-se utilizar o comando 'julia –p n' com n correspondendo ao número de processos, sendo que normalmente esse valor é igual ao número de CPU *threads* da máquina. Julia tem, também, suporte *built-in* para dois tipos de *clusters*: *clusters* locais, especificado com a opção '-p', e *clusters* de múltiplas máquinas, especificados por ' —machine-file'.

Funções como *addprocs*, *rmprocs* e *workers* estão disponíveis como meios de adicionar, remover e identificar processos em *clusters*.

O módulo *Distributed* é automaticamente adicionado aos processos *worker*, mas deve ser explicitamente carregado no caso do processo *master*.

- **Comunicação de Informação**

  O envio de mensagens e a troca de informação constituem a maioria do *overhead* em computação distribuída, sendo que reduzir o número de mensagens e de informação enviada é fundamental para alcançar melhor eficiência e escalabilidade. Com este objetivo, é importante saber utilizar as operações de movimentação de informação *fetch* e *spawnat* da forma mais eficiente possível, uma vez que o seu uso incorreto poderá gerar um maior volume de dados transmitidos [61]. O método *fetch* é caracterizado como uma operação de transferência de informação explícita, uma vez que faz um pedido direto para que um objeto seja movido para a máquina local, enquanto que *spawnat* é considerado uma operação de transferência de informação implícita, sendo que o volume de informação que é necessário transmitir depende do contexto em que a operação é utilizada.

- **Referências Remotas, AbstractChannel e Distributed Garbage Collection**

  Referências remotas [62] referem sempre uma implementação concreta da estrutura abstrata *AbstractChannel*. Duas dessas implementações são *Channel* e *RemoteChannel*. É sempre necessária uma implementação concreta do *AbstractChannel* de modo a poder executar comandos relativos a troca de informação como 'put!', 'take!', 'fetch' e 'wait'.

  Objetos aludidos por referências remotas apenas podem ser libertados quando todas as suas referências no *cluster* estiverem eliminadas.

  O nó onde o valor de um objeto é guardado mantém sempre memória dos *workers* que têm referência à sua informação. Sempre que um *RemoteChannel* ou um *Future* é serializado para um *worker*, o nó apontado por essa referência é notificado, e sempre que o *RemoteChannel* ou *Future* é *garbage collected* localmente, o nó a que pertence o valor é igualmente notificado. Referências remotas são apenas válidas no contexto do *cluster* em execução.

O momento em que um objeto é *garbage collected* localmente depende do tamanho do objeto e do consequente efeito na memória do sistema.

- **Shared Arrays e Distributed Garbage Collection**

  Arrays partilhados (*Shared Arrays*) [63] utilizam a memória partilhada de um sistema para mapear o *array* entre os diversos processos. Neste tipo de *arrays*, cada processo tem acesso ao *array* completo, contrariamente a outras estruturas em que cada processo apenas pode aceder a uma parte do *array* e em que não é possível a processos diferentes acederem à mesma informação.

  O acesso e alocação de valores em *Shared Arrays* funciona de forma semelhante ao acesso e alocação de valores em *arrays* normais e é eficiente porque a memória partilhada está disponível nos processos locais.

  Os *Shared Arrays* estão dependentes da *garbage collection* do nó criador para que este liberte as referências em todos os *workers* participantes.

  Uma vez que todos os processos participantes têm acesso à memória partilhada, é necessário ter cuidado para não criar conflitos nos casos em que vários processos escrevam para o *array* em simultâneo, nos mesmos espaços de memória.

- **Gestores de Clusters**

  O lançamento e gestão de processos Julia em *clusters* é feito através de gestores de *clusters* [64]. Um *ClusterManager* é responsável por lançar os processos *worker* no *cluster*, gerir os eventos durante o tempo de vida do *worker* e, opcionalmente, providenciar troca de informação.

  Julia oferece dois tipos de gestores de *clusters* embutidos: *LocalManager*, quando são invocadas as operações addprocs() e addprocs(np::Integer), e SSHManager, quando é utilizada a operação addprocs(hostnames::Array) em conjunto com uma lista de *hostnames*. O *LocalManager* é utilizado para lançar *workers* adicionais dentro do mesmo *host*, tirando partido de *hardware multicore* com capacidade *multiprocessing*. O SSHManager é utilizado para lançar *workers* em *hosts* remotos, via SSH.

CAPÍTULO 3

IMPLEMENTAÇÕES

Neste capítulo é feita uma introdução aos conceitos de *benchmarks* e *microbenchmarks*, é analisado o algoritmo de otimização DMS, procede-se à descrição dos aspetos mais importantes da implementação do algoritmo nas linguagens Python, Julia e Matlab e e faz-se uma análise dos *microbenchmarks* implementados.

3.1. Introdução a Benchmarks e Microbenchmarks

Na avaliação de desempenho de qualquer sistema existem duas abordagens frequentemente utilizadas: avaliar o comportamento do sistema em causa numa carga de trabalho (ou aplicação) típica, e verificar como as primitivas do sistema, responsáveis por operações fundamentais, se comportam relativamente ao seu desempenho específico. Estas duas noções estão relacionadas com os conceitos de *benchmark* e *microbenchmark*.

Ao recorrer a *benchmarks* estamos a avaliar não só as operações principais que fazem parte do sistema em estudo, mas também como o sistema se comporta como um todo. Este conceito é muito importante uma vez que, por exemplo, pode haver um sistema que é eficiente no lançamento de *threads* mas que, ao mesmo tempo, a sua gestão de memória provoca o aparecimento de *bottlenecks* que fazem com que uma aplicação real tenha um mau desempenho. Assim, ao usar-se como *benchmarks* as aplicações reais, ou programas que tenham perfil e

carga de trabalho idêntica às dessas aplicações, obteremos resultados semelhantes aos do sistema real, o que permite prever mais corretamente o seu comportamento de execução.

Os *microbenchmarks* têm por objetivo avaliar como operações de um sistema, relativas a diferentes primitivas e fundamentais para o desempenho do sistema em estudo, se comportam individualmente. Diferentes primitivas têm impactos diferentes no desempenho final da aplicação, sendo por isso fundamental o seu estudo no contexto da avaliação do sistema final, bem como a identificação dos *bottlenecks* e das partes do sistema que necessitem ser otimizadas. No contexto do nosso trabalho, focamos o estudo dos *microbenchmarks* no conjunto de primitivas relacionadas com a criação e comunicação entre *threads* (para ambiente *multithreading*) e a criação e comunicação entre processos (para ambiente *multiprocessing*).

Neste trabalho inspirámo-nos em pacotes de *benchmarks* populares e frequentemente utilizados na avaliação de sistemas e recorremos ao uso do algoritmo de otimização DMS para elaboração dos nossos *benchmarks*. Este algoritmo baseia-se em buscas direcionadas, ou seja, na avaliação de funções numa sucessão de pontos, procurando o máximo (ou mínimo, dependente do contexto) dessa função em determinada zona do seu domínio. Este processo é repetido para a vizinhança de cada ponto avaliado, utilizando normalmente métodos numéricos para gerir a busca com o objetivo de encontrar melhores resultados e, eventualmente, convergir os resultados para a solução ótima.

Tendo em conta os objetivos específicos desta Dissertação, o nosso foco principal estará na análise de desempenho do algoritmo DMS do ponto de vista de uma aplicação real através de diferentes *benchmarks*. Avaliaremos também o desempenho das operações principais relativas a execução paralela e das ações de lançamento de *workers* e comunicação entre os mesmos, através da criação de *microbenchmarks*.

Em seguida faremos uma breve descrição de três *benchmarks* populares que visam a análise de *performance* de sistemas. Estes *benchmarks* não foram utilizados nesta Dissertação devido a ser inviável a sua escrita nas diferentes linguagens dentro do tempo disponível, sendo que foram estudados como inspiração para a elaboração dos *benchmarks* criados nesta Dissertação.

- **LINPACK Benchmarks**

  O LINPACK *Benchmark*, que tem sido atualizado ao longo dos anos, é atualmente utilizado por cientistas em todo o mundo para avaliar a *performance* dos seus sistemas.

  O LINPACK *Benchmart* suite [23] (que faz parte do popular pacote de álgebra linear com o mesmo nome) oferece diferentes tipos de *benchmarks* com o objetivo de avaliar a *performance* na resolução de equações lineares em sistemas de densidade computacional elevada: *benchmarks* para matrizes 100 x 100 (LINPACK 100), matrizes 1000 x 1000 (LINPACK 1000), e um *benchmark* personalizável ao algoritmo em estudo e à memória disponível do sistema a ser avaliado (HPLinpack).

  O LINPACK 100 é bastante semelhante ao *benchmark* inicialmente publicado em 1979, no qual os sistemas de equações são resolvidos através de *Gaussien elimination* com *parcial pivoting*. É importante notar que os computadores modernos têm capacidade de memória e desempenho tão elevadas, que este tipo de *benchmark* já não permite testar da melhor forma a *performance* desses computadores, devido às matrizes serem demasiado pequenas.

  O LINPACK 1000 oferece um nível de *performance* mais próximo do limite do sistema, uma vez que utiliza um problema de matriz de dimensão 1000x1000 e oferece possibilidade de efetuar alterações ao algoritmo.

  O HPLinpack (*Highly-Parallel Linpack*) permite a testagem de *clusters* de computadores paralelos, o que não é abrangido pelos dois *benchmarks* anteriormente referidos. O seu tamanho é variável de modo a utilizar todos os recursos do *cluster*. Os resultados obtidos por este *benchmark* são utilizados na lista Top500.

- **DEISA Benchmarks**

  DEISA (*Distribute European Infrastructure for Supercomputing Applications*) [24] é um consórcio de alguns dos maiores centros nacionais de supercomputação europeus criado com o objetivo de avançar a Ciência da Computação através do melhoramento e reforço de infraestruturas de *distributed high performance computing* [22] na Europa.

A DEISA Benchmark Suite [25] corresponde a uma seleção de *benchmarks* de aplicações HPC (*High Performance Computing*). As aplicações usadas nesta *benchamark suite* foram escolhidas com propósito de representar áreas científicas e classes de programas chave na Europa. As aplicações estão divididas em sete categorias: Astrofísica, CFD e Combustão, Ciências da Terra e Pesquisa Climática, Informática, Ciência dos Materiais, Física de Plasma e Quantum ChromoDynamics.

Conforme os recursos de *hardware* disponíveis para cada tipo de programação, a *performance* destas aplicações pode ter variações em diferentes arquiteturas. O objetivo destes *benchmarks* é quantificar essas diferenças para as aplicações descritas, e em arquiteturas HPC.

- **SPEC Benchmarks**

SPEC (*Standard Performance Evaluation Corportation*) [26] é uma organização sem fins lucrativos que procura estabelecer, manter e apoiar *benchmarks* de avaliação de *performance* e eficiência energética, para as gerações mais recentes de sistemas computacionais.

SPEC oferece uma grande variedade de *benchmarks* que podemos caracterizar através dos setores em que atuam [27]: Cloud; CPU; Performance Gráfica e de *Workstation*; HPC com OpenMP, MPI, OpenACC e OpenCL; Java *Client/Server*; Armazenamento; Potência; Virtualização e *Web Servers*. Alguns dos seus *benchmarks* mais antigos foram reformados, muitos deles com o aparecimento de *benchmarks* mais recentes para os mesmos setores de avaliação.

A SPEC Benchmark Suite [28] começou, quando foi lançada, por disponibilizar 10 *benchmarks*. Em 1992, foram adicionados mais 20 *benchmarks* individuais, os quais já incluem testes para *multitasking* e *performance* de disco, e foram divididos relativamente a *integer* e *floating point*: Cint92 e CFP92. Em 1995, SPEC introduziu as suites Cint95 e CFP95, adicionando mais 18 *benchmarks* relacionados com a avaliação da *performance* de processadores, sistema de memória, e compiladores.

Desde 1995, foram lançados novos *benchmarks* para avaliação de CPU em 2000, 2006, e 2017, sendo este último a versão mais recente até ao momento.

O SPEC CPU 2017 [29] contém um conjunto de 43 *benchmarks*, divididos em 4 secções: SPECrate 2017 Integer, SPECrate 2017 Floating Point, SPECspeed 2017 Integer e SPECspeed 2017 Floating Point.

3.2. Algoritmo DMS

3.2.1. Algoritmos de Otimização sem Derivadas

Como visto anteriormente, existe a necessidade de definir uma aplicação que permita avaliar os sistemas em estudo. Assim, sendo este trabalho relacionado com o projeto BoostDFO, no qual se estudam algoritmos de otimização de funções sem derivadas (DFO), fazemos o desenvolvimento de uma bateria de testes (*benchmark*) baseada num destes algoritmos de otimização, implementando-o nas várias linguagens. Este desenvolvimento torna-se assim um dos objetivos do trabalho, de modo a permitir a comparação das diferentes linguagens, neste contexto, relativamente ao seu desempenho.

Os algoritmos de otimização sem derivadas baseiam-se numa abordagem de busca no espaço de soluções para a função a otimizar, procurando a que melhor satisfaz os requisitos impostos. Normalmente a grande diferença entre estes algoritmos refere-se aos métodos usados para guiar a busca e acelerar a sua convergência para a solução ótima (ou a melhor solução possível).

A Figura 4 mostra, de forma simplificada, a base de um algoritmo de busca direcionada [35], o qual tem semelhanças ao algoritmo utilizado nesta Dissertação. Esta figura demonstra, de forma geral, a existência de uma fase inicial de descoberta dos pontos melhores candidatos, com base em heurísticas, e uma segunda fase na qual se avalia a função nesses pontos, de forma independente, com o objetivo de descobrir os melhores valores para a função.

Em seguida faremos uma breve descrição de dois algoritmos de otimização estudados no âmbito do projeto BoostDFO, sendo que o nosso trabalho incidirá na utilização do algoritmo DMS.

```
1  Set parameters 0 < γ_dec < 1 ≤ γ_inc
2  Choose initial point x_0 and step size α_0 > 0
3  for k = 0, 1, 2, ... do
4  |  Choose and order a finite set Y_k ⊂ ℝ^n                          // (search step)
5  |  x_k^+ ← test_descent(f, x_k, Y_k)
6  |  if x_k^+ = x_k then
7  |  |  Choose and order poll directions D_k ⊂ ℝ^n                    // (poll step)
8  |  |_ x_k^+ ← test_descent(f, x_k, {x_k + α_k d_i : d_i ∈ D_k})
9  |  if x_k^+ = x_k then
10 |  |  α_{k+1} ← γ_inc α_k
11 |  else
12 |  |_ α_{k+1} ← γ_dec α_k
13 |_ x_{k+1} ← x_k^+
```

Figura 4 - Base de Algoritmo de Busca Direcionada [35]

- **Algoritmo de Otimização SID-PSM**

  SID-PSM [32][34] é um método de busca direcionada com cíclicos iterativos, que permite a resolução de problemas de DFO (*Derivative-Free Optimization*) procurando uma solução que minimiza a função objetivo.

  Os passos deste algoritmo correspondem a: Initialization Step; Search Step; Pool Step; Compute Descent Indicators; Order Directions e Update the Mesh Size Parameter. Nos passos Search e Pool procura-se obter novas potenciais soluções, a partir das melhores soluções conhecidas e tendo em conta a direção mais promissora. Todos os passos, com exceção do Initialization Step, são repetidos em cada iteração, até se encontrar a melhor solução.

  Nestes algoritmos pode haver apenas uma função-objetivo de *input*, embora possam ser consideradas diversas restrições aos valores de solução.

- **Algoritmo de Otimização DMS**

  DMS [34] é um algoritmo de otimização com ciclo iterativo e multiobjectivo, contrariamente a outros algoritmos, como o SID-PSM, que procuram um único ponto que otimiza uma função-objetivo. Sendo multiobjectivo, este algoritmo considera como função-objetivo uma composição de várias funções, possivelmente contraditórias, procurando soluções que as otimizem. Neste algoritmo procuram-se soluções que correspondam a otimizações de Pareto, ou seja, soluções possíveis para todas as funções, nas quais não é seja possível encontrar uma melhor solução para uma das funções sem ser pior para outras.

Este algoritmo consiste, à semelhança do SID-PSM, num passo de inicialização (Initialization Step) seguido de um ciclo com vários passos (Search Step, Pool Step e Parameters Update) onde se vai obtendo e atualizando um conjunto de soluções cada vez melhores.

3.2.2. Estrutura do algoritmo DMS e Funções-Objetivo

No contexto deste trabalho, pretendemos utilizar a implementação do algoritmo de otimização DMS nas linguagens Python, Julia e Matlab como *benchmark* para avaliar e comparar o desempenho destas linguagens. Este algoritmo é constituído, nas diversas implementações, pelos seguintes ficheiros:

- **driver_dms**

  Este driver é responsável pela forma e número de vezes que o algoritmo DMS é executado, de modo a obter os resultados pretendidos para as diferentes avaliações. Este driver foi alterado em todas as linguagens relativamente à sua implementação original de modo a produzir os resultados relativos às diferentes parametrizações de forma facilmente comparável e a organizar a informação obtida em diferentes ficheiros de texto, consoante a função-objetivo em estudo e o modo de paralelismo utilizado. De modo a aumentar a fiabilidade dos resultados obtidos, o driver extrai sempre 5 resultados (como é possível observar no Apêndice A), provenientes de 5 execuções do algoritmo para cada um dos modos de execução em estudo, sendo que é utilizada a média dessas execuções para efeitos de avaliação e comparação de resultados entre linguagens.

- **parameters_dms**

  Corresponde ao ficheiro de parâmetros do algoritmo. Este ficheiro permite executar o algoritmo DMS em diversas condições. Alguns dos parâmetros que têm maior relevância no contexto deste trabalho são o modo de paralelismo utilizado (*multiprocessing*, *multithreading* ou execução sequencial), a função-objetivo a otimizar (ZDT1, ZDT2, ZDT3, ZDT4, ZDT6, Styrene), o sistema operativo que se está a utilizar (Windows, Linux) e, no caso das linguagens Matlab e Julia, nos modos *multiprocessing* e *multithreading,* se o tempo de execução extraído inclui, ou não, o tempo de inicialização do *pool* de *workers* (modos DMSInit e DriverInit, respetivamente).

- **dms**

  Ficheiro relativo à função principal dms do algoritmo. A função presente neste ficheiro utiliza os seus dados de *input* e as definições do ficheiro de parâmetros, executa o algoritmo com apoio das funções auxiliares (que se encontram nos ficheiros descritos em seguida), produz os resultados da otimização da função-objetivo e retorna o tempo que a execução demorou. A execução da função dms pode ser dividida em quatro partes: Initialization Step, Search Step, Poll Step e Parameters Update.

  É também neste ficheiro que são utilizados os mecanismos de paralelismo correspondentes aos modos *multiprocessing* e *multithreading*.

- **match_point**

  Ficheiro com função auxiliar match_point do algoritmo DMS. Esta função é responsável por analisar uma lista de pontos previamente avaliados e tentar encontrar o ponto providenciado pelo otimizador. Quando esta pesquisa é bem-sucedida, a função retorna o valor da função-objetivo correspondente anteriormente calculado.

  A função match_point é utilizada no Initialization Step e no Poll Step do algoritmo DMS.

- **paretodominance**

  Ficheiro com função auxiliar paretodominance do algoritmo DMS. Esta função verifica se determinado vetor satisfaz o critério de dominância de Pareto.

  A função paretodominance é utilizada no Initialization Step e no Poll Step do algoritmo DMS.

- **search_step**

  Ficheiro com função auxiliar search_step do algoritmo DMS. Esta função está presente no Search Step mas, no contexto deste trabalho, nunca é utilizada.

- **sort_gamma**

  Ficheiro com função auxiliar sort_gamma do algoritmo DMS. Esta função ordena a lista de pontos recebida de acordo com o maior espaço entre dois pontos consecutivos.

  A função sort_gamma é utilizada no Poll Step do algoritmo DMS.

- **func_C**

  Ficheiro com função auxiliar func_C do algoritmo DMS. Esta função está presente no Initialization Step e Poll Step mas, no contexto deste trabalho, nunca é utilizada.

Como foi referido anteriormente, o algoritmo DMS é responsável pela otimização de funções-objetivo. As funções-objetivo implementadas e estudadas no âmbito desta Dissertação são:

- **Conjunto ZDT**

  Foram implementadas 5 funções-objetivo deste conjunto [65]: ZDT1, ZDT2, ZDT3, ZDT4 e ZDT6. Estas funções são semelhantes na sua forma de implementação, embora produzam resultados e tempos de execução diferentes.

- **styrene_dms**

  Esta função-objetivo [66] permite obter os resultados mais relevantes no que respeita à eficiência dos modos de paralelismo das diferentes linguagens. Entre outros detalhes, a função Styrene cria um ficheiro de *input*, chama um ficheiro executável com o *input* do ficheiro criado, guarda os resultados da execução num ficheiro de *output*, analisa o ficheiro e devolve o resultado correspondente. Todo este processo faz com que esta função-objetivo tenha uma carga de trabalho bastante elevada e que, consequentemente, demore muito mais tempo a executar do que as funções-objetivo do conjunto ZDT. Deste modo, os tempos de execução obtidos permitem obter uma melhor análise dos benefícios do uso das diferentes formas de paralelismo de cada linguagem, para os diferentes tamanhos de *pool* de *workers* que serão estudados. Estes benefícios, nas funções-objetivo ZDT, não são

tão evidentes, uma vez que estas funções têm uma carga de trabalho menor e, assim, o *overhead* do uso de paralelismo acaba por muitas vezes anular as vantagens do mesmo. Todas estas análises serão apresentadas em maior detalhe no capítulo 4.

É muito importante referir que a manipulação e testagem do algoritmo DMS não é o objetivo desta Dissertação. Este algoritmo é utilizado somente como uma ferramenta que, ao ser implementada nas linguagens Python, Julia e Matlab com o maior grau de semelhança possível, permite analisar e comparar as linguagens, em particular a nível de paralelismo, de modo a estudar a sua eficiência. Deste modo, muitas das variáveis do ficheiro de parâmetros parameters_dms mantiveram-se inalteradas durante toda a testagem nas três linguagens, uma vez que não afetavam ou estavam relacionadas com os objetivos desta Dissertação. Assim, apenas as seguintes variáveis do ficheiro parameters_dms foram manipuladas nesta Dissertação para obter as diferentes análises em todas as linguagens:

- **parallel_ver**

  Esta variável, criada no âmbito desta Dissertação, permite determinar se o algoritmo será executado com paralelismo *multiprocessing*, *multithreading* ou se a execução será totalmente sequencial.

- **objectiveFunction**

  Esta variável, criada no âmbito desta Dissertação, permite escolher a função-objetivo a ser otimizada pelo algoritmo: ZDT1, ZDT2, ZDT3, ZDT4, ZDT6 ou Styrene.

- **envVar**

  Esta variável, criada no âmbito desta Dissertação, permite identificar se o ambiente em que se está a executar o algoritmo DMS é Windows ou Linux. Esta variável é relevante nos casos em que se está a otimizar a função-objetivo Styrene, de modo a determinar qual o ficheiro executável que esta função chama.

- **dmsInit**

  Esta variável, criada no âmbito desta Dissertação, está disponível apenas para as linguagens Julia e Matlab. O seu objetivo é permitir escolher,

nos casos em que se executa o algoritmo DMS em modo paralelo (*multiprocessing* ou *multithreading*), se a criação do *pool* de *workers* é feita no ficheiro driver_dms ou no ficheiro dms, correspondendo aos modos de inicialização DriverInit e DMSInit, respetivamente. Esta decisão determina se o tempo de criação do *pool* é incluído no tempo de execução do algoritmo DMS que ultimamente é utilizado para as análises e comparações de resultados entre linguagens.

A motivação para a criação desta variável é analisar o impacto que a inicialização do *pool* de *workers* tem no tempo total de execução do algoritmo nas diferentes linguagens. O interesse de realizar este estudo resulta do facto de, durante a implementação do algoritmo em Julia e Matlab, se notar uma demora de múltiplos segundos no início da execução do algoritmo que não era observável em Python, e que aumentava de forma substancial o tempo total de execução destas linguagens. Assim, tornámos este acontecimento num dos casos de estudo desta Dissertação para as linguagens Julia e Matlab. No caso da linguagem Python, esta análise é feita apenas nos *microbenchmarks*, uma vez que a diferença de tempos obtidos com ou sem criação de *pool* de *workers* no algoritmo DMS é desprezável.

- **output**

  Esta variável faz parte do ficheiro de parâmetros inicial do algoritmo DMS. O seu objetivo passa por determinar se as iterações criadas durante a execução do DMS são mostradas no ecrã em tempo real, ou se apenas os resultados finais são apresentados. Quando esta variável está definida para mostrar todas as iterações no ecrã é também possível visualizar, no final da execução do algoritmo para as funções-objetivo do conjunto ZDT, o gráfico correspondente aos resultados obtidos.

- **list**

  Esta variável faz parte do ficheiro de parâmetros inicial do algoritmo DMS. Embora nunca seja diretamente manipulada nesta Dissertação, a variável define os valores iniciais dos limites do domínio e ponto onde começa a busca consoante a função-objetivo a otimizar seja a função Styrene ou uma das funções do conjunto ZDT.

Todas as outras variáveis do ficheiro de parâmetros do algoritmo DMS permanecem inalteradas ao longo desta Dissertação, mantendo os valores que tinham por predefinição no algoritmo DMS inicialmente fornecido.

Cada uma das implementações do algoritmo DMS nas diferentes linguagens teve características e desafios únicos que serão descritos ao longo dos próximos subcapítulos.

3.3. Metodologia de Implementação

Como referido no capítulo 1, esta Dissertação está relacionada com o projeto BoostDFO, no qual foi implementada uma versão do algoritmo DMS na linguagem Matlab já com mecanismos de paralelismo *multiprocessing* incluídos. Nesta Dissertação utilizámos essa versão do algoritmo (tendo, no entanto, retirado alguns dos detalhes específicos ao projeto anterior) como base para as implementações feitas em Julia e Python, assim como para alterações posteriores à implementação Matlab de modo a abranger um maior número de situações a analisar.

No processo de implementação do algoritmo DMS nas linguagens Python e Julia, foi seguida a mesma metodologia:

1º) Começar por implementar a versão sequencial do algoritmo, usando como base o algoritmo Matlab;

2º) Após ter a versão sequencial finalizada e a produzir os mesmos resultados que o programa Matlab, implementar os métodos de paralelismo;

3º) Fazer as implementações em Python e Julia o mais parecidas possível à implementação Matlab. Deste modo, ao analisarmos os resultados obtidos em cada uma das linguagens, teremos acesso aos dados mais fiáveis para comparação de *performance* das linguagens em cada uma das métricas em estudo.

É de salientar a importância de os métodos de paralelismo implementados serem o mais semelhantes possível entre linguagens. Para este efeito, o código paralelizado é o mesmo (as funções-objetivo, no Poll Step) e recorre-se em todas as linguagens a uma estrutura *master-worker*, ou seja, é criado um *pool* de *workers* (os quais podem ser processos ou *threads*, consoante o modo de paralelismo), e a execução das funções-objetivo é repartida pelos *workers* disponíveis.

Nos subcapítulos seguintes iremos abordar cada uma das implementações do algoritmo DMS, descrevendo alguns dos detalhes mais relevantes no contexto deste trabalho. No caso das implementações em Python e Julia, iremos também analisar algumas das principais diferenças relativamente à implementação base na linguagem Matlab.

3.4. Implementação DMS Matlab

Uma vez que Matlab é a linguagem em que o algoritmo DMS está originalmente implementado, a autoria do algoritmo nesta linguagem não vem primariamente desta Dissertação. No entanto, foram feitas algumas alterações ao DMS Matlab de modo a extrair as métricas necessárias para comparação com as outras linguagens:

- **driver_dms.m**

  O ficheiro driver_dms.m foi modificado na totalidade. Inicialmente, este ficheiro apenas permitia fazer uma execução do algoritmo DMS com uma função-objetivo intitulada de func_F (que corresponde à função ZDT1). Após as alterações feitas, o driver_dms permite guardar todos os resultados obtidos para qualquer combinação de variáveis do ficheiro de parâmetros que façam sentido no âmbito da Dissertação.

  De forma resumida, o driver_dms funciona da seguinte forma: Começa por verificar qual a função-objetivo a ser otimizada através da análise do ficheiro de parâmetros e com base nisso efetua a inicialização das respetivas variáveis com os valores necessários para o funcionamento dessas funções-objetivo. Após esse passo, é criado o ficheiro de texto onde serão guardados os resultados obtidos, personalizando o nome do ficheiro com a informação corresponde à função-objetivo, ao modo de paralelismo (*multiprocessing*, *multithreading*, sequencial), e ao modo de inicialização DMSInit ou DriverInit (no caso dos modos de paralelismo *multiprocessing* ou *multithreading*). Após o ficheiro de texto estar criado, iniciam-se as diversas execuções do algoritmo DMS (cujos dados estão disponíveis no Apêndice A). Se o modo de execução escolhido for sequencial, o driver executará o algoritmo DMS 5 vezes, e guardará os tempos de execução devolvidos no ficheiro de texto. Se o modo for *multiprocessing*, o algoritmo será executado 5 vezes para

cada um dos diferentes tamanhos do *pool* de *workers* em estudo, começando com pool de 16 processos, após isso 8 processos, depois 4, 2 e por fim um *pool* com 1 processo. Se o modo for *multithreading*, o processo é semelhante ao utilizado para *multiprocessing*, com exceção de os *workers* corresponderem a *threads* em vez de processos. Quer para paralelismo *multiprocessing* ou *multithreading*, o *pool* de *workers* é sempre eliminado após a conclusão das respetivas execuções, de modo a não afetar as execuções seguintes. A eliminação do *pool* de *workers,* à semelhança da criação do mesmo, pode ser feita quer no ficheiro driver_dms, quer no dms, dependendo de se estar a executar o algoritmo no modo de inicialização DriverInit ou DMSInit, respetivamente.

- **dms.m**

  No ficheiro dms.m foi adicionado à versão inicialmente fornecida a criação de *pool* de processos/*threads* com tamanho personalizável (de entre as opções 1,2,4,8 e 16) para os casos em que o tempo de criação do *pool* de *workers* conta para os resultados da avaliação (modo DMSInit). Foi também adicionada a eliminação do *pool* nas mesmas condições. A criação do pool de *threads* origina a implementação do modo de paralelismo *multithreading* nesta linguagem.

- **parameters_dms.m**

  No ficheiro parameters_dms.m foram adicionadas as variáveis para decidir o modo de paralelismo, a função-objetivo, o ambiente de execução e o modo de inicialização do *pool* de *workers*. Foi também tornado automática a atribuição do valor correto à variável *list*, consoante a função-objetivo escolhida.

A implementação de paralelismo, no DMS Matlab, está divida em duas partes: criação de *pool* de *workers* e paralelização da avaliação da função-objetivo.

A criação do *pool* de *workers*, quer para *multiprocessing* quer para *multithreading* é feita recorrendo à função parpool. No caso do modo *multiprocessing*, a implementação utilizada é p=parpool('*local*', *numWorkers*), com *numWorkers* sendo o tamanho do *pool*. Para o modo *multithreading*, a implementação é p=parpool('*threads*'). A variável p permite identificar o *pool* de *workers* criado, o que é

necessário para a sua eliminação no final das execuções correspondentes. É também importante notar que, no caso do ambiente *multithreading*, o processo para definir o tamanho do *pool* de *threads* é mais complexo, não sendo possível determiná-lo da mesma forma que em ambiente *multiprocessing*. Neste caso, é necessário recorrer a uma estratégia diferente em que primeiro se define, através do método maxNumCompThreads [40], o número máximo de *threads* disponíveis, guardando o valor previamente definido, e só então se utiliza o método parpool('threads'), que utilizará o número de *threads* definido. No momento em que se pretende eliminar o *pool* de *threads*, repõe-se o antigo valor máximo de *threads* que tinha sido guardado.

A paralelização da avaliação da função-objetivo é feita através do método parfor. Este é o método utilizado no algoritmo DMS proveniente do projeto BoostDFO e foi também assim implementado na versão utilizada nesta Dissertação. O método parfor [41] permite fazer a paralelização de um ciclo-for recorrendo ao *pool* de *workers* definido (ou a um *pool* paralelo de tamanho *default*, se não existir nenhum especificado). É este mecanismo de paralelismo que tentamos reproduzir nas implementações Python e Julia de forma a obter resultados fiáveis para efetuar comparações de *performance*.

Há ainda uma situação relativa à execução do algoritmo DMS Matlab em modo *multithreading* que é importante referir: A função-objetivo Styrene não pode ser otimizada neste modo de paralelismo. A razão devesse ao facto de a implementação da função Styrene recorrer à criação e escrita de ficheiros no seu modo de funcionamento, sendo que estes tipos de operações não são permitidos em ambiente *multithreading* nesta linguagem, retornando o erro "*Use of function fopen is not supported on a thread-based parallel pool*".

3.5. Implementação DMS Python

A implementação do algoritmo DMS em Python foi feita de raiz no âmbito desta Dissertação. Procurámos que cada função pertencente ao algoritmo nesta linguagem ficasse o mais semelhante possível à implementação em Matlab, e que o mesmo sucedesse para as funções-objetivo a otimizar. De um modo geral podemos afirmar que, na maioria dos casos, a linguagem Python oferece alternativas viáveis e diretas para os métodos implementados em Matlab, não tendo havido

muitas situações em que não fosse possível encontrar nesta linguagem mecanismos de programação diretamente correspondentes aos do programa Matlab. É importante referir que grande parte dos métodos matemáticos necessários para a implementação do algoritmo nesta linguagem, bem como mecanismos para manipulação de *arrays* foram encontrados na biblioteca NumPy [67], sendo que esta biblioteca teve um papel fundamental no sucesso da implementação do algoritmo DMS Python.

Em seguida iremos abordar alguns dos detalhes de implementação em que se observou um maior contraste entre as implementações em Python e Matlab, assim como algumas das principais situações que proporcionaram um maior desafio de implementação:

- A forma de retornar o output de funções em Python é diferente da de Matlab. Em Python é necessário o uso de uma instrução return explícita, ao invés de Matlab onde a determinação das variáveis a retornar como output é feita na mesma instrução que a declaração da função;
- Em Matlab, a função-objetivo que se pretende otimizar no contexto do algoritmo é passada, no driver_dms, como parâmetro na chamada da função dms. Em Python não é utilizada a mesma metodologia. Nesta linguagem, a função-objetivo não é passada como parâmetro, sendo ao invés importada diretamente dentro da função dms. A determinação da função-objetivo a otimizar é feita através da análise do ficheiro de parâmetros;
- A chamada a posições de arrays, em Python, é feita através do uso de parênteses retos, contrariamente a Matlab em que essa chamada é feita com parênteses curvos. Assim, em Matlab, não existe distinção na forma como se chama uma função com os respetivos parâmetros e como se indica a posição de um *array*, mas essa distinção é clara no caso da linguagem Python;
- Em Python, o índice da primeira posição de um *array* é representado como índice de valor 0. Em Matlab, no entanto, o índice da primeira posição corresponde ao índice de valor 1. Este fator complicou, por vezes, a implementação do algoritmo em Python, uma

vez que todos os índices utilizados na implementação modelo Matlab tiveram de ser alterados.

- A escrita de texto formatado é diferente em Matlab e Python, sendo que foi necessário recorrer a implementações diferentes nas duas linguagens para produzir os mesmos resultados;
- Na implementação Matlab recorre-se, por vezes, ao uso da função logical. Em Python esta função não tem uma tradução direta, nem existe outra função que desempenhe o mesmo papel. Assim, é necessário recorrer a diferentes métodos para obter a mesma funcionalidade, nesta linguagem;
- Existem diversas situações ao longo do algoritmo DMS em Matlab em que uma variável começa por ter um valor escalar, e ao longo da execução do algoritmo e das diferentes iterações, esta variável recebe novos valores, transformando-se num *array* de múltiplas posições. Isto acontecesse devido a, em Matlab, ser possível adicionar novos valores a uma variável através de atribuição de valores a posições previamente não existentes. Por exemplo, para uma variável escalar x=2, em Matlab é possível fazer a operação x(2)=4 e, a partir deste momento, a variável corresponderá a um array de duas posições com valores 2 e 4. Do mesmo modo, em Matlab, é possível chamar uma variável escalar quer através do nome da variável, quer através da posição 1 dessa variável. Por exemplo, para x=2, Matlab devolverá o valor 2 quer se faça a chamada da variável como x ou como x(1).

  Estes tipos de situações não são possíveis na linguagem Python. Nesta linguagem não é possível transformar uma variável de valor escalar num *array* da mesma forma que em Matlab, nem é permitido referirmo-nos a uma variável escalar através de índices. Isto levou a que fosse necessário fazer as implementações relacionadas com estes casos de forma diferente, sendo que muitas vezes se optou por inicializar as variáveis escalares como *arrays* de uma única posição.
- Em Matlab é possível criar uma variável através da atribuição de um valor a um índice, sem ser necessária qualquer inicialização

prévia da variável. Por exemplo, pode ser criada uma variável com um único valor através do comando x(1)=5, ou até criar um *array* de duas entradas através do comando x(2)=5, sendo que neste caso é atribuído o valor 0 à primeira entrada do *array*. Em Python, não é possível a criação de variáveis desta forma. É necessário iniciar as variáveis como *arrays* para poder fazer a chamada dos seus valores através de índices, e é apenas permitido chamar índices que já existam no *array*;

- A linguagem Matlab tem um mecanismo para, numa só instrução, ordenar um *array* e obter um segundo *array* correspondente aos índices dos valores ordenados. Este é um mecanismo que é utilizado na função auxiliar sort_gamma do algoritmo DMS. Este processo não é tão simples de efetuar em Python como em Matlab, por duas razões: 1) o mecanismo para obter o *array* ordenado e, ao mesmo tempo, um segundo *array* com os respetivos índices não está implementado em Python, e 2) não há forma direta de ordenar um *array* por ordem descendente, sendo que a ordenação predefinida é ascendente.

  É relativamente simples obter a ordenação descendente do *array* por processos alternativos, após algum estudo e raciocínio. Obter o *array* de índices, no entanto, demonstrou ser um processo mais complexo. Para este efeito, recorri à função argsort com parâmetro kind='stable'. Fazer com que os índices obtidos fossem os corretos durante toda a execução do algoritmo foi um processo que necessitou de extensa pesquisa, verificando-se que apenas produz os mesmos resultados que a versão Matlab com a parametrização kind='stable'. As outras parametrizações disponíveis, como kind='mergesort' ou kind= 'quicksort', produzem os resultados corretos até determinada iteração da execução do DMS, mas a partir daí os resultados começam a retornar diferentes da versão Matlab.

A implementação de paralelismo, no DMS Python, está divida em duas partes: criação do *pool* de *workers* e paralelização das avaliações da função-objetivo.

A criação do *pool* de *workers*, quer estes sejam processos ou *threads*, é feita com recurso ao *package multiprocessing* da Python Standard Library. Este *package* permite criar um *pool* de processos (para o modo *multiprocessing*) através do comando p= *multiprocessing*.Pool(*numWorkers*) [42] e um pool de *threads* (para modo *multithreading*) com recurso ao comando pThread=*multiprocessing*.ThreadPool(numWorkers) [43]. A criação do *pool* acontece no Initialization Step do algoritmo, e este processo é tão eficiente (quando comparado com o tempo de criação de *pool* das linguagens Matlab e Julia, como será demonstrado no capítulo 4) que a diferença de tempos de execução do algoritmo DMS Python quando se inclui ou exclui o tempo da criação do *pool* é desprezável. Esta é a razão de a implementação do algoritmo em Python ser a única cujo ficheiro de parâmetros não inclui a variável dmsInit, pois não é relevante fazer o estudo da diferença de tempos de execução do algoritmo DMS Python entre modos DMSInit e DriverInit.

A paralelização da avaliação da função-objetivo é feita através dos comandos p.map e pThread.map, dependendo de se estar a executar o algoritmo em modo *multiprocessing* ou *multithreading*, respetivamente. A variável p utilizada nestes comandos corresponde ao *pool* de *workers* criado. Há ainda uma particularidade do paralelismo efetuado em modo *multiprocessing* que não existe em mais nenhuma linguagem: neste modo, é possível executar o comando map com diferentes chunksizes [44]. Os chunksizes correspondem a uma estimativa da carga de trabalho atribuída a cada um dos processos do *pool* e, no nosso trabalho, iremos utilizar os resultados obtidos neste modo de paralelismo do DMS Python com os chunksizes de valores 1, 4 e 8 para análise e comparação de *performance* com as outras linguagens.

3.6. Implementação DMS Julia

A implementação do algoritmo DMS em Julia foi feita de raiz no âmbito desta Dissertação. Procurámos que cada função pertencente ao algoritmo nesta linguagem ficasse o mais semelhante possível à implementação em Matlab e que o mesmo sucedesse para as funções-objetivo a otimizar. Durante a implementação do algoritmo DMS Julia tivemos a oportunidade de confirmar que a experiência de programação nesta linguagem tem bastantes semelhanças à da linguagem Matlab, como foi referido no estudo da linguagem feito no capítulo 2. No entanto,

e possivelmente devido a Julia ser uma linguagem recente e ainda em desenvolvimento, deparámo-nos com diversos casos em que certas funções ou comandos presentes no DMS Matlab não tinham métodos iguais ou equivalentes na linguagem Julia. Isto levou a que em certas situações fosse necessário utilizar metodologias menos eficientes para produzir os mesmos resultados de certos comandos existentes em Matlab. Devido a este fator consideramos que, de um modo geral, a implementação do algoritmo DMS em Julia foi mais complexa do que a implementação em Python.

Quando o algoritmo DMS Julia é executado em modo *multiprocessing*, à semelhança do que acontece na linguagem Matlab, a criação do *pool* de *workers* pode ocorrer em dois modos, consoante a parametrização do ficheiro de parâmetros: DMSInit ou DriverInit. No modo DMSInit, o *pool* de *workers* é criado na função dms e o tempo da sua criação é incluído no tempo de execução total extraído. No modo DriverInit, o *pool* é criado no ficheiro driver_dms e, deste modo, o tempo da sua inicialização não conta para o tempo total extraído. Estes modos foram implementados com objetivo de fazer um estudo sobre o impacto que o tempo de criação do *pool* de *workers* tem na *performance* do algoritmo.

Em seguida, iremos abordar alguns dos detalhes de implementação em que se observou um maior contraste entre as implementações DMS em Julia e Matlab, assim como algumas das principais situações que proporcionaram um maior desafio de implementação:

- A forma de retornar o output de funções em Julia é diferente de em Matlab. Em Julia é necessário o uso de uma instrução return explícita, ao invés de Matlab onde a determinação das variáveis a retornar como output é feita na mesma instrução que a declaração da função;
- A chamada a posições de *arrays*, em Julia, é feita através do uso de parênteses retos, contrariamente a Matlab em que essa chamada é feita com parênteses curvos. Assim, em Julia, a distinção entre a forma de chamar uma função com os respetivos parâmetros e de indicar a posição de um *array* é clara, contrariamente ao que acontece em Matlab;

- A escrita de texto formatado é diferente em Matlab e Julia, sendo que foi necessário recorrer a implementações diferentes nas duas linguagens para produzir os mesmos resultados;
- Na implementação Matlab recorre-se, por vezes, ao uso da função logical. Em Julia, no entanto, esta função não tem uma tradução direta, nem existe outra função que desempenhe o mesmo papel. Assim, é necessário recorrer a diferentes métodos para obter a mesma funcionalidade, nesta linguagem;
- Em Matlab, um *array* de dimensões 1x3 (1 linha e 3 colunas) representa-se por [1 1 1] ou [1, 1, 1] e um *array* com dimensões 3x1 representa-se [1; 1; 1]. Em Julia, os mesmos *arrays* produzem resultados de dimensões diferentes: apenas o *array* [1 1 1] mantém as dimensões 1x3. O *array* [1, 1, 1], nesta linguagem, tem dimensões 3x0 e mesmo acontece com o *array* [1; 1; 1].
- Em Matlab, a função-objetivo que se pretende otimizar no contexto do algoritmo é passada, no driver_dms, como parâmetro na chamada da função dms. Em Julia não é utilizada a mesma metodologia. Nesta linguagem, a função-objetivo não é passada como parâmetro sendo, ao invés, utilizado o comando include("nomeFuncaoObjetivo.jl"). No caso de o algoritmo estar a executar em modo *multiprocessing*, em particular, o comando include deverá utilizar a anotação @everywhere para que a função-objetivo seja reconhecida não só no processo *master*, como em todos os processos do *pool* de *workers*. A determinação da função-objetivo a otimizar é feita através da análise do ficheiro de parâmetros;
- Em Matlab é possível utilizar *if-statements* em que a condição é numérica em vez de lógica, sendo considerada verdadeira se diferente de zero e falsa se igual a zero. Em Julia, no entanto, não é possível utilizar valores não-booleanos como condição válida, tendo-se substituído por comparações com o valor zero ;
- Em Matlab é possível criar uma variável através da atribuição de um valor a um índice, sem ser necessária qualquer inicialização prévia da variável. Por exemplo, pode ser criada uma variável com um único valor através do comando x(1)=5, ou até criar um *array*

de duas entradas através do comando x(2)=5, sendo que neste caso é atribuído o valor 0 à primeira entrada do *array*. Em Julia, não é possível a criação de variáveis desta forma. É necessário iniciar as variáveis como *arrays* para poder fazer a chamada dos seus valores através de índices, e apenas é permitido chamar índices que já existam no *array*;

- -O comando length(var) tem funcionalidades diferentes em Matlab e Julia, o que pode originar situações de má-utilização da instrução para utilizadores menos familiarizados. Em Matlab, o comando length(var) devolve a maior dimensão da variável, sendo que, em Julia, o mesmo comando devolve o número total de elementos da variável;
- Em Matlab, o comando find(condição) retorna os índices correspondentes aos valores que satisfazem a condição. Em Julia, no entanto, não existe nenhum comando que implemente esta funcionalidade de forma igual. Isto leva a que seja necessário implementar formas diferentes e, possivelmente, menos eficientes de obter os mesmos resultados.

A implementação dos métodos de paralelismo, no DMS Julia, é feita de forma distinta quando se executa o algoritmo no modo *multiprocessing* ou *multithreading*:

- No modo *multiprocessing*, a implementação de paralelismo inclui a criação de um *pool* de *workers* (processos) e a paralelização da avaliação da função-objetivo. O *pool* de *workers* é criado através do comando addprocs(numWorkers) [45], com numWorkers sendo o número de processos a ser criado. Uma vez que no driver_dms a função dms é chamada 5 vezes para cada um dos diferentes tamanhos do *pool* de *workers* (como observável no Apêndice A), é necessário remover os processos do antigo *pool* antes da criação de um novo, utilizando para esse efeito o comando rmprocs(workers()) [46]. A paralelização do código, por sua vez, é feita com recurso às anotações @sync @distributed [47] utilizados no ciclo-for, bem como à estrutura SharedArray [48]. A anotação @distributed permite executar o ciclo-for com recurso aos diferentes processos do

pool de *workers*. A anotação @sync garante que o código seguinte apenas seja executado após todos os processos do *pool* de *workers* terem acabado de processar a sua carga de trabalho. O SharedArray é uma estrutura partilhada por todos os *workers* e que, assim, permite guardar os valores dos respetivos processos num array comum.

- No modo *multithreading* a determinação do tamanho do *pool* de *threads* é feita antes da execução do algoritmo DMS através do valor da variável ambiente JULIA_NUM_THREADS [49]. Assim, durante a execução do algoritmo apenas se tem de proceder à paralelização da função-objetivo, utilizando para isso a anotação @threads [50] do package Threads no ciclo-for. Esta anotação permite utilizar o ciclo-for como uma região *multithreaded*, tomando assim partido das diferentes *threads* disponíveis. O facto de este modo de execução não incluir a criação de *pool* de *workers* leva a que não exista distinção entre modos de inicialização DMSInit e DriverInit neste contexto.

3.7. Implementação Microbenchmarks

Neste subcapítulo iremos descrever os *microbenchmarks* implementados no âmbito desta Dissertação. Estes *microbenchmarks* têm por objetivo simular, de forma o mais isolada possível, os mecanismos de paralelismo *multiprocessing* e *multithreading* utilizados nas implementações do algoritmo DMS nas diferentes linguagens. Estes *microbenchmarks* irão permitir-nos analisar o tempo relacionado com a criação de *pool* de *workers* e com a execução do código paralelizável em cada uma das linguagens. Foi também criado um *microbenchmark* para simular a execução sequencial do algoritmo para efeitos de comparação de *performance* com os *microbenchmarks* relativos aos modos de paralelismo. Neste contexto, o *microbenchmark* sequencial utiliza um ciclo-for sequencial, não havendo criação de *pool* de *workers*.

Os *microbenchmarks* são constituídos pelos mesmos ficheiros em todas as linguagens:

- **driver_microbenchmarks**

  Este ficheiro é responsável por executar os *microbenchmarks* em conformidade com os parâmetros do ficheiro parameters_microbenchmarks e guardar os resultados nos respetivos ficheiros de texto. De forma resumida, o driver começa por criar o nome do ficheiro onde vão ser guardados os resultados obtidos com base em 1) linguagem 2) no modo de execução ser *multiprocessing*, *multithreading*, ou sequencial 3) na criação do *pool* de *workers* ser feita em modo DMSInit ou DriverInit, no caso dos modos de paralelismo e 4) no número de colunas da matriz utilizada como objeto a distribuir pelos *workers* (sendo os números de colunas em estudo 20, 100 e 500). Após os ficheiros de texto estarem criados, procede-se à execução dos *microbenchmarks*. Para cada *microbenchmark*, devido aos tempos de execução serem tão reduzidos (com exceção dos resultados obtidos para o modo de inicialização DMSInit das linguagens Julia e Matlab, como será visto no capítulo 4), há o risco de as primitivas de medição de tempo das linguagens não terem precisão suficiente para expressar corretamente os resultados de uma única execução. Assim, tomámos a opção de medir o tempo de 50 execuções seguidas para cada *microbenchmark*, e utilizar a média (resultado de dividir o tempo total obtido por 50) como valor que é registado no ficheiro de texto. Todo este processo corresponde apenas a um dos tempos registados para determinado modo de execução sendo que, à semelhança do que acontece no ficheiro driver_dms do algoritmo DMS, este processo é repetido 5 vezes nas mesmas condições para efeitos de fiabilidade dos resultados obtidos, utilizando-se a média das 5 execuções para efeitos de comparações entre linguagens.

  No caso do modo *sequential*, o processo descrito corresponde à totalidade dos resultados extraídos. No caso dos modos *multiprocessing* e *multithreading*, o processo descrito é efetuado para cada um dos diferentes tamanhos do *pool* de *workers* em estudo e são extraídos os resultados para ambos os modos de inicialização DMSInit e DriverInit, à semelhança dos *benchmarks* do algoritmo DMS. No caso da linguagem Julia, no entanto, o modo *multithreading* é feito de forma diferente: o número de *threads* disponíveis para o *pool* utilizado é determinado antes da execução através de uma variável

ambiente JULIA_NUM_THREADS, sendo que os resultados extraídos neste modo não são distinguidos entre DMSInit e DriverInit.

- **parameters_microbenchmarks**

  Ficheiro de parâmetros onde é possível determinar se o modo de execução é *multiprocessing*, *multithreading* ou sequencial, se a criação de *pool* de *workers* é feita em modo DMSInit ou DriverInit e quais as dimensões da matriz utilizada no processo de paralelismo;

- **objectiveFunction**

  Ficheiro que simula a função-objetivo a otimizar e que é utilizado no processo de paralelismo. No caso dos *microbenchmarks*, como o objetivo é verificar a eficiência dos mecanismos de paralelismo, esta função apenas devolve o parâmetro que recebeu quando foi chamada, de forma a simular a menor carga de trabalho possível;

- **multiprocessing_microbenchmarks**

  Ficheiro contendo a criação do *pool* de processos (que é apenas executado no modo DMSInit) e a execução da função-objetivo em modo *multiprocessing*, com os mesmos métodos que foram utilizados no algoritmo DMS para cada uma das linguagens.

- **multithreading_microbenchmarks**

  Ficheiro contendo a criação de *pool* de *threads* (para as linguagens Python e Matlab e apenas executado no modo DMSInit) e a execução da função-objetivo em modo *multithreading*, com os mesmos métodos que foram utilizados no algoritmo DMS para cada uma das linguagens.

- **sequential_microbenchmarks**

  Ficheiro contendo a execução sequencial da função-objetivo, da mesma forma que acontece no algoritmo DMS nas diferentes linguagens.

CAPÍTULO 4

AVALIAÇÕES

4.1. Estratégia de Avaliação e Objetivos

Neste capítulo iremos proceder à análise e comparação dos resultados obtidos no âmbito dos *benchmarks* e *microbenchmarks* implementados para as linguagens em estudo.

As avaliações efetuadas nesta Dissertação englobam a execução do algoritmo de otimização DMS implementado nas linguagens Python, Julia e Matlab, utilizando as funções objetivo Styrene, ZDT1, ZDT2, ZDT3, ZDT4 e ZDT6 e correspondem aos estudos comparativos dos resultados das diferentes execuções do algoritmo. O algoritmo DMS foi implementado de forma semelhante em todas as linguagens, como descrito no capítulo 3, de modo que as conclusões retiradas da comparação dos tempos de execução não sejam influenciadas por alterações ao algoritmo.

Os principais parâmetros que são manipulados no algoritmo DMS em todas as linguagens de modo a obter os dados necessários para os diferentes estudos são a função-objetivo a otimizar, o modo de paralelismo utilizado (*multiprocessing*, *multithreading* e sequencial), o tamanho do *pool* de *workers* (no caso dos modos *multiprocessing* e *multithreading*) e a escolha do modo de inicialização DMSInit ou DriverInit, (no caso dos modos paralelos e com exceções da linguagem Python e do modo *multithreading* da linguagem Julia, como referido no capítulo 3*)*.

Os *microbenchmarks* correspondem à execução de programas de pequena dimensão, escritos em Python, Julia e Matlab, que implementam as estratégias de

execução *multiprocessing*, *multithreading* e sequencial utilizadas no algoritmo DMS. Estes *microbenchmarks* utilizam diferentes matrizes no processo paralelo/sequencial, de modo a obter resultados para diferentes cargas de trabalho, e têm por objetivo permitir a análise dos *overheads*, de forma isolada, dos mecanismos usados para programação paralela em cada linguagem.

Os principais parâmetros manipulados nos *microbenchmarks* para obter os dados necessários sobre os diferentes casos em estudo são o modo de execução (*multiprocessing*, *multithreading* e sequencial), o número de colunas da matriz utilizada (uma vez que este parâmetro está relacionado com a carga de trabalho da execução paralela/sequencial), o tamanho do *pool* de *workers* (no caso dos modos *multiprocessing* e *multithreading*) e o modo de inicialização do *pool* de *workers* (modos DMSInit e DriverInit).

Como foi referido anteriormente, e como se pode verificar nas tabelas de resultados presentes no Apêndice A, foram extraídos 5 resultados para cada um dos tipos de execuções feitas nesta Dissertação a nível de *benchmarks* e *microbenchmarks*, de modo a assegurar a fiabilidade dos dados utilizados na análise e comparação de resultados. Para todos os casos envolvendo paralelismo são calculadas duas médias diferentes, T1-T5 e T2-T5. A média T1-T5 corresponde à média dos 5 resultados obtidos para cada um dos tipos de execução. A média T2-T5 corresponde à média dos 4 últimos resultados obtidos, ou seja, a média excluindo os resultados da primeira execução. O motivo para calcular estas duas médias deriva de ser observável em certos casos (especialmente em certas execuções paralelas de DMS Julia e DMS Matlab) uma discrepância dos tempos da primeira execução relativamente às restantes 4. Este facto, em nosso entender, deve-se a que na primeira execução sejam feitas algum tipo de inicializações que não sejam necessárias repetir nas restantes e também ao carregamento dos ficheiros para *buffers* do sistema que resultem num aumento do tempo de execução do algoritmo durante a primeira execução. Com estas médias podemos estudar se existem diferenças significativas entre linguagens também neste aspeto e se esta discrepância, ao comparar a *performance* entre linguagens, pode levar a conclusões diferentes.

Na secção 4.2 serão apresentados e discutidos os resultados obtidos para todos os casos de estudo desta Dissertação. Esta secção está divida em *Benchmarks* e *Microbenchmarks*. Os principais estudos apresentados em *Benchmarks* são:

- Para cada função-objetivo, analisar as linguagens que produziram os resultados mais eficientes e menos eficientes em cada um dos modos *multiprocessing*, *multithreading* e sequencial. Esta análise, no caso dos modos de execução paralela, é feita quer para cada um dos diferentes tamanhos de *pool* de *workers* (de entre os tamanhos 1,2,4,8 e 16) quer para o conjunto de todos os *pool* de *workers*;
- Para cada funções-objetivo, determinar quais foram as execuções mais eficientes e menos eficientes para o conjunto de todos os modos de execução (*multiprocessing*, *multithreading* e sequencial);
- Estudo das diferenças de resultados obtidos entre os modos de inicialização DMSInit e DriverInit (quando aplicável) de modo a analisar o impacto que o tempo de criação de *pool* de *workers* tem no desempenho do programa;
- Estudo de potenciais diferenças de conclusões obtidas utilizando as médias T1-T5 e T2-T5, de forma a analisar se a diferença de tempos de execução entre a primeira execução e restantes 4 tem impacto na comparação de *performance* das linguagens;

É importante relembrar alguns aspetos (referidos em maior detalhe no capítulo 3) relevantes para a análise dos *Benchmarks*:

- No DMS Python são analisados três modos diferentes da execução *multiprocessing* relativos aos diferentes chunksizes implementados;
- O DMS Python não tem distinção entre modo DMSInit e DriverInit;
- O DMS Julia não tem distinção entre modo DMSInit e DriverInit para execução em modo *multithreading*;
- Em DMS Matlab não é possível executar a função-objetivo Styrene em modo *multithreading*.

Relativamente à secção *Microbenchmarks* do próximo subcapítulo, os principais estudos apresentados são:

- Comparação da eficiência dos *microbenchmarks* de cada linguagem para cada um dos modos *multiprocessing*, *multithreading* e sequencial. Todas as

análises feitas são relativas aos dados obtidos com três tipos de matrizes (matrizes com 20, 100 e 500 colunas) e são realizadas para os modos DMSInit e DriverInit quando aplicável, utilizando ambas as médias T1-T5 e T2-T5;

- Comparação dos tempos de execução obtidos com modos de inicialização DMSInit e DriverInit nas três linguagens, para as execuções paralelas *multiprocessing* e *multithreading*.

Procedemos à execução de todos os programas na mesma máquina, com os mesmos parâmetros e nas mesmas condições. A máquina utilizada pertence ao *cluster* do Departamento de Informática da FCT-UNL, e tem as seguintes características, de acordo com a documentação oficial do *cluster* [39]:

- CPU: 2 X Intel Xeon E5-2609 v4
- Total Cores/Threads: 16/16
- Memory: 32 GiB DDR4 2400 MHz
- Network: 2 x 1 Gbps
- Main Disk: 110 GB SSD

4.2. Apresentação de Resultados

As figuras presentes nesta secção relativas aos resultados das diferentes execuções contêm diversas abreviaturas para os diferentes modos de execução implementados. As notações utilizadas podem ser encontradas no Apêndice B, bem como a descrição do significado dos seus componentes. Para maior comodidade, no entanto, apresentaremos aqui os significados dos constituintes das abreviaturas utilizadas ao longo desta secção:

- J corresponde a "Julia", P corresponde a "Python", M corresponde a "Matlab";
- MP corresponde a "Multiprocessing", MT corresponde a "Multithreading", S corresponde a "Sequencial";
- C1 corresponde a "Chunksize=1", C4 corresponde a "Chunksize=4", C8 corresponde a "Chunksize=8";
- DMSInit corresponde a "modo de inicialização DMSInit", DriverInit corresponde a "modo de inicialização DriverInit";
- T1 corresponde a "média T1-T5", T2 corresponde a "média T2-T5";

- Col20 corresponde a "Matriz de 20 Colunas", Col100 corresponde a "Matriz de 100 Colunas", Col500 corresponde a "Matriz de 500 Colunas".

Todas as figuras e tabelas relativas aos estudos efetuados nesta secção estão disponíveis no Apêndice B.

4.2.1. Benchmarks

4.2.1.1. Styrene

4.2.1.1.1. Multiprocessing

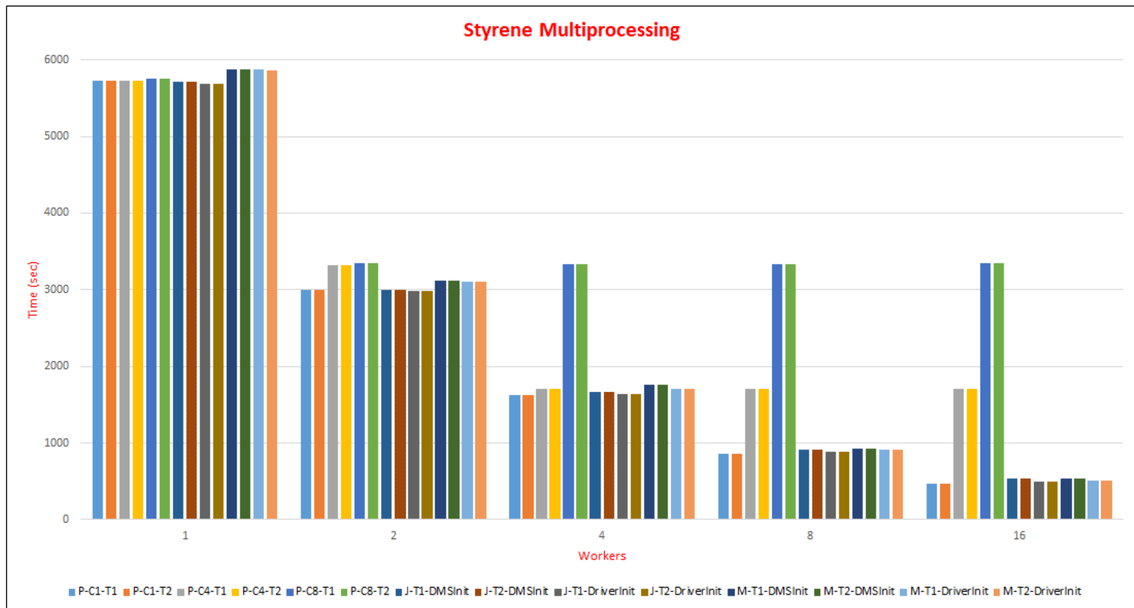


Figura 5 - Benchmarks Styrene Multiprocessing

A figura 5 mostra os resultados obtidos nas diferentes linguagens em *multiprocessing* para os modos de inicialização DMSInit e DriverInit e médias T1-T5 e T2-T5. Esta figura foi aumentada de modo a facilitar a sua análise, sendo que está disponível com melhor qualidade no Apêndice B (figura 27) assim como a respetiva tabela de resultados (tabela 26).

Os resultados do estudo da eficiência das linguagens para ambos os modos de inicialização e ambas as médias, considerando o melhor chunksize de Python para cada *pool size*, são:

- Julia é o mais eficiente para *pools* de 1 e 2 *workers* e Python é o mais eficiente para *pools* de 4, 8 e 16 *workers*;

- Matlab é o menos eficiente em todos os *pool sizes* (com exceção do modo DMSInit com média T1-T5 no qual Julia é o menos eficiente para *pool* de 16 *workers*);
- De entre todas as linguagens e *pool sizes*, a melhor execução é Python com *pool* com 16 *workers* e a pior é Matlab com *pool* com 1 *worker*.

Quando fazemos o mesmo estudo considerando todos os chunksizes de Python, a principal diferença para os resultados acima descritos é que Python com *chunksize*=8 passa a ser a implementação menos eficiente para *pools* de 2, 4, 8 e 16 workers, ao invés de Matlab.

As observações gerais mais relevantes do estudo de Styrene-Multiprocessing são:

- De modo geral e considerando o melhor *chunksize* de Python, a implementação em Python é a mais eficiente (só sendo menos eficiente que Julia para *pools* de 1 e 2 *workers* por uma margem muito reduzida) e a implementação em Matlab é a menos eficiente;
- As implementações Matlab, Julia e Python (com *chunksize*=1) produzem resultados mais eficientes quanto maior o tamanho do *pool* de *workers*;
- Para a implementação Python, os resultados obtidos com *chunksize*=1 são sempre melhores que os obtidos com *chunksize*=4 e *chunksize*=8. A implementação com *chunksize*=4 deixa de ter melhorias de eficiência para *pools* de tamanho superior a 2 *workers* e o mesmo acontece com *chunksize*=8 para *pools* de tamanho superior a 1 *worker*.

4.2.1.1.2. Multithreading

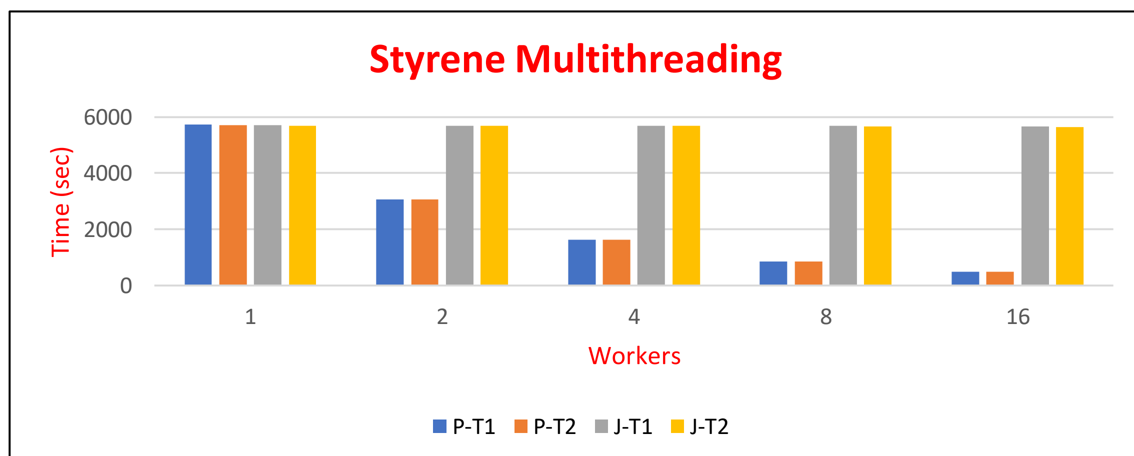


Figura 6 - Benchmarks Styrene Multithreading

A figura 6 mostra os resultados obtidos nas linguagens Python e Julia em *multithreading* para as médias T1-T5 e T2-T5. A figura está disponível no Apêndice B (figura 33) assim como a respetiva tabela de resultados (tabela 32).

As observações gerais mais relevantes do estudo de Styrene-Multithreading para ambas as médias são:

- Python é a linguagem mais eficiente para todos os *pool sizes* com exceção de pool de 1 *thread,* para o qual Julia é a mais eficiente;
- De entre todos os *pool sizes*, a execução mais eficiente é Python com *pool* de 16 *worker* e a menos eficiente é Python com *pool* de 1 *worker*;
- A eficiência de ambas as linguagens melhora com o aumento do tamanho do *pool* de *workers*, embora as melhorias de eficiência em Julia sejam muito reduzidas (praticamente negligenciáveis) A explicação que nos parece mais plausível para a falta de melhoria de eficiência em Julia com o aumento do *pool size* é: uma vez que a função-objetivo Styrene faz chamada a um ficheiro executável, o que requer a utilização de um mecanismo *lock* [51] para assegurar *thread-safety*, este procedimento resulta em as diferentes *threads* se comportarem de modo sequencial durante grande parte da execução do algoritmo e que, assim, se reduza o potencial benefício do uso das *threads* disponíveis, sendo a carga de trabalho paralelizável muito reduzida.

4.2.1.1.3. Sequential

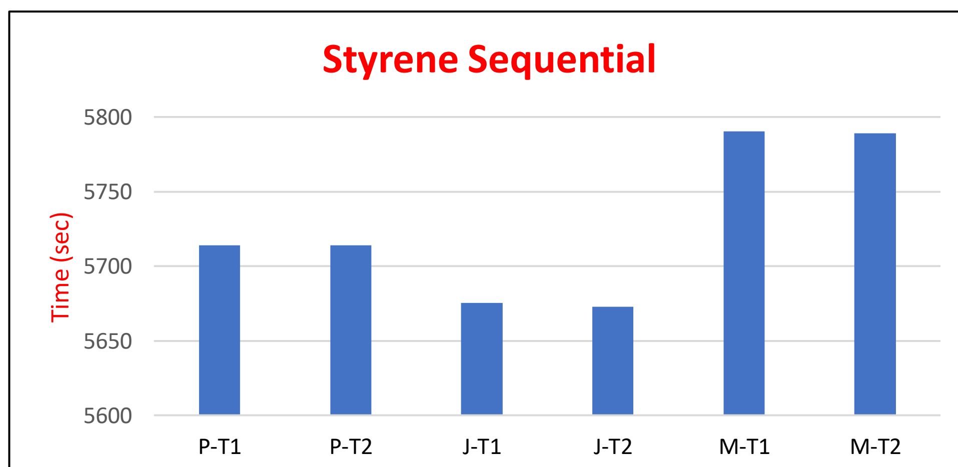


Figura 7 - Benchmarks Styrene Sequential

A figura 7 mostra os resultados obtidos na execução sequencial das diferentes linguagens, para as médias T1-T5 e T2-T5. A figura está disponível no Apêndice B (figura 39), assim como a sua respetiva tabela de resultados (tabela 38).

Para Styrene-Sequencial com ambas as médias observamos que:

- A linguagem mais eficiente é Julia e a menos eficiente é Matlab.

4.2.1.1.4. Execução mais Eficiente de entre todos os Modos

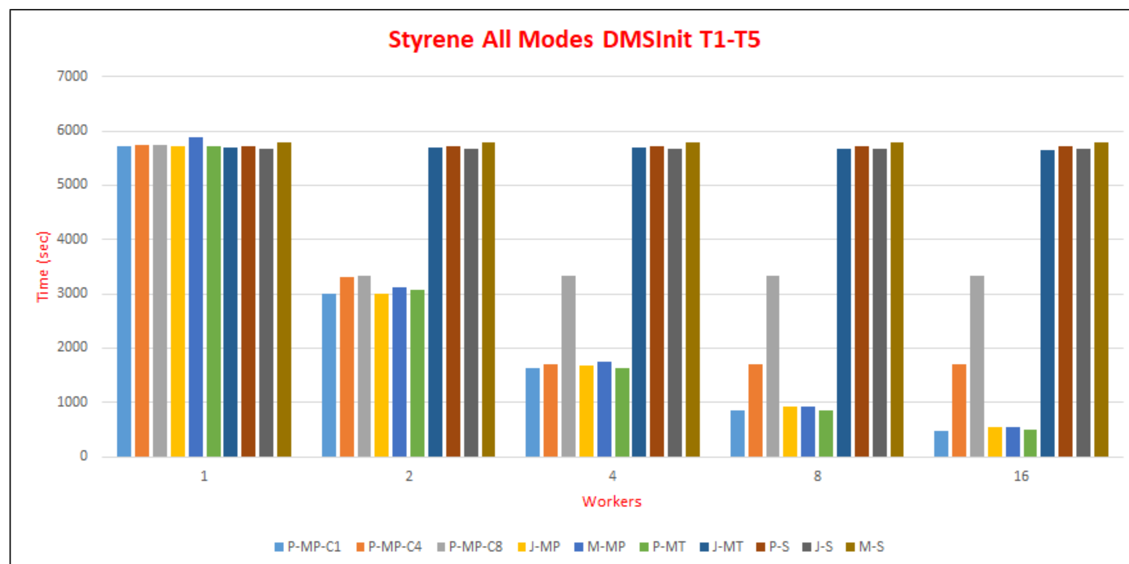


Figura 8 - Benchmarks Styrene DMSInit T1-T5

A figura 8 mostra os resultados das execuções *multiprocessing*, *multithreading* e sequencial de todas as linguagens apenas para o modo de inicialização DMSInit e média T1-T5. Esta figura foi aumentada de modo a facilitar a sua análise, sendo que está disponível com melhor qualidade, em conjunto com as restantes figuras e tabelas relativas a todas as combinações dos modos DMSInit e DriverInit e das médias T1-T5 e T2-T5, no Apêndice B (figuras 45-48 e tabelas 44-47). É também importante fazer notar que as execuções em modo sequencial não utilizam *pools* de *workers*, sendo que estão representadas nestas figuras com valores iguais para cada um dos *pool sizes* de modo a facilitar a comparação de resultados.

Os resultados do estudo da eficiência das linguagens com ambos os modos de inicialização e ambas as médias, quer considerando todos os *chunksizes* ou apenas o melhor *chunksize* de Python, são:

- Julia-Sequencial é o mais eficiente comparando com *pool* de 1 *worker*, Julia-*Multiprocessing* é o mais eficiente para *pool* de 2 *workers*, Python-*Multithreading* é o mais eficiente para *pool* de 4 *workers* e Python-*Multiprocessing* (com chunksize=1) é o mais eficiente para *pools* de 8 e 16 *workers*;
- Matlab-*Multiprocessing* é o menos eficiente para *pool* de 1 *worker* e Matlab-Sequencial é o menos eficiente para todos os restantes *pool sizes*;
- De entre todos os *pool sizes*, a execução mais eficiente é Python-*Multiprocessing* (com chunksize=1) com *pool* de 16 *workers* e a menos eficiente é Matlab-*Multiprocessing* com *pool* de 1 *worker*.

Algumas das conclusões mais relevantes observadas para a função Styrene, de entre todos os modos de execução:

- Python foi a linguagem mais eficiente em ambos os modos de paralelismo (*multithreading* e *multiprocessing)* para os *pools* de 4, 8 e 16 *workers*;
- Julia foi a linguagem mais eficiente em ambos os modos de paralelismo (*multithreading* e *multiprocessing*) para *pool* de 1 *worker,* assim como a mais eficiente em execução sequencial.
- Matlab não foi a linguagem mais eficiente em nenhuma execução da função Styrene;

4.2.1.2. Conjunto ZDT

As funções ZDT têm cargas de trabalho muito inferiores às da função Styrene, sendo que permitem analisar e obter conclusões acerca da eficiência dos modos de paralelismo das linguagens num contexto diferente, no qual muitas vezes o *overhead* das computações paralelas excede os benefícios da paralelização do código.

Das funções-objetivo pertencentes ao conjunto ZDT, as funções ZDT1 e ZDT2 têm geralmente cargas de trabalho mais elevadas e as funções ZDT3 e ZDT6 têm geralmente cargas de trabalho mais reduzidas.

Em seguida iremos analisar cada uma das funções ZDT nas execuções *multiprocessing*, *multithreading* e sequencial das diferentes linguagens para modos de inicialização DMSInit e DriverInit e médias T1-T5 e T2-T5.

4.2.1.2.1. Multiprocessing

As figuras e tabelas correspondentes às execuções *multiprocessing* das funções ZDT estão disponíveis no Apêndice B com melhor visibilidade (figuras 28-32 e tabelas 27-31).

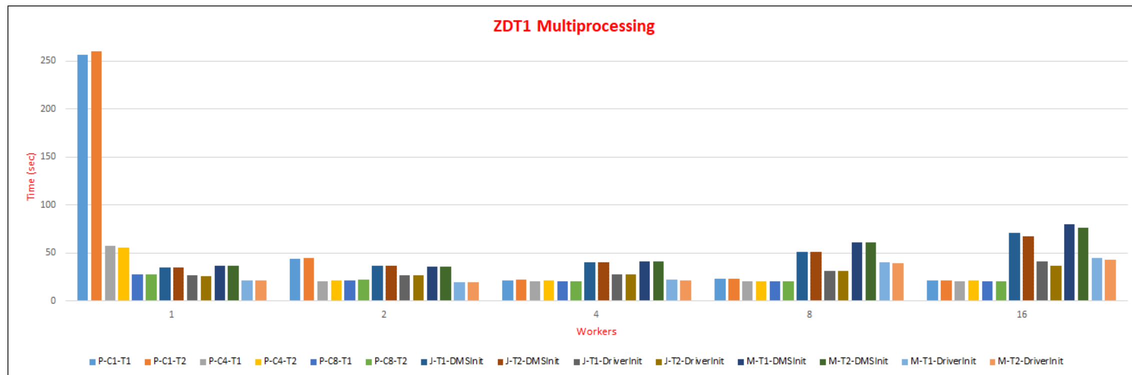


Figura 9 - Benchmarks ZDT1 Multiprocessing

- ZDT1- Estudo comparativo de eficiência das linguagens para DMSInit com ambas as médias, considerando o melhor *chunksize* de Python em cada um dos tamanhos do *pool* de *workers*:

  - Python é o mais eficiente para todos os *pool sizes*;
  - Matlab é o menos eficiente para todos os *pool sizes* exceto para *pool* de 2 *workers,* no qual Julia é o menos eficiente;
  - De entre todos os *pools sizes*, a execução mais eficiente é Python com *pool* de 4 *workers* e a menos eficiente é Matlab com *pool* de 16 *workers*.

- ZDT1- Principais diferenças para os resultados acima descritos quando consideramos todos os *chunksizes* de Python:

  - Python com *chunksize*=1 é o menos eficiente para *pools* com 1 e 2 *workers*;
  - De entre todos os *pools sizes*, a execução menos eficiente é Python com *chunksize*=1 e *pool* com 1 *worker.*

- ZDT1- Diferenças de resultados obtidos com DriverInit:

  - Matlab é o mais eficiente para *pools* com 1 e 2 *workers* e Julia é o menos eficiente para *pools* de 4 *workers*;

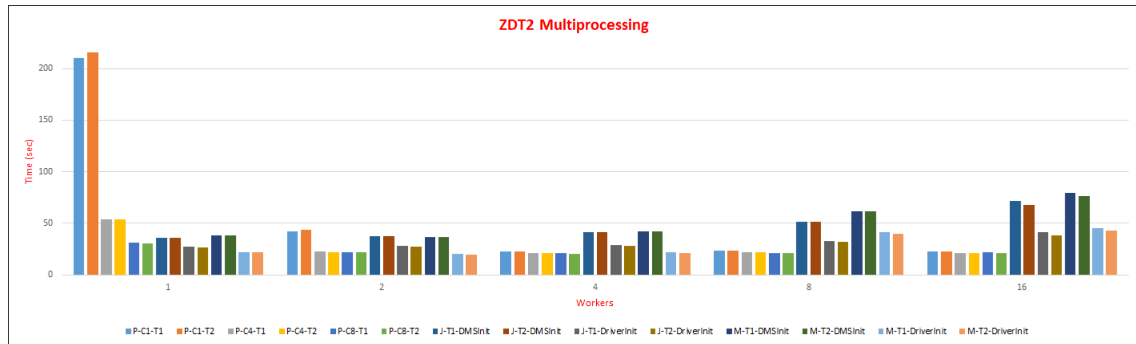


Figura 10 - Benchmarks ZDT2 Multiprocessing

- ZDT2- Estudo comparativo de eficiência das linguagens para DMSInit com ambas as médias, considerando o melhor *chunksize* de Python em cada um dos tamanhos do *pool* de *workers*:
  - Python é o mais eficiente para todos os *pool sizes*;
  - Matlab é o menos eficiente para todos os *pool sizes* exceto para *pool* de 2 *workers*, no qual Julia é o menos eficiente;
  - De entre todos os *pools sizes*, a execução mais eficiente é Python com *pool* de 4 *workers* e a menos eficiente é Matlab com *pool* de 16 *workers*.

- ZDT2- Principais diferenças para os resultados acima descritos quando consideramos todos os *chunksizes* de Python:
  - Python com *chunksize*=1 é o menos eficiente para *pools* com 1 e 2 *workers*;
  - De entre todos os *pools sizes*, a execução menos eficiente é Python com chunksize=1 e *pool* com 1 *worker*.

- ZDT2- Diferenças de resultados obtidos com DriverInit:
  - Considerando o melhor *chunksize* de Python: Matlab é o mais eficiente para *pools* com 1 e 2 *workers*, Python é o menos eficiente para *pool* de 1 *worker* e Julia é o menos eficiente para pools de 4 *workers*;
  - Considerando todos os *chunksizes* de Python Matlab é o mais eficiente para *pools* com 1 e 2 *workers* e Julia é o menos eficiente para pools de 4 *workers*;

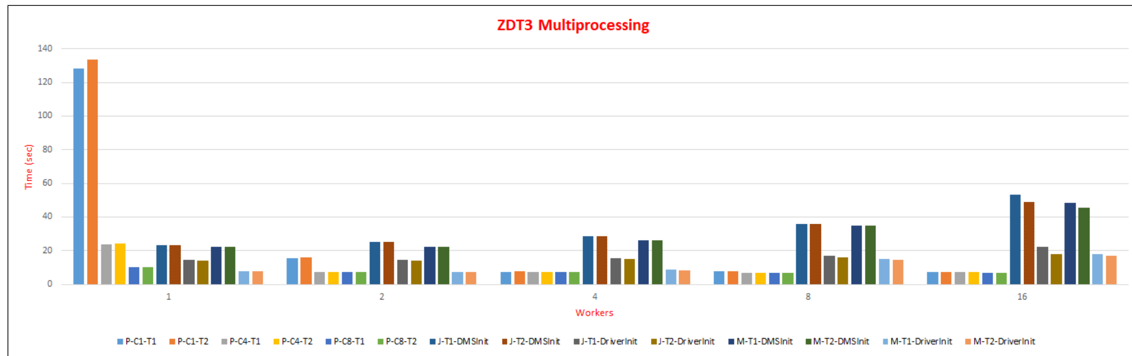


Figura 11 - Benchmarks ZDT3 Multiprocessing

- ZDT3- Estudo comparativo de eficiência das linguagens para DMSInit com ambas as médias, considerando o melhor *chunksize* de Python em cada um dos tamanhos do *pool* de *workers*:
  - Python é o mais eficiente para todos os *pool sizes*;
  - Julia é o menos eficiente para todos os *pool sizes*;
  - De entre todos os *pools sizes*, a execução mais eficiente é Python com *pool* de 16 *workers* e a menos eficiente é Julia com *pool* de 16 *workers*.
- ZDT3- Principais diferenças para os resultados acima descritos quando consideramos todos os *chunksizes* de Python:
  - Python com *chunksize*=1 é o menos eficiente para *pool* com 1 *worker*;
  - De entre todos os *pools sizes*, a execução menos eficiente é Python com chunksize=1 e *pool* com 1 *worker*.
- ZDT3- Diferenças de resultados obtidos com DriverInit:
  - Considerando o melhor *chunksize* de Python: Matlab é o mais eficiente para *pools* com 1 *worker;*
  - Considerando todos os *chunksizes* de Python: Matlab é o mais eficiente para *pools* com 1 *worker* e Python com *chunksize*=1 é o menos eficiente para *pools* de 2 *workers;*

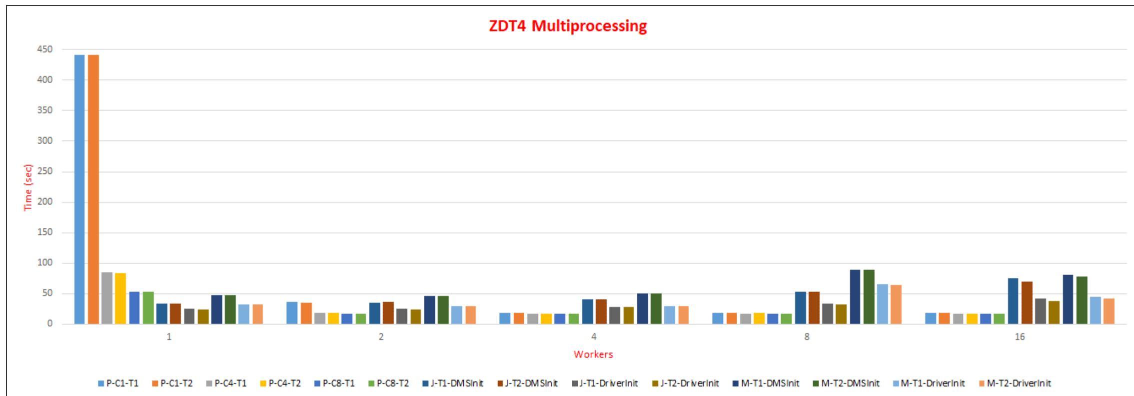


Figura 12 - Benchmarks ZDT4 Multiprocessing

- ZDT4- Estudo comparativo de eficiência das linguagens para DMSInit com ambas as médias, considerando o melhor *chunksize* de Python em cada um dos tamanhos do *pool* de *workers*:
    - Python é o mais eficiente para todos os *pool sizes* exceto para *pool* com 1 *worker*, para o qual Julia é o mais eficiente;
    - Matlab é o menos eficiente para todos os *pool sizes* exceto para *pool* com 1 *worker*, para o qual Python é o menos eficiente;
    - De entre todos os *pools sizes*, a execução mais eficiente é Python com *pool* de 4 *workers* e a menos eficiente é Matlab com *pool* de 8 *workers*.
- ZDT4- Principais diferenças para os resultados acima descritos quando consideramos todos os *chunksizes* de Python:
    - De entre todos os *pools sizes*, a execução menos eficiente é Python com chunksize=1 e *pool* com 1 *worker*.
- ZDT4- Diferenças de resultados obtidos com DriverInit:
    - Considerando todos os *chunksizes* de Python: Python com chunksize=1 é o menos eficiente para *pool* de 2 *workers*.

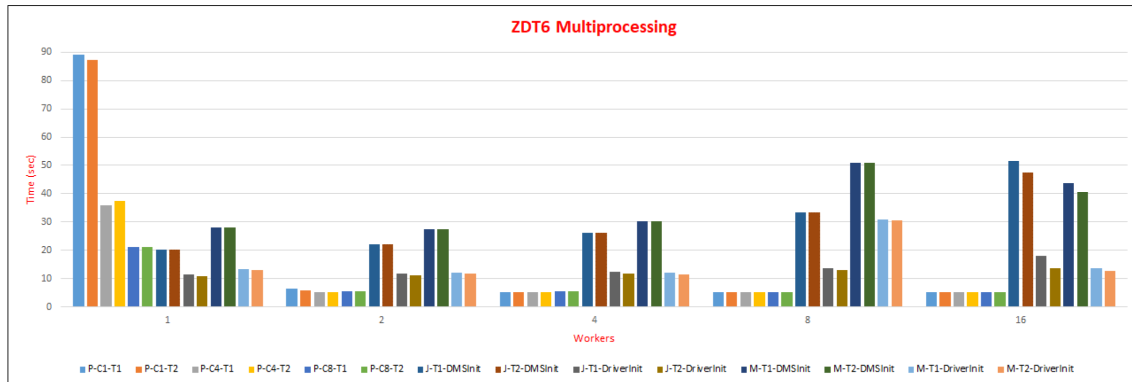


Figura 13 - Benchmarks ZDT6 Multiprocessing

- ZDT6- Estudo comparativo de eficiência das linguagens para DMSInit com ambas as médias, considerando o melhor *chunksize* de Python em cada um dos tamanhos do *pool* de *workers*:
  - Python é o mais eficiente para todos os *pool sizes* exceto para *pool* com 1 *worker*, para o qual Julia é o mais eficiente;
  - Matlab é o menos eficiente para todos os *pool sizes* exceto para *pool* com 16 *workers*, para o qual Julia é o menos eficiente;
  - De entre todos os *pools sizes*, a execução mais eficiente é Python com *pool* com 8 *workers* e a menos eficiente é Julia com *pool* de 16 *workers*.

- ZDT6- Principais diferenças para os resultados acima descritos quando consideramos todos os *chunksizes* de Python:
  - Python com *chunksize*=1 é o menos eficiente para *pool* com 1 *worker*;
  - De entre todos os *pools sizes*, a execução menos eficiente é Python com chunksize=1 e *pool* com 1 *worker*.

- ZDT6- Diferenças de resultados obtidos com DriverInit:
  - Considerando o melhor *chunksize* de Python: Python é o menos eficiente para *pool* com 1 *worker* e Julia é o menos eficiente para *pools* com 4 *workers*;
  - Considerando todos os *chunksizes* de Python: Julia é o menos eficiente para *pools* com 4 *workers*;

Algumas das principais conclusões gerais da execução *multiprocessing* para o conjunto das funções-objetivo ZDT são:

- De forma geral, Python é a linguagem mais eficiente (quando consideramos os tempos do seu melhor *chunksize*) em todas as situações estudadas para todas as funções ZDT. As únicas exceções são, para o modo DMSInit, o *pool* de 1 *worker* de algumas funções para as quais Julia é o mais eficiente e, para o modo DriverInit, os *pools* de 1 e 2 *workers* de algumas funções ZDT para as quais Matlab é o mais eficiente.
- Matlab nunca foi a linguagem mais eficiente no modo de inicialização DMSInit, ou seja, quando se conta o tempo total de execução do programa incluindo a criação de *pool* de *workers*;
- Na linguagem Python, o *chunksize*=8 é sempre o melhor e o *chunksize*=1 é sempre o pior (contrariamente ao que acontece com a função Styrene), sendo que este último produz tempos muito superiores a todas as outras execuções do modo *multiprocessing.* Acreditamos que seria relevante fazer um estudo mais exaustivo relativo aos resultados obtidos com chunksize=1 para averiguar o motivo da sua *performance* neste contexto;
- Para todas as funções ZDT, os tempos de execução das linguagens Matlab e Julia têm tendência a piorar com o aumento do *pool size* (contrariamente ao que acontece com a função Styrene) e os tempos de execução da linguagem Python têm tendência a diminuir ou manter-se semelhantes com o aumento do *pool size*;
- As conclusões obtidas utilizando as médias T1-T5 e T2-T5 são iguais em todos os casos;

4.2.1.2.2. Multithreading

As figuras e tabelas correspondentes às execuções *multithreading* das funções ZDT estão disponíveis no Apêndice B (figuras 34-38 e tabelas 33-37).

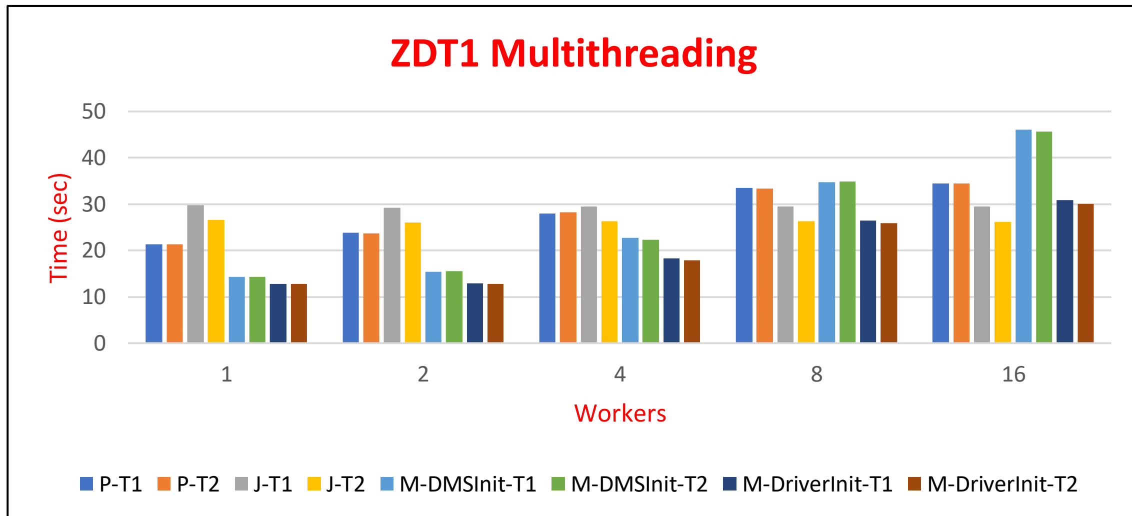


Figura 14 - Benchmarks ZDT1 Multithreading

- ZDT1- Estudo comparativo de eficiência das linguagens para DMSInit e médias T1-T5:
  - Matlab é o mais eficiente para os *pools* de 1, 2 e 4 *workers* e Julia é o mais eficiente para os *pools* de 8 e 16 *workers*;
  - Julia é o menos eficiente para os *pools* de 1, 2 e 4 *workers* e Matlab é o menos eficiente para os *pools* de 8 e 16 *workers*;
  - De entre todos os *pools sizes*, a execução mais eficiente é Matlab com *pool* de 1 *worker* e a menos eficiente é Matlab com *pool* de 16 *workers*.

- ZDT1- Principais diferenças de resultados observadas:
  - Para DriverInit e médias T1-T5: Matlab é o mais eficiente para *pool* de 8 *workers* e Python é o menos eficiente para pools com 8 e 16 *workers*;
  - Para DriverInit e médias T2-T5: Matlab é o mais eficiente para *pool* de 8 *workers* e Python é o menos eficiente para *pools* com 4, 8 e 16 *workers*;
  - Para DMSInit e médias T2-T5: Python é o menos eficiente para pool com 4 *workers*.

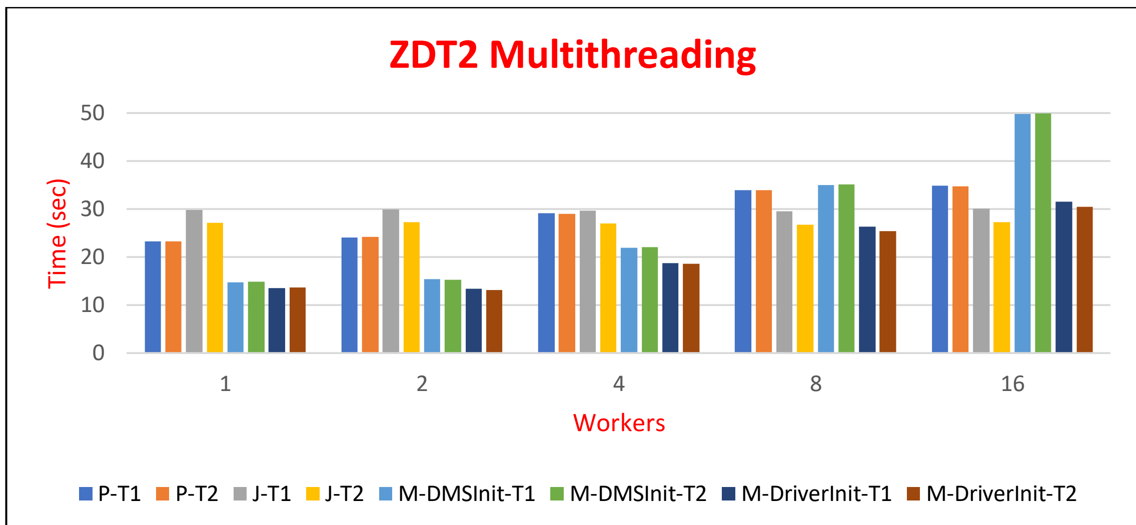


Figura 15 - Benchmarks ZDT2 Multithreading

- Todos os estudos feitos para a função ZDT2, com ambos os modos de inicialização e ambas as médias, produzem resultados iguais aos da função ZDT1.

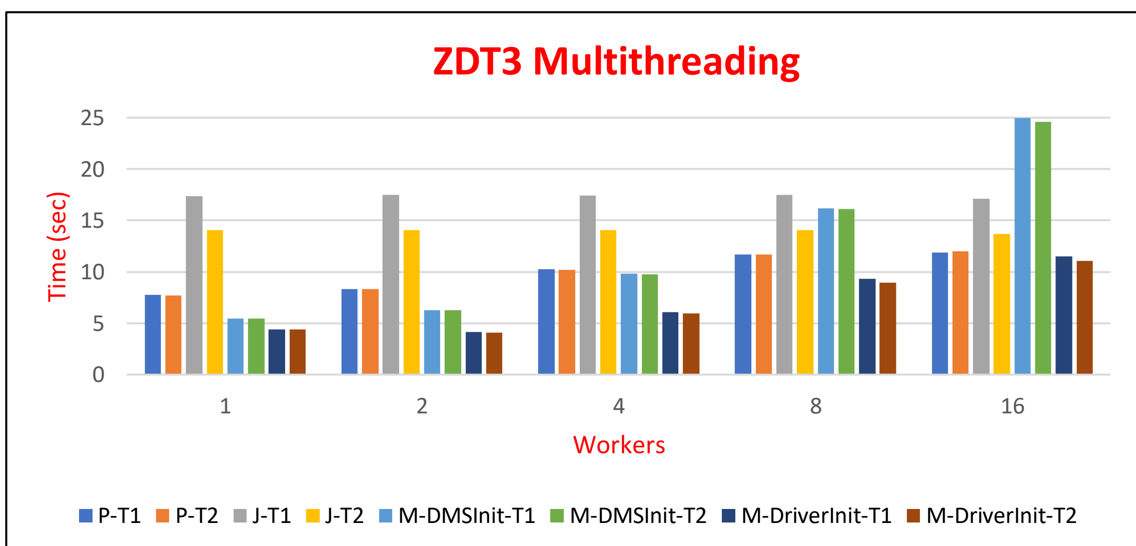


Figura 16 - Benchmarks ZDT3 Multithreading

- ZDT3- Estudo comparativo de eficiência das linguagens para DMSInit e médias T1-T5:

  - Matlab é o mais eficiente para os *pools* de 1, 2 e 4 *workers* e Python é o mais eficiente para os *pools* de 8 e 16 *workers*;
  - Julia é o menos eficiente para todos os *pool sizes* exceto *pool* de 16 *workers*, para o qual Matlab é o menos eficiente;

- De entre todos os *pools sizes*, a execução mais eficiente é Matlab com *pool* de 1 *worker* e o menos eficiente é Matlab com *pool* de 16 *workers.*

- ZDT3- Principais diferenças de resultados observadas:
  - Para DriverInit com ambas as médias: Para todos os *pool sizes*, Matlab passa a ser o mais eficiente e Julia o menos eficiente;
  - Para DMSInit e médias T2-T5: Matlab é o menos eficiente para *pool* de 8 *workers*.

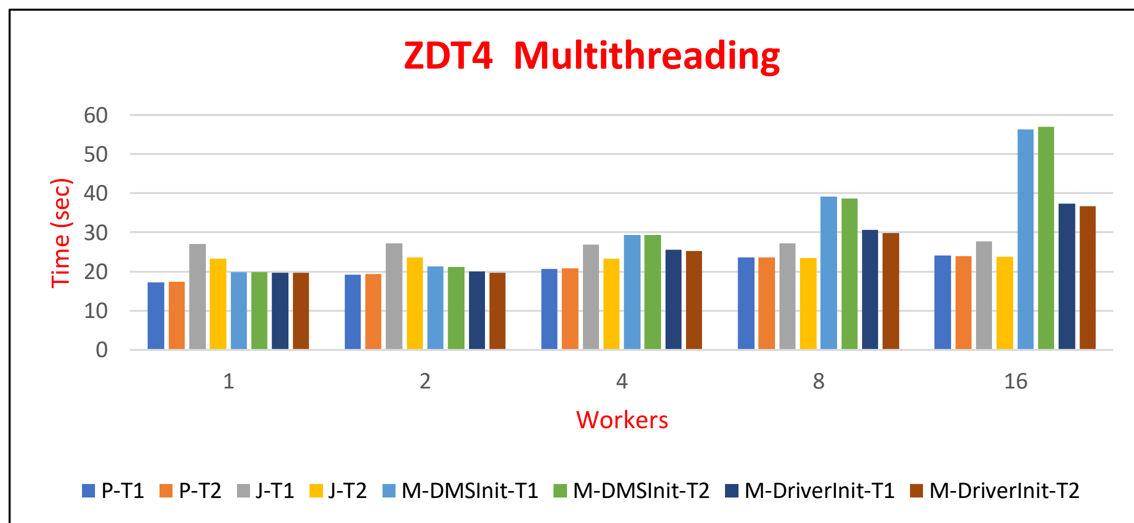


Figura 17 - Benchmarks ZDT4 Multithreading

- -ZDT4- Estudo comparativo de eficiência das linguagens para DMSInit com ambas as médias:
  - Python é o mais eficiente para todos os *pool sizes;*
  - Julia é o menos eficiente para os *pools* de 1 e 2 *workers* e Matlab é o menos eficiente para os *pools* de 4, 8 e 16 *workers;*
  - De entre todos os *pools sizes*, a execução mais eficiente é Python com *pool* de 1 *worker* e a menos eficiente é Matlab com *pool* de 16 *workers*.

- ZDT4- Principais diferenças de resultados observadas:
  - Para DriverInit com ambas as médias: Julia é o menos eficiente para pool de 4 workers;

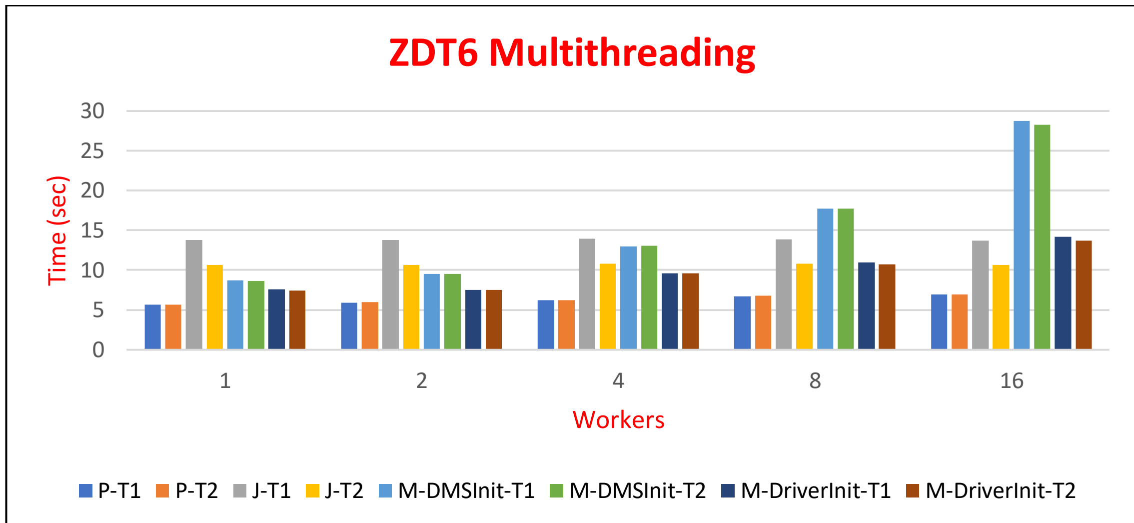


Figura 18 - Benchmarks ZDT6 Multithreading

- ZDT6- Estudo comparativo de eficiência das linguagens para DMSInit e médias T1-T5:
  - Python é o mais eficiente para todos os *pool sizes*;
  - Julia é o menos eficiente para os *pools* de 1, 2 e 4 *workers* e Matlab é o menos eficiente para os *pools* de 8 e 16 *workers;*
  - De entre todos os *pools sizes*, a execução mais eficiente é Python com *pool* de 1 *worker* e a menos eficiente é Matlab com *pool* de 16 *workers;*
- ZDT6- Principais diferenças de resultados observadas:
  - Para DriverInit com ambas as médias: Julia é o menos eficiente para *pool* de 8 *workers*;
  - Para DMSInit e médias T2-T5: Matlab é o menos eficiente para *pool* de 4 *workers*.

Algumas das principais conclusões gerais da execução *multithreading* para o conjunto das funções-objetivo ZDT são:

- A eficiência das linguagens varia bastante com a função-objetivo em estudo e com o *pool size*, não sendo possível retirar conclusões gerais claras sobre o comportamento das linguagens neste modo de execução. No entanto é possível concluir, para ambos os modos de inicialização, que Python é a línguagem mais eficiente para duas funções ZDT, Matlab é a línguagem mais eficiente para pools de menor dimensão (1, 2 e 4 workers) em 3 funções ZDT e Julia é apenas a mais eficiente para pools de

dimensão elevada (16 workers). As restantes situações variam consoante o modo de inicialização;

- Os tempos de execução das linguagens Matlab e Python têm tendência a piorar com o aumento do *pool size* e os tempos de execução da linguagem Julia, por sua vez, permanecem semelhantes em todos os *pool sizes,* não sendo observável a melhoria de desempenho ligeira que acontece com a função Styrene. A razão deste acontecimento, em nossa entender e após alguma pesquisa, parece derivar de o ambiente *multithreading* de Julia apenas produzir melhorias de desempenho relativas à utilização de múltiplas *threads* quando a carga de trabalho é suficiente elevada. No entanto, este é um tópico para o qual consideramos relevante efetuar um estudo mais exaustivo, analisando as condições em que o ambiente *multithreading* em Julia produz melhorias de desempenho significativas.

4.2.1.2.3. Sequential

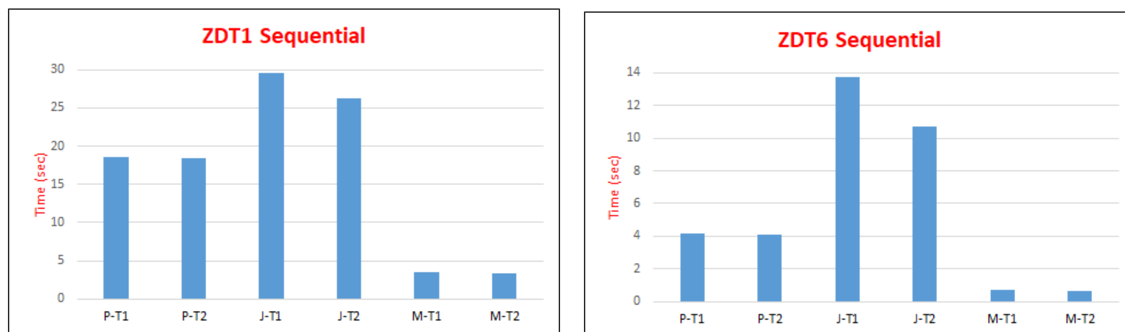


Figura 19 - Benchmarks ZDT1 e ZDT6 Sequential

As conclusões obtidas para todas as funções ZDT em modo sequencial são iguais. Assim, apresentamos de forma representativa os gráficos das funções ZDT1 e ZDT6. As figuras e tabelas correspondentes às execuções sequenciais do conjunto ZDT estão disponíveis no Apêndice B (figuras 40-44 e tabelas 39-43).

Para todas as funções ZDT, quer com médias T1-T5 ou T2-T5, Matlab é sempre a linguagem mais eficiente e Julia é sempre a menos eficiente.

4.2.1.2.4. Execução mais Eficiente de entre todos os Modos

Nesta secção são estudadas as execuções mais eficientes e menos eficientes de entre todos os modos de execução (*multiprocessing*, *multithreading* e sequencial) para todas as funções ZDT. Devido aos resultados obtidos neste estudo terem

bastantes variações, o que torna complexa a sua interpretação, optámos por adicionar ao Apêndice B (secção B.1.5) o estudo individual de cada uma das funções-objetivo. Assim, iremos apresentar nesta secção apenas as conclusões mais relevantes que foram retiradas deste estudo. Todas as figuras e tabelas de resultados utilizadas (relativas a ambos os modos de inicialização e ambas as médias) estão também disponíveis no Apêndice B (figuras 49-68 e tabelas 48-67).

Conclusões gerais observadas para o conjunto ZDT, de entre todos os modos de execução, são:

- Matlab-Sequencial é o mais eficiente de entre todos os modos de execução (*multiprocessing*, *multithreading*, sequencial) por uma margem elevada para todas as funções ZDT;
- Não é possível extrair conclusões comuns relativas às execuções menos eficientes de entre todos os modos de execução, uma vez que estas variam consoante a função-objetivo em análise e o tamanho do *pool* de *workers*;
- De forma geral, os resultados obtidos variam bastante entre os diferentes modos de execução (*multiprocessing*, *multithreading*, sequencial) sendo que é recomendado observar as conclusões gerais destacadas em cada um dos modos, não sendo viável destacar conclusões comuns entre eles;
- De forma geral (e com algumas exceções referidas nos diferentes modos de execução) podemos afirmar que o tempo de execução das funções ZDT tende a piorar com o aumento do *pool* de *workers* para os modos de execução paralelos.

4.2.1.3. Comparações de performance com implementação base

Nesta secção apresentamos o *speedup* [71] (métrica comparativa de *performance* entre implementações paralelas e sequenciais a executar o mesmo problema) e as melhorias de *performance* relativas às implementações feitas no âmbito desta Dissertação, quando comparadas com a implementação do algoritmo DMS inicialmente fornecido.

Como referido em capítulos anteriores, a implementação base do algoritmo que utilizámos no início desta Dissertação foi escrita em Matlab e dispõe de mecanismos de computação sequencial e computação paralela *multiprocessing*. As

implementações relativas às linguagens Python e Julia com mecanismos de programação *multiprocessing*, *multithreading* e sequencial foram realizadas nesta Dissertação, assim como a implementação dos mecanismos de programação *multithreading* em Matlab.

Tabela 1 - Styrene Multiprocessing - Speedup (vs. Matlab Sequential)

Styrene Multiprocessing - Speedup (vs. Matlab Sequential)						
Workers	P-C1-T1	Speedup	J-T1-DMSInit	Speedup	M-T1-DMSInit	Speedup
16	473,3	12,23	539,5	10,73	536,9	10,79
8	858,5	6,75	912,9	6,34	931,9	6,21
4	1630,0	3,55	1667,4	3,47	1756,5	3,30
2	2997,1	1,93	2996,3	1,93	3120,6	1,86
1	5724,3	1,01	5717,5	1,01	5874,4	0,99
Sequential	5714,0	1,01	5675,5	1,02	5790,3	1,00
	Python		Julia		Matlab	

A tabela 1 permite visualizar o *speedup* entre os tempos de execução obtidos para os modos *multiprocessing* (nos diferentes *pool sizes*) e sequencial das três linguagens e o modo sequencial da implementação (base) Matlab. Esta tabela é relativa à função-objetivo Styrene (sendo que esta é a função que proporciona as melhores análises de *performance* das linguagens) com modo DMSInit e a média T1-T5. Como é possível verificar, os valores obtidos são semelhantes para as três linguagens nos diferentes casos de estudo e o melhor *speedup* corresponde a Python-*Multiprocessing*-Chunksize=1.

Tabela 2 - Parallel Implementations- Performance Ratio

Parallel Implementations					
Func.	Best Initial Value		Best Final Value		Ratio
	Configuration	T1-T5	Configuration	T1-T5	
Styrene	M-MP-16-DMSInit	536,9	P-MP-16-C1	473,3	1,13
ZDT1	M-MP-2-DMSInit	36,1	M-MT-1-DMSInit	14,4	2,51
ZDT2	M-MP-2-DMSInit	36,8	M-MT-1-DMSInit	14,7	2,50
ZDT3	M-MP-1-DMSInit	22,2	M-MT-1-DMSInit	5,5	4,05
ZDT4	M-MP-2-DMSInit	45,5	P-MP-4-C4	16,8	2,71
ZDT6	M-MP-2-DMSInit	27,5	P-MP-8-C4	5,1	5,40

A tabela 2 representa as melhorias de *performance* (calculadas de forma igual ao *speedup*) entre a melhor execução paralela de cada uma das funções-objetivo implementadas e a melhor execução do modo *multiprocessing* da linguagem Matlab, para o modo DMSInit com média T1-T5. Este estudo pretende comparar

a *performance* de paralelismo das implementações feitas nesta Dissertação com a implementação Matlab inicialmente fornecida.

4.2.2. Microbenchmarks

4.2.2.1. Multiprocessing

As figuras apresentadas em seguida correspondem aos resultados da execução dos *microbenchmarks multiprocessing* nas três linguagens para os modos de inicialização DMSInit e DriverInit e médias T1-T5, utilizando matrizes de 20, 100 e 500 colunas. Todas as figuras e tabelas utilizadas neste estudo, incluindo as respetivas às médias T2-T5, estão disponíveis no Apêndice B com melhor visibilidade (figuras 70-73 e tabelas 70-73).

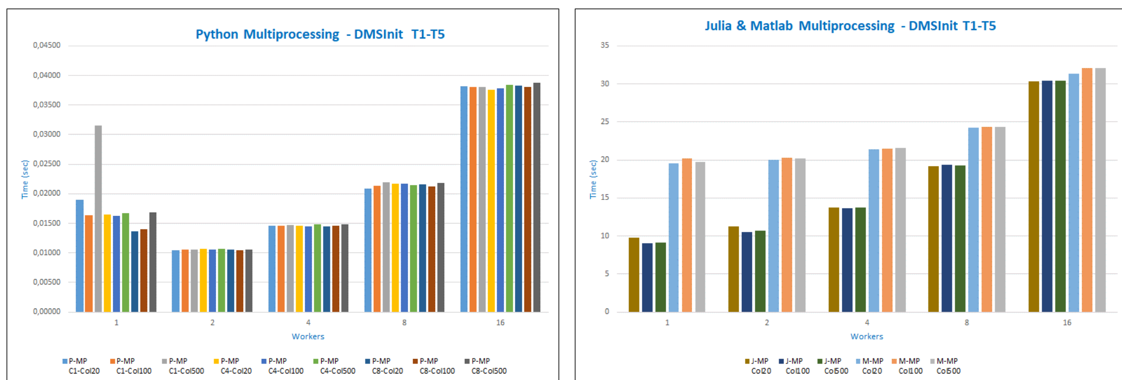


Figura 20 - Microbenchmarks Multiprocessing DMSInit T1-T5

- Resultados obtidos com DMSInit:
  - Existe uma diferença muito acentuada entre os tempos de execução obtidos em Python e nas linguagens Matlab e Julia;
  - Para todos os *pool sizes*, Python é sempre o mais eficiente e Matlab é sempre o menos eficiente.
  - Para todas as linguagens, o tempo de execução tende a aumentar com o aumento do tamanho do *pool* de *workers*, com exceção do *pool* de 1 *worker* da linguagem Python que apresenta valores mais elevados que o esperado;
  - Para a linguagem Python, não existe nenhuma diferença relevante para este estudo entre os tempos de execução obtidos com os diferentes *chunksizes*;

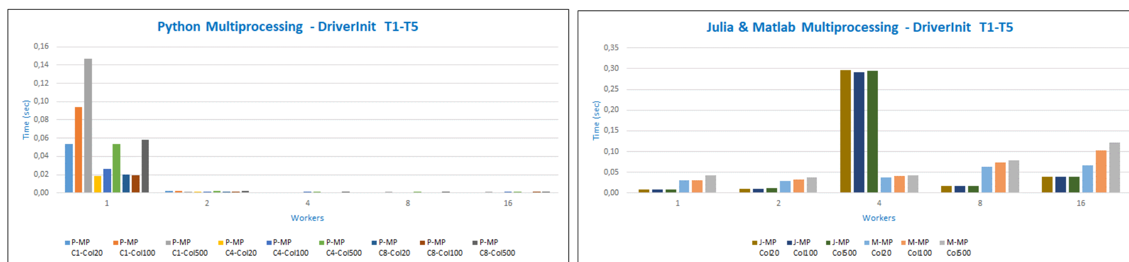


Figura 21 - Microbenchmarks Multiprocessing DriverInit T1-T5

- Resultados obtidos com DriverInit:
  - Já não existe uma disparidade tão acentuada de tempos de execução entre Python e as linguagens Julia e Matlab como é observável no modo de inicialização DMSInit, o que demonstra o impacto que a criação do *pool* de *workers* tem no tempo de execução das linguagens;
  - Python é a linguagem mais eficiente para todos os *pool sizes* exceto para *pool* de 1 *worker* (no qual apresenta valores anómalos), em que Julia é a mais eficiente. Para o *pool* de 4 *workers*, Julia apresenta valores muito elevados para os quais não houve oportunidade de investigar uma explicação;
  - Python tem tendência, de modo geral, a ficar mais eficiente com o aumento do *pool size*. Para as linguagens Julia e Matlab, o tempo de execução tem tendência a piorar com o aumento do *pool size*;

Conclusões de Microbenchmarks Multiprocessing:

- Existe uma diferença muito elevada entre os tempos de execução das linguagens Matlab e Julia nos modos DMSInit e DriverInit. Esta diferença salienta o tempo consumido pela criação do *pool* de *workers* nestas duas linguagens. Esta diferença também existe na linguagem Python, mas não é tão acentuada;
- Python é a linguagem mais eficiente em todos os casos de estudo para ambos os modos de inicialização, com a única exceção sendo o *pool* de 1 *worker* em modo DriverInit para o qual Julia é o mais eficiente. Não houve oportunidade de investigar mais aprofundadamente a razão da discrepância de valores associada ao *pool* de 1 *worker* de Python, sendo que seria um tópico relevante sobre o qual efetuar estudos futuros;
- Em Python, de um modo geral, o tempo de execução tende a aumentar com o aumento do *pool size* para DMSInit e tende a diminuir nas mesmas

condições para DriverInit. Em nosso entender, a explicação para este acontecimento parece derivar de o *overhead* de criação do *pool* de *workers* exceder os benefícios de paralelização em DMSInit, mas que quando o tempo de criação de *pool* não é contabilizado (DriverInit), um maior *pool* de *workers* corresponde a um melhor tempo de execução devido a uma maior produtividade no processamento da carga de trabalho;

- No caso das linguagens Julia e Matlab os tempos de execução aumentam ligeiramente com o aumento do *pool size* no modo DriverInit e aumentam na ordem dos segundos no modo DMSInit (devido ao *overhead* da criação do pool de *workers* neste modo);
- As conclusões obtidas são, de forma geral, semelhantes para os diferentes tamanhos de matrizes utilizados e são iguais para ambas as médias.

4.2.2.2. Multithreading

As figuras apresentadas em seguida correspondem aos resultados obtidos da execução dos *microbenchmarks multithreading* nas três linguagens, para ambos os modos de inicialização, médias T1-T5 e diferentes matrizes. Todas as figuras e tabelas utilizadas neste estudo, incluindo as respetivas à média T2-T5, estão disponíveis no Apêndice B (figuras 74-77 e tabelas 74-77).

É importante relembrar que, para a linguagem Julia, não existe distinção entre o modo DMSInit e DriverInit, uma vez que a criação do *pool* de *threads* é feita antes da execução do programa através de uma variável ambiente. Desta forma, utilizamos os mesmos dados relativos à execução Julia nos dois modos de inicialização, apesar de estes serem mais relevantes no contexto do modo DriverInit onde são avaliadas as *performances* de todas as linguagens excluindo o tempo de criação de *pool* de *workers*.

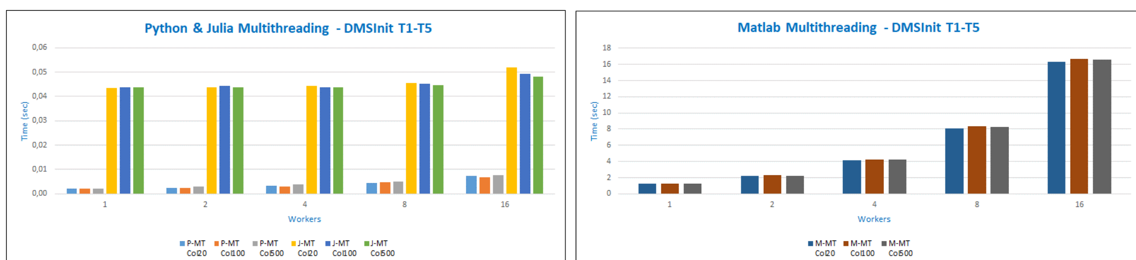


Figura 22 - Microbenchmarks Multithreading DMSInit T1-T5

- Resultados obtidos no modo DMSInit:
  - Existe uma diferença muito elevada entre os tempos de execução de Matlab e das linguagens Python e Julia;
  - Para todos os *pool sizes*, Python tem sempre os melhores tempos de execução e Matlab tem sempre os piores;
  - Para todas as linguagens, o tempo de execução tende a aumentar com o aumento do *pool size*. No caso da linguagem Julia esse aumento é muito ligeiro, sendo praticamente negligenciável;

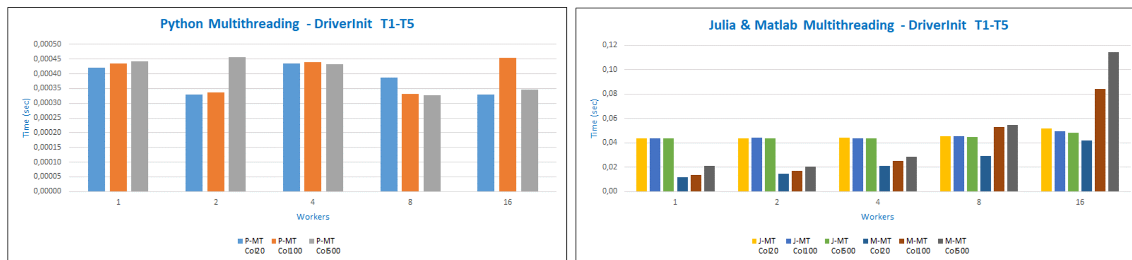


Figura 23 - Microbenchmarks Multithreading DriverInit T1-T5

- Resultados obtidos no modo DriverInit:
  - Já não existe uma disparidade tão acentuada de tempos de execução entre Matlab e as linguagens Julia e Python como é observável no modo DMSInit sendo que, efetivamente, Matlab passa a ter melhor desempenho que Julia para alguns dos *pool sizes*. Isto demonstra o impacto da criação do *pool* no tempo de execução de Matlab para execução *multithreading*;
  - Python é o mais eficiente para todos os *pool sizes*. Julia é o menos eficiente para *pools* de 1, 2 e 4 *workers* e Matlab é o menos eficiente (excluíndo os resultados com matriz de 20 colunas) para *pools* de 8 e 16 *workers*;
  - A linguagem Matlab é a única em que os tempos de execução aumentam de forma significativa com o tamanho da matriz utilizada, e esse aumento de tempo é tanto mais acentuado quando maior o *pool size*;
  - Para Matlab e Julia, o tempo de execução tende a piorar com o aumento do *pool size* (sendo que no caso da linguagem Julia, esse aumento é praticamente negligenciável), enquanto que os tempos de execução de Python não mostram um comportamento claro;

Conclusões de Microbenchmarks Multithreading:

- Python tem os resultados mais eficientes em todos os *pool sizes* para DMSInit e DriverInit;
- Os tempos de execução de Julia mantêm-se praticamente constantes para todos os *pool sizes*, como acontece na execução do algoritmo DMS Julia com as funções ZDT. Como referido anteriormente, acreditamos que este acontecimento se deve ao modo *multithreading* da linguagem Julia produzir melhorias de *performance* apenas para cargas de trabalho suficientemente elevadas;
- Existe uma diferença muito significativa entre os tempos de execução da linguagem Matlab nos modos DMSInit e DriverInit. Este fator é indicativo do impacto que a criação do *pool* de *workers* tem na *performance* desta linguagem, neste contexto;
- De modo geral, os tempos de execução nas diferentes linguagens pioram com o aumento do tamanho do *pool* de *workers* (sendo Python em modo DriverInit a exceção), o que é indicativo de um *overhead* proporcional ao tamanho do *pool*;
- As conclusões obtidas são, de forma geral, semelhantes para os diferentes tamanhos de matrizes utilizados e são iguais para ambas as médias.

4.2.2.3. Sequential

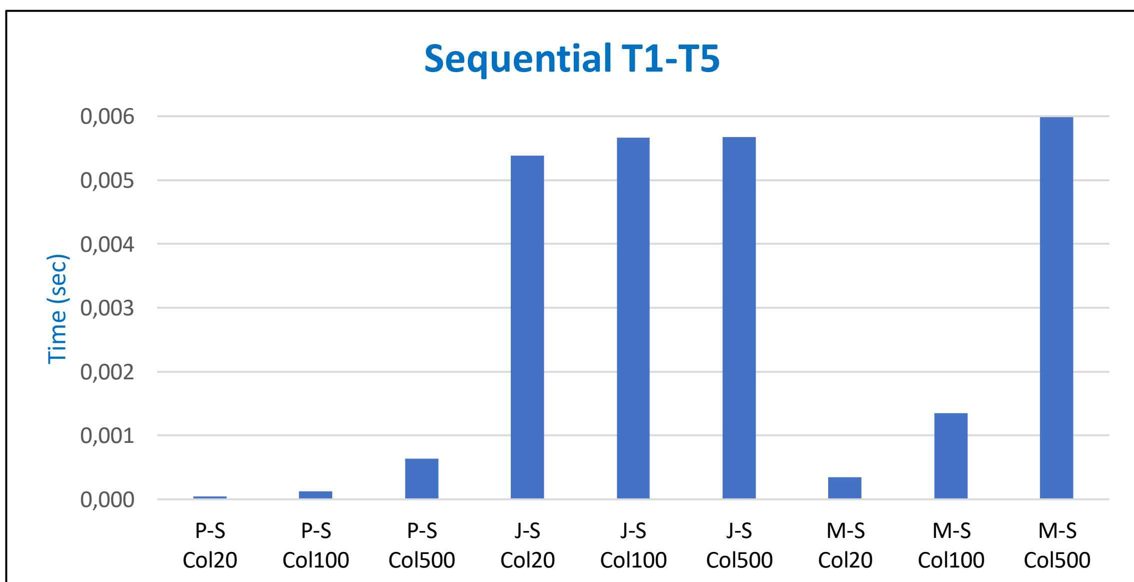


Figura 24 - Microbenchmarks Sequential T1-T5

As figuras e tabelas relativas ao modo sequencial, para médias T1-T5 e T2-T5, estão disponíveis no Apêndice B (figuras 78-79 e tabelas 78-79).

- **Conclusões de Microbenchmarks Sequencial:**
  - A linguagem Python tem os melhores tempos de execução para todos os tamanhos das matrizes em estudo. A execução mais eficiente é Python com matriz de 20 colunas e a menos eficiente é Matlab com matriz de 500 colunas;
  - Os tempos de execução da linguagem Matlab aumentam consideravelmente com o aumento de número de colunas da matriz utilizada. Este aumento também é visível na linguagem Python, embora seja menos acentuado;
  - As conclusões obtidas são semelhantes para as médias T1-T5 e T2-T5.

4.2.2.4. Diferenças entre modos DMSInit e DriverInit:

Apesar de a diferença de tempos de execução entre os modos DMSInit e DriverInit para os *microbenchmarks* poder ser deduzida através das análises feitas anteriormente, as figuras apresentados nesta secção permitem visualizar essa diferença de forma mais clara. A figura 30 apresentada abaixo corresponde aos resultados da execução *multiprocessing* para matriz de 20 colunas, sendo que as conclusões são as mesmas para todos os tamanhos de matrizes, neste estudo. As figuras e tabelas utilizadas neste estudo estão disponíveis no Apêndice B com melhor visibilidade (figuras 80-83 e tabelas 80-83).

4.2.2.4.1. Multiprocessing

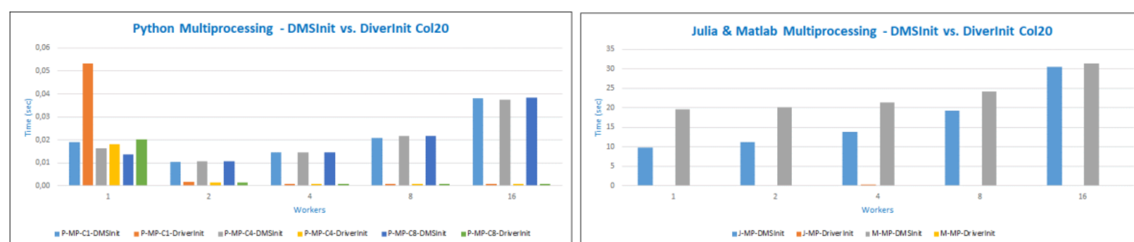


Figura 25 - Microbenchmarks Multiprocessing - DMSInit vs. DriverInit - 20 Cols.

- Para Python, o modo DMSInit produz sempre tempos mais elevados que DriverInit (com excepção do *pool* de 1 *worker*) e a diferença de tempos é cada vez mais acentuada quanto maior o *pool size*;

- Para Julia e Matlab a diferença de tempos entre DMSInit e DriverInit também aumenta com o tamanho do *pool size*, sendo esta diferença de uma magnitude muito superior à da linguagem Python (por exemplo, para o *pool* de 4 *workers*, Matlab com DMSInit demora 21.38 segundos e Matlab com DriverInit demora 0.0365 segundos).

4.2.2.4.2. Multithreading

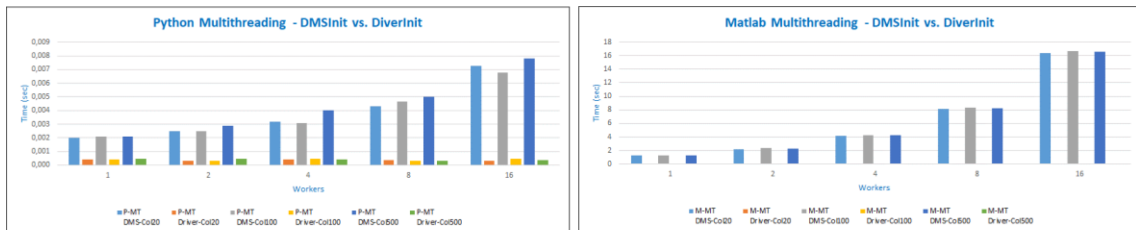


Figura 26 - Microbenchmarks Multithreading - DMSInit vs. DriverInit

- As conclusões para execução *multithreading* são semelhantes às de *multiprocessing*. Para todas as linguagens, o modo DMSInit tem sempre tempos mais elevados que DriverInit e a diferença de tempos é cada vez mais acentuada quanto maior o *pool size*. No entanto, para Matlab, a diferença entre os dois modos é de uma magnitude muito superior à de Python. Sendo que Julia não tem distinção entre modo DMSInit e DriverInit em *multithreading* não é possível fazer este estudo neste contexto.

Capítulo 5

Conclusões

As linguagens Python, Julia e Matlab são linguagens de programação muito populares entre cientistas e engenheiros de diferentes áreas, sendo que todas oferecem mecanismos para programação paralela. Com esta Dissertação pretendeu-se comparar estas linguagens e contribuir para uma melhor compreensão do desempenho esperado, em particular usando paralelismo. Para este efeito, recorremos à implementação do algoritmo de otimização DMS em Python, Julia e Matlab de forma o mais semelhante possível.

Os objetivos definidos no início da Dissertação foram:

- Implementação do algoritmo DMS nas linguagens Python e Julia com métodos para executar as funções-objetivo de forma paralela, recorrendo a *pools* de processos (*multiprocessing*) e a *pools* de threads (*multithreading*). No mesmo contexto, tivemos por objetivo adicionar mecanismos de paralelismo *multithreading* à implementação existente na linguagem Matlab;
- Implementação de funções-objetivo nas linguagens Python e Julia, como forma de avaliar o algoritmo DMS para diversas cargas de trabalho;
- Utilização das diferentes implementações do algoritmo DMS como forma de avaliar e comparar a *performance* do algoritmo nas diferentes linguagens, em especial no contexto de execução paralela;
- Criação de *microbenchmarks* que simulem o processo de paralelismo implementado nas diferentes linguagens para o algoritmo DMS, de modo a

avaliar e comparar a *performance* das primitivas responsáveis pelo paralelismo e pela criação do respetivo *pool* de *workers*.

Todos estes objetivos foram cumpridos: O algoritmo DMS foi implementado nas linguagens Python e Julia com mecanismos para execução sequencial e execução paralela *multithreading* e *multiprocessing*, sendo possível recorrer a *pools* de *workers* de diferentes tamanhos de modo a expandir as análises e comparações de *performance* possíveis para cada linguagem. Foi adicionada à implementação Matlab a opção de execução do algoritmo com paralelismo *multithreading*, assim como mecanismos para escolha do número de *workers* que constituem os *pools* paralelos, quer de processos ou *threads*. Foram implementadas 6 funções-objetivo nas linguagens Python e Julia que permitem o estudo do algoritmo DMS para diversas cargas de trabalho. Foram criadas diversas experiências para avaliar e comparar a *performance* do algoritmo DMS em Python, Julia e Matlab para execuções sequenciais e paralelas, sendo que o paralelismo é analisado para diferentes *pool sizes*. Foram criados *microbenchmarks* com vista a estudar, de forma isolada, as principais primitivas responsáveis pela execução paralela, nas 3 linguagens.

Para além dos objetivos inicialmente definidos no âmbito da Dissertação, foram realizadas as seguintes implementações e estudos adicionais:

- Implementação dos modos de inicialização DMSInit e DriverInit utilizados nas execuções paralelas, os quais permitem obter os tempos de execução do algoritmo DMS incluindo e excluindo o tempo de criação do *pool* de *workers*, respetivamente. Estes modos permitem analisar o impacto que o tempo de criação do *pool* paralelo tem no tempo total de execução do algoritmo e como esse impacto difere entre as linguagens. São efetuados estudos relativos a estes modos nos *benchmarks* e *microbenchmarks* implementados;
- Estudo comparativo das diversas execuções e análises feitas ao longo da Dissertação, a nível de *benchmarks* e *microbenchmarks*, utilizando as médias T1-T5 e T2-T5. Como foi explicado em maior detalhe nos capítulos 3 e 4, são sempre extraídos 5 resultados para cada um dos tipos de execução feitos nesta Dissertação e é utilizada sua média nos estudos do capítulo 4. No entanto, analisando os tempos extraídos das 5 execuções (disponíveis no Apêndice A), é possível observar situações nas linguagens

Julia e Matlab em que a primeira execução produz tempos superiores aos obtidos nas 4 execuções seguintes. Isto levantou a questão sobre se deveríamos basear as comparações relacionadas com os diferentes objetivos nas médias das 5 execuções feitas (médias T1-T5), ou na média das últimas 4 execuções, excluindo a primeira (médias T2-T5). Assim, este estudo permite avaliar se existem diferenças de conclusões para as diferentes análises feitas na Dissertação quando utilizamos cada uma das duas médias;

- Estudo comparativo de *performance* para o modo de paralelismo *multiprocessing* da linguagem Python relativamente aos diferentes *chunksizes* implementados, de forma a analisar quais os que resultam em maior e menor eficiência do algoritmo para os diferentes tamanhos do *pool* de *workers* e diferentes funções-objetivo.

Todos os estudos efetuados, quer dos objetivos iniciais quer dos posteriormente adicionados, produziram os resultados necessários para as análises e comparações pretendidas.

De modo geral, não houve impedimentos significativos aos objetivos que esta Dissertação se propôs a alcançar. O maior imprevisto que ocorreu ao longo da Dissertação foi relativo à impossibilidade de executar o algoritmo DMS na linguagem Matlab em modo *multithreading* para a função-objetivo Styrene. A causa desse impedimento, como explicado nos capítulos 3 e 4, foi a impossibilidade de executar um dos comandos internos da função Styrene em ambiente *multithreaded*. Esta impossibilidade provém de o ambiente *multithreaded* da linguagem Matlab apenas permitir a execução de um subconjunto das instruções disponíveis para ambiente *multiprocessing*, não estando deste modo relacionada com alguma falha relativa ao trabalho realizado durante a Dissertação. Em todo o caso, isto levou a que a comparação de *performance multithreading* com a função Styrene fosse efetuada apenas para as linguagens Python e Julia.

Algumas das conclusões mais relevantes obtidas através dos *benchmarks* implementados no capítulo 4 foram:

Execução Sequencial:

- Para a função-objetivo Styrene: Julia é a linguagem mais eficiente e Matlab é a linguagem menos eficiente;

- Para o conjunto ZDT: Matlab é a mais eficiente e Julia a menos eficiente. Matlab-Sequencial é, também, a execução mais eficiente do conjunto dos modos *multiprocessing*, *multithreading* e sequencial, sinal de que a utilização de programação paralela para estas funções não produz qualquer melhoria de *performance* devido à carga de trabalho a ser executada ser tão reduzida.

Execução Multiprocessing:

- Para a função-objetivo Styrene: Python é, de modo geral, a linguagem mais eficiente (só sendo menos eficiente que Julia para *pools* de 1 e 2 *workers* por uma margem muito reduzida) e Matlab é a menos eficiente;
- Para o conjunto ZDT: Python é, de modo geral, a linguagem mais eficiente. As únicas exceções são, para o modo DMSInit, o *pool* de 1 *worker* de algumas funções em que Julia é o mais eficiente e, para o modo DriverInit, os *pools* de 1 e 2 *workers* de algumas funções em que Matlab é o mais eficiente;
- Matlab não é a linguagem mais eficiente em nenhum caso de estudo quando contabilizamos o tempo de criação do *pool* de *workers* (modo DMSInit);
- Para a função Styrene, os tempos de execução de todas as linguagens (quando consideramos o melhor *chunksize* de Python) diminuem com o aumento do tamanho do *pool* de *workers*. Para as funções do conjunto ZDT, os tempos de execução das linguagens Julia e Matlab aumentam e os de Python mantêm-se semelhantes nas mesmas condições.
- Relativamente aos *chunksizes* da linguagem Python: Para a função Styrene, os resultados obtidos com *chunksize* 1 são sempre melhores que os obtidos com *chunksize* 4 e *chunksize* 8, sendo que ambos os *chunksizes* 4 e 8 deixam de ter melhorias de desempenho para os *pools* de maior dimensão (4, 8 e 16 *workers)*. Para as funções do conjunto ZDT, o *chunksize* 8 é o melhor para os *pools* de 1 e 2 *workers* e *chunksize* 1 é o pior, sendo que os valores de todos os *chunksizes* são semelhantes para os restantes *pool sizes*. Também para estas funções, o *chunksize* 1 produz valores muito superiores ao esperado, sendo que não tivemos oportunidade de fazer um estudo mais exaustivo para encontrar a razão desse acontecimento;

- Nas linguagens Matlab e Julia existe uma disparidade de tempos de execução entre os modos DMSInit e DriverInit que aumenta proporcionalmente com o tamanho do *pool* de *workers*. Esta disparidade é causada pelo tempo de criação do *pool* de *workers* e chega a atingir diferenças de tempo superiores a 15 segundos para os *pools* de 8 e 16 *workers* em ambas as linguagens;
- As linguagens mais eficientes nos diferentes casos de estudo de todas as funções-objetivo foram sempre as mesmas para ambas as médias T1-T5 e T2-T5;

Execução Multithreading:

- Para a função Styrene: Python é a linguagem mais eficiente (só sendo menos eficiente que Julia para *pool* de 1 *thread)*. É importante relembrar, no entanto, que Matlab não foi executado neste modo de paralelismo, não sendo assim incluído nesta comparação;
- Para o conjunto ZDT: A eficiência das linguagens varia bastante com a função-objetivo em estudo e com o *pool size*, não sendo possível retirar conclusões claras sobre o comportamento geral das linguagens neste modo de execução. No entanto é possível concluir, para ambos os modos de inicialização, que Python é a linguagem mais eficiente para duas funções ZDT, Matlab é a linguagem mais eficiente para pools de menor dimensão (1, 2 e 4 workers) em 3 funções ZDT e Julia é apenas a mais eficiente para pools de dimensão elevada (16 workers). As restantes situações variam consoante o modo de inicialização.
- Para a função Styrene, a eficiência de ambas as linguagens melhora com o aumento do tamanho do *pool* de *workers* (embora as melhorias de eficiência em Julia sejam negligenciáveis). Para o conjunto ZDT, os tempos de execução das linguagens Matlab e Python têm tendência a piorar com o aumento do *pool size* e os tempos de execução da linguagem Julia, por sua vez, permanecem semelhantes em todos os *pool sizes.*
- Na linguagem Matlab existe uma disparidade de tempos de execução entre os modos DMSInit e DriverInit que aumenta proporcionalmente com o tamanho do *pool* de *workers*. Esta disparidade é causada pelo tempo de criação do *pool* de *workers* e chega a atingir diferenças de tempo superiores a 8 segundos para os pools de 8 e 16 *workers;*

- Para a função Styrene, obtêm-se os mesmos resultados com ambas as médias. No caso do conjunto ZDT existem pequenas diferenças entre os resultados obtidos com T1-T5 e T2-T5, mas de modo geral as conclusões são semelhantes;

Foi também feito o estudo de comparação de *performance* entre a linguagem mais eficiente e a segunda mais eficiente para cada um dos modos de execução (*multiprocessing*, *multithreading*, sequencial), de modo a analisar potenciais situações em que uma execução seja muito mais eficiente que todas as outras no mesmo contexto. Este estudo foi realizado para todas as funções-objetivo com o modo DMSInit e média T1-T5 e corresponde à figura 68, presente no Apêndice B. As situações com um ratio (métrica semelhante ao *speedup* obtida através da divisão do tempo menos eficiente pelo mais eficiente) mais elevado para cada um dos modos de execução foram:

- Modo *multiprocessing* da função ZDT6, sendo Python-Chunksize=4 com *pool* de 8 *workers* a execução mais eficiente (ratio 4);
- Modo *multithreading* da função Styrene, sendo Python com *pool* de 16 *workers* a execução mais eficiente (ratio 11.35);
- Modo sequencial da função ZDT3, sendo Matlab a execução mais eficiente (ratio 6.94).

Após a conclusão desta Dissertação identificámos diversos estudos que podem ser relevantes para os utilizadores das linguagens Python, Julia e Matlab e que podem ser realizados em trabalhos futuros. Alguns exemplos de potenciais estudos são:

- Análise e comparação de *performance* das linguagens Python, Julia e Matlab a nível de execução paralela, em que Python recorre a bibliotecas diferentes da *Python Standard Library* para a implementação dos mecanismos de paralelismo. Essa análise permitiria completar este estudo com o de outras bibliotecas disponíveis em Python;
- Completar o estudo comparativo de *performance* das três linguagens a nível de paralelismo *multithreading* utilizando uma função-objetivo que proporcione uma carga de trabalho semelhante ou superior à oferecida pela função Styrene, mas em que o seu funcionamento interno seja possível executar no modo *multithreading* de Matlab. A impossibilidade de

executar a função Styrene em Matlab neste modo deixou a questão acerca da sua *performance* relativa às outras linguagens, neste contexto;

- Expandir o estudo do modo de paralelismo *multithreading* da linguagem Julia. Nesta Dissertação, os tempos extraídos para Julia em modo *multithreading* não demonstraram o comportamento esperado para os diferentes tamanhos de *pool* de *workers* utilizados, sendo que se mantiveram relativamente constantes. Embora tenhamos encontrado algumas justificações plausíveis para esses acontecimentos (apresentadas nos capítulos 3 e 4), acreditamos que seria benéfico para os utilizadores da linguagem Julia um estudo mais exaustivo no qual se avaliassem as condições em que o ambiente *multithreading* pode levar a melhorias de *performance* dos algoritmos utilizados, e os casos em que não produz alterações de *performance* significativas;
- Estudo de mecanismos que permitam diminuir o tempo de criação de *pools* de *workers* para as linguagens Julia e Matlab. Nesta Dissertação, Python revelou-se a linguagem mais eficiente na maioria das comparações feitas durante os estudos de *performance* das execuções paralelas em modo DMSInit das 3 linguagens. Analisando os *microbenchmarks* implementados nesta Dissertação relativos à comparação entre os modos de inicialização DMSInit e DriverInit, é notório o tempo de criação de *pool* de *workers* em Julia e Matlab. Este fator levanta a questão sobre qual seria a *performance* destas duas linguagens a nível de paralelismo, se a criação do *pool* de *workers* fosse mais eficiente.

Referências

[1] Sérgio Filipe Faustino Tavares. *Contributions to the development of an integrated toolbox of solvers in Derivative-Free Optimization*. Dissertação de Mestrado em Engenharia Informática, FCT/UNL, 2020. URL: https://docentes.fct.unl.pt/algb/files/masterthesis_sergiotavares.pdf (acesso mais recente em 24.11.2021).

[2] Chase Coleman, Spencer Lyon, Lilia Maliar and Serguei Maliar. "Matlab, Python, Julia: What to Choose in Economics?" In *Computational Economics*, 2020. DOI: https://doi.org/10.1007/s10614-020-09983-3

[3] Open Source Initiative. *The Open Source Definition*. URL: https://opensource.org/docs/osd (acesso mais recente em 24.11.2021).

[4] Gary King. "Replication, Replication." PS: Political Science and Politics 28, no. 3 (1995): 444-52. DOI: https://doi.org/10.2307/420301.

[5] Science Direct. *Numerical Scheme*. URL: https://www.sciencedirect.com/topics/engineering/numerical-scheme (acesso mais recente em 24.11.2021).

[6] Jan Gmys, Tiago Carneiro, Nouredine Melab, El-Ghazali Talbi and Daniel Tuyttens. "A comparative study of high-productivity high-performance programming languages for parallel metaheuristics". In: *Swarm and Evolutionary Computation*, Volume 57, 2020. DOI: https://doi.org/10.1016/j.swevo.2020.100720

[7] S.Boragan Aruoba and Jesús Fernández-Villaverde. "A comparison of programming languages in macroeconomics". In: *Journal of Economic Dynamics and Control*, Volume 58, Pages 265-273, 2015, DOI: https://doi.org/10.1016/j.jedc.2015.05.009

[8] Guido Van Rossum and Fred L. Drake Jr. *An Introduction to Python*. Network Theory Limited, 2003.

[9] Linda Dailey Paulson Paulson, "Developers shift to dynamic programming languages". In *Computer*, vol. 40, no. 2, pp. 12-15, 2007, DOI: https://doi.org/10.1109/MC.2007.53.

[10] Python. *Parallel Processing and Multiprocessing in Python*. URL: https://wiki.python.org/moin/ParallelProcessing (acesso mais recente em 24.11.2021).

[11] GreatLearning. *34 Open-Source Python Libraries You Should Know About*. URL: https://www.mygreatlearning.com/blog/open-source-python-libraries/ (acesso mais recente em 24.11.2021).

[12] Python. *Python documentation*. URL: https://docs.python.org/ (acesso mais recente em 24.11.2021).

[13] MathWorks. *Parallel Computing Toolbox*. URL: https://www.mathworks.com/help/parallel-computing/ (acesso mais recente em 24.11.2021).

[14] MathWorks. *Parallel Computing Toolbox User's Guide*. The MathWorks, Inc, 2020.

[15] MathWorks. *MATLAB Parallel Server*. URL: https://www.mathworks.com/help/matlab-parallel-server/ (acesso mais recente em 24.11.2021).

[16] MathWorks. *Choose Between Thread-Based and Process-Based Environments*. URL: https://www.mathworks.com/help/parallel-computing/choose-between-thread-based-and-process-based-environments.html (acesso mais recente em 24.11.2021).

[17] MathWorks. *Asynchronous Parallel Programming*. URL: https://www.mathworks.com/help/parallel-computing/asynchronous-parallel-programming.html (acesso mais recente em 24.11.2021).

[18] Julia. *Parallel Computing*. URL: https://docs.julialang.org/en/v1/manual/parallel-computing/ (acesso mais recente em 24.11.2021).

[19] Julia. *Asynchronous Programming*. URL: https://docs.julialang.org/en/v1/manual/asynchronous-programming/ (acesso mais recente em 24.11.2021).

[20] Julia. *Multi-Threading*. URL: https://docs.julialang.org/en/v1/manual/multi-threading/ (acesso mais recente em 24.11.2021).

[21] Julia. *Multi-processing and Distributed Computing*. URL: https://docs.julialang.org/en/v1/manual/distributed-computing/ (acesso mais recente em 24.11.2021).

[22] Wolfgang Gentzsch, Denis Girou, Alison Kennedy, Hermann Lederer, Johannes Reetz , Morris Riedel, Andreas Schott, Andrea Vanni, Mariano Vazquez and Jules Wolfrat. "DEISA-Distributed European Infrastructure for Supercomputing Applications". In: *J Grid Computing* 9:259–277, 2011. DOI: https://doi.org/10.1007/s10723-011-9183-2.

[23] Jack. J Dongarra, Piotr Luszczek and Antoine Petitet. "The LINPACK Benchamark: past, present and future". In: *Concurrency and Computation.: Practice and Experience* 15:803–820, 2003. DOI: https://doi.org/10.1002/cpe.728.

[24] CORDIS. *Distributed European Infrastructure for supercomputing applications*. URL: https://cordis.europa.eu/project/id/508830 (acesso mais recente em 24.11.2021).

[25] Michèle Weiland. "Profile of scientific applications on HPC architectures using the DEISA Benchmark Suite". In: *Computer Science - Research and Development* 25, 33–39, 2010. DOI: https://doi.org/10.1007/s00450-010-0103-7.

[26] The Standard Performance Evaluation Corporation (SPEC). URL: https://www.spec.org/ (acesso mais recente em 24.11.2021).

[27] The Standard Performance Evaluation Corporation. *SPEC's Benchmarks*. URL: https://www.spec.org/benchmarks.html (acesso mais recente em 24.11.2021).

[28] Jozo J. Dujmovic and Ivo Dujmovic. "Evolution and Evaluation of SPEC Benchmarks". In: *ACM SIGMETRICS Performance Evaluation Review* (December 1998). DOI: https://doi.org/10.1145/306225.306228

[29] Standard Performance Evaluation Corporation. *CPU 2017 Metrics*. URL: https://www.spec.org/cpu2017/Docs/overview.html#metrics. (acesso mais recente em 24.11.2021).

[30] Dijkstra E.W. "Cooperating Sequential Processes". In: *Hansen P.B. (eds) The Origin of Concurrent Programming*. Springer, New York, NY, 1968. DOI: https://doi.org/10.1007/978-1-4757-3472-0_2

[31] Lisandro Dalcin. *MPI for Python*, 2020. URL: https://mpi4py.readthedocs.io/en/stable/ (acesso mais recente em 24.11.2021).

[32] Jason J. Roberts, Benjamin D. Best, Daniel C. Dunn, Eric A. Treml and Patrick N. Halpin. "Marine Geospatial Ecology Tools: An integrated framework for ecological geoprocessing with ArcGIS, Python, R, MATLAB, and C++", In: *Environmental Modelling & Software*, Volume 25, Issue 10, Pages 1197-1207, 2010. DOI: https://doi.org/10.1016/j.envsoft.2010.03.029.

[33] Dimitri Yatsenko, Jacob Reimer, Alexander S. Ecker, Edgar Y. Walker, Fabian Sinz, Philipp Berens, Andreas Hoenselaar, R. James Cotton, Athanassios S. Siapas and Andreas S. Tolias. "DataJoint: managing big scientific data using MATLAB or Python", 2015. DOI: https://doi.org/10.1101/031658.

[34] A. L. Custódio, J. F. A. Madeira, A. I. F. Vaz and L. N. Vicente. "Direct Multisearch for Multiobjective Optimization". In: *SIAM Journal on Optimization*. 21(3). 1109- 1140. DOI: https://doi.org/10.1137/10079731X

[35] Jeffrey Larson, Matt Menickelly and Stefan M. Wild. Derivative-free optimization methods. In: *Acta Numerica* 28, p. 287-404, 2019. DOI: https://doi.org/10.1017/S0962492919000060. URL: https://arxiv.org/abs/1904.11585 (acesso mais recente em 24.11.2021).

[36] Agile Scientific. *Julia in a nutshell*. URL: https://agilescientific.com/blog/2014/9/4/julia-in-a-nutshell.html (acesso mais recente em 24.11.2021).

[37] Python Stable Releases. URL: https://www.python.org/downloads/widows/ (acesso mais recente em 24.11.2021).

[38] Towards Data Science. *5 Ways Julia Is Better Than Python*. URL: https://towardsdatascience.com/5-ways-julia-is-better-than-python-334cc66d64ae (acesso mais recente em 24.11.2021).

[39] DI-CLUSTER WIKI. *Hardware Details*. URL: https://cluster.di.fct.unl.pt/docs/technical/#hardware-details (acesso mais recente em 24.11.2021).

[40] MathWorks. *Control maximum number of computational threads*. URL: https://www.mathworks.com/help/matlab/ref/maxnumcompthreads.html (acesso mais recente em 24.11.2021).

[41] MathWorks. *Execute for-loop iterations in parallel on workers*. URL: https://www.mathworks.com/help/parallel-computing/parfor.html#d123e61374 (acesso mais recente em 24.11.2021).

[42] Python. *Process Pools*. URL: https://docs.python.org/3/library/multiprocessing.html#module-multiprocessing.pool (acesso mais recente em 24.11.2021).

[43] Python. *The multiprocessing.dummy module*. URL: https://docs.python.org/3/library/multiprocessing.html#module-multiprocessing.dummy (acesso mais recente em 24.11.2021).

[44] Python. *Map*. URL: https://docs.python.org/3/library/multiprocessing.html#multiprocessing.pool.Pool.map (acesso mais recente em 24.11.2021).

[45] Julia. *Distributed.addprocs*. URL: https://docs.julialang.org/en/v1/stdlib/Distributed/#Distributed.addprocs (acesso mais recente em 24.11.2021).

[46] Julia. *Distributed.rmprocs*. URL: https://docs.julialang.org/en/v1/stdlib/Distributed/#Distributed.rmprocs (acesso mais recente em 24.11.2021).

[47] Julia. *Distributed.@distributed*. URL: https://docs.julialang.org/en/v1/stdlib/Distributed/#Distributed.@distributed (acesso mais recente em 24.11.2021).

[48] Julia. *SharedArrays.SharedArray*. URL: https://docs.julialang.org/en/v1/stdlib/SharedArrays/#SharedArrays.SharedArray (acesso mais recente em 24.11.2021).

[49] Julia. *JULIA_NUM_THREADS*. URL: https://docs.julialang.org/en/v1/manual/environment-variables/#JULIA_NUM_THREADS (acesso mais recente em 24.11.2021).

[50] Julia. *Base.Threads.@threads*. URL: https://docs.julialang.org/en/v1/base/multi-threading/#Base.Threads.@threads (acesso mais recente em 24.11.2021).

[51] Julia. *Base.Threads.SpinLock*. URL: https://docs.julialang.org/en/v1/base/multi-threading/#Base.Threads.SpinLock (acesso mais recente em 24.11.2021).

[52] Dispy. *Dispy: Distributed and Parallel Computing with/for Python*. URL: https://dispy.org/ (acesso mais recente em 24.11.2021).

[53] RAY. URL: https://www.ray.io/ (acesso mais recente em 24.11.2021).

[54] Python. *Thread-based parallelism*. URL: https://docs.python.org/3/library/threading.html (acesso mais recente em 24.11.2021).

[55] GeeksforGeeks. *Python Daemon Threads*. URL: https://www.geeksforgeeks.org/python-daemon-threads/ (acesso mais recente em 24.11.2021).

[56] Python. *Multiprocessing — Process-based parallelism*. URL: https://docs.python.org/3/library/multiprocessing.html (acesso mais recente em 24.11.2021).

[57] Python. *Multiprocessing — Process-based parallelism - Sharing state between processes*. URL: https://docs.python.org/3/library/multiprocessing.html#sharing-state-between-processes (acesso mais recente em 24.11.2021).

[58] Python. *Multiprocessing.shared_memory — Provides shared memory for direct access across processes*. URL: https://docs.python.org/3/library/multiprocessing.shared_memory.html (acesso mais recente em 24.11.2021).

[59] Python. *Concurrent.futures — Launching parallel tasks*. URL: https://docs.python.org/3/library/concurrent.futures.html (acesso mais recente em 24.11.2021).

[60] Python. *Queue — A synchronized queue class*. URL: https://docs.python.org/3/library/queue.html (acesso mais recente em 24.11.2021).

[61] Julia. *Multi-processing and Distributed Computing – Data Movement*. URL: https://docs.julialang.org/en/v1/manual/distributed-compting/#Data-Movement (acesso mais recente em 24.11.2021).

[62] Julia. *Multi-processing and Distributed Computing – Remote References and AbstractChannels*. URL: https://docs.julialang.org/en/v1/manual/distributed-computing/#Remote-References-and-AbstractChannels (acesso mais recente em 24.11.2021).

[63] Julia. *Multi-processing and Distributed Computing – Shared Arrays*. URL: https://docs.julialang.org/en/v1/manual/distributed-compting/#man-shared-arrays (acesso mais recente em 24.11.2021).

[64] Julia. *Multi-processing and Distributed Computing – ClusterManagers*. URL: https://docs.julialang.org/en/v1/manual/distributed-computing/#ClusterManagers (acesso mais recente em 24.11.2021).

[65] C++ API documentation » ZDT test suite. URL: https://esa.github.io/pagmo2/docs/cpp/problems/zdt.html (acesso mais recente em 24.11.2021).

[66] Charles Audet, Vincent Béchard & Sébastien Le Digabel. Nonsmooth optimization through Mesh Adaptive Direct Search and Variable Neighborhood Search. J Glob Optim 41, p. 299–318 (2008). DOI: https://doi.org/10.1007/s10898-007-9234-1. URL: https://link.springer.com/article/10.1007%2Fs10898-007-9234-1#citeas.

[67] NumPy. URL: https://numpy.org/ (acesso mais recente em 24.11.2021).

[68] SciPy. URL: https://scipy.org/ (acesso mais recente em 24.11.2021).

[69] TensorFlow. URL: https://www.tensorflow.org/learn (acesso mais recente em 24.11.2021).

[70] Python. *_thread – Low-level threading API*¶. URL: https://docs.python.org/3/library/_thread.html (acesso mais recente em 24.11.2021).

[71] IGI Global. URL: https://www.igi-global.com/dictionary/multi-threaded-architectures/27999 (acesso mais recente em 24.11.2021).

APÊNDICES

APÊNDICE

RESULTADOS EXPERIMENTAIS

Apresentação das tabelas com os resultados extraídos das diferentes execuções relativas aos Benchmarks e Microbenchmarks implementados.

A.1. Benchmarks

Tabela 3 - Resultados Benchmarks Python Multiprocessing (1/2)

Python - Multiprocessing (1/2)										
Func.	Workers	Chunk Size	Time (sec)					Average (sec)		Standard Deviation
			T1	T2	T3	T4	T5	T1-T5	T2-T5	T1-T5
Styrene	1	1	5727,542	5726,211	5718,633	5725,394	5723,961	5724,348	5723,550	3,085
	1	4	5717,736	5720,256	5768,834	5722,929	5723,217	5730,594	5733,809	19,223
	1	8	5717,705	5719,785	5753,538	5773,991	5791,467	5751,297	5759,695	29,172
	2	1	3002,328	2994,782	2997,098	2996,352	2994,976	2997,107	2995,802	2,749
	2	4	3312,037	3314,731	3314,063	3314,591	3313,014	3313,687	3314,100	1,022
	2	8	3334,775	3335,516	3334,636	3365,251	3337,067	3341,449	3343,118	11,932
	4	1	1629,922	1629,653	1631,612	1629,493	1629,206	1629,977	1629,991	0,850
	4	4	1702,334	1701,194	1698,930	1702,053	1701,099	1701,122	1700,819	1,196
	4	8	3335,567	3338,208	3335,716	3335,092	3335,629	3336,042	3336,161	1,104
	8	1	857,864	859,686	857,352	859,147	858,239	858,458	858,606	0,849
	8	4	1703,997	1703,051	1702,863	1701,175	1704,799	1703,177	1702,972	1,219
	8	8	3332,934	3335,603	3334,315	3338,544	3335,924	3335,464	3336,096	1,867
	16	1	473,907	472,920	473,820	471,610	474,295	473,310	473,161	0,962
	16	4	1702,201	1702,671	1702,820	1701,861	1701,195	1702,150	1702,137	0,586
	16	8	3336,999	3337,997	3370,263	3336,232	3335,869	3343,472	3345,090	13,415
ZDT1	1	1	236,697	271,071	281,010	240,031	254,734	256,708	261,711	17,201
	1	4	65,614	50,525	54,596	53,923	62,653	57,462	55,424	5,697
	1	8	26,950	27,632	27,492	28,763	27,054	27,578	27,735	0,646
	2	1	42,829	46,065	41,590	48,685	41,562	44,146	44,476	2,801
	2	4	20,258	21,414	20,657	22,717	19,977	21,004	21,191	0,983
	2	8	19,701	21,346	22,642	22,203	21,864	21,551	22,014	1,017
	4	1	21,416	22,317	22,508	22,309	20,874	21,885	22,002	0,632
	4	4	19,064	20,100	22,757	22,074	20,876	20,974	21,452	1,328
	4	8	19,810	21,145	20,973	19,515	20,974	20,483	20,652	0,679
	8	1	23,250	22,950	22,978	23,426	22,704	23,062	23,015	0,251
	8	4	20,596	20,883	20,815	21,179	20,802	20,855	20,920	0,188
	8	8	21,706	20,900	21,001	20,641	20,688	20,987	20,807	0,383
	16	1	21,935	21,876	21,845	21,936	21,820	21,882	21,869	0,047
	16	4	20,670	21,095	20,940	21,328	21,047	21,016	21,103	0,214
	16	8	20,627	20,810	20,655	20,638	21,097	20,765	20,800	0,178
ZDT2	1	1	187,341	208,168	251,374	200,190	204,675	210,350	216,102	21,691
	1	4	54,431	50,398	48,693	49,328	66,947	53,959	53,841	6,794
	1	8	32,278	29,187	34,151	30,020	28,866	30,901	30,556	2,016
	2	1	34,279	45,077	45,104	43,394	42,547	42,080	44,031	4,023
	2	4	22,583	22,510	22,564	21,084	22,671	22,283	22,207	0,601
	2	8	22,634	21,117	22,536	22,469	19,918	21,735	21,510	1,065
	4	1	22,015	22,583	22,306	23,514	22,346	22,553	22,687	0,513
	4	4	19,533	19,683	20,445	21,764	22,203	20,726	21,024	1,081
	4	8	21,114	21,314	20,070	20,436	20,578	20,702	20,599	0,454
	8	1	22,713	23,263	23,377	23,322	23,612	23,257	23,393	0,297
	8	4	21,953	21,051	21,492	22,080	21,607	21,637	21,558	0,364
	8	8	21,479	21,822	21,195	21,193	20,662	21,270	21,218	0,382
	16	1	22,358	22,532	22,213	22,332	22,518	22,391	22,399	0,120
	16	4	21,169	21,258	21,348	21,268	21,331	21,275	21,301	0,063
	16	8	21,846	21,107	21,528	21,414	21,725	21,524	21,444	0,257

Tabela 4 - Resultados Benchmarks Python Multiprocessing (2/2)

Python - Multiprocessing (2/2)										
Func.	Workers	Chunk Size	Time (sec)					Average (sec)		Standard Deviation
			T1	T2	T3	T4	T5	T1-T5	T2-T5	T1-T5
ZDT3	1	1	106,772	125,611	141,893	138,891	128,750	128,383	133,786	12,392
	1	4	21,025	21,673	18,901	32,843	24,071	23,703	24,372	4,859
	1	8	10,344	10,130	9,935	10,200	9,879	10,098	10,036	0,171
	2	1	13,920	16,184	16,689	14,729	16,195	15,543	15,949	1,044
	2	4	7,498	7,542	7,399	6,900	7,473	7,363	7,329	0,236
	2	8	7,433	7,563	6,949	7,008	6,809	7,153	7,082	0,292
	4	1	6,559	8,007	6,941	7,797	7,464	7,354	7,552	0,536
	4	4	6,605	7,503	6,701	7,461	7,218	7,097	7,221	0,377
	4	8	6,675	7,344	7,502	7,060	7,030	7,122	7,234	0,285
	8	1	7,575	7,705	7,543	7,526	7,754	7,621	7,632	0,092
	8	4	7,117	6,910	7,240	7,213	6,765	7,049	7,032	0,184
	8	8	6,836	6,972	6,911	6,977	7,000	6,939	6,965	0,059
	16	1	7,519	7,458	7,654	7,490	7,494	7,523	7,524	0,068
	16	4	6,753	7,151	7,073	7,156	7,275	7,082	7,164	0,176
	16	8	6,861	6,961	6,976	6,911	6,708	6,883	6,889	0,097
ZDT4	1	1	444,047	438,087	492,276	388,402	446,339	441,830	441,276	32,981
	1	4	92,234	77,660	85,899	69,991	102,747	85,706	84,074	11,355
	1	8	56,244	45,935	57,180	47,223	62,064	53,729	53,100	6,177
	2	1	37,536	39,915	29,919	37,878	34,951	36,040	35,666	3,443
	2	4	18,424	18,412	18,295	18,104	16,608	17,969	17,855	0,690
	2	8	18,314	18,294	18,241	18,240	15,373	17,692	17,537	1,160
	4	1	17,750	18,270	17,457	18,770	18,484	18,146	18,246	0,480
	4	4	16,155	17,868	16,107	16,912	16,930	16,795	16,954	0,643
	4	8	17,675	18,152	16,984	17,099	16,798	17,342	17,258	0,500
	8	1	18,359	18,129	18,707	17,721	17,880	18,159	18,109	0,350
	8	4	17,534	17,610	17,740	18,137	17,523	17,709	17,752	0,228
	8	8	17,212	17,357	17,436	17,558	17,166	17,346	17,379	0,144
	16	1	18,211	17,999	18,346	18,030	18,207	18,159	18,145	0,128
	16	4	17,486	17,374	17,392	17,476	17,270	17,400	17,378	0,078
	16	8	17,143	17,191	17,171	17,157	17,199	17,172	17,179	0,021
ZDT6	1	1	97,405	88,284	92,220	83,732	84,500	89,228	87,184	5,085
	1	4	30,296	32,576	43,400	40,123	33,528	35,985	37,407	4,942
	1	8	21,668	21,408	22,116	21,275	19,826	21,259	21,156	0,772
	2	1	9,591	5,756	5,695	6,053	5,590	6,537	5,774	1,535
	2	4	5,303	5,143	5,247	4,977	5,206	5,175	5,143	0,112
	2	8	5,623	5,580	5,384	5,508	5,528	5,524	5,500	0,081
	4	1	5,004	5,172	5,239	5,135	5,242	5,158	5,197	0,087
	4	4	5,122	4,980	4,996	5,218	5,341	5,131	5,134	0,136
	4	8	5,257	5,459	5,450	5,698	5,546	5,482	5,538	0,143
	8	1	5,166	5,190	5,205	5,172	5,210	5,189	5,194	0,017
	8	4	5,073	5,096	5,128	5,029	5,143	5,094	5,099	0,041
	8	8	5,287	5,230	5,346	5,325	5,317	5,301	5,305	0,040
	16	1	5,330	5,229	5,399	5,282	5,334	5,315	5,311	0,057
	16	4	5,305	5,215	5,309	5,337	5,254	5,284	5,279	0,044
	16	8	5,470	5,426	5,420	5,305	5,376	5,399	5,381	0,056

Tabela 5 - Resultados Benchmarks Python Multithreading

Python - Multithreading									
Func.	Threads	Time (sec)					Average (sec)		Standard Deviation
		T1	T2	T3	T4	T5	T1-T5	T2-T5	T1-T5
Styrene	1	5717,837	5714,201	5719,413	5714,430	5717,333	5716,643	5716,345	2,022
	2	3053,026	3052,420	3053,730	3054,350	3107,282	3064,161	3066,945	21,570
	4	1627,222	1629,080	1632,118	1627,857	1625,855	1628,427	1628,728	2,119
	8	862,481	862,357	859,889	861,643	860,897	861,454	861,197	0,966
	16	498,170	491,863	498,509	501,097	499,780	497,884	497,812	3,183
ZDT1	1	21,615	21,603	21,272	19,776	22,667	21,387	21,330	0,932
	2	23,916	23,740	24,055	23,981	23,077	23,754	23,713	0,354
	4	26,703	28,549	28,477	28,319	27,781	27,966	28,282	0,686
	8	33,805	33,099	34,141	33,511	32,748	33,461	33,375	0,494
	16	34,690	34,822	34,324	34,733	33,990	34,512	34,467	0,311
ZDT2	1	23,003	23,182	23,170	23,214	23,208	23,155	23,193	0,078
	2	23,700	23,527	24,265	24,420	24,322	24,047	24,133	0,361
	4	29,351	28,808	27,385	30,127	29,540	29,042	28,965	0,930
	8	33,932	33,849	33,916	33,975	33,714	33,877	33,863	0,091
	16	35,175	34,981	33,885	34,915	35,184	34,828	34,741	0,483
ZDT3	1	7,997	7,767	7,853	7,496	7,771	7,777	7,722	0,163
	2	8,308	8,330	8,266	8,297	8,391	8,319	8,321	0,042
	4	10,536	10,900	9,420	10,251	10,117	10,245	10,172	0,492
	8	11,624	11,590	11,823	11,843	11,556	11,687	11,703	0,121
	16	11,632	11,604	12,279	12,186	11,818	11,904	11,972	0,280
ZDT4	1	16,673	17,294	18,036	18,124	15,760	17,177	17,304	0,884
	2	19,170	19,330	19,401	19,339	19,062	19,261	19,283	0,125
	4	20,252	21,014	20,857	21,412	19,955	20,698	20,810	0,526
	8	23,551	23,341	23,017	24,311	23,711	23,586	23,595	0,431
	16	24,483	24,312	24,189	24,382	23,155	24,104	24,009	0,484
ZDT6	1	5,636	5,695	5,558	5,738	5,781	5,682	5,693	0,078
	2	5,893	5,975	5,917	6,029	5,830	5,929	5,938	0,068
	4	6,113	5,953	6,471	6,195	6,211	6,189	6,208	0,168
	8	6,558	6,576	7,256	6,633	6,563	6,717	6,757	0,271
	16	6,657	6,891	7,249	7,086	6,687	6,914	6,978	0,228

Tabela 6 - Resultados Benchmarks Python Sequential

Python - Sequential								
Func.	Time (sec)					Average (sec)		Standard Deviation
	T1	T2	T3	T4	T5	T1-T5	T2-T5	T1-T5
Styrene	5713,666	5715,915	5708,985	5716,039	5715,220	5713,965	5714,040	2,630
ZDT1	19,396	18,401	18,388	18,444	18,356	18,597	18,398	0,400
ZDT2	19,865	18,934	18,898	18,891	18,903	19,098	18,906	0,384
ZDT3	6,683	6,463	6,464	6,466	6,462	6,508	6,464	0,088
ZDT4	15,328	15,023	15,011	14,979	14,968	15,062	14,995	0,135
ZDT6	4,183	4,112	4,142	4,105	4,108	4,130	4,117	0,030

Tabela 7 - Resultados Benchmarks Julia Multiprocessing - DriverInit

Julia - Multiprocessing - DriverInit									
Func.	Workers	Time (sec)					Average (sec)		Standard Deviation
		T1	T2	T3	T4	T5	T1-T5	T2-T5	T1-T5
Styrene	1	5730,784	5681,125	5680,045	5685,206	5677,717	5690,975	5681,023	20,052
	2	2986,662	2975,516	2973,983	2973,836	3005,276	2983,055	2982,153	12,091
	4	1654,482	1641,378	1641,617	1644,374	1642,671	1644,904	1642,510	4,904
	8	893,400	880,340	879,423	881,091	880,286	882,908	880,285	5,273
	16	526,354	491,316	486,581	488,931	492,194	497,075	489,755	14,770
ZDT1	1	28,657	25,938	25,884	25,814	26,571	26,573	26,052	1,077
	2	28,968	26,510	26,394	26,516	26,501	26,978	26,480	0,996
	4	30,282	27,459	28,129	27,760	27,337	28,193	27,671	1,080
	8	34,674	31,072	32,181	30,648	30,612	31,837	31,128	1,527
	16	57,477	37,692	36,748	37,387	36,990	41,259	37,204	8,116
ZDT2	1	29,753	26,925	26,648	26,669	26,722	27,343	26,741	1,209
	2	29,656	27,324	27,144	27,208	27,310	27,728	27,246	0,966
	4	30,808	28,264	27,817	28,208	27,934	28,606	28,056	1,113
	8	36,230	32,233	32,696	31,657	31,858	32,935	32,111	1,685
	16	55,498	38,806	37,129	36,890	39,316	41,528	38,035	7,047
ZDT3	1	16,795	14,091	14,082	14,035	14,111	14,623	14,080	1,086
	2	16,903	14,271	14,189	14,199	14,172	14,747	14,208	1,078
	4	17,877	15,080	15,578	14,445	14,378	15,471	14,870	1,281
	8	19,445	16,134	16,305	16,161	16,336	16,876	16,234	1,287
	16	39,302	18,132	17,928	17,725	18,540	22,325	18,081	8,493
ZDT4	1	27,517	23,935	24,295	23,923	24,854	24,905	24,252	1,349
	2	28,346	24,715	24,415	24,653	24,515	25,329	24,575	1,512
	4	30,396	26,612	27,669	28,439	27,255	28,074	27,494	1,304
	8	36,033	33,423	32,365	31,874	31,602	33,059	32,316	1,612
	16	61,305	37,927	37,942	38,316	37,986	42,695	38,043	9,306
ZDT6	1	13,796	11,024	11,007	10,915	11,030	11,554	10,994	1,121
	2	13,728	11,483	11,048	11,050	11,010	11,664	11,148	1,047
	4	14,812	11,894	12,015	11,964	11,868	12,511	11,935	1,152
	8	16,422	13,156	13,130	13,285	13,079	13,815	13,163	1,306
	16	35,800	13,799	13,707	13,521	13,506	18,066	13,633	8,867

Tabela 8 - Resultados Benchmarks Julia Multiprocessing - DMSInit

Julia - Multiprocessing - DMSInit									
Func.	Workers	Time (sec)					Average (sec)		Standard Deviation
		T1	T2	T3	T4	T5	T1-T5	T2-T5	T1-T5
Styrene	1	5734,533	5753,824	5699,979	5702,413	5696,948	5717,539	5713,291	22,656
	2	2998,757	2997,829	2994,511	2994,737	2995,725	2996,312	2995,701	1,694
	4	1665,310	1668,177	1667,581	1667,766	1668,130	1667,393	1667,913	1,065
	8	915,913	911,833	911,295	910,701	914,948	912,938	912,194	2,089
	16	562,578	531,201	535,489	531,787	536,328	539,477	533,701	11,723
ZDT1	1	35,333	35,584	35,339	35,434	35,383	35,414	35,435	0,092
	2	37,005	36,828	37,001	37,670	37,086	37,118	37,146	0,288
	4	40,891	41,324	41,037	40,355	40,618	40,845	40,833	0,335
	8	51,397	51,305	50,031	51,385	50,801	50,984	50,880	0,525
	16	87,292	67,696	67,106	66,923	66,568	71,117	67,073	8,096
ZDT2	1	36,267	36,072	36,093	36,331	36,377	36,228	36,218	0,124
	2	37,507	37,724	37,511	37,627	37,444	37,563	37,576	0,100
	4	41,252	41,729	40,856	40,817	41,083	41,147	41,121	0,331
	8	51,038	51,512	51,422	52,053	50,771	51,359	51,440	0,438
	16	86,445	68,244	68,382	67,453	67,725	71,650	67,951	7,405
ZDT3	1	23,174	23,212	23,530	23,500	23,301	23,344	23,386	0,146
	2	24,997	24,934	25,564	24,722	24,926	25,029	25,036	0,283
	4	28,186	28,633	28,297	28,099	28,726	28,388	28,439	0,248
	8	35,994	35,417	36,524	36,030	35,882	35,969	35,963	0,354
	16	70,315	49,138	49,119	49,168	48,650	53,278	49,019	8,520
ZDT4	1	33,543	34,459	33,685	34,140	34,206	34,007	34,123	0,341
	2	35,563	35,752	35,524	35,842	36,165	35,769	35,821	0,230
	4	41,469	41,434	41,385	40,596	40,936	41,164	41,088	0,343
	8	54,031	53,333	53,313	53,751	53,658	53,617	53,513	0,270
	16	93,401	70,160	69,490	70,197	70,014	74,652	69,965	9,378
ZDT6	1	20,196	20,339	20,395	20,452	20,429	20,362	20,404	0,091
	2	22,292	22,946	21,919	22,078	21,825	22,212	22,192	0,400
	4	26,542	25,910	26,838	25,433	26,459	26,237	26,160	0,501
	8	33,772	33,249	33,541	33,318	33,337	33,443	33,361	0,191
	16	67,048	47,181	47,403	47,869	47,568	51,414	47,505	7,820

Tabela 9 - Resultados Benchmarks Julia Multithreading

Julia - Multithreading									
Func.	Threads	Time (sec)					Average (sec)		Standard Deviation
		T1	T2	T3	T4	T5	T1-T5	T2-T5	T1-T5
Styrene	1	5723,468	5682,110	5679,731	5683,137	5718,926	5697,474	5690,976	19,454
	2	5722,466	5708,167	5683,628	5669,972	5672,040	5691,255	5683,452	20,690
	4	5705,541	5678,276	5675,621	5676,626	5689,074	5685,028	5679,899	11,330
	8	5689,621	5669,086	5668,863	5669,248	5668,385	5673,040	5668,895	8,295
	16	5669,925	5648,595	5647,103	5646,539	5650,048	5652,442	5648,071	8,826
ZDT1	1	42,497	26,613	26,570	26,369	26,497	29,709	26,512	6,394
	2	42,160	26,000	26,155	25,718	26,062	29,219	25,984	6,472
	4	42,162	26,233	26,411	26,504	26,256	29,513	26,351	6,325
	8	42,430	26,255	26,494	26,443	25,979	29,520	26,293	6,457
	16	42,875	26,186	26,100	25,864	26,275	29,460	26,106	6,709
ZDT2	1	40,600	27,026	27,312	27,166	26,676	29,756	27,045	5,426
	2	40,651	27,162	27,243	27,198	27,265	29,904	27,217	5,373
	4	40,623	27,131	27,393	26,368	26,685	29,640	26,894	5,503
	8	40,794	27,257	27,185	26,216	26,164	29,523	26,705	5,654
	16	41,094	27,275	27,579	26,956	26,845	29,950	27,164	5,578
ZDT3	1	30,596	14,347	14,009	13,917	14,015	17,377	14,072	6,611
	2	31,119	14,048	14,004	14,034	14,161	17,473	14,062	6,823
	4	30,723	14,316	13,995	13,920	14,033	17,398	14,066	6,664
	8	31,148	14,069	14,068	13,967	14,063	17,463	14,042	6,842
	16	30,680	13,714	13,698	13,671	13,712	17,095	13,699	6,792
ZDT4	1	41,473	23,531	23,271	23,278	23,353	26,981	23,358	7,246
	2	41,740	23,633	23,573	23,622	23,562	27,226	23,598	7,257
	4	41,600	23,427	23,366	23,078	22,962	26,887	23,208	7,359
	8	41,831	23,536	23,634	23,513	23,302	27,163	23,497	7,335
	16	43,040	23,692	23,966	23,994	23,604	27,659	23,814	7,692
ZDT6	1	26,296	10,572	10,628	10,569	10,648	13,743	10,604	6,277
	2	26,368	10,575	10,599	10,576	10,674	13,758	10,606	6,305
	4	26,534	10,725	10,792	10,733	10,795	13,916	10,761	6,309
	8	25,967	11,093	10,768	10,714	10,662	13,841	10,809	6,065
	16	26,027	10,693	10,614	10,561	10,529	13,685	10,599	6,171

Tabela 10 - Resultados Benchmarks Julia Sequential

Julia - Sequential								
Func.	Time (sec)					Average (sec)		Standard Deviation
	T1	T2	T3	T4	T5	T1-T5	T2-T5	T1-T5
Styrene	5685,437	5665,462	5703,218	5660,678	5662,605	5675,480	5672,991	16,454
ZDT1	42,233	26,216	26,489	26,282	26,249	29,494	26,309	6,370
ZDT2	40,237	26,974	26,792	27,377	26,571	29,590	26,929	5,330
ZDT3	30,635	14,369	13,976	13,898	13,798	17,335	14,010	6,653
ZDT4	41,148	23,885	23,248	23,188	23,410	26,976	23,432	7,090
ZDT6	26,078	10,735	10,695	10,604	10,721	13,767	10,689	6,156

Tabela 11 - Resultados Benchmarks Matlab Multiprocessing - DriverInit

Matlab - Multiprocessing - DriverInit									
Func.	Workers	Time (sec)					Average (sec)		Standard Deviation
		T1	T2	T3	T4	T5	T1-T5	T2-T5	T1-T5
Styrene	1	5929,177	5866,462	5865,731	5870,312	5868,708	5880,078	5867,804	24,603
	2	3106,260	3104,371	3103,591	3105,620	3104,098	3104,788	3104,420	0,994
	4	1710,860	1710,546	1705,508	1708,904	1705,439	1708,251	1707,599	2,364
	8	910,803	910,121	908,134	906,972	905,295	908,265	907,630	2,020
	16	512,547	506,938	503,643	505,957	505,215	506,860	505,438	3,041
ZDT1	1	22,474	21,160	21,106	21,186	21,241	21,433	21,173	0,522
	2	21,287	19,866	19,616	19,603	19,624	19,999	19,677	0,651
	4	26,691	22,464	20,684	20,757	21,071	22,333	21,244	2,272
	8	45,608	40,298	38,768	39,061	38,526	40,452	39,163	2,649
	16	52,857	45,950	43,777	41,268	41,045	44,979	43,010	4,329
ZDT2	1	23,409	21,783	21,603	21,522	21,642	21,992	21,637	0,714
	2	21,997	19,826	19,754	19,620	19,776	20,195	19,744	0,904
	4	25,827	21,643	20,838	20,959	20,532	21,960	20,993	1,967
	8	44,802	40,910	39,240	40,256	39,881	41,018	40,072	1,968
	16	53,527	46,418	44,208	41,728	40,657	45,307	43,253	4,571
ZDT3	1	8,713	7,783	7,774	7,715	7,591	7,915	7,716	0,405
	2	8,470	7,426	7,057	7,023	6,918	7,379	7,106	0,572
	4	10,179	8,824	8,093	7,581	8,107	8,557	8,151	0,903
	8	17,168	15,487	14,226	13,974	15,076	15,186	14,691	1,133
	16	21,997	17,429	17,637	16,483	17,236	18,156	17,196	1,960
ZDT4	1	34,102	33,029	33,037	32,097	32,323	32,918	32,622	0,701
	2	31,151	29,119	29,086	28,671	28,678	29,341	28,888	0,925
	4	33,452	29,911	28,824	29,537	28,740	30,093	29,253	1,736
	8	71,324	65,551	62,743	64,322	64,282	65,644	64,225	2,976
	16	55,317	44,594	42,598	39,789	39,718	44,403	41,675	5,756
ZDT6	1	14,253	13,090	12,869	13,042	13,014	13,253	13,004	0,505
	2	13,488	11,783	11,768	11,650	11,507	12,039	11,677	0,731
	4	14,472	12,018	11,056	11,820	11,113	12,096	11,502	1,247
	8	32,904	31,301	30,646	30,575	29,445	30,974	30,492	1,135
	16	17,805	13,885	12,574	12,560	11,799	13,725	12,704	2,148

Tabela 12 - Resultados Benchmarks Matlab Multiprocessing - DMSInit

Matlab - Multiprocessing - DMSInit									
Func.	Workers	Time (sec)					Average (sec)		Standard Deviation
		T1	T2	T3	T4	T5	T1-T5	T2-T5	T1-T5
Styrene	1	5876,838	5871,511	5874,720	5877,405	5871,410	5874,377	5873,762	2,544
	2	3122,635	3121,106	3121,072	3116,634	3121,775	3120,644	3120,146	2,084
	4	1724,130	1726,342	1737,880	1865,490	1728,757	1756,520	1764,617	54,685
	8	928,796	929,365	928,071	944,574	928,525	931,866	932,634	6,368
	16	549,290	534,355	533,705	532,575	534,361	536,857	533,749	6,250
ZDT1	1	36,370	36,292	36,728	37,696	37,192	36,856	36,977	0,527
	2	36,288	36,039	36,046	35,983	35,991	36,069	36,015	0,112
	4	42,374	41,889	42,500	40,741	41,049	41,711	41,545	0,703
	8	61,846	60,421	61,771	61,219	60,297	61,111	60,927	0,652
	16	92,096	76,864	76,274	76,862	77,116	79,842	76,779	6,133
ZDT2	1	37,506	37,159	37,352	37,547	41,296	38,172	38,338	1,568
	2	37,071	38,606	36,209	36,116	36,098	36,820	36,757	0,964
	4	41,524	40,292	41,024	41,205	44,936	41,796	41,864	1,621
	8	61,710	62,275	60,476	61,677	62,025	61,633	61,613	0,618
	16	91,833	76,558	75,670	76,899	76,278	79,448	76,351	6,206
ZDT3	1	22,142	22,216	22,135	22,212	22,142	22,169	22,176	0,036
	2	22,557	22,154	22,600	22,899	22,410	22,524	22,516	0,244
	4	25,962	26,162	25,647	26,269	26,294	26,067	26,093	0,240
	8	34,544	34,953	35,516	34,594	33,873	34,696	34,734	0,538
	16	59,521	45,869	45,264	45,046	45,781	48,296	45,490	5,621
ZDT4	1	47,874	47,682	48,097	48,408	47,945	48,001	48,033	0,243
	2	45,631	45,669	45,383	45,505	45,433	45,524	45,498	0,110
	4	50,346	49,681	50,589	50,127	49,766	50,102	50,041	0,343
	8	90,075	91,358	89,722	88,042	89,188	89,677	89,578	1,086
	16	92,929	79,264	78,103	77,226	77,859	81,076	78,113	5,963
ZDT6	1	28,305	28,166	27,822	28,770	27,882	28,189	28,160	0,341
	2	27,437	27,725	27,543	28,032	26,790	27,505	27,522	0,411
	4	30,126	30,130	30,235	29,502	30,664	30,131	30,133	0,372
	8	51,422	51,055	50,761	50,517	51,215	50,994	50,887	0,322
	16	56,117	41,121	40,417	40,166	40,912	43,747	40,654	6,195

Tabela 13 - Resultados Benchmarks Matlab Multithreading - DriverInit

Matlab - Multithreading - DriverInit									
Func.	Threads	Time (sec)					Average (sec)		Standard Deviation
		T1	T2	T3	T4	T5	T1-T5	T2-T5	T1-T5
Styrene	*This function cannot execute in Multithreading mode in Matlab*								
ZDT1	1	12,823	12,599	13,252	12,534	12,672	12,776	12,764	0,257
	2	13,496	12,977	13,161	12,424	12,564	12,924	12,781	0,391
	4	19,488	18,404	17,885	17,586	17,801	18,233	17,919	0,683
	8	28,586	26,362	25,880	25,714	25,413	26,391	25,842	1,140
	16	34,162	31,075	30,096	29,777	29,315	30,885	30,066	1,737
ZDT2	1	12,991	13,544	13,140	13,619	13,977	13,454	13,570	0,353
	2	14,284	13,433	13,276	12,729	12,653	13,275	13,023	0,588
	4	19,089	18,552	19,907	17,933	17,758	18,648	18,538	0,786
	8	30,412	25,960	25,124	24,854	25,244	26,319	25,296	2,079
	16	35,691	31,534	30,066	29,982	29,989	31,453	30,393	2,200
ZDT3	1	4,455	4,397	4,571	4,337	4,321	4,416	4,407	0,091
	2	4,474	4,037	3,936	4,260	4,093	4,160	4,082	0,189
	4	6,406	6,238	5,921	5,790	5,931	6,057	5,970	0,228
	8	10,838	8,946	8,957	8,932	9,011	9,337	8,961	0,751
	16	13,151	10,786	11,487	10,809	11,197	11,486	11,070	0,872
ZDT4	1	19,766	20,088	19,465	19,824	19,550	19,739	19,732	0,219
	2	20,633	19,936	19,133	20,520	19,401	19,925	19,747	0,593
	4	27,159	25,904	24,242	23,961	26,913	25,636	25,255	1,324
	8	33,714	30,769	29,721	29,455	29,369	30,606	29,829	1,632
	16	39,764	36,604	38,046	36,048	36,295	37,351	36,748	1,391
ZDT6	1	8,025	7,726	7,523	7,033	7,544	7,570	7,457	0,323
	2	7,547	7,837	7,587	7,412	7,160	7,509	7,499	0,222
	4	9,511	9,859	9,514	9,657	9,242	9,557	9,568	0,202
	8	12,199	11,276	10,138	10,681	10,646	10,988	10,685	0,705
	16	16,108	14,218	13,295	14,107	13,192	14,184	13,703	1,047

Tabela 14 - Resultados Benchmarks Matlab Multithreading - DMSInit

Matlab - Multithreading - DMSInit									
Func.	Threads	Time (sec)					Average (sec)		Standard Deviation
		T1	T2	T3	T4	T5	T1-T5	T2-T5	T1-T5
Styrene	*This function cannot execute in Multithreading mode in Matlab*								
ZDT1	1	14,719	14,152	14,214	14,171	14,521	14,355	14,264	0,226
	2	15,138	15,631	15,289	15,169	15,855	15,416	15,486	0,280
	4	24,363	23,088	22,578	22,267	21,218	22,703	22,288	1,031
	8	34,235	33,748	35,910	35,303	34,354	34,710	34,829	0,783
	16	47,594	45,214	45,045	45,694	46,487	46,007	45,610	0,938
ZDT2	1	14,365	15,169	14,226	14,745	15,111	14,723	14,813	0,381
	2	15,739	15,294	15,162	14,946	15,481	15,324	15,221	0,271
	4	21,312	22,192	21,737	21,888	22,147	21,855	21,991	0,319
	8	34,550	34,781	34,613	34,712	36,238	34,979	35,086	0,635
	16	49,253	45,583	45,628	46,880	61,229	49,714	49,830	5,909
ZDT3	1	5,639	5,350	5,419	5,583	5,403	5,479	5,438	0,112
	2	6,264	6,140	6,102	6,527	6,299	6,266	6,267	0,150
	4	10,017	10,035	9,631	9,965	9,317	9,793	9,737	0,280
	8	16,503	16,103	16,181	15,985	16,241	16,203	16,127	0,173
	16	26,490	24,569	24,767	24,385	24,697	24,982	24,604	0,765
ZDT4	1	20,335	20,045	19,698	19,240	20,142	19,892	19,781	0,386
	2	22,098	21,358	20,732	21,597	21,179	21,393	21,217	0,452
	4	29,399	29,136	28,837	29,748	29,389	29,302	29,278	0,303
	8	40,655	38,626	38,919	38,313	39,015	39,106	38,718	0,813
	16	53,904	57,409	57,134	55,018	58,334	56,360	56,974	1,639
ZDT6	1	8,930	8,501	8,700	8,524	8,827	8,696	8,638	0,167
	2	9,388	9,414	9,588	9,228	9,800	9,484	9,507	0,195
	4	12,417	12,695	13,043	13,798	12,808	12,952	13,086	0,468
	8	17,610	17,611	17,671	17,690	17,999	17,716	17,743	0,145
	16	30,482	29,171	27,795	28,401	27,610	28,692	28,244	1,048

Tabela 15 - Resultados Benchmarks Matlab Sequential

Matlab - Sequential								
Func.	Time (sec)					Average (sec)		Standard Deviation
	T1	T2	T3	T4	T5	T1-T5	T2-T5	T1-T5
Styrene	5795,006	5790,278	5789,772	5789,767	5786,726	5790,310	5789,136	2,664
ZDT1	4,104	3,191	3,219	3,670	3,686	3,574	3,442	0,339
ZDT2	3,721	3,053	3,031	3,060	3,029	3,179	3,043	0,272
ZDT3	1,362	0,847	0,822	0,832	0,827	0,938	0,832	0,212
ZDT4	3,551	2,959	2,955	2,963	2,968	3,079	2,961	0,236
ZDT6	1,166	0,653	0,631	0,631	0,640	0,744	0,639	0,211

A.2. Microbenchmarks

Tabela 16 - Resultados Microbenchmarks Python Multiprocessing (1/2)

Python - Multiprocessing (1/2)											
Cols.	Init	Workers	Chunk Size	Time (sec)					Average (sec)		Standard Deviation
				T1	T2	T3	T4	T5	T1-T5	T2-T5	T1-T5
		1	1	0,078817	0,041277	0,041760	0,050799	0,053440	0,053219	0,046819	0,013675
		1	4	0,019309	0,024474	0,016570	0,016443	0,014242	0,018208	0,017932	0,003521
		1	8	0,016251	0,020388	0,023213	0,014304	0,027440	0,020319	0,021336	0,004729
		2	1	0,002048	0,001898	0,002502	0,001566	0,001387	0,001880	0,001838	0,000389
		2	4	0,001706	0,001925	0,001020	0,001542	0,001153	0,001469	0,001410	0,000338
		2	8	0,001890	0,001347	0,001735	0,001399	0,001719	0,001618	0,001550	0,000209
		4	1	0,000806	0,000825	0,000828	0,000869	0,000874	0,000840	0,000849	0,000027
20	DriverInit	4	4	0,000846	0,000928	0,000873	0,000897	0,000915	0,000892	0,000903	0,000029
		4	8	0,000812	0,000783	0,000979	0,000892	0,000909	0,000875	0,000891	0,000070
		8	1	0,000862	0,000843	0,000809	0,000835	0,000835	0,000837	0,000831	0,000017
		8	4	0,000795	0,000790	0,000777	0,000776	0,000788	0,000785	0,000783	0,000007
		8	8	0,000792	0,000767	0,000762	0,000784	0,000762	0,000773	0,000769	0,000012
		16	1	0,001077	0,000940	0,000887	0,000869	0,000905	0,000936	0,000900	0,000074
		16	4	0,000933	0,000861	0,000857	0,000886	0,000866	0,000881	0,000868	0,000028
		16	8	0,000903	0,000864	0,000910	0,000895	0,000874	0,000889	0,000886	0,000017
		1	1	0,016586	0,014646	0,020869	0,024084	0,018643	0,018966	0,019561	0,003294
		1	4	0,020394	0,014065	0,014996	0,015642	0,017221	0,016464	0,015481	0,002218
		1	8	0,015928	0,014390	0,012974	0,010068	0,014956	0,013663	0,013097	0,002036
		2	1	0,010447	0,010557	0,010616	0,010489	0,010237	0,010469	0,010475	0,000130
		2	4	0,010888	0,010576	0,010495	0,010613	0,010637	0,010642	0,010580	0,000132
		2	8	0,010549	0,010507	0,010685	0,010590	0,010676	0,010601	0,010615	0,000070
		4	1	0,014264	0,014400	0,014729	0,014644	0,014782	0,014564	0,014639	0,000199
20	DMSInit	4	4	0,014672	0,014275	0,014621	0,014608	0,014696	0,014574	0,014550	0,000153
		4	8	0,014540	0,014472	0,014633	0,014287	0,014517	0,014490	0,014477	0,000114
		8	1	0,021806	0,020308	0,020267	0,020935	0,020946	0,020852	0,020614	0,000559
		8	4	0,021708	0,021608	0,020994	0,022312	0,021992	0,021723	0,021727	0,000439
		8	8	0,021507	0,021569	0,021767	0,021163	0,022000	0,021601	0,021625	0,000279
		16	1	0,039771	0,036986	0,037858	0,038896	0,037089	0,038120	0,037707	0,001072
		16	4	0,037419	0,036683	0,038215	0,037748	0,038032	0,037619	0,037670	0,000540
		16	8	0,038670	0,038920	0,037371	0,037812	0,038665	0,038288	0,038192	0,000592
		1	1	0,075490	0,077999	0,077361	0,086877	0,151681	0,093882	0,098480	0,029166
		1	4	0,029278	0,028240	0,024081	0,027199	0,021795	0,026119	0,025329	0,002775
		1	8	0,017980	0,013024	0,014240	0,020959	0,028720	0,018985	0,019236	0,005617
		2	1	0,002174	0,002537	0,000978	0,002524	0,001566	0,001956	0,001901	0,000603
		2	4	0,001436	0,001562	0,001863	0,001596	0,001517	0,001595	0,001635	0,000144
		2	8	0,001022	0,001768	0,002237	0,001323	0,001683	0,001607	0,001753	0,000413
		4	1	0,000870	0,000938	0,000943	0,000986	0,000977	0,000943	0,000961	0,000041
100	DriverInit	4	4	0,001015	0,000944	0,000977	0,000985	0,000969	0,000978	0,000969	0,000023
		4	8	0,000973	0,000943	0,000951	0,000971	0,000959	0,000959	0,000956	0,000011
		8	1	0,000798	0,000831	0,000993	0,000990	0,000963	0,000915	0,000944	0,000083
		8	4	0,000853	0,000838	0,000813	0,000840	0,000806	0,000830	0,000824	0,000018
		8	8	0,000843	0,000837	0,000815	0,000818	0,000848	0,000832	0,000830	0,000013
		16	1	0,000989	0,000963	0,000835	0,000808	0,000905	0,000900	0,000878	0,000070
		16	4	0,000993	0,000995	0,000947	0,000985	0,000994	0,000983	0,000980	0,000018
		16	8	0,001024	0,000979	0,000990	0,000982	0,000960	0,000987	0,000978	0,000021

Tabela 17 - Resultados Microbenchmarks Python Multiprocessing (2/2)

Python - Multiprocessing (2/2)											
Cols.	Init	Workers	Chunk Size	Time (sec)					Average (sec)		Standard Deviation T1-T5
				T1	T2	T3	T4	T5	T1-T5	T2-T5	
		1	1	0,017614	0,020155	0,017584	0,012815	0,013918	0,016417	0,016118	0,002683
		1	4	0,015157	0,012862	0,021769	0,015507	0,016084	0,016276	0,016556	0,002957
		1	8	0,012315	0,011424	0,017790	0,015025	0,013361	0,013983	0,014400	0,002249
		2	1	0,010281	0,010777	0,010323	0,010572	0,010646	0,010520	0,010580	0,000190
		2	4	0,010558	0,010436	0,010501	0,010527	0,010559	0,010516	0,010506	0,000046
		2	8	0,010583	0,010526	0,010491	0,010392	0,010436	0,010486	0,010461	0,000067
		4	1	0,014781	0,014816	0,014784	0,014548	0,014291	0,014644	0,014610	0,000201
100	DMSInit	4	4	0,014221	0,014448	0,014369	0,014622	0,014597	0,014451	0,014509	0,000148
		4	8	0,014092	0,014665	0,014670	0,014685	0,014719	0,014566	0,014685	0,000238
		8	1	0,021236	0,021200	0,020942	0,021652	0,021631	0,021332	0,021356	0,000272
		8	4	0,021318	0,022085	0,022410	0,021967	0,020917	0,021739	0,021845	0,000543
		8	8	0,021767	0,021081	0,020562	0,020699	0,022022	0,021226	0,021091	0,000577
		16	1	0,037937	0,038821	0,036988	0,038208	0,038323	0,038055	0,038085	0,000606
		16	4	0,037897	0,037369	0,037897	0,038674	0,037061	0,037780	0,037750	0,000550
		16	8	0,038234	0,038307	0,038168	0,038497	0,037112	0,038064	0,038021	0,000488
		1	1	0,163168	0,172562	0,099195	0,180398	0,120799	0,147224	0,143239	0,031628
		1	4	0,060879	0,044161	0,048800	0,041776	0,071183	0,053360	0,051480	0,011080
		1	8	0,056322	0,077037	0,045201	0,056398	0,057613	0,058514	0,059062	0,010300
		2	1	0,001527	0,001737	0,002001	0,000992	0,001000	0,001451	0,001433	0,000401
		2	4	0,001464	0,002170	0,002411	0,002167	0,001574	0,001957	0,002081	0,000370
		2	8	0,002038	0,002225	0,001629	0,001551	0,001841	0,001857	0,001812	0,000251
		4	1	0,000963	0,000971	0,000991	0,000934	0,000948	0,000961	0,000961	0,000019
500	DriverInit	4	4	0,001041	0,001005	0,001048	0,001033	0,001044	0,001034	0,001033	0,000015
		4	8	0,001011	0,000998	0,001021	0,000994	0,000991	0,001003	0,001001	0,000011
		8	1	0,000849	0,000988	0,001149	0,001135	0,001129	0,001050	0,001100	0,000116
		8	4	0,001065	0,001021	0,001003	0,001021	0,001012	0,001024	0,001014	0,000021
		8	8	0,001010	0,001020	0,001033	0,001025	0,000992	0,001016	0,001018	0,000014
		16	1	0,001016	0,001058	0,001065	0,001020	0,001049	0,001042	0,001048	0,000020
		16	4	0,000972	0,001008	0,001039	0,001027	0,001039	0,001017	0,001028	0,000025
		16	8	0,001066	0,001009	0,001021	0,001010	0,001015	0,001024	0,001014	0,000021
		1	1	0,034408	0,042415	0,033419	0,028413	0,018974	0,031526	0,030805	0,007716
		1	4	0,015090	0,019361	0,018159	0,017616	0,013360	0,016717	0,017124	0,002182
		1	8	0,017635	0,020443	0,013742	0,017039	0,015069	0,016786	0,016573	0,002297
		2	1	0,010509	0,010771	0,010605	0,010420	0,010647	0,010590	0,010611	0,000120
		2	4	0,010684	0,010554	0,010647	0,010644	0,010763	0,010658	0,010652	0,000068
		2	8	0,010499	0,010634	0,010362	0,010536	0,010586	0,010523	0,010530	0,000093
		4	1	0,014730	0,014789	0,014785	0,014771	0,014672	0,014749	0,014754	0,000044
500	DMSInit	4	4	0,014736	0,014909	0,014801	0,014676	0,014730	0,014770	0,014779	0,000080
		4	8	0,014999	0,014668	0,014926	0,014780	0,014861	0,014847	0,014809	0,000115
		8	1	0,021915	0,021990	0,022168	0,021911	0,021522	0,021901	0,021898	0,000211
		8	4	0,021686	0,020588	0,022136	0,021244	0,021835	0,021498	0,021451	0,000538
		8	8	0,021409	0,022436	0,021728	0,022185	0,021424	0,021836	0,021943	0,000411
		16	1	0,038642	0,036721	0,037768	0,038128	0,038739	0,038000	0,037839	0,000730
		16	4	0,038819	0,038318	0,039177	0,039018	0,036731	0,038413	0,038311	0,000889
		16	8	0,038667	0,038087	0,039788	0,038125	0,039172	0,038768	0,038793	0,000647

Tabela 18 - Resultados Microbenchmarks Python Multithreading

Python - Multithreading										
Cols.	Init	Workers	Time (sec)					Average (sec)		Standard Deviation T1-T5
			T1	T2	T3	T4	T5	T1-T5	T2-T5	
20	DriverInit	1	0,000341	0,000355	0,000474	0,000470	0,000469	0,000422	0,000442	0,000060
		2	0,000326	0,000324	0,000320	0,000334	0,000346	0,000330	0,000331	0,000009
		4	0,000323	0,000373	0,000497	0,000490	0,000494	0,000435	0,000464	0,000073
		8	0,000308	0,000286	0,000397	0,000462	0,000484	0,000387	0,000407	0,000079
		16	0,000321	0,000272	0,000324	0,000348	0,000386	0,000330	0,000333	0,000037
20	DMSInit	1	0,001988	0,001964	0,002015	0,002107	0,001993	0,002013	0,002020	0,000050
		2	0,002508	0,002511	0,002628	0,002384	0,002492	0,002505	0,002504	0,000077
		4	0,003182	0,003225	0,003130	0,003144	0,003175	0,003171	0,003169	0,000033
		8	0,004361	0,004344	0,004327	0,004285	0,004331	0,004330	0,004322	0,000025
		16	0,007287	0,007198	0,007146	0,007384	0,007294	0,007262	0,007256	0,000083
100	DriverInit	1	0,000348	0,000461	0,000454	0,000445	0,000470	0,000436	0,000458	0,000045
		2	0,000334	0,000340	0,000332	0,000339	0,000333	0,000336	0,000336	0,000003
		4	0,000311	0,000422	0,000479	0,000495	0,000488	0,000439	0,000471	0,000069
		8	0,000354	0,000321	0,000323	0,000329	0,000328	0,000331	0,000325	0,000012
		16	0,000380	0,000420	0,000468	0,000486	0,000521	0,000455	0,000474	0,000050
100	DMSInit	1	0,002128	0,002150	0,002155	0,002045	0,002077	0,002111	0,002107	0,000043
		2	0,002470	0,002476	0,002483	0,002436	0,002460	0,002465	0,002464	0,000016
		4	0,003061	0,002960	0,003071	0,003111	0,003092	0,003059	0,003059	0,000052
		8	0,004663	0,004694	0,004648	0,004736	0,004610	0,004670	0,004672	0,000043
		16	0,006693	0,006791	0,006835	0,006847	0,006778	0,006789	0,006813	0,000054
500	DriverInit	1	0,000349	0,000446	0,000476	0,000458	0,000478	0,000441	0,000465	0,000048
		2	0,000323	0,000490	0,000495	0,000497	0,000477	0,000456	0,000490	0,000067
		4	0,000338	0,000383	0,000490	0,000485	0,000473	0,000434	0,000458	0,000062
		8	0,000319	0,000303	0,000335	0,000341	0,000336	0,000327	0,000329	0,000014
		16	0,000315	0,000354	0,000334	0,000348	0,000383	0,000347	0,000355	0,000023
500	DMSInit	1	0,002115	0,002100	0,002109	0,002050	0,001992	0,002073	0,002063	0,000047
		2	0,002856	0,002885	0,002953	0,002892	0,002870	0,002891	0,002900	0,000033
		4	0,004108	0,003876	0,003945	0,003976	0,004078	0,003997	0,003969	0,000086
		8	0,005040	0,005007	0,004946	0,005103	0,004931	0,005005	0,004997	0,000063
		16	0,007854	0,007667	0,007688	0,008119	0,007698	0,007805	0,007793	0,000170

Tabela 19 - Resultados Microbenchmarks Python Sequential

Python - Sequential								
Cols.	Time (sec)					Average (sec)		Standard Deviation T1-T5
	T1	T2	T3	T4	T5	T1-T5	T2-T5	
20	0,000044	0,000043	0,000043	0,000044	0,000043	0,000043	0,000043	0,0000005
100	0,000122	0,000121	0,000121	0,000122	0,000121	0,000121	0,000121	0,0000005
500	0,000594	0,000592	0,000641	0,000671	0,000671	0,000634	0,000644	0,0000351

Tabela 20 - Resultados Microbenchmarks Julia Multiprocessing

Julia - Multiprocessing										
Cols.	Init	Workers	Time (sec)					Average (sec)		Standard Deviation
			T1	T2	T3	T4	T5	T1-T5	T2-T5	T1-T5
20	DriverInit	1	0,039838	0,000984	0,000739	0,000821	0,000973	0,008671	0,000879	0,015584
		2	0,040957	0,002095	0,001589	0,001957	0,001657	0,009651	0,001825	0,015654
		4	0,288936	0,292473	0,296024	0,299813	0,303499	0,296149	0,297952	0,005158
		8	0,048887	0,007784	0,007224	0,006913	0,013850	0,016932	0,008943	0,016180
		16	0,130373	0,016843	0,017671	0,014113	0,014497	0,038699	0,015781	0,045857
20	DMSInit	1	9,690856	9,799238	9,794217	9,818582	9,663182	9,753215	9,768805	0,063351
		2	11,287506	11,223625	11,328604	11,261223	11,322216	11,284635	11,283917	0,039049
		4	13,842179	13,645218	13,648302	13,815285	13,776824	13,745562	13,721407	0,083309
		8	19,261717	19,266377	19,002250	19,230696	19,171700	19,186548	19,167756	0,098141
		16	31,434815	30,221096	30,191304	29,979434	30,031290	30,371588	30,105781	0,539479
100	DriverInit	1	0,039883	0,000960	0,000750	0,000809	0,000851	0,008651	0,000843	0,015616
		2	0,041672	0,002370	0,001594	0,001606	0,001618	0,009772	0,001797	0,015953
		4	0,284693	0,288488	0,291788	0,295230	0,298642	0,291768	0,293537	0,004900
		8	0,048484	0,007934	0,009408	0,006795	0,006845	0,015893	0,007746	0,016323
		16	0,134113	0,015262	0,017443	0,014116	0,014723	0,039131	0,015386	0,047504
100	DMSInit	1	9,022640	9,052373	9,026412	9,029450	9,074382	9,041051	9,045654	0,019632
		2	10,552120	10,522076	10,582204	10,549642	10,524984	10,546205	10,544727	0,021796
		4	13,728316	13,676915	13,641777	13,611238	13,713940	13,674437	13,660968	0,043659
		8	19,364293	19,344355	19,365328	19,319352	19,238716	19,326409	19,316938	0,046929
		16	31,484411	30,124488	30,063351	30,218812	30,086050	30,395422	30,123175	0,547079
500	DriverInit	1	0,040446	0,000857	0,000950	0,001060	0,001280	0,008919	0,001037	0,015764
		2	0,044884	0,001620	0,005667	0,001736	0,001778	0,011137	0,002700	0,016943
		4	0,288280	0,291848	0,295234	0,298634	0,302136	0,295226	0,296963	0,004879
		8	0,050889	0,007180	0,007188	0,011121	0,007030	0,016682	0,008130	0,017173
		16	0,129597	0,017238	0,016226	0,014467	0,016616	0,038829	0,016137	0,045393
500	DMSInit	1	9,206839	9,143538	9,062872	8,994794	9,152147	9,112038	9,088338	0,074501
		2	10,755585	10,571312	10,750554	10,634424	10,581635	10,658702	10,634481	0,079986
		4	13,809434	13,686014	13,621043	13,951770	13,478844	13,709421	13,684418	0,161406
		8	19,312890	19,304530	19,172720	19,406336	19,055089	19,250313	19,234669	0,122745
		16	31,960474	30,009361	30,222316	29,959904	29,988106	30,428032	30,044922	0,771837

Tabela 21 - Resultados Microbenchmarks Julia Multithreading

Julia - Multithreading									
Cols.	Workers	Time (sec)					Average (sec)		Standard Deviation
		T1	T2	T3	T4	T5	T1-T5	T2-T5	T1-T5
20	1	0,046051	0,043776	0,042440	0,042741	0,042791	0,043560	0,042937	0,001324
	2	0,047221	0,043584	0,042439	0,042455	0,042881	0,043716	0,042840	0,001801
	4	0,048256	0,043414	0,043838	0,043241	0,043389	0,044428	0,043471	0,001924
	8	0,047281	0,043550	0,049240	0,043613	0,043810	0,045499	0,045053	0,002340
	16	0,065084	0,054412	0,044318	0,049593	0,046193	0,051920	0,048629	0,007423
100	1	0,046804	0,043246	0,042862	0,042812	0,042782	0,043701	0,042926	0,001560
	2	0,049161	0,043974	0,042584	0,042850	0,042916	0,044297	0,043081	0,002478
	4	0,046682	0,043654	0,042774	0,042509	0,043456	0,043815	0,043098	0,001494
	8	0,048568	0,044535	0,044055	0,044570	0,044482	0,045242	0,044411	0,001673
	16	0,059343	0,049304	0,050690	0,043607	0,043985	0,049386	0,046897	0,005717
500	1	0,046497	0,043293	0,043152	0,043114	0,042832	0,043778	0,043098	0,001368
	2	0,046626	0,043792	0,042564	0,042809	0,042721	0,043702	0,042972	0,001524
	4	0,046788	0,043492	0,042288	0,043097	0,042480	0,043629	0,042839	0,001637
	8	0,046805	0,042356	0,043452	0,042965	0,047342	0,044584	0,044029	0,002069
	16	0,054642	0,054346	0,044064	0,043249	0,043740	0,048008	0,046350	0,005303

Tabela 22 - Resultados Microbenchmarks Julia Sequential

Julia - Sequential								
Cols.	Time (sec)					Average (sec)		Standard Deviation
	T1	T2	T3	T4	T5	T1-T5	T2-T5	T1-T5
20	0,007308	0,004140	0,004670	0,005385	0,005426	0,005386	0,004905	0,001073
100	0,007851	0,004124	0,004856	0,005509	0,005987	0,005665	0,005119	0,001260
500	0,007848	0,004271	0,005049	0,005595	0,005619	0,005676	0,005134	0,001191

Tabela 23 - Resultados Microbenchmarks Matlab Multiprocessing

Matlab - Multiprocessing										
Cols.	Init	Workers	Time (sec)					Average (sec)		Standard Deviation T1-T5
			T1	T2	T3	T4	T5	T1-T5	T2-T5	
20	DriverInit	1	0,037825	0,027460	0,027791	0,027946	0,028028	0,029810	0,027806	0,004012
		2	0,039433	0,027899	0,025271	0,024548	0,026480	0,028726	0,026050	0,005473
		4	0,048241	0,034975	0,033455	0,029412	0,036470	0,036511	0,033578	0,006320
		8	0,067316	0,064538	0,067345	0,055826	0,056695	0,062344	0,061101	0,005078
		16	0,088891	0,063819	0,062791	0,062204	0,053020	0,066145	0,060459	0,012015
20	DMSInit	1	19,686060	19,574985	19,446517	19,444994	19,566835	19,543878	19,508333	0,090518
		2	19,892806	20,089352	19,951572	20,235975	20,024456	20,038832	20,075339	0,118796
		4	21,222672	21,473875	21,479148	21,383948	21,354158	21,382760	21,422782	0,093851
		8	24,094109	24,251539	24,157948	24,229590	24,398748	24,226387	24,259456	0,102501
		16	34,270283	30,632847	30,584676	30,655040	30,612875	31,351144	30,621360	1,459753
100	DriverInit	1	0,038228	0,028090	0,029241	0,028603	0,028898	0,030612	0,028708	0,003827
		2	0,042919	0,029850	0,029707	0,028660	0,028899	0,032007	0,029279	0,005475
		4	0,057662	0,041371	0,032822	0,036457	0,038199	0,041302	0,037212	0,008633
		8	0,088713	0,073163	0,067374	0,071570	0,068887	0,073941	0,070249	0,007656
		16	0,138337	0,102539	0,088016	0,089708	0,092726	0,102265	0,093247	0,018724
100	DMSInit	1	20,391019	20,153745	20,291533	20,184248	20,020270	20,208163	20,162449	0,125876
		2	20,458153	19,994757	20,268669	20,409135	20,355844	20,297312	20,257101	0,163805
		4	21,463277	21,302243	21,732013	21,473497	21,433847	21,480975	21,485400	0,139682
		8	24,135859	24,485985	24,403595	24,229540	24,366555	24,324307	24,371419	0,125463
		16	36,349137	31,186852	31,075959	30,929660	30,897215	32,087765	31,022422	2,133228
500	DriverInit	1	0,051184	0,040552	0,039926	0,039917	0,039891	0,042294	0,040072	0,004452
		2	0,048465	0,038015	0,034420	0,034633	0,033876	0,037882	0,035236	0,005488
		4	0,052829	0,043673	0,035887	0,045316	0,034722	0,042485	0,039900	0,006637
		8	0,092949	0,079462	0,071248	0,072333	0,075808	0,078360	0,074713	0,007842
		16	0,163647	0,117812	0,107405	0,112111	0,109006	0,121996	0,111584	0,021127
500	DMSInit	1	19,767656	19,700133	19,529090	19,906777	19,770144	19,734760	19,726536	0,122820
		2	20,334623	20,333822	20,079560	20,085244	19,949876	20,156625	20,112126	0,152880
		4	21,543337	21,612904	21,521132	21,370190	21,622068	21,533926	21,531574	0,090629
		8	24,297039	24,290618	24,451168	24,308976	24,339365	24,337433	24,347532	0,059283
		16	36,413207	31,262776	31,021920	30,907346	30,926685	32,106387	31,029682	2,157120

Tabela 24 - Resultados Microbenchmarks Matlab Multithreading

Matlab - Multithreading										
Cols.	Init	Workers	Time (sec)					Average (sec)		Standard Deviation T1-T5
			T1	T2	T3	T4	T5	T1-T5	T2-T5	
20	DriverInit	1	0,013059	0,011350	0,011970	0,011303	0,010889	0,011714	0,011378	0,000756
		2	0,014977	0,013124	0,015074	0,013195	0,015878	0,014450	0,014318	0,001099
		4	0,021558	0,019882	0,023355	0,020795	0,020477	0,021213	0,021127	0,001200
		8	0,031857	0,028593	0,032109	0,025628	0,026091	0,028856	0,028105	0,002747
		16	0,068195	0,035510	0,031361	0,034556	0,039941	0,041913	0,035342	0,013425
20	DMSInit	1	1,239705	1,260582	1,245171	1,289810	1,256371	1,258328	1,262984	0,017435
		2	2,210481	2,201698	2,186868	2,185601	2,191778	2,195285	2,191486	0,009476
		4	4,176202	4,100631	4,130274	4,106031	4,142782	4,131184	4,119930	0,027306
		8	8,127783	8,104699	8,146352	8,135262	8,098223	8,122464	8,121134	0,018253
		16	16,757576	16,314248	16,160986	16,308432	16,213510	16,350950	16,249294	0,211412
100	DriverInit	1	0,014333	0,012366	0,013954	0,014149	0,013430	0,013646	0,013475	0,000708
		2	0,017415	0,016598	0,014988	0,017325	0,018414	0,016948	0,016831	0,001138
		4	0,026623	0,026381	0,024013	0,024134	0,024068	0,025044	0,024649	0,001194
		8	0,056579	0,048792	0,047049	0,057278	0,054222	0,052784	0,051835	0,004135
		16	0,110537	0,078800	0,073197	0,075606	0,083594	0,084347	0,077799	0,013550
100	DMSInit	1	1,284228	1,256207	1,251617	1,271316	1,276279	1,267929	1,263855	0,012250
		2	2,301823	2,264047	2,332873	2,321023	2,326813	2,309316	2,311189	0,024914
		4	4,266814	4,220992	4,313215	4,196985	4,314936	4,262588	4,261532	0,047656
		8	8,360980	8,435237	8,273448	8,223535	8,306654	8,319971	8,309719	0,072952
		16	17,166108	16,561958	16,454616	16,625342	16,427855	16,647176	16,517443	0,269151
500	DriverInit	1	0,021127	0,022666	0,021920	0,020144	0,018933	0,020958	0,020916	0,001314
		2	0,021434	0,022136	0,020284	0,020238	0,019007	0,020620	0,020416	0,001079
		4	0,029079	0,028913	0,027509	0,026416	0,030534	0,028490	0,028343	0,001412
		8	0,061713	0,056440	0,051392	0,052835	0,051444	0,054765	0,053028	0,003930
		16	0,142663	0,097930	0,114424	0,117115	0,098791	0,114185	0,107065	0,016253
500	DMSInit	1	1,243571	1,238372	1,289883	1,293291	1,258007	1,264625	1,269888	0,022961
		2	2,272773	2,252949	2,237069	2,217096	2,222089	2,240395	2,232301	0,020469
		4	4,254865	4,171376	4,224551	4,200546	4,214691	4,213206	4,202791	0,027482
		8	8,296503	8,221601	8,230015	8,203241	8,256495	8,241571	8,227838	0,032378
		16	17,165881	16,492442	16,401504	16,454383	16,357950	16,574432	16,426570	0,299237

Tabela 25 - Resultados Microbenchmarks Matlab Sequential

Matlab - Sequential								
Cols.	Time (sec)					Average (sec)		Standard Deviation T1-T5
	T1	T2	T3	T4	T5	T1-T5	T2-T5	
20	0,000382	0,000375	0,000369	0,000333	0,000291	0,000350	0,000342	0,000034
100	0,002010	0,001198	0,001184	0,001175	0,001198	0,001353	0,001189	0,000329
500	0,007072	0,005786	0,005674	0,005729	0,005660	0,005984	0,005712	0,000546

APÊNDICE

B

Comparação dos Resultados Experimentais

Apresentação de tabelas e gráficos utilizados no estudo comparativo de performance das linguagens Python, Julia e Matlab, para Benchmarks e Microbenchmarks.

B.1. Benchmarks

B.1.1. Multiprocessing

Notação utilizada:

α-Cβ-Tγ-λ α = linguagem de programação: P(ython), J(ulia) ou M(atlab).

β = chunksize (apenas aplicável no caso de Python): 1, 4 ou 8.

γ = tempos considerados: 1 para médias T1-T5 ou 2 para médias T2-T5.

	λ = modo de inicialização (não aplicável no caso de Python): DMSInit ou DriverInit.
Exemplos:	P-C8-T2 corresponde à linguagem Python, com chunksize 8 e médias T2-T5.
	J-T1-DMSInit corresponde à linguagem Julia com médias T1-T5 e DMSInit.

Tabela 26 - Resultados Benchmarks Multiprocessing: Styrene T1-T5 e T2-T5

Multiprocessing - Styrene T1-T5 e T2-T5														
Workers	**P-C1-T1**	**P-C1-T2**	**P-C4-T1**	**P-C4-T2**	**P-C8-T1**	**P-C8-T2**	**J-T1-DMSInit**	**J-T2-DMSInit**	**J-T1-DriverInit**	**J-T2-DriverInit**	**M-T1-DMSInit**	**M-T2-DMSInit**	**M-T1-DriverInit**	**M-T2-DriverInit**
1	5724,348	5723,550	5730,594	5733,809	5751,297	5759,695	5717,539	5713,291	5690,975	5681,023	5874,377	5873,762	5880,078	5867,804
2	2997,107	2995,802	3313,687	3314,100	3341,449	3343,118	2996,312	2995,701	2983,055	2982,153	3120,644	3120,146	3104,788	3104,420
4	1629,977	1629,991	1701,122	1700,819	3336,042	3336,161	1667,393	1667,913	1644,904	1642,510	1756,520	1764,617	1708,251	1707,599
8	858,458	858,606	1703,177	1702,972	3335,464	3336,096	912,938	912,194	882,908	880,285	931,866	932,634	908,265	907,630
16	473,310	473,161	1702,150	1702,137	3343,472	3345,090	539,477	533,701	497,075	489,755	536,857	533,749	506,860	505,438

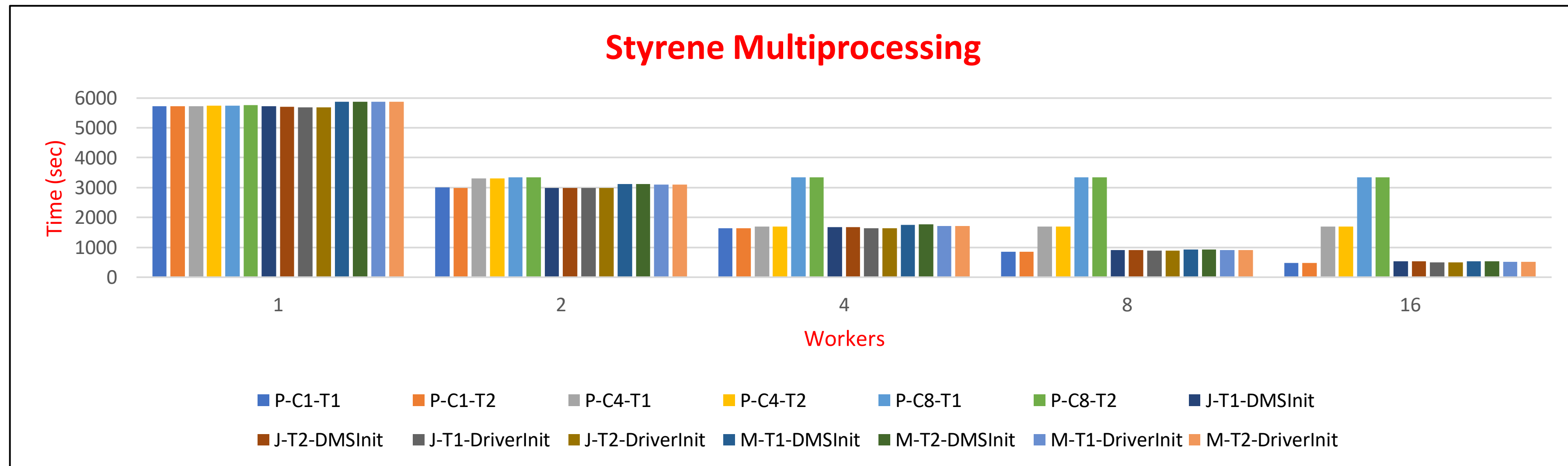


Figura 27 - Benchmarks Styrene Multiprocessing

Tabela 27 - Resultados Benchmarks Multiprocessing: ZDT1 T1-T5 e T2-T5

Multiprocessing - ZDT1 T1-T5 e T2-T5														
Workers	P-C1-T1	P-C1-T2	P-C4-T1	P-C4-T2	P-C8-T1	P-C8-T2	J-T1-DMSInit	J-T2-DMSInit	J-T1-DriverInit	J-T2-DriverInit	M-T1-DMSInit	M-T2-DMSInit	M-T1-DriverInit	M-T2-DriverInit
1	256.708	261.711	57.462	55.424	27.578	27.735	35.414	35.435	26.573	26.052	36.856	36.977	21.433	21.173
2	44.146	44.476	21.004	21.191	21.551	22.014	37.118	37.146	26.978	26.480	36.069	36.015	19.999	19.677
4	21.885	22.002	20.974	21.452	20.483	20.652	40.845	40.833	28.193	27.671	41.711	41.545	22.333	21.244
8	23.062	23.015	20.855	20.920	20.987	20.807	50.984	50.880	31.837	31.128	61.111	60.927	40.452	39.163
16	21.882	21.869	21.016	21.103	20.765	20.800	71.117	67.073	41.259	37.204	79.842	76.779	44.979	43.010

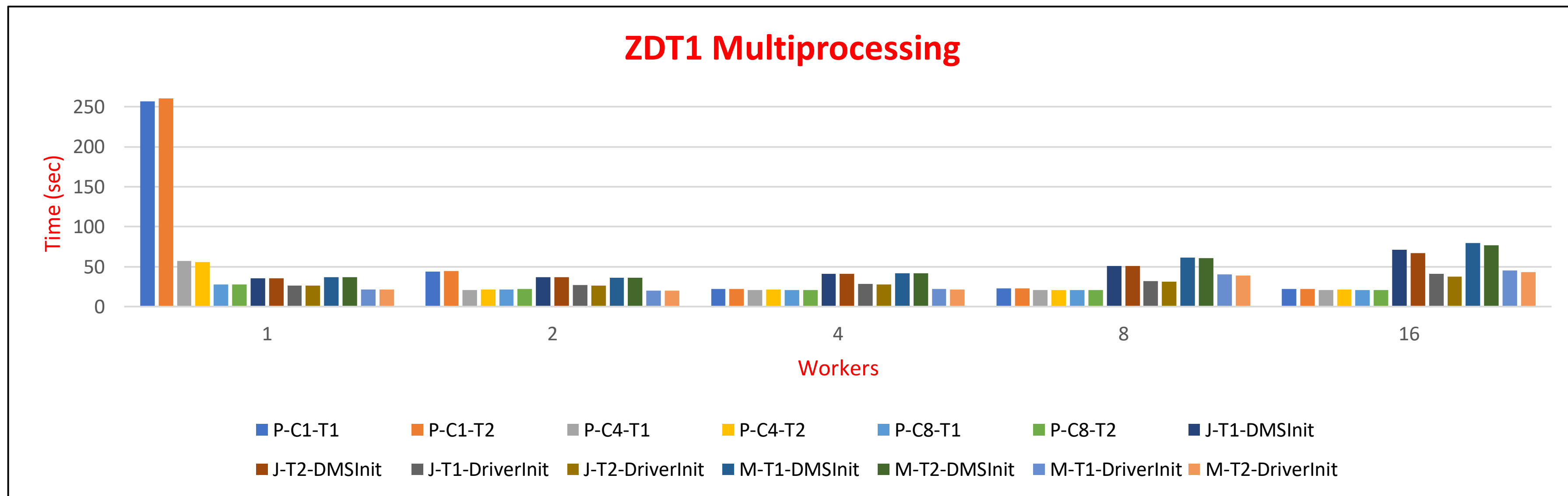


Figura 28 - Benchmarks ZDT1 Multiprocessing

Tabela 28 - Resultados Benchmarks Multiprocessing: ZDT2 T1-T5 e T2-T5

Multiprocessing - ZDT2 T1-T5 e T2-T5														
Workers	**P-C1-T1**	**P-C1-T2**	**P-C4-T1**	**P-C4-T2**	**P-C8-T1**	**P-C8-T2**	**J-T1-DMSInit**	**J-T2-DMSInit**	**J-T1-DriverInit**	**J-T2-DriverInit**	**M-T1-DMSInit**	**M-T2-DMSInit**	**M-T1-DriverInit**	**M-T2-DriverInit**
1	210,350	216,102	53,959	53,841	30,901	30,556	36,228	36,218	27,343	26,741	38,172	38,338	21,992	21,637
2	42,080	44,031	22,283	22,207	21,735	21,510	37,563	37,576	27,728	27,246	36,820	36,757	20,195	19,744
4	22,553	22,687	20,726	21,024	20,702	20,599	41,147	41,121	28,606	28,056	41,796	41,864	21,960	20,993
8	23,257	23,393	21,637	21,558	21,270	21,218	51,359	51,440	32,935	32,111	61,633	61,613	41,018	40,072
16	22,391	22,399	21,275	21,301	21,524	21,444	71,650	67,951	41,528	38,035	79,448	76,351	45,307	43,253

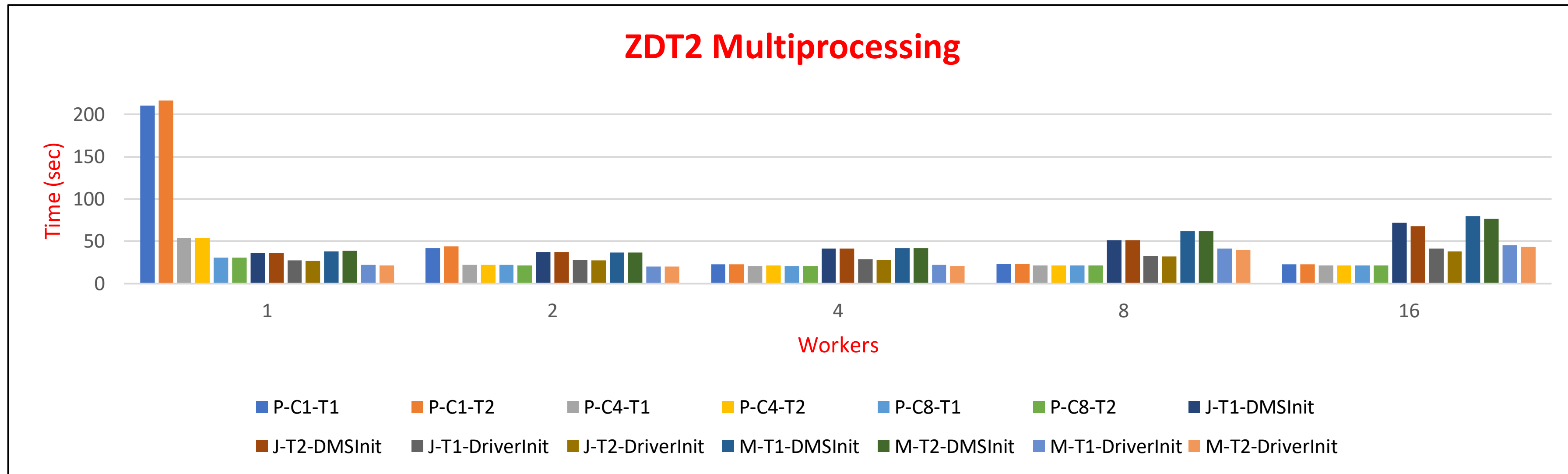


Figura 29 - Benchmarks ZDT2 Multiprocessing

Tabela 29 - Resultados Benchmarks Multiprocessing: ZDT3 T1-T5 e T2-T5

Multiprocessing - ZDT3 T1-T5 e T2-T5														
Workers	**P-C1-T1**	**P-C1-T2**	**P-C4-T1**	**P-C4-T2**	**P-C8-T1**	**P-C8-T2**	**J-T1-DMSInit**	**J-T2-DMSInit**	**J-T1-DriverInit**	**J-T2-DriverInit**	**M-T1-DMSInit**	**M-T2-DMSInit**	**M-T1-DriverInit**	**M-T2-DriverInit**
1	128,383	133,786	23,703	24,372	10,098	10,036	23,344	23,386	14,623	14,080	22,169	22,176	7,915	7,716
2	15,543	15,949	7,363	7,329	7,153	7,082	25,029	25,036	14,747	14,208	22,524	22,516	7,379	7,106
4	7,354	7,552	7,097	7,221	7,122	7,234	28,388	28,439	15,471	14,870	26,067	26,093	8,557	8,151
8	7,621	7,632	7,049	7,032	6,939	6,965	35,969	35,963	16,876	16,234	34,696	34,734	15,186	14,691
16	7,523	7,524	7,082	7,164	6,883	6,889	53,278	49,019	22,325	18,081	48,296	45,490	18,156	17,196

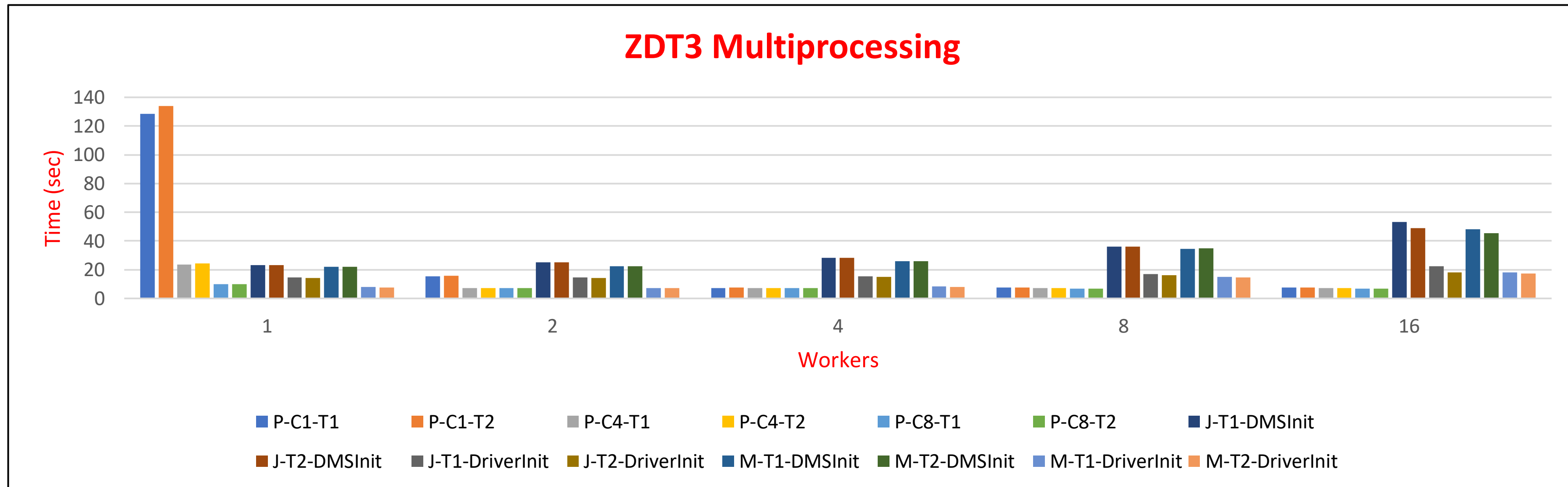


Figura 30 - Benchmarks ZDT3 Multiprocessing

Tabela 30 - Resultados Benchmarks Multiprocessing: ZDT4 T1-T5 e T2-T5

Multiprocessing - ZDT4 T1-T5 e T2-T5														
Workers	**P-C1-T1**	**P-C1-T2**	**P-C4-T1**	**P-C4-T2**	**P-C8-T1**	**P-C8-T2**	**J-T1-DMSInit**	**J-T2-DMSInit**	**J-T1-DriverInit**	**J-T2-DriverInit**	**M-T1-DMSInit**	**M-T2-DMSInit**	**M-T1-DriverInit**	**M-T2-DriverInit**
1	441,830	441,276	85,706	84,074	53,729	53,100	34,007	34,123	24,905	24,252	48,001	48,033	32,918	32,622
2	36,040	35,666	17,969	17,855	17,692	17,537	35,769	35,821	25,329	24,575	45,524	45,498	29,341	28,888
4	18,146	18,246	16,795	16,954	17,342	17,258	41,164	41,088	28,074	27,494	50,102	50,041	30,093	29,253
8	18,159	18,109	17,709	17,752	17,346	17,379	53,617	53,513	33,059	32,316	89,677	89,578	65,644	64,225
16	18,159	18,145	17,400	17,378	17,172	17,179	74,652	69,965	42,695	38,043	81,076	78,113	44,403	41,675

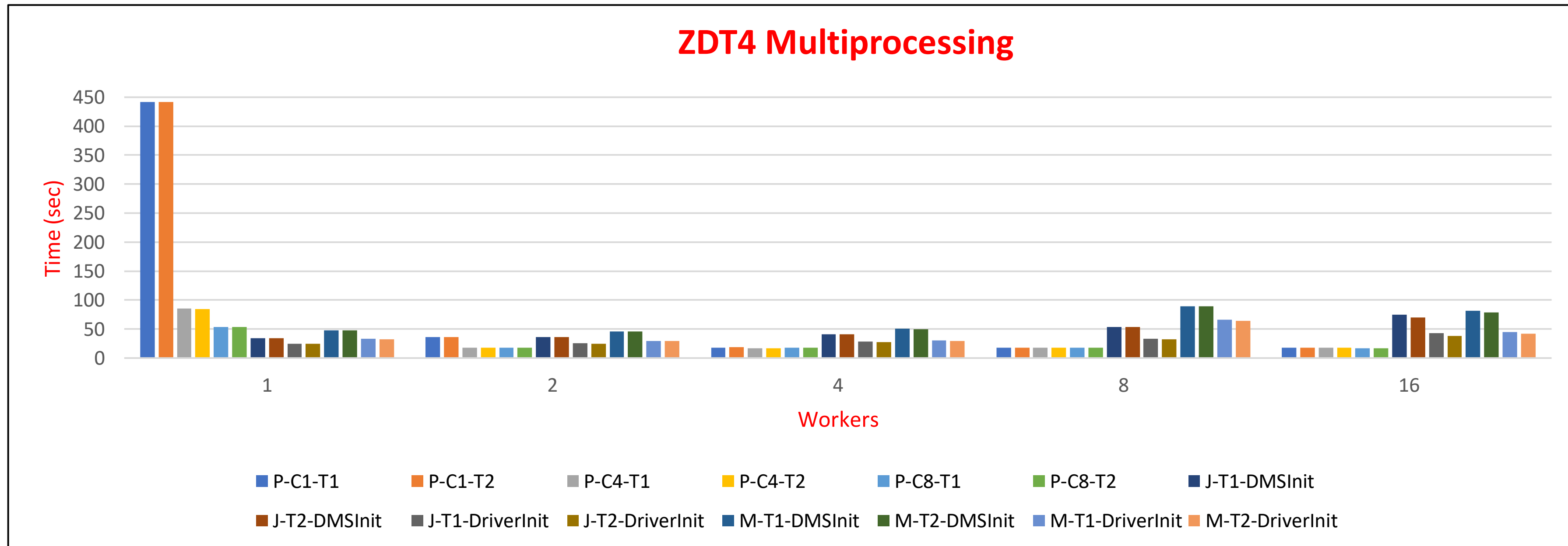


Figura 31 - Benchmarks ZDT4 Multiprocessing

Tabela 31 - Resultados Benchmarks Multiprocessing: ZDT6 T1-T5 e T2-T5

Multiprocessing - ZDT6 T1-T5 e T2-T5														
Workers	**P-C1-T1**	**P-C1-T2**	**P-C4-T1**	**P-C4-T2**	**P-C8-T1**	**P-C8-T2**	**J-T1-DMSInit**	**J-T2-DMSInit**	**J-T1-DriverInit**	**J-T2-DriverInit**	**M-T1-DMSInit**	**M-T2-DMSInit**	**M-T1-DriverInit**	**M-T2-DriverInit**
1	89,228	87,184	35,985	37,407	21,259	21,156	20,362	20,404	11,554	10,994	28,189	28,160	13,253	13,004
2	6,537	5,774	5,175	5,143	5,524	5,500	22,212	22,192	11,664	11,148	27,505	27,522	12,039	11,677
4	5,158	5,197	5,131	5,134	5,482	5,538	26,237	26,160	12,511	11,935	30,131	30,133	12,096	11,502
8	5,189	5,194	5,094	5,099	5,301	5,305	33,443	33,361	13,815	13,163	50,994	50,887	30,974	30,492
16	5,315	5,311	5,284	5,279	5,399	5,381	51,414	47,505	18,066	13,633	43,747	40,654	13,725	12,704

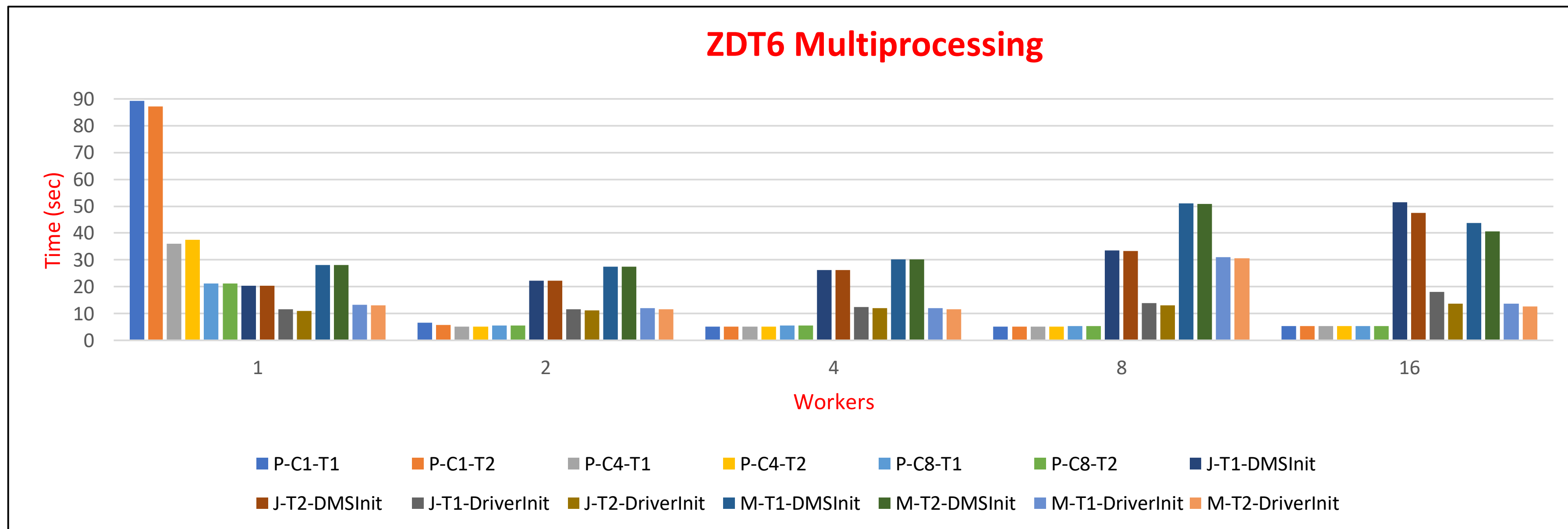


Figura 32 - Benchmarks ZDT6 Multiprocessing

B.1.2. Multithreading

Notação utilizada:

α-λ-Tγ α = linguagem de programação: P(ython), J(ulia) ou M(atlab).

λ = modo de inicialização (apenas aplicável no caso de Matlab): DMSInit ou DriverInit.

γ = tempos considerados: 1 para médias T1-T5 ou 2 para médias T2-T5.

Exemplo: M-DriverInit-T1 corresponde à linguagem Matlab com médias T1-T5 e DriverInit.

Tabela 32 - Resultados Benchmarks Multithreading: Styrene T1-T5 e T2-T5

Multithreading - Styrene T1-T5 e T2-T5				
Workers	P-T1	P-T2	J-T1	J-T2
1	5716,643	5716,345	5697,474	5690,976
2	3064,161	3066,945	5691,255	5683,452
4	1628,427	1628,728	5685,028	5679,899
8	861,454	861,197	5673,040	5668,895
16	497,884	497,812	5652,442	5648,071

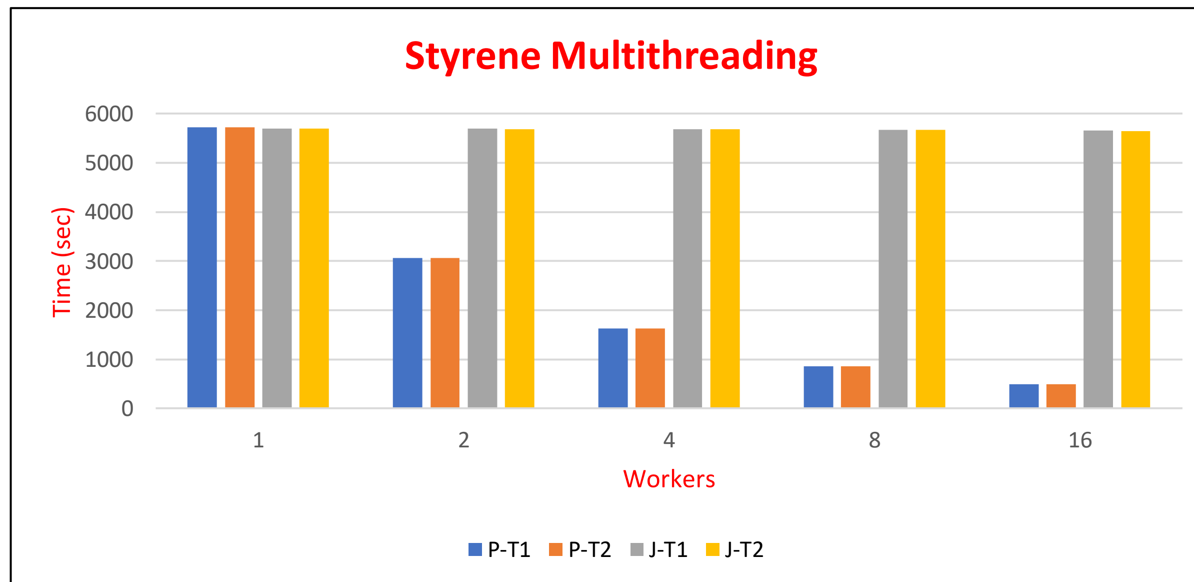


Figura 33 - Benchmarks Styrene Multithreading

Tabela 33 - Resultados Benchmarks Multithreading: ZDT1 T1-T5 e T2-T5

Multithreading - ZDT1 T1-T5 e T2-T5								
Workers	P-T1	P-T2	J-T1	J-T2	M-DMSInit-T1	M-DMSInit-T2	M-DriverInit-T1	M-DriverInit-T2
1	21,387	21,330	29,709	26,512	14,355	14,264	12,776	12,764
2	23,754	23,713	29,219	25,984	15,416	15,486	12,924	12,781
4	27,966	28,282	29,513	26,351	22,703	22,288	18,233	17,919
8	33,461	33,375	29,520	26,293	34,710	34,829	26,391	25,842
16	34,512	34,467	29,460	26,106	46,007	45,610	30,885	30,066

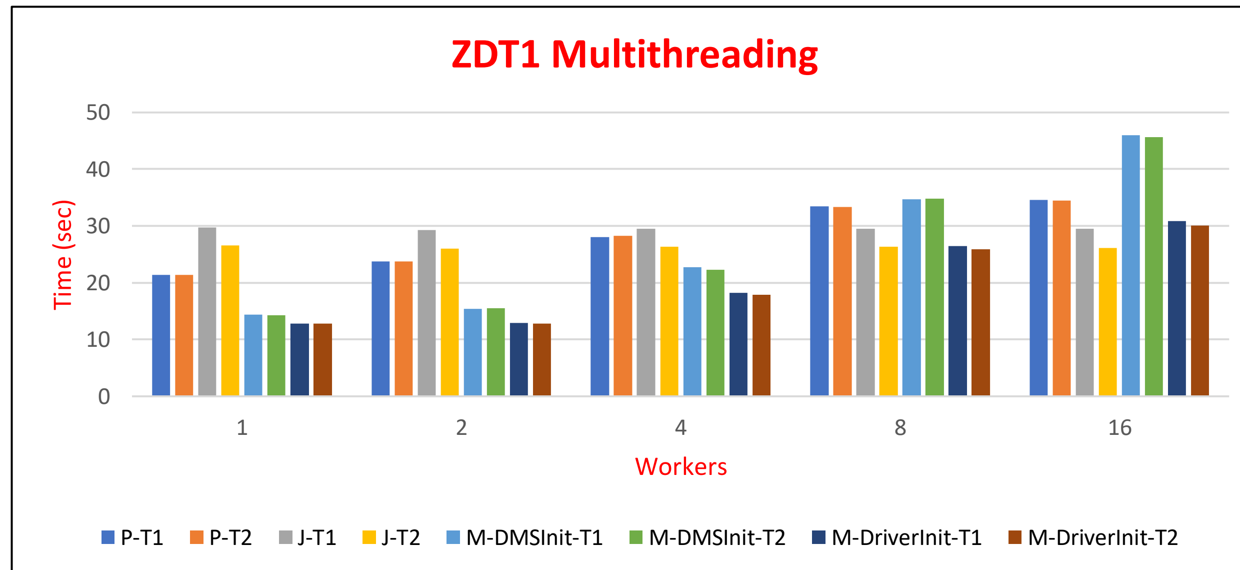


Figura 34 - Benchmarks ZDT1 Multithreading

Tabela 34 - Resultados Benchmarks Multithreading: ZDT2 T1-T5 e T2-T5

Multithreading - ZDT2 T1-T5 e T2-T5								
Workers	P-T1	P-T2	J-T1	J-T2	M-DMSInit-T1	M-DMSInit-T2	M-DriverInit-T1	M-DriverInit-T2
1	23,155	23,193	29,756	27,045	14,723	14,813	13,454	13,570
2	24,047	24,133	29,904	27,217	15,324	15,221	13,275	13,023
4	29,042	28,965	29,640	26,894	21,855	21,991	18,648	18,538
8	33,877	33,863	29,523	26,705	34,979	35,086	26,319	25,296
16	34,828	34,741	29,950	27,164	49,714	49,830	31,453	30,393

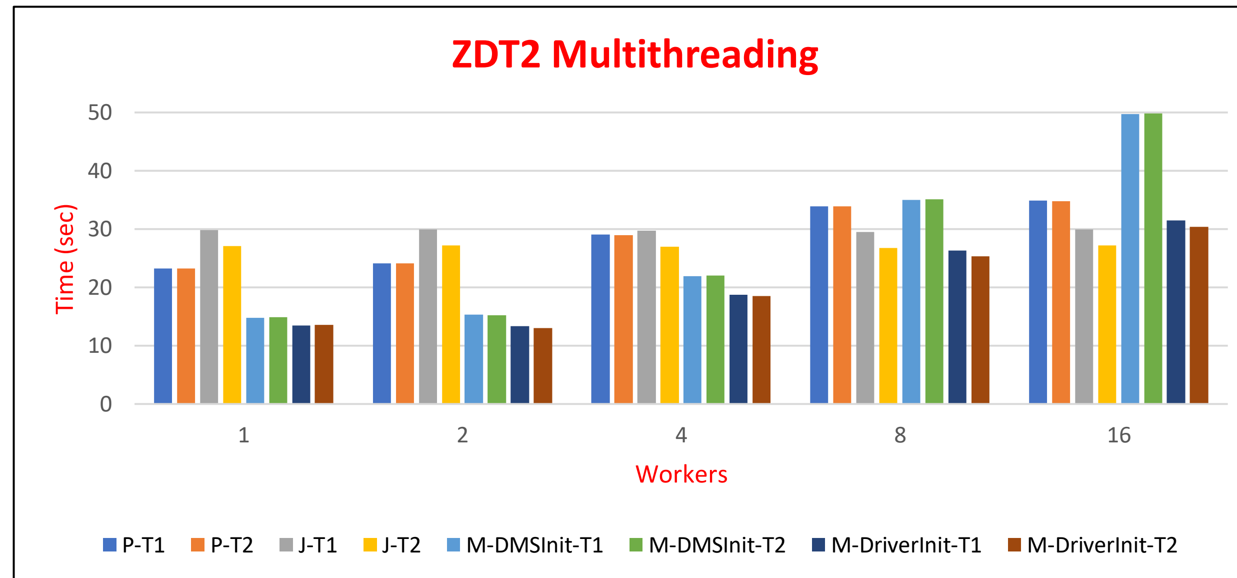


Figura 35 - Benchmarks ZDT2 Multithreading

Tabela 35 - Resultados Benchmarks Multithreading: ZDT3 T1-T5 e T2-T5

Multithreading - ZDT3 T1-T5 e T2-T5								
Workers	P-T1	P-T2	J-T1	J-T2	M-DMSInit-T1	M-DMSInit-T2	M-DriverInit-T1	M-DriverInit-T2
1	7,777	7,722	17,377	14,072	5,479	5,438	4,416	4,407
2	8,319	8,321	17,473	14,062	6,266	6,267	4,160	4,082
4	10,245	10,172	17,398	14,066	9,793	9,737	6,057	5,970
8	11,687	11,703	17,463	14,042	16,203	16,127	9,337	8,961
16	11,904	11,972	17,095	13,699	24,982	24,604	11,486	11,070

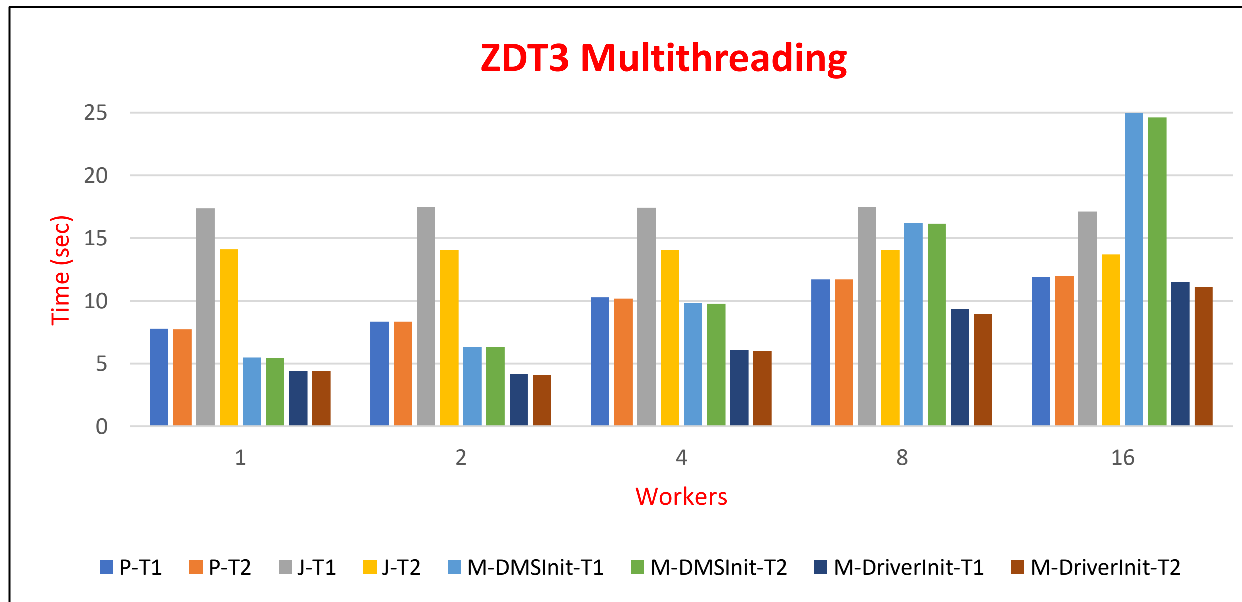


Figura 36 - Benchmarks ZDT3 Multithreading

Tabela 36 - Resultados Benchmarks Multithreading: ZDT4 T1-T5 e T2-T5

Multithreading - ZDT4 T1-T5 e T2-T5								
Workers	P-T1	P-T2	J-T1	J-T2	M-DMSInit-T1	M-DMSInit-T2	M-DriverInit-T1	M-DriverInit-T2
1	17,177	17,304	26,981	23,358	19,892	19,781	19,739	19,732
2	19,261	19,283	27,226	23,598	21,393	21,217	19,925	19,747
4	20,698	20,810	26,887	23,208	29,302	29,278	25,636	25,255
8	23,586	23,595	27,163	23,497	39,106	38,718	30,606	29,829
16	24,104	24,009	27,659	23,814	56,360	56,974	37,351	36,748

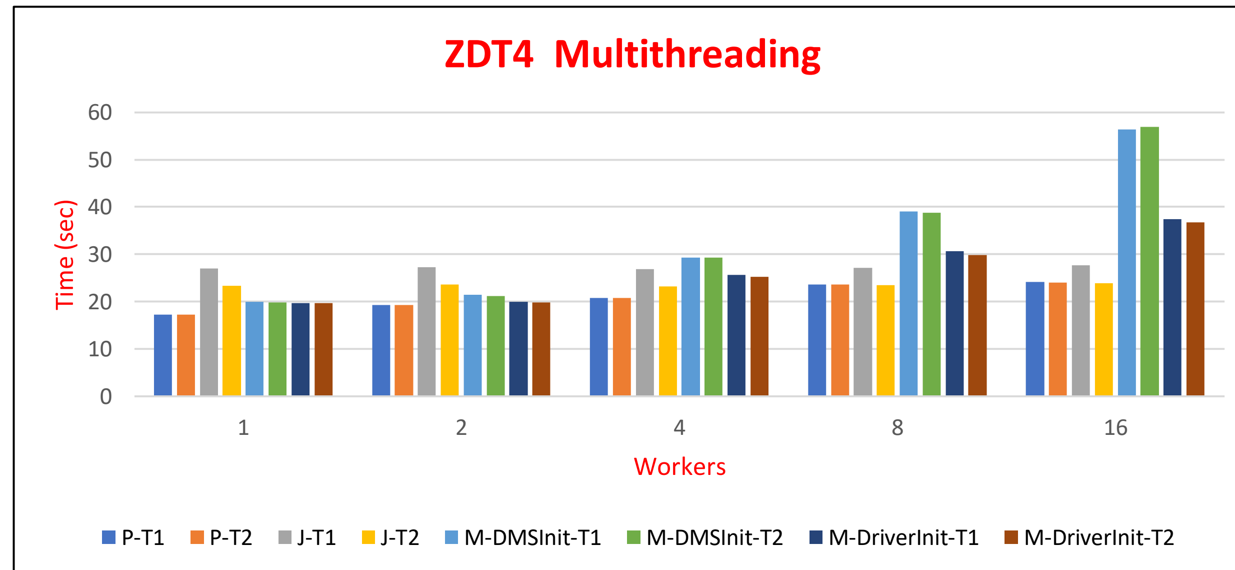


Figura 37 - Benchmarks ZDT4 Multithreading

Tabela 37 - Resultados Benchmarks Multithreading: ZDT6 T1-T5 e T2-T5

Multithreading - ZDT6 DMSInit T1-T5 e T2-T5								
Workers	P-T1	P-T2	J-T1	J-T2	M-DMSInit-T1	M-DMSInit-T2	M-DriverInit-T1	M-DriverInit-T2
1	5,682	5,693	13,743	10,604	8,696	8,638	7,570	7,457
2	5,929	5,938	13,758	10,606	9,484	9,507	7,509	7,499
4	6,189	6,208	13,916	10,761	12,952	13,086	9,557	9,568
8	6,717	6,757	13,841	10,809	17,716	17,743	10,988	10,685
16	6,914	6,978	13,685	10,599	28,692	28,244	14,184	13,703

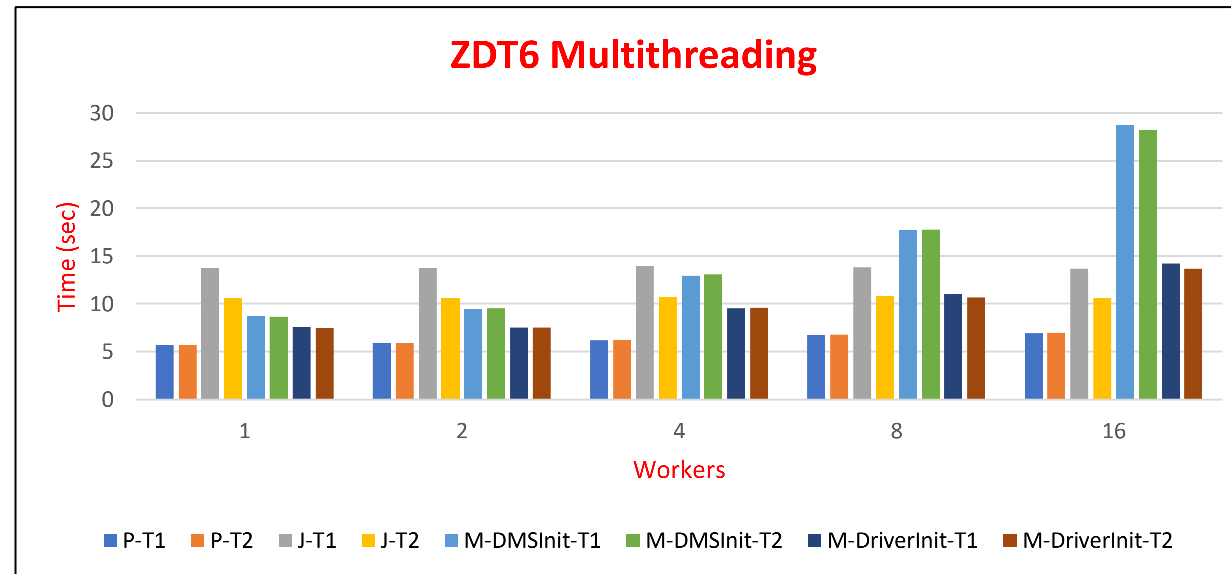


Figura 38 - Benchmarks ZDT6 Multithreading

B.1.3. Sequential

Notação utilizada:

α-Tγ α = linguagem de programação: P(ython), J(ulia) ou M(atlab).

γ = tempos considerados: 1 para médias T1-T5 ou 2 para médias T2-T5.

Exemplo : P-T2 corresponde à linguagem Python com médias T2-T5.

Tabela 38 - Resultados Benchmarks Sequential: Styrene

Sequential - Styrene					
P-T1	P-T2	J-T1	J-T2	M-T1	M-T2
5713,965	5714,040	5675,480	5672,991	5790,310	5789,136

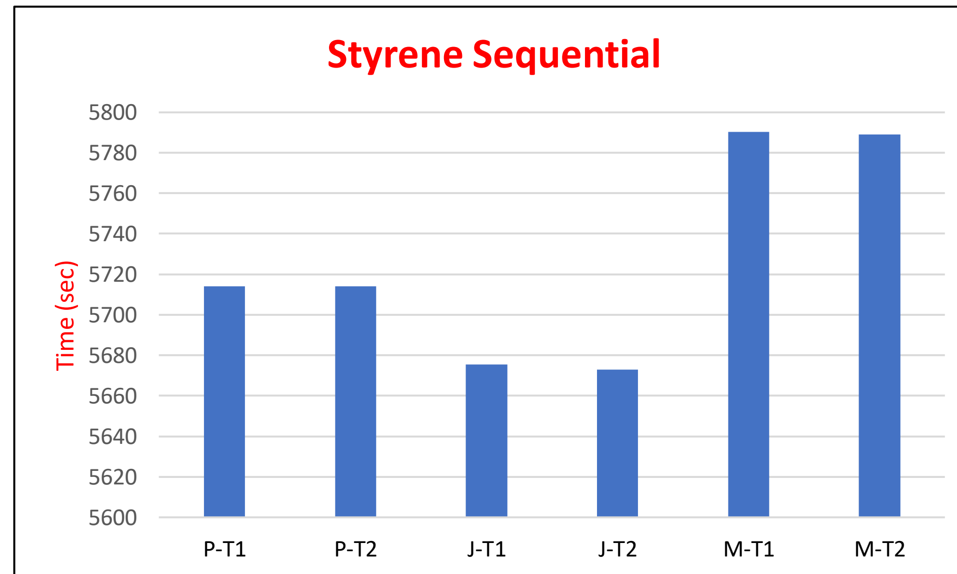


Figura 39 - Benchmarks Styrene Sequential

Tabela 39 - Resultados Benchmarks Sequential: ZDT1

Sequential - ZDT1					
P-T1	**P-T2**	**J-T1**	**J-T2**	**M-T1**	**M-T2**
18,597	18,398	29,494	26,309	3,574	3,442

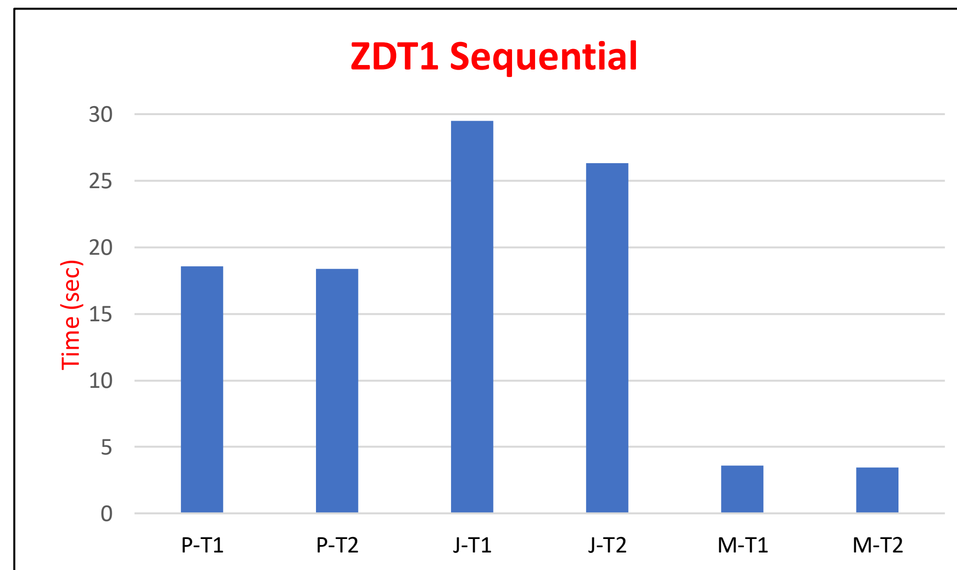


Figura 40 - Benchmarks ZDT1 Sequential

Tabela 40 - Resultados Benchmarks Sequential: ZDT2

Sequential - ZDT2					
P-T1	**P-T2**	**J-T1**	**J-T2**	**M-T1**	**M-T2**
19,098	18,906	29,590	26,929	3,179	3,043

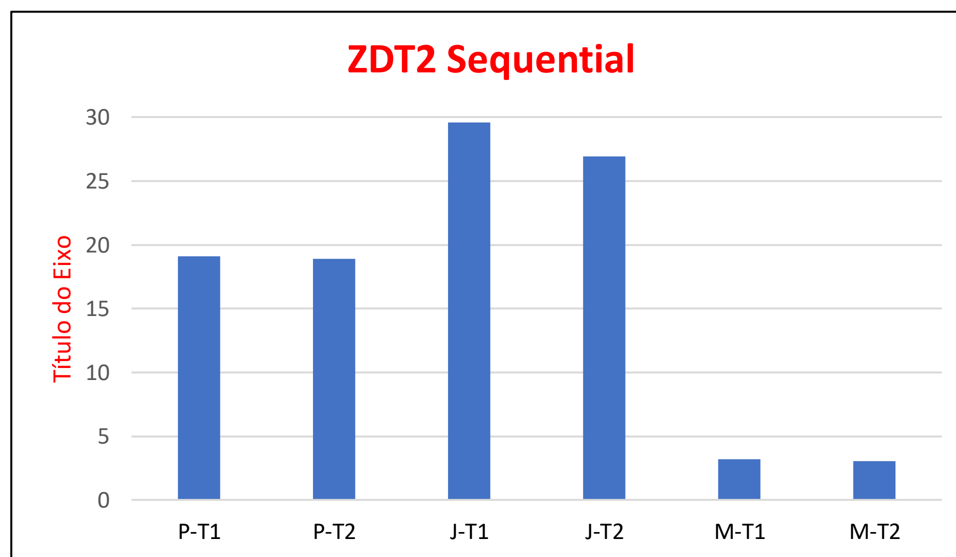


Figura 41 - Benchmarks ZDT2 Sequential

Tabela 41 - Resultados Benchmarks Sequential: ZDT3

Sequential - ZDT3					
P-T1	P-T2	J-T1	J-T2	M-T1	M-T2
6,508	6,464	17,335	14,010	0,938	0,832

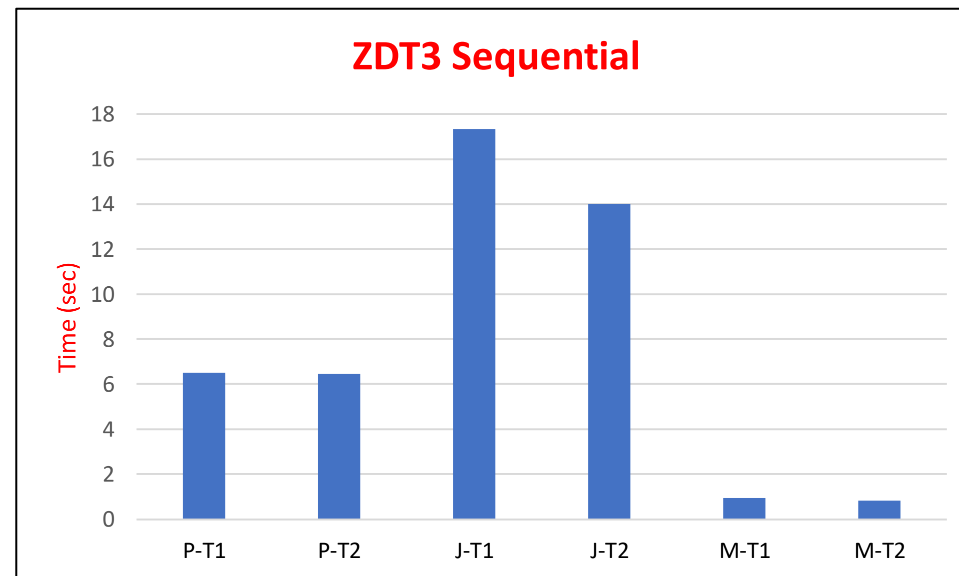


Figura 42 - Benchmarks ZDT3 Sequential

Tabela 42 - Resultados Benchmarks Sequential: ZDT4

Sequential - ZDT4					
P-T1	P-T2	J-T1	J-T2	M-T1	M-T2
15,062	14,995	26,976	23,432	3,079	2,961

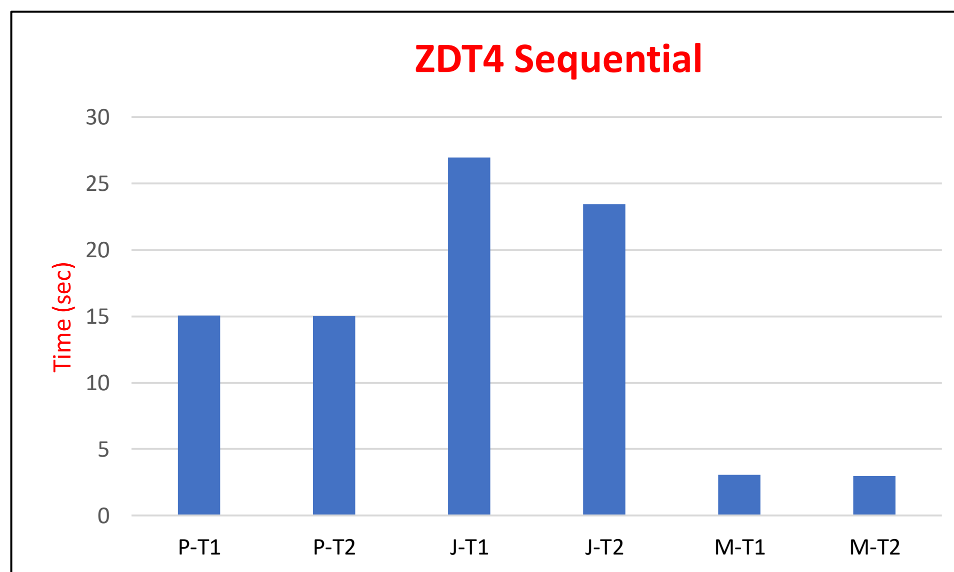


Figura 43 - Benchmarks ZDT4 Sequential

Tabela 43 - Resultados Benchmarks Sequential: ZDT6

Sequential - ZDT6					
P-T1	P-T2	J-T1	J-T2	M-T1	M-T2
4,130	4,117	13,767	10,689	0,744	0,639

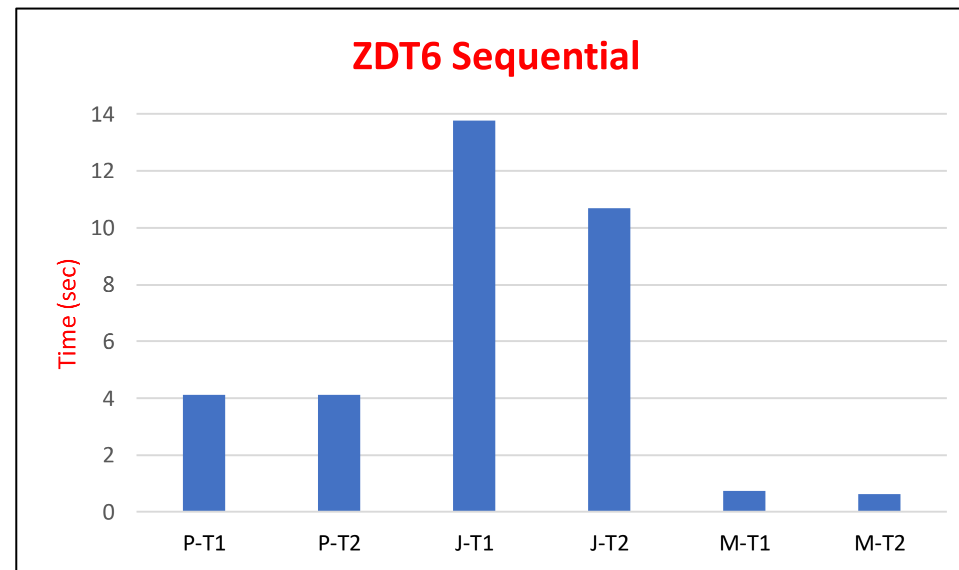


Figura 44 - Benchmarks ZDT6 Sequential

B.1.4. Todos os Modos

Notação utilizada:

α-φ-Cβ α = linguagem de programação: P(ython), J(ulia) ou M(atlab).

φ = modo: MP para Multiprocessing, MT para Multiththreading e S para Sequential.

β = chunksize (apenas aplicável no caso de Python com Multiprocessing): 1, 4 ou 8.

Exemplos: P-MP-C8 corresponde à linguagem Python com Multiprocessing e chunksize 8.

J-MT corresponde à linguagem Julia com Multithreading.

Tabela 44 - Resultados Benchmarks Styrene DMSInit T1-T5

Styrene DMSInit T1-T5										
Workers	**P-MP-C1**	**P-MP-C4**	**P-MP-C8**	**J-MP**	**M-MP**	**P-MT**	**J-MT**	**P-S**	**J-S**	**M-S**
1	5724,348	5730,594	5751,297	5717,539	5874,377	5716,643	5697,474	5713,965	5675,480	5790,310
2	2997,107	3313,687	3341,449	2996,312	3120,644	3064,161	5691,255	5713,965	5675,480	5790,310
4	1629,977	1701,122	3336,042	1667,393	1756,520	1628,427	5685,028	5713,965	5675,480	5790,310
8	858,458	1703,177	3335,464	912,938	931,866	861,454	5673,040	5713,965	5675,480	5790,310
16	473,310	1702,150	3343,472	539,477	536,857	497,884	5652,442	5713,965	5675,480	5790,310

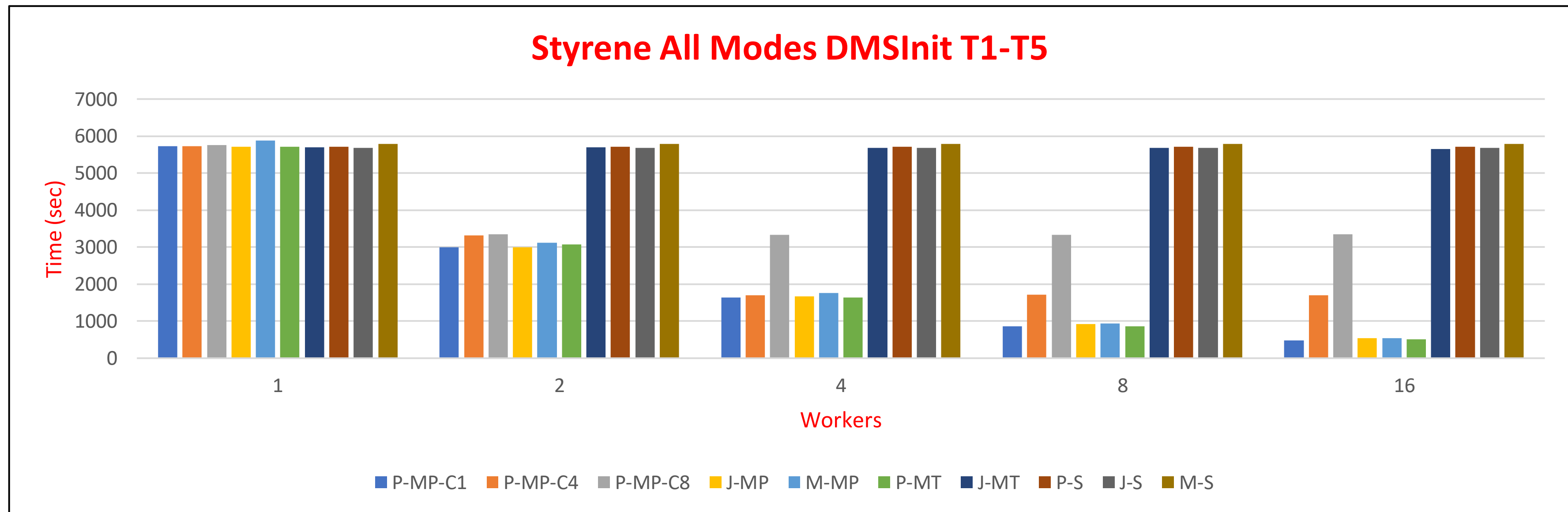


Figura 45 - Benchmarks Styrene DMSInit T1-T5

Tabela 45 - Resultados Benchmarks Styrene DMSInit T2-T5

Styrene DMSInit T2-T5										
Workers	P-MP-C1	P-MP-C4	P-MP-C8	J-MP	M-MP	P-MT	J-MT	P-S	J-S	M-S
1	5723,550	5733,809	5759,695	5713,291	5873,762	5716,345	5690,976	5714,040	5672,991	5789,136
2	2995,802	3314,100	3343,118	2995,701	3120,146	3066,945	5683,452	5714,040	5672,991	5789,136
4	1629,991	1700,819	3336,161	1667,913	1764,617	1628,728	5679,899	5714,040	5672,991	5789,136
8	858,606	1702,972	3336,096	912,194	932,634	861,197	5668,895	5714,040	5672,991	5789,136
16	473,161	1702,137	3345,090	533,701	533,749	497,812	5648,071	5714,040	5672,991	5789,136

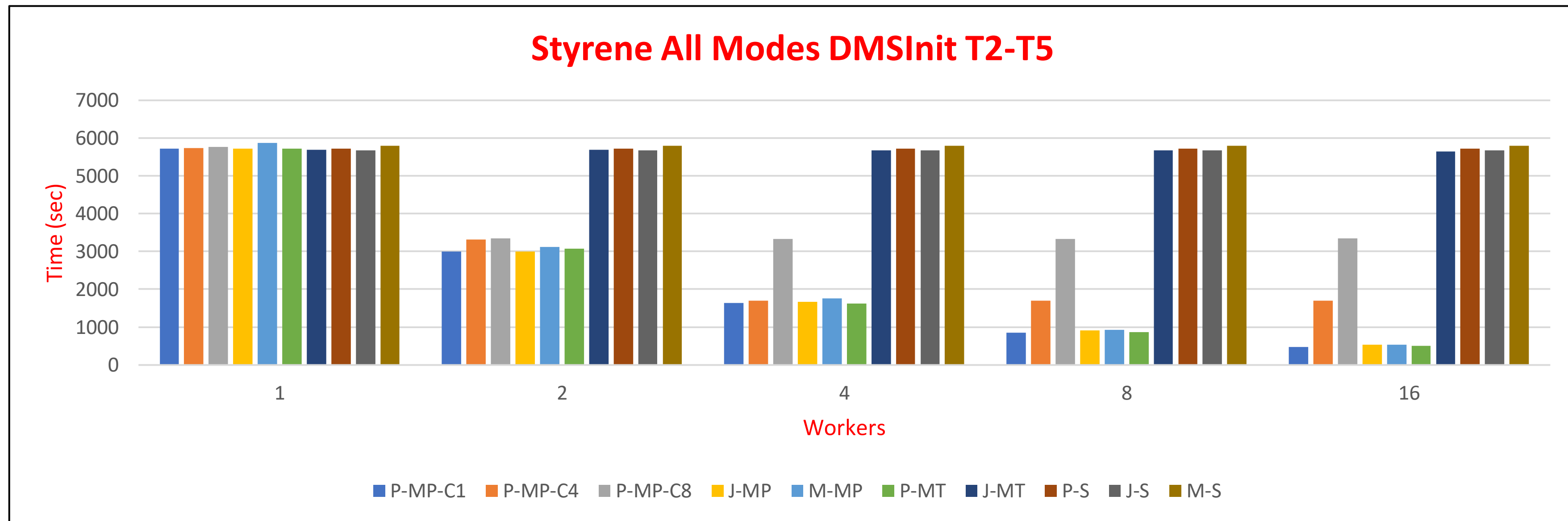


Figura 46 - Benchmarks Styrene DMSInit T2-T5

Tabela 46 - Resultados Benchmarks Styrene DriverInit T1-T5

Styrene DriverInit T1-T5										
Workers	**P-MP-C1**	**P-MP-C4**	**P-MP-C8**	**J-MP**	**M-MP**	**P-MT**	**J-MT**	**P-S**	**J-S**	**M-S**
1	5724,348	5730,594	5751,297	5690,975	5880,078	5716,643	5697,474	5713,965	5675,480	5790,310
2	2997,107	3313,687	3341,449	2983,055	3104,788	3064,161	5691,255	5713,965	5675,480	5790,310
4	1629,977	1701,122	3336,042	1644,904	1708,251	1628,427	5685,028	5713,965	5675,480	5790,310
8	858,458	1703,177	3335,464	882,908	908,265	861,454	5673,040	5713,965	5675,480	5790,310
16	473,310	1702,150	3343,472	497,075	506,860	497,884	5652,442	5713,965	5675,480	5790,310

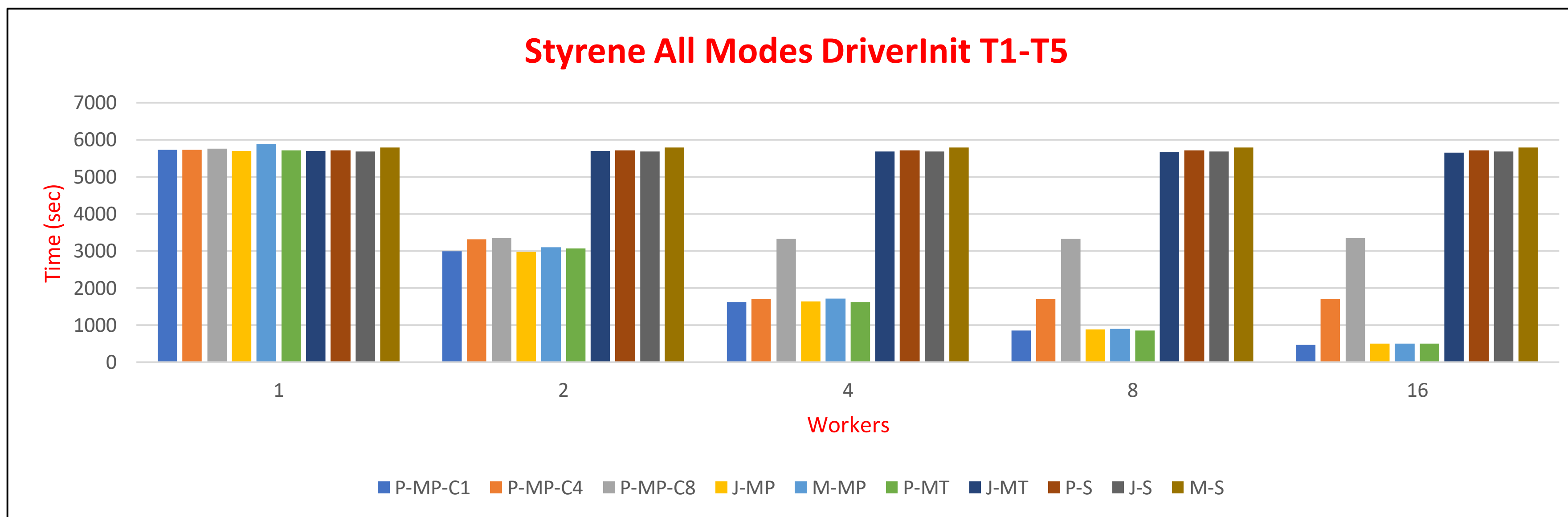


Figura 47 - Benchmarks Styrene DriverInit T1-T5

Tabela 47 - Resultados Benchmarks Styrene DriverInit T2-T5

Styrene DriverInit T2-T5										
Workers	**P-MP-C1**	**P-MP-C4**	**P-MP-C8**	**J-MP**	**M-MP**	**P-MT**	**J-MT**	**P-S**	**J-S**	**M-S**
1	5723,550	5733,809	5759,695	5681,023	5867,804	5716,345	5690,976	5714,040	5672,991	5789,136
2	2995,802	3314,100	3343,118	2982,153	3104,420	3066,945	5683,452	5714,040	5672,991	5789,136
4	1629,991	1700,819	3336,161	1642,510	1707,599	1628,728	5679,899	5714,040	5672,991	5789,136
8	858,606	1702,972	3336,096	880,285	907,630	861,197	5668,895	5714,040	5672,991	5789,136
16	473,161	1702,137	3345,090	489,755	505,438	497,812	5648,071	5714,040	5672,991	5789,136

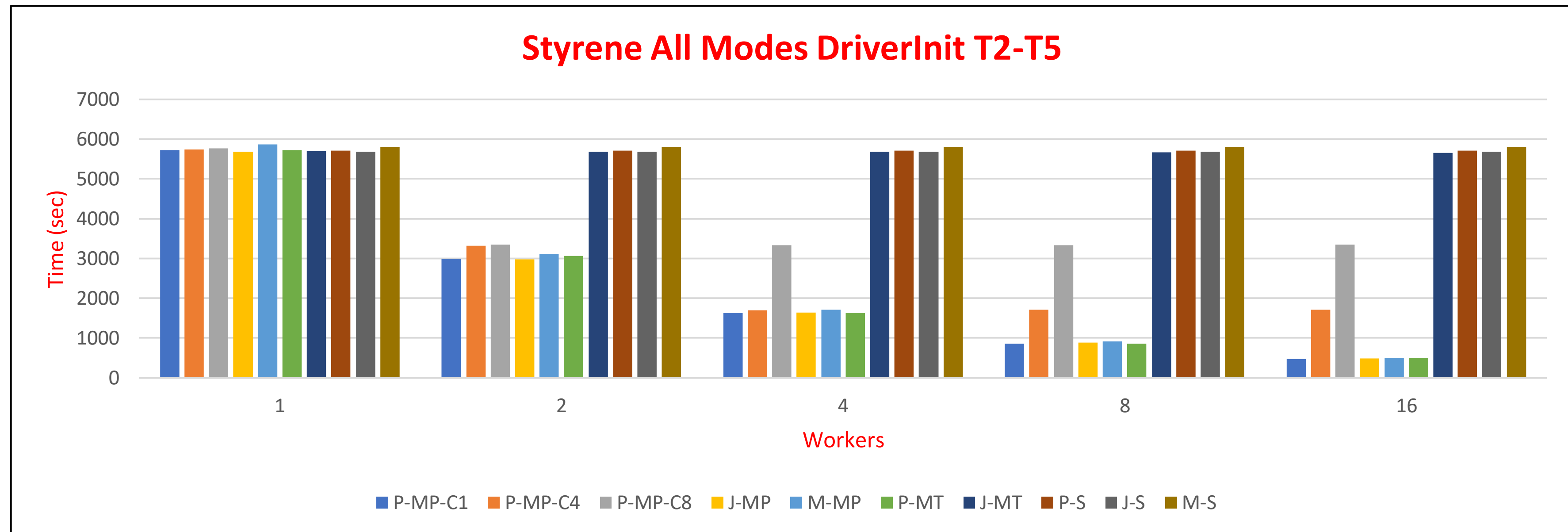


Figura 48 - Benchmarks Styrene DriverInit T2-T5

Tabela 48 - Resultados Benchmarks ZDT1 DMSInit T1-T5

ZDT1 DMSInit T1-T5											
Workers	**P-MP-C1**	**P-MP-C4**	**P-MP-C8**	**J-MP**	**M-MP**	**P-MT**	**J-MT**	**M-MT**	**P-S**	**J-S**	**M-S**
1	256,708	57,462	27,578	35,414	36,856	21,387	29,709	14,355	18,597	29,494	3,574
2	44,146	21,004	21,551	37,118	36,069	23,754	29,219	15,416	18,597	29,494	3,574
4	21,885	20,974	20,483	40,845	41,711	27,966	29,513	22,703	18,597	29,494	3,574
8	23,062	20,855	20,987	50,984	61,111	33,461	29,520	34,710	18,597	29,494	3,574
16	21,882	21,016	20,765	71,117	79,842	34,512	29,460	46,007	18,597	29,494	3,574

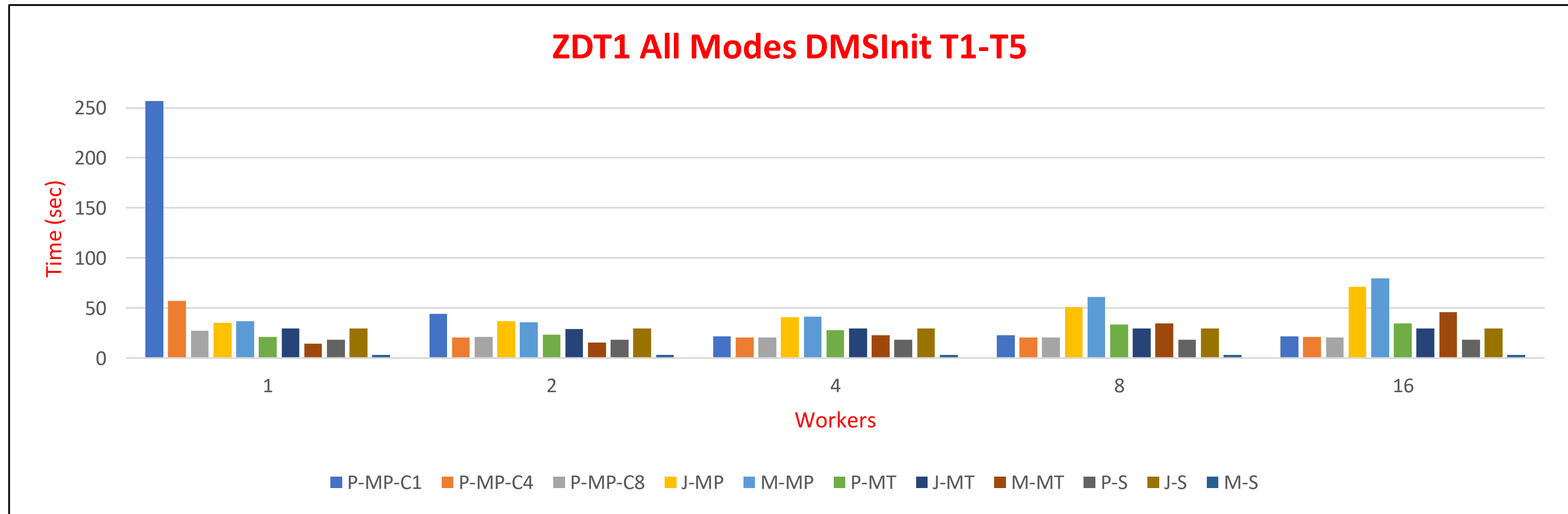


Figura 49 - Benchmarks ZDT1 DMSInit T1-T5

Tabela 49 - Resultados Benchmarks ZDT1 DMSInit T2-T5

ZDT1 DMSInit T2-T5											
Workers	**P-MP-C1**	**P-MP-C4**	**P-MP-C8**	**J-MP**	**M-MP**	**P-MT**	**J-MT**	**M-MT**	**P-S**	**J-S**	**M-S**
1	261,711	55,424	27,735	35,435	36,977	21,330	26,512	14,264	18,398	26,309	3,442
2	44,476	21,191	22,014	37,146	36,015	23,713	25,984	15,486	18,398	26,309	3,442
4	22,002	21,452	20,652	40,833	41,545	28,282	26,351	22,288	18,398	26,309	3,442
8	23,015	20,920	20,807	50,880	60,927	33,375	26,293	34,829	18,398	26,309	3,442
16	21,869	21,103	20,800	67,073	76,779	34,467	26,106	45,610	18,398	26,309	3,442

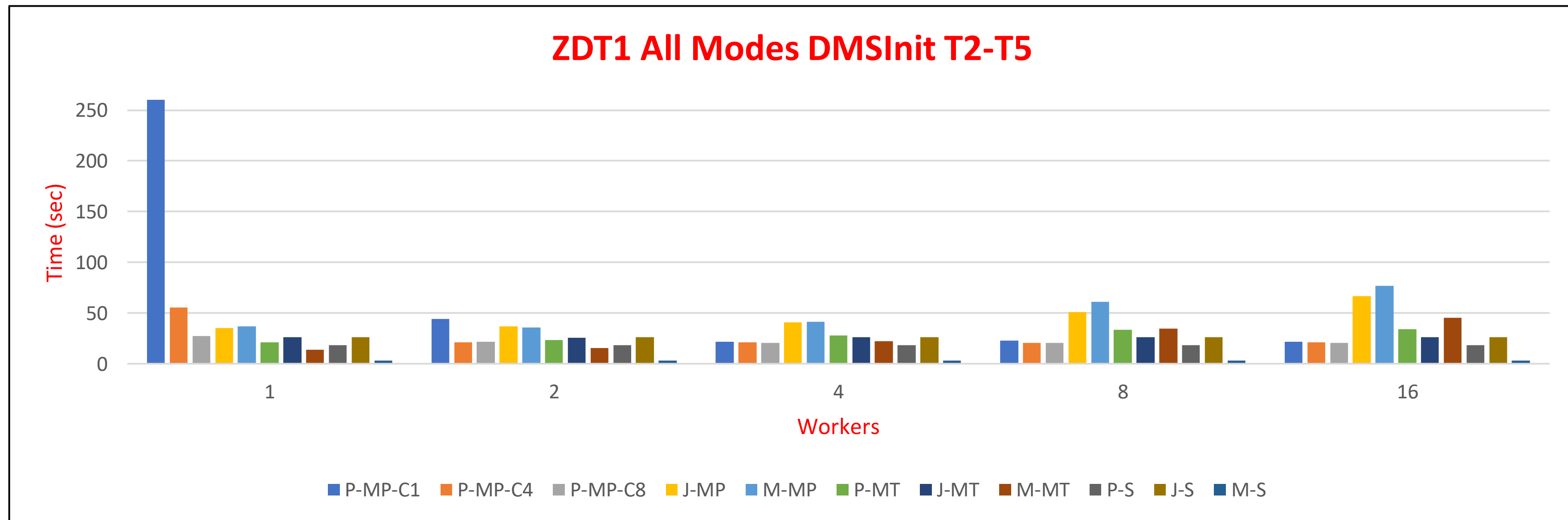


Figura 50 - Benchmarks ZDT1 DMSInit T2-T5

Tabela 50 - Resultados Benchmarks ZDT1 DriverInit T1-T5

ZDT1 DriverInit T1-T5											
Workers	**P-MP-C1**	**P-MP-C4**	**P-MP-C8**	**J-MP**	**M-MP**	**P-MT**	**J-MT**	**M-MT**	**P-S**	**J-S**	**M-S**
1	256,708	57,462	27,578	26,573	21,433	21,387	29,709	12,776	18,597	29,494	3,574
2	44,146	21,004	21,551	26,978	19,999	23,754	29,219	12,924	18,597	29,494	3,574
4	21,885	20,974	20,483	28,193	22,333	27,966	29,513	18,233	18,597	29,494	3,574
8	23,062	20,855	20,987	31,837	40,452	33,461	29,520	26,391	18,597	29,494	3,574
16	21,882	21,016	20,765	41,259	44,979	34,512	29,460	30,885	18,597	29,494	3,574

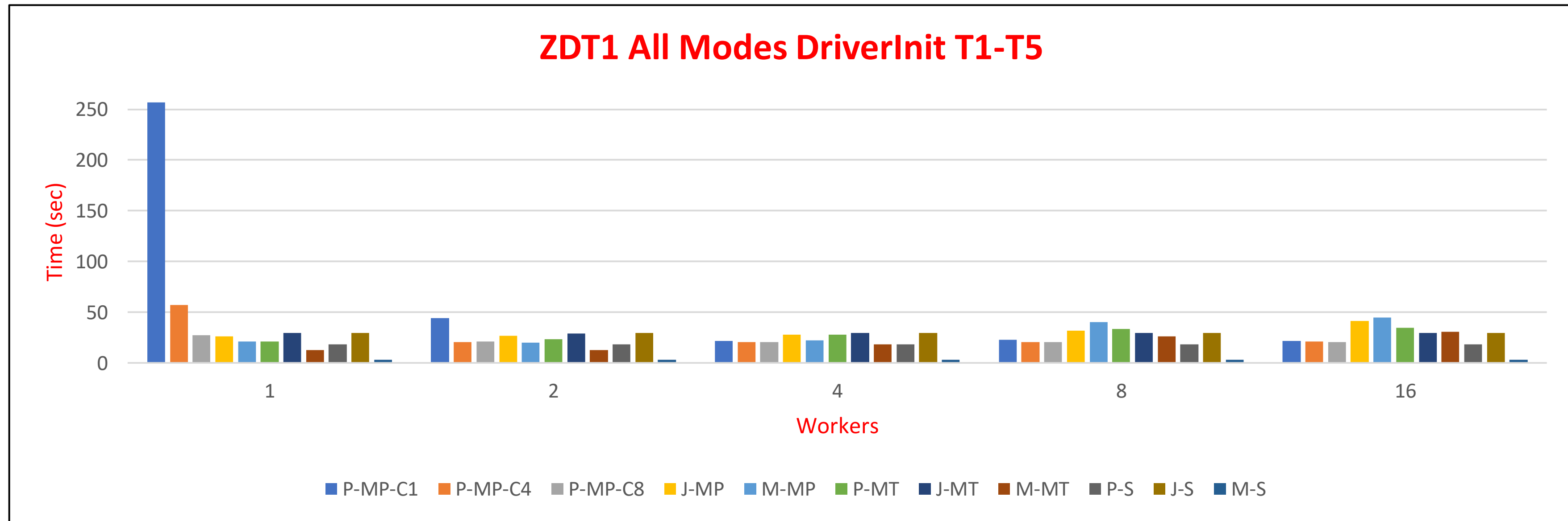


Figura 51 - Benchmarks ZDT1 DriverInit T1-T5

Tabela 51 - Resultados Benchmarks ZDT1 DriverInit T2-T5

ZDT1 DriverInit T2-T5											
Workers	**P-MP-C1**	**P-MP-C4**	**P-MP-C8**	**J-MP**	**M-MP**	**P-MT**	**J-MT**	**M-MT**	**P-S**	**J-S**	**M-S**
1	261,711	55,424	27,735	26,052	21,173	21,330	26,512	12,764	18,398	26,309	3,442
2	44,476	21,191	22,014	26,480	19,677	23,713	25,984	12,781	18,398	26,309	3,442
4	22,002	21,452	20,652	27,671	21,244	28,282	26,351	17,919	18,398	26,309	3,442
8	23,015	20,920	20,807	31,128	39,163	33,375	26,293	25,842	18,398	26,309	3,442
16	21,869	21,103	20,800	37,204	43,010	34,467	26,106	30,066	18,398	26,309	3,442

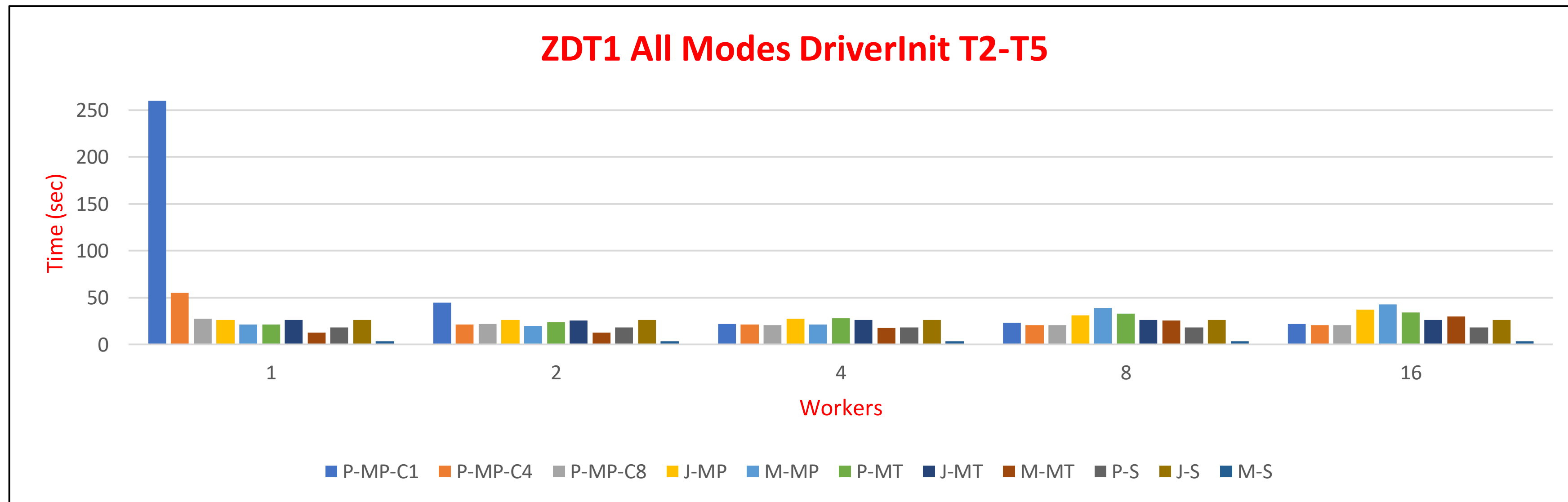


Figura 52 - Benchmarks ZDT1 DriverInit T2-T5

Tabela 52 - Resultados Benchmarks ZDT2 DMSInit T1-T5

ZDT2 DMSInit T1-T5											
Workers	**P-MP-C1**	**P-MP-C4**	**P-MP-C8**	**J-MP**	**M-MP**	**P-MT**	**J-MT**	**M-MT**	**P-S**	**J-S**	**M-S**
1	210,350	53,959	30,901	36,228	38,172	23,155	29,756	14,723	19,098	29,590	3,179
2	42,080	22,283	21,735	37,563	36,820	24,047	29,904	15,324	19,098	29,590	3,179
4	22,553	20,726	20,702	41,147	41,796	29,042	29,640	21,855	19,098	29,590	3,179
8	23,257	21,637	21,270	51,359	61,633	33,877	29,523	34,979	19,098	29,590	3,179
16	22,391	21,275	21,524	71,650	79,448	34,828	29,950	49,714	19,098	29,590	3,179

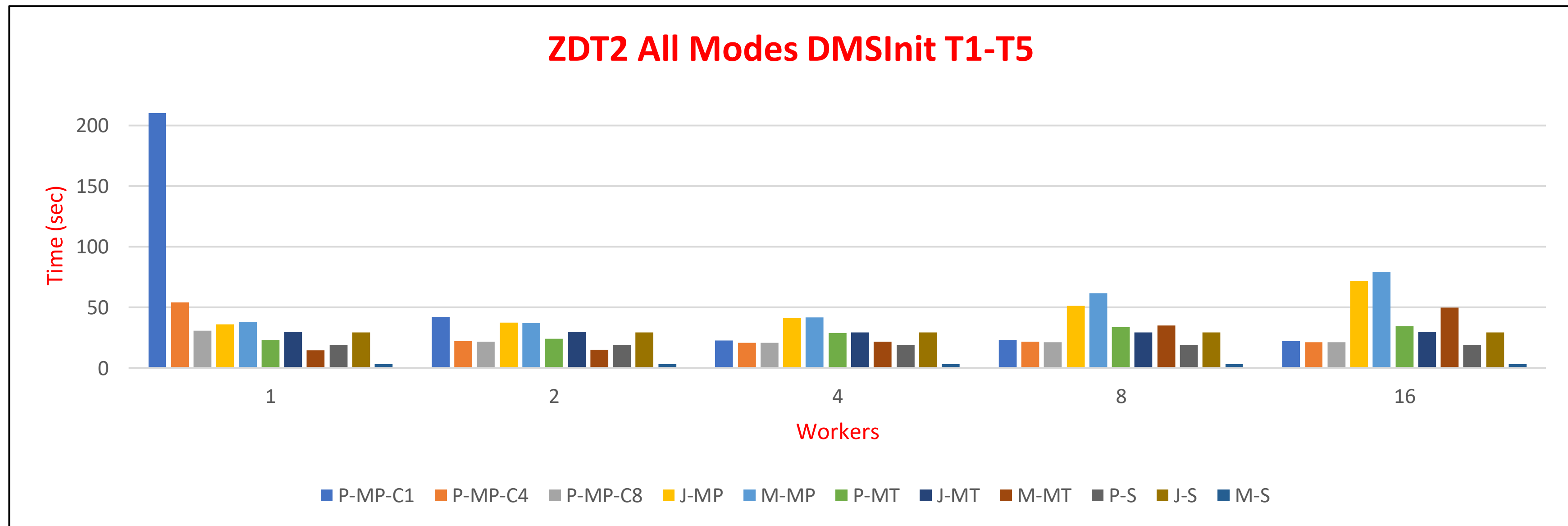


Figura 53 - Benchmarks ZDT2 DMSInit T1-T5

Tabela 53 - Resultados Benchmarks ZDT2 DMSInit T2-T5

ZDT2 DMSInit T2-T5											
Workers	P-MP-C1	P-MP-C4	P-MP-C8	J-MP	M-MP	P-MT	J-MT	M-MT	P-S	J-S	M-S
1	216,102	53,841	30,556	36,218	38,338	23,193	27,045	14,813	18,906	26,929	3,043
2	44,031	22,207	21,510	37,576	36,757	24,133	27,217	15,221	18,906	26,929	3,043
4	22,687	21,024	20,599	41,121	41,864	28,965	26,894	21,991	18,906	26,929	3,043
8	23,393	21,558	21,218	51,440	61,613	33,863	26,705	35,086	18,906	26,929	3,043
16	22,399	21,301	21,444	67,951	76,351	34,741	27,164	49,830	18,906	26,929	3,043

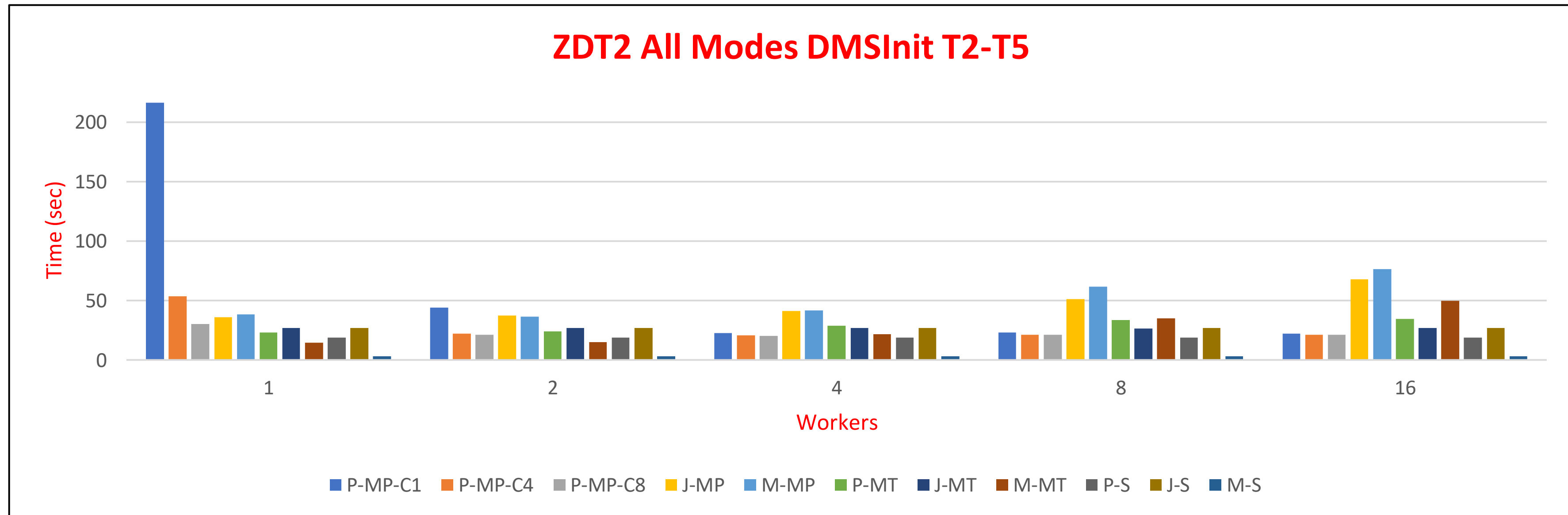


Figura 54 - Benchmarks ZDT2 DMSInit T2-T5

Tabela 54 - Resultados Benchmarks ZDT2 DriverInit T1-T5

ZDT2 DriverInit T1-T5											
Workers	P-MP-C1	P-MP-C4	P-MP-C8	J-MP	M-MP	P-MT	J-MT	M-MT	P-S	J-S	M-S
1	210,350	53,959	30,901	27,343	21,992	23,155	29,756	13,454	19,098	29,590	3,179
2	42,080	22,283	21,735	27,728	20,195	24,047	29,904	13,275	19,098	29,590	3,179
4	22,553	20,726	20,702	28,606	21,960	29,042	29,640	18,648	19,098	29,590	3,179
8	23,257	21,637	21,270	32,935	41,018	33,877	29,523	26,319	19,098	29,590	3,179
16	22,391	21,275	21,524	41,528	45,307	34,828	29,950	31,453	19,098	29,590	3,179

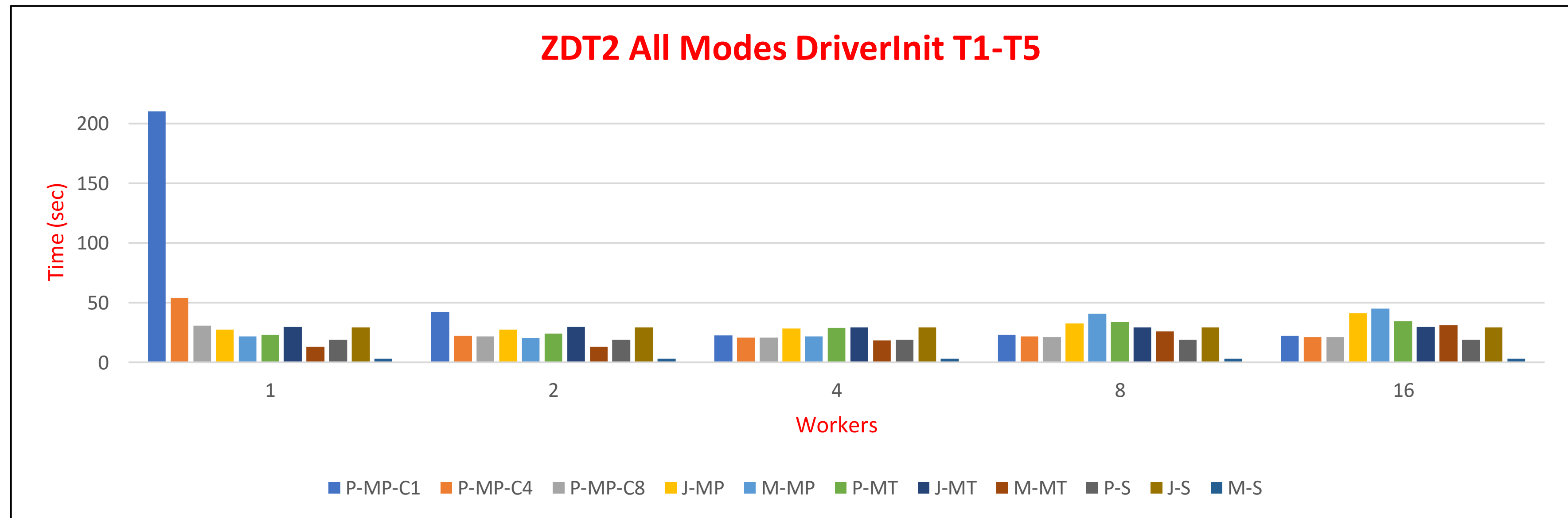


Figura 55 - Benchmarks ZDT2 DriverInit T1-T5

Tabela 55 - Resultados Benchmarks ZDT2 DriverInit T2-T5

ZDT2 DriverInit T2-T5											
Workers	**P-MP-C1**	**P-MP-C4**	**P-MP-C8**	**J-MP**	**M-MP**	**P-MT**	**J-MT**	**M-MT**	**P-S**	**J-S**	**M-S**
1	216,102	53,841	30,556	26,741	21,637	23,193	27,045	13,570	18,906	26,929	3,043
2	44,031	22,207	21,510	27,246	19,744	24,133	27,217	13,023	18,906	26,929	3,043
4	22,687	21,024	20,599	28,056	20,993	28,965	26,894	18,538	18,906	26,929	3,043
8	23,393	21,558	21,218	32,111	40,072	33,863	26,705	25,296	18,906	26,929	3,043
16	22,399	21,301	21,444	38,035	43,253	34,741	27,164	30,393	18,906	26,929	3,043

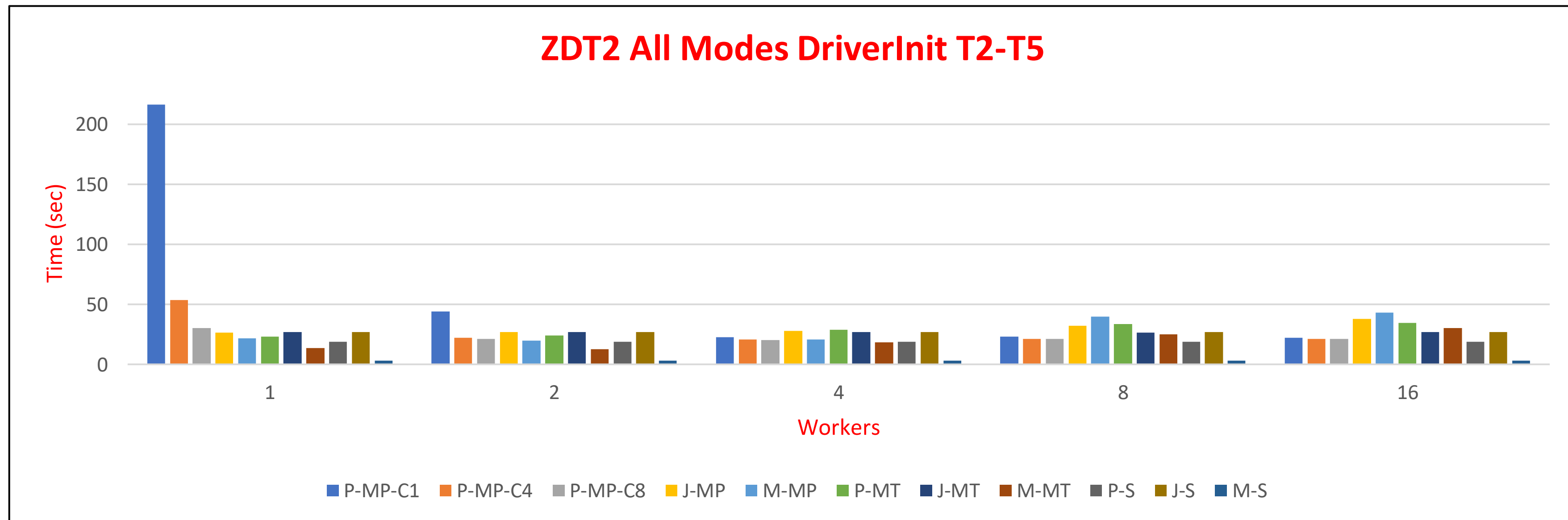


Figura 56 - Benchmarks ZDT2 DriverInit T2-T5

Tabela 56 - Resultados Benchmarks ZDT3 DMSInit T1-T5

ZDT3 DMSInit T1-T5											
Workers	**P-MP-C1**	**P-MP-C4**	**P-MP-C8**	**J-MP**	**M-MP**	**P-MT**	**J-MT**	**M-MT**	**P-S**	**J-S**	**M-S**
1	128,383	23,703	10,098	23,344	22,169	7,777	17,377	5,479	6,508	17,335	0,938
2	15,543	7,363	7,153	25,029	22,524	8,319	17,473	6,266	6,508	17,335	0,938
4	7,354	7,097	7,122	28,388	26,067	10,245	17,398	9,793	6,508	17,335	0,938
8	7,621	7,049	6,939	35,969	34,696	11,687	17,463	16,203	6,508	17,335	0,938
16	7,523	7,082	6,883	53,278	48,296	11,904	17,095	24,982	6,508	17,335	0,938

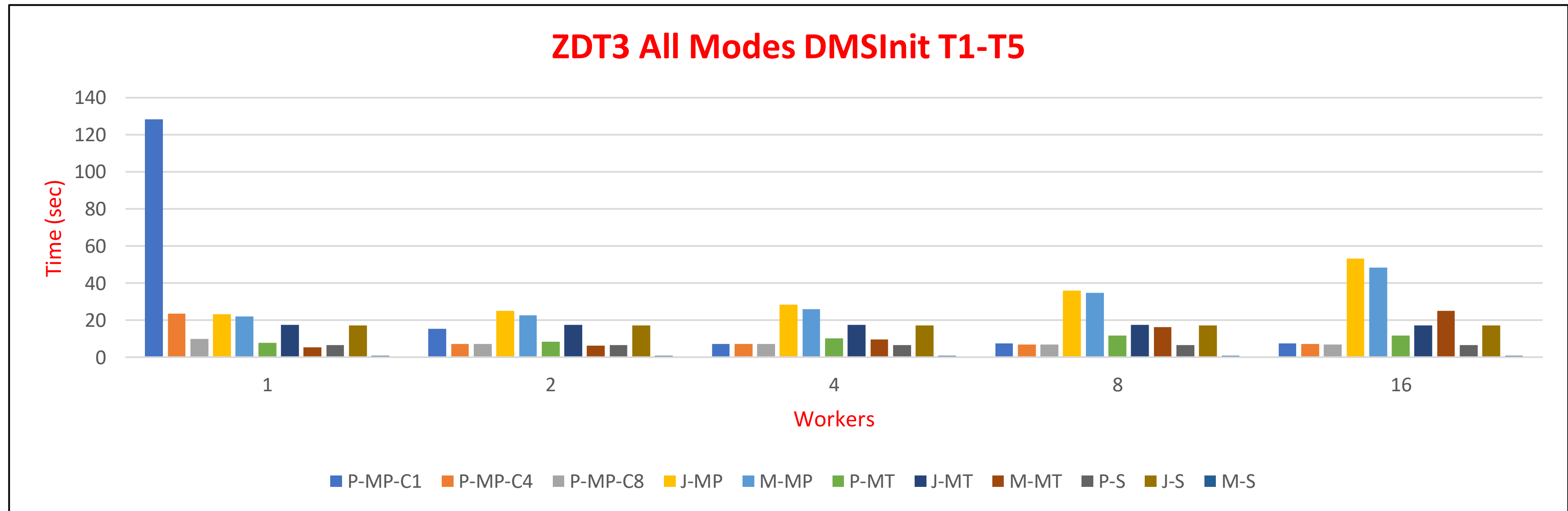


Figura 57 - Benchmarks ZDT3 DMSInit T1-T5

Tabela 57 - Resultados Benchmarks ZDT3 DMSInit T2-T5

ZDT3 DMSInit T2-T5											
Workers	**P-MP-C1**	**P-MP-C4**	**P-MP-C8**	**J-MP**	**M-MP**	**P-MT**	**J-MT**	**M-MT**	**P-S**	**J-S**	**M-S**
1	133,786	24,372	10,036	23,386	22,176	7,722	14,072	5,438	6,464	14,010	0,832
2	15,949	7,329	7,082	25,036	22,516	8,321	14,062	6,267	6,464	14,010	0,832
4	7,552	7,221	7,234	28,439	26,093	10,172	14,066	9,737	6,464	14,010	0,832
8	7,632	7,032	6,965	35,963	34,734	11,703	14,042	16,127	6,464	14,010	0,832
16	7,524	7,164	6,889	49,019	45,490	11,972	13,699	24,604	6,464	14,010	0,832

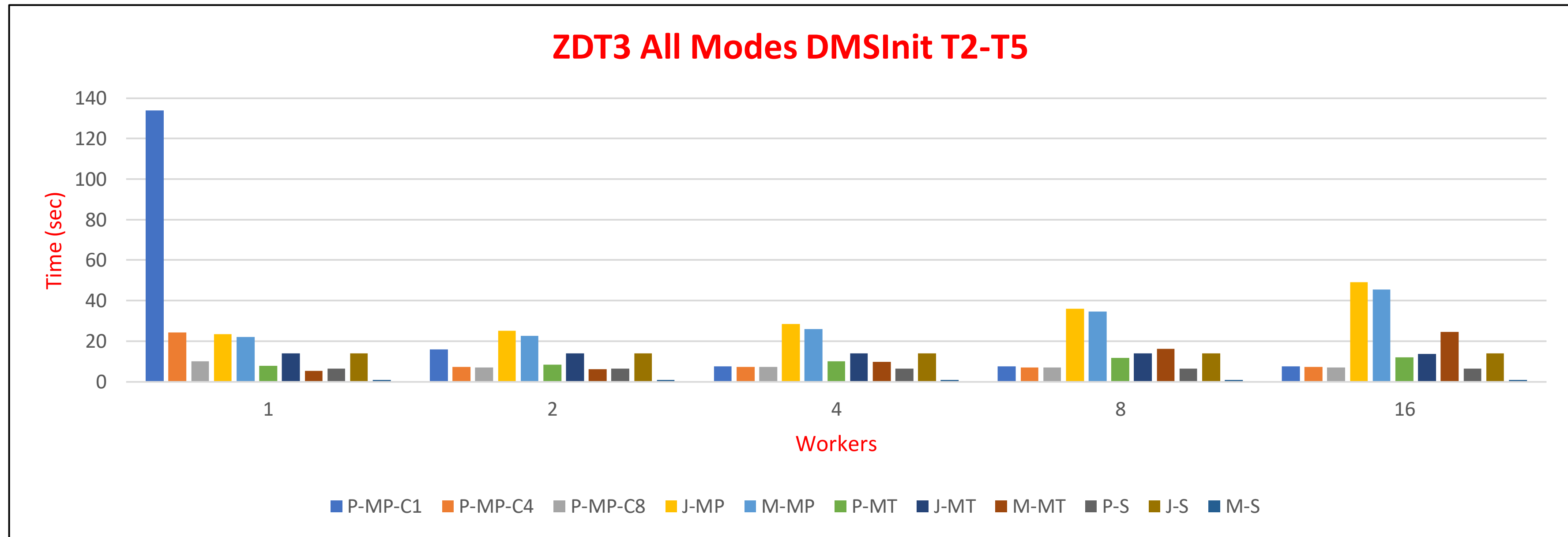


Figura 58 - Benchmarks ZDT3 DMSInit T2-T5

Tabela 58 - Resultados Benchmarks ZDT3 DriverInit T1-T5

ZDT3 DriverInit T1-T5											
Workers	**P-MP-C1**	**P-MP-C4**	**P-MP-C8**	**J-MP**	**M-MP**	**P-MT**	**J-MT**	**M-MT**	**P-S**	**J-S**	**M-S**
1	128,383	23,703	10,098	14,623	7,915	7,777	17,377	4,416	6,508	17,335	0,938
2	15,543	7,363	7,153	14,747	7,379	8,319	17,473	4,160	6,508	17,335	0,938
4	7,354	7,097	7,122	15,471	8,557	10,245	17,398	6,057	6,508	17,335	0,938
8	7,621	7,049	6,939	16,876	15,186	11,687	17,463	9,337	6,508	17,335	0,938
16	7,523	7,082	6,883	22,325	18,156	11,904	17,095	11,486	6,508	17,335	0,938

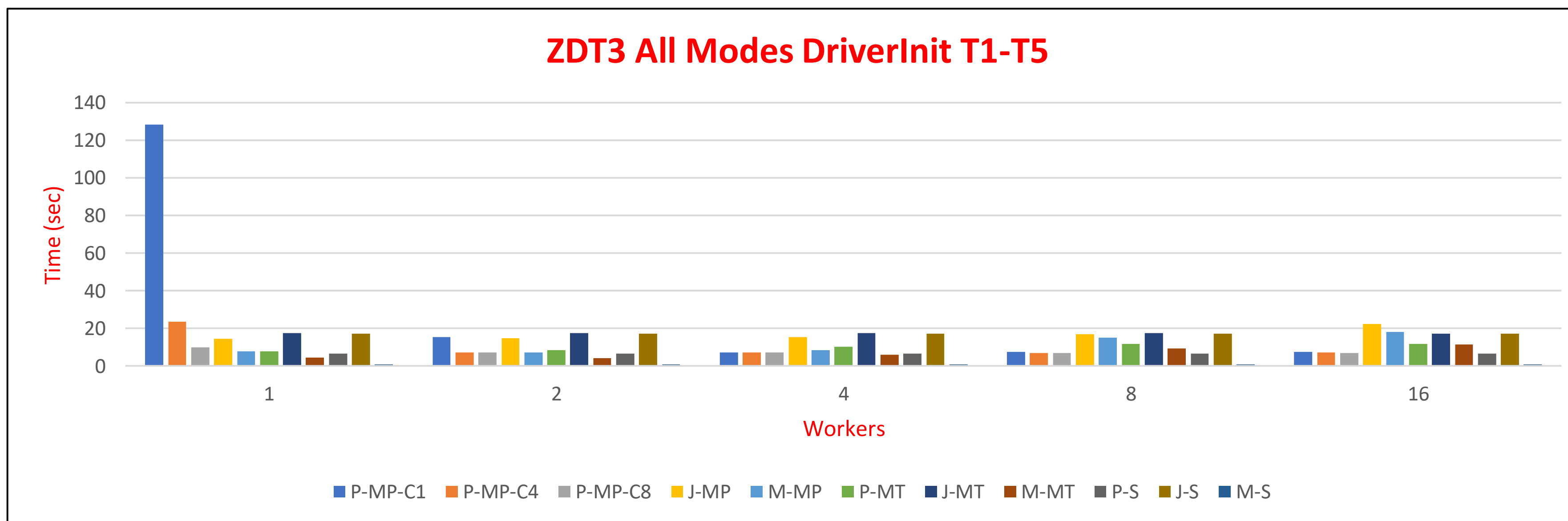


Figura 59 - Benchmarks ZDT3 DriverInit T1-T5

Tabela 59 - Resultados Benchmarks ZDT3 DriverInit T2-T5

ZDT3 DriverInit T2-T5											
Workers	**P-MP-C1**	**P-MP-C4**	**P-MP-C8**	**J-MP**	**M-MP**	**P-MT**	**J-MT**	**M-MT**	**P-S**	**J-S**	**M-S**
1	133,786	24,372	10,036	14,080	7,716	7,722	14,072	4,407	6,464	14,010	0,832
2	15,949	7,329	7,082	14,208	7,106	8,321	14,062	4,082	6,464	14,010	0,832
4	7,552	7,221	7,234	14,870	8,151	10,172	14,066	5,970	6,464	14,010	0,832
8	7,632	7,032	6,965	16,234	14,691	11,703	14,042	8,961	6,464	14,010	0,832
16	7,524	7,164	6,889	18,081	17,196	11,972	13,699	11,070	6,464	14,010	0,832

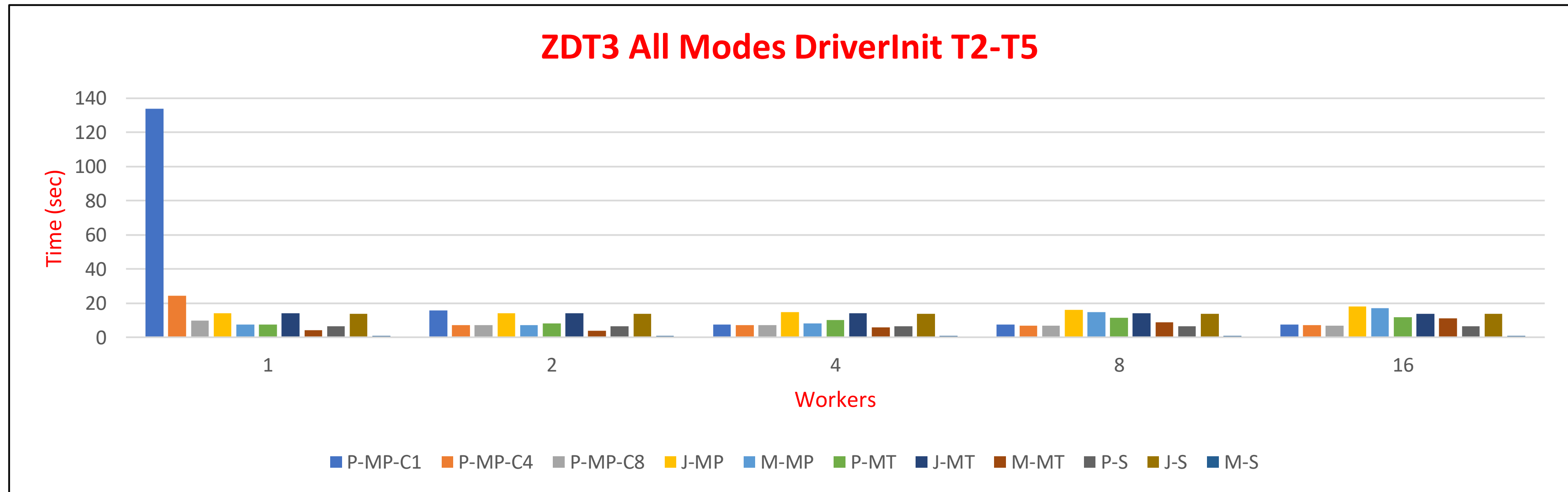


Figura 60 - Benchmarks ZDT3 DriverInit T2-T5

Tabela 60 - Resultados Benchmarks ZDT4 DMSInit T1-T5

ZDT4 DMSInit T1-T5											
Workers	**P-MP-C1**	**P-MP-C4**	**P-MP-C8**	**J-MP**	**M-MP**	**P-MT**	**J-MT**	**M-MT**	**P-S**	**J-S**	**M-S**
1	441,830	85,706	53,729	34,007	48,001	17,177	26,981	19,892	15,062	26,976	3,079
2	36,040	17,969	17,692	35,769	45,524	19,261	27,226	21,393	15,062	26,976	3,079
4	18,146	16,795	17,342	41,164	50,102	20,698	26,887	29,302	15,062	26,976	3,079
8	18,159	17,709	17,346	53,617	89,677	23,586	27,163	39,106	15,062	26,976	3,079
16	18,159	17,400	17,172	74,652	81,076	24,104	27,659	56,360	15,062	26,976	3,079

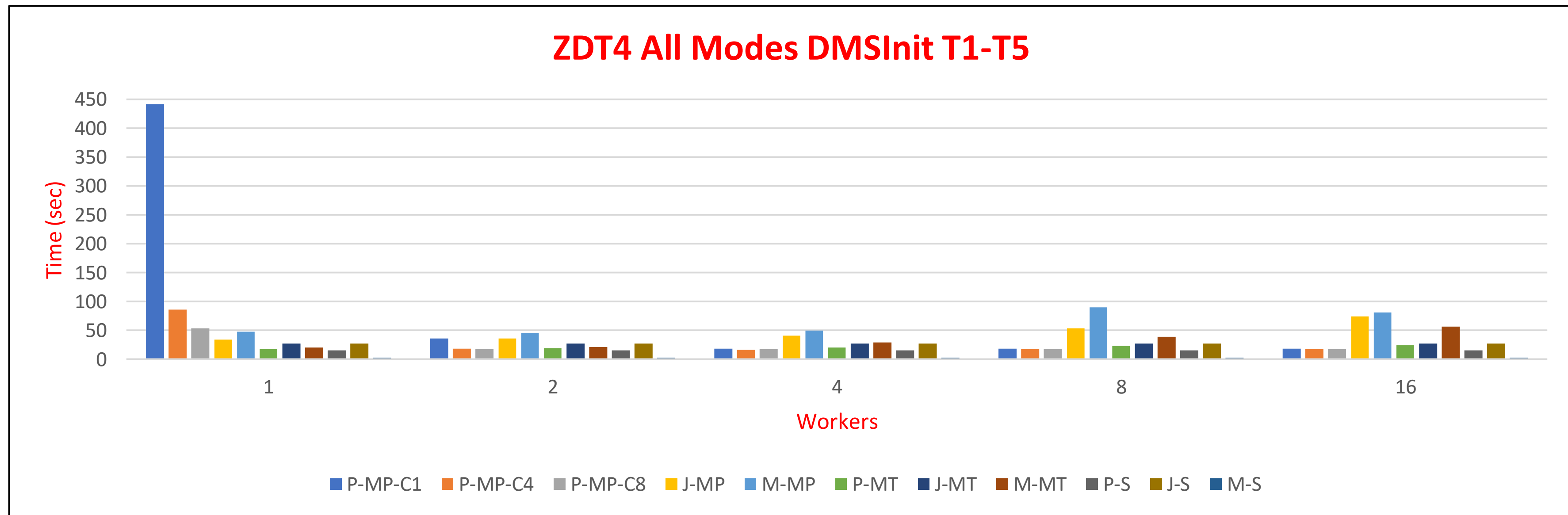


Figura 61 - Benchmarks ZDT4 DMSInit T1-T5

Tabela 61 - Resultados Benchmarks ZDT4 DMSInit T2-T5

ZDT4 DMSInit T2-T5											
Workers	P-MP-C1	P-MP-C4	P-MP-C8	J-MP	M-MP	P-MT	J-MT	M-MT	P-S	J-S	M-S
1	441,276	84,074	53,100	34,123	48,033	17,304	23,358	19,781	14,995	23,432	2,961
2	35,666	17,855	17,537	35,821	45,498	19,283	23,598	21,217	14,995	23,432	2,961
4	18,246	16,954	17,258	41,088	50,041	20,810	23,208	29,278	14,995	23,432	2,961
8	18,109	17,752	17,379	53,513	89,578	23,595	23,497	38,718	14,995	23,432	2,961
16	18,145	17,378	17,179	69,965	78,113	24,009	23,814	56,974	14,995	23,432	2,961

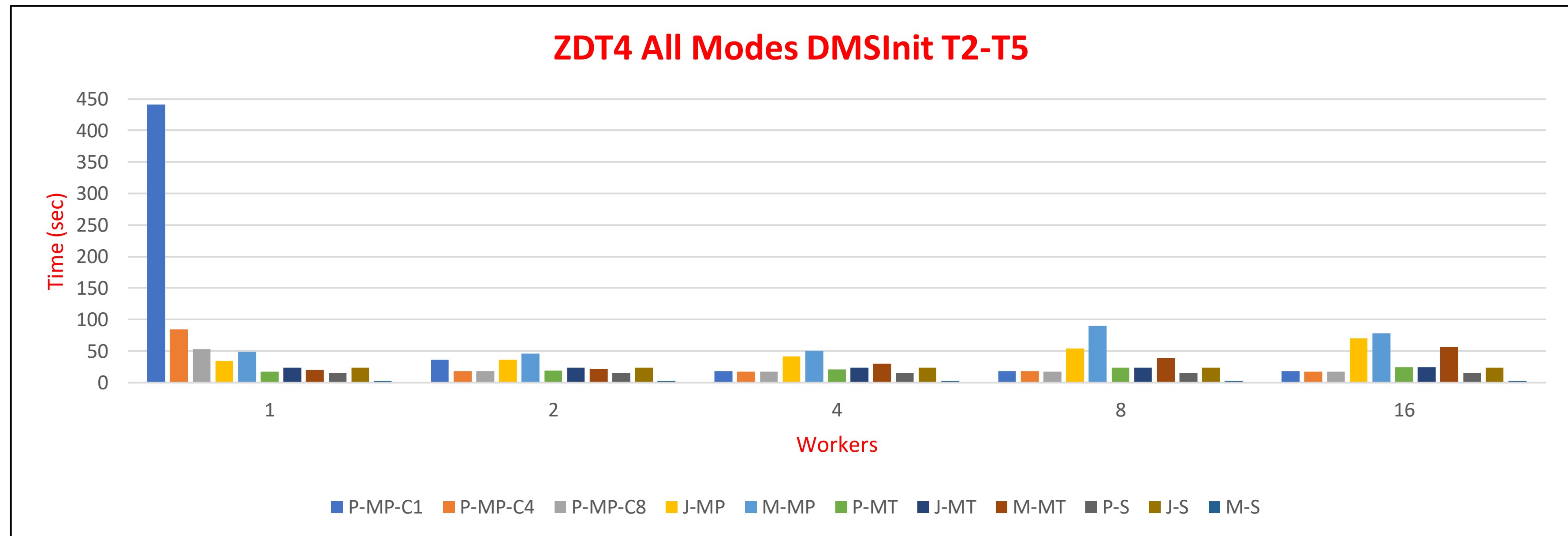


Figura 62 - Benchmarks ZDT4 DMSInit T2-T5

Tabela 62 - Resultados Benchmarks ZDT4 DriverInit T1-T5

ZDT4 DriverInit T1-T5											
Workers	P-MP-C1	P-MP-C4	P-MP-C8	J-MP	M-MP	P-MT	J-MT	M-MT	P-S	J-S	M-S
1	441,830	85,706	53,729	24,905	32,918	17,177	26,981	19,739	15,062	26,976	3,079
2	36,040	17,969	17,692	25,329	29,341	19,261	27,226	19,925	15,062	26,976	3,079
4	18,146	16,795	17,342	28,074	30,093	20,698	26,887	25,636	15,062	26,976	3,079
8	18,159	17,709	17,346	33,059	65,644	23,586	27,163	30,606	15,062	26,976	3,079
16	18,159	17,400	17,172	42,695	44,403	24,104	27,659	37,351	15,062	26,976	3,079

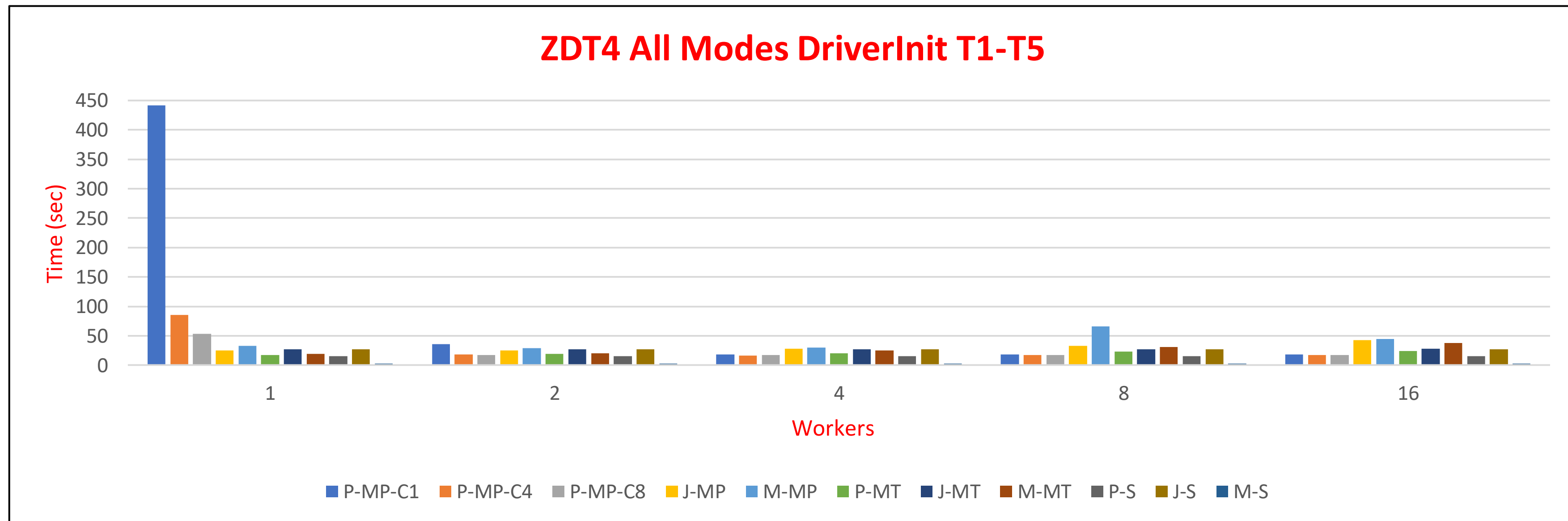


Figura 63 - Benchmarks ZDT4 DriverInit T1-T5

Tabela 63 - Resultados Benchmarks ZDT4 DriverInit T2-T5

ZDT4 DriverInit T2-T5											
Workers	**P-MP-C1**	**P-MP-C4**	**P-MP-C8**	**J-MP**	**M-MP**	**P-MT**	**J-MT**	**M-MT**	**P-S**	**J-S**	**M-S**
1	441,276	84,074	53,100	24,252	32,622	17,304	23,358	19,732	14,995	23,432	2,961
2	35,666	17,855	17,537	24,575	28,888	19,283	23,598	19,747	14,995	23,432	2,961
4	18,246	16,954	17,258	27,494	29,253	20,810	23,208	25,255	14,995	23,432	2,961
8	18,109	17,752	17,379	32,316	64,225	23,595	23,497	29,829	14,995	23,432	2,961
16	18,145	17,378	17,179	38,043	41,675	24,009	23,814	36,748	14,995	23,432	2,961

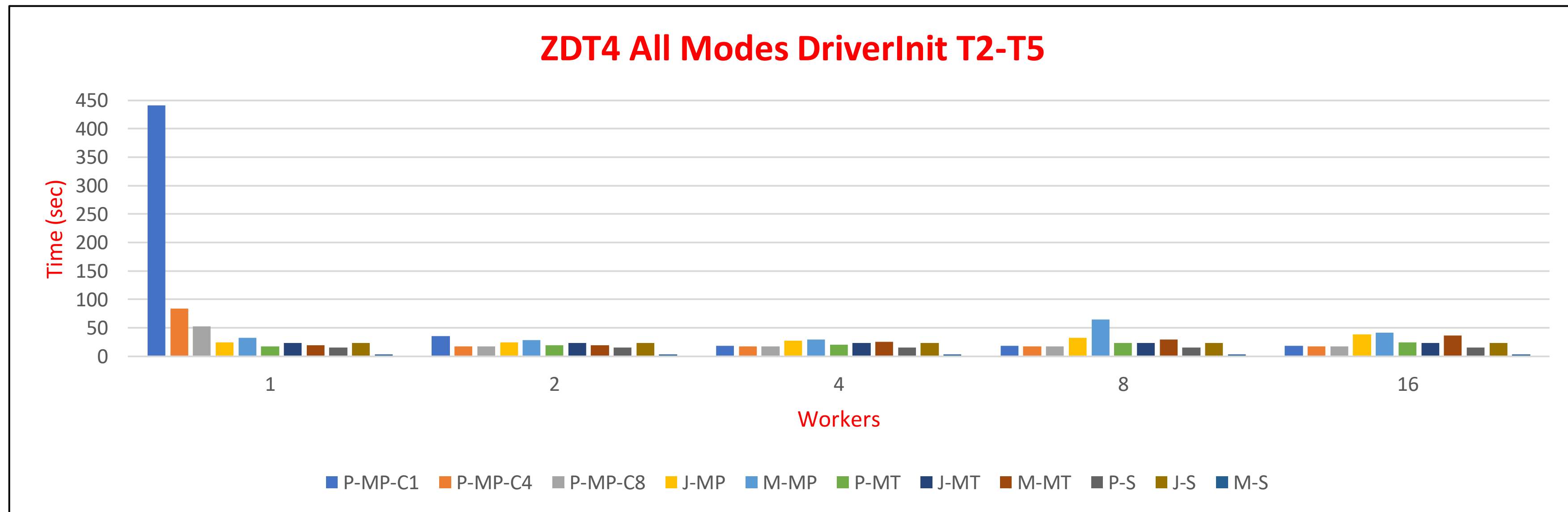


Figura 64 - Benchmarks ZDT4 DriverInit T2-T5

Tabela 64 - Resultados Benchmarks ZDT6 DMSInit T1-T5

ZDT6 DMSInit T1-T5											
Workers	**P-MP-C1**	**P-MP-C4**	**P-MP-C8**	**J-MP**	**M-MP**	**P-MT**	**J-MT**	**M-MT**	**P-S**	**J-S**	**M-S**
1	89,228	35,985	21,259	20,362	28,189	5,682	13,743	8,696	4,130	13,767	0,744
2	6,537	5,175	5,524	22,212	27,505	5,929	13,758	9,484	4,130	13,767	0,744
4	5,158	5,131	5,482	26,237	30,131	6,189	13,916	12,952	4,130	13,767	0,744
8	5,189	5,094	5,301	33,443	50,994	6,717	13,841	17,716	4,130	13,767	0,744
16	5,315	5,284	5,399	51,414	43,747	6,914	13,685	28,692	4,130	13,767	0,744

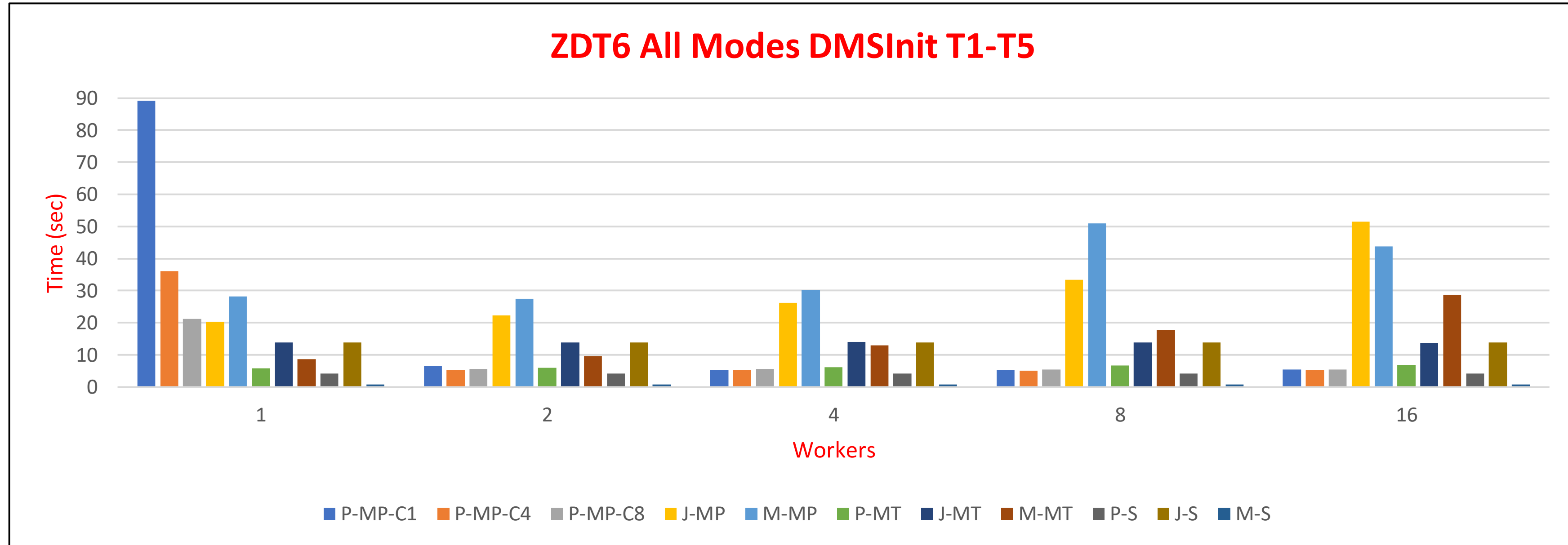


Figura 65 - Benchmarks ZDT6 DMSInit T1-T5

Tabela 65 - Resultados Benchmarks ZDT6 DMSInit T2-T5

ZDT6 DMSInit T2-T5											
Workers	**P-MP-C1**	**P-MP-C4**	**P-MP-C8**	**J-MP**	**M-MP**	**P-MT**	**J-MT**	**M-MT**	**P-S**	**J-S**	**M-S**
1	87,184	37,407	21,156	20,404	28,160	5,693	10,604	8,638	4,117	10,689	0,639
2	5,774	5,143	5,500	22,192	27,522	5,938	10,606	9,507	4,117	10,689	0,639
4	5,197	5,134	5,538	26,160	30,133	6,208	10,761	13,086	4,117	10,689	0,639
8	5,194	5,099	5,305	33,361	50,887	6,757	10,809	17,743	4,117	10,689	0,639
16	5,311	5,279	5,381	47,505	40,654	6,978	10,599	28,244	4,117	10,689	0,639

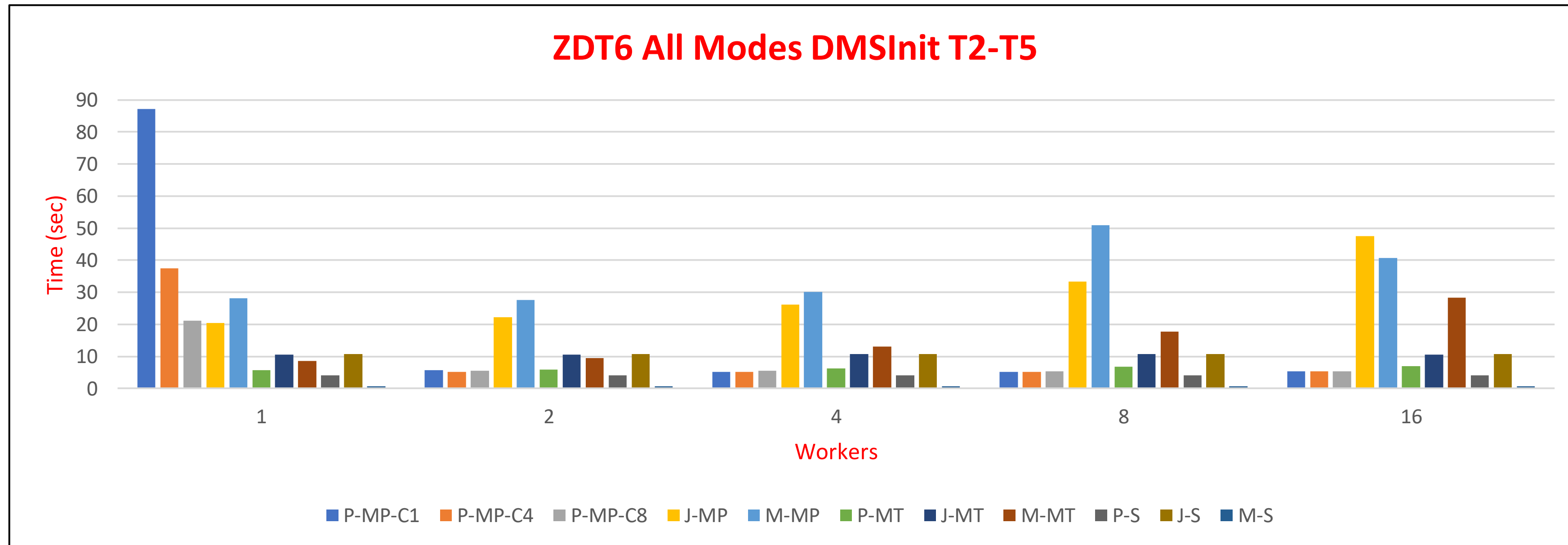


Figura 66 - Benchmarks ZDT6 DMSInit T2-T5

Tabela 66 - Resultados Benchmarks ZDT6 DriverInit T1-T5

ZDT6 DriverInit T1-T5											
Workers	P-MP-C1	P-MP-C4	P-MP-C8	J-MP	M-MP	P-MT	J-MT	M-MT	P-S	J-S	M-S
1	89,228	35,985	21,259	11,554	13,253	5,682	13,743	7,570	4,130	13,767	0,744
2	6,537	5,175	5,524	11,664	12,039	5,929	13,758	7,509	4,130	13,767	0,744
4	5,158	5,131	5,482	12,511	12,096	6,189	13,916	9,557	4,130	13,767	0,744
8	5,189	5,094	5,301	13,815	30,974	6,717	13,841	10,988	4,130	13,767	0,744
16	5,315	5,284	5,399	18,066	13,725	6,914	13,685	14,184	4,130	13,767	0,744

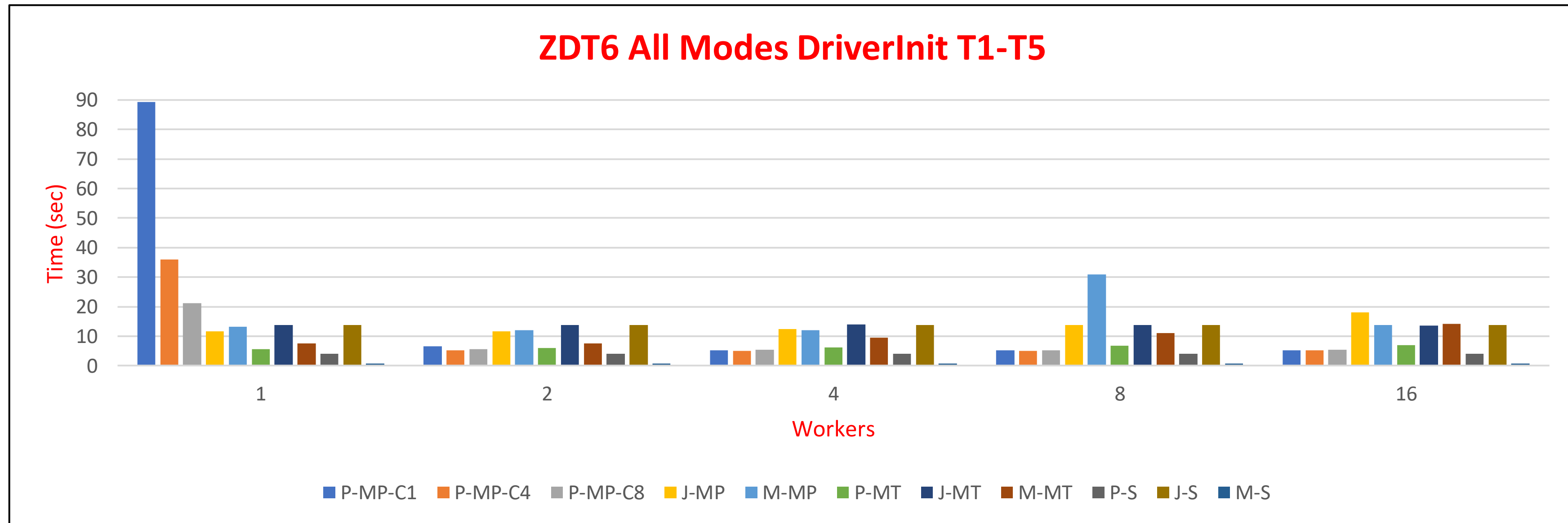


Figura 67 - Benchmarks ZDT6 DriverInit T1-T5

Tabela 67 - Resultados Benchmarks ZDT6 DriverInit T2-T5

ZDT6 DriverInit T2-T5											
Workers	**P-MP-C1**	**P-MP-C4**	**P-MP-C8**	**J-MP**	**M-MP**	**P-MT**	**J-MT**	**M-MT**	**P-S**	**J-S**	**M-S**
1	87,184	37,407	21,156	10,994	13,004	5,693	10,604	7,457	4,117	10,689	0,639
2	5,774	5,143	5,500	11,148	11,677	5,938	10,606	7,499	4,117	10,689	0,639
4	5,197	5,134	5,538	11,935	11,502	6,208	10,761	9,568	4,117	10,689	0,639
8	5,194	5,099	5,305	13,163	30,492	6,757	10,809	10,685	4,117	10,689	0,639
16	5,311	5,279	5,381	13,633	12,704	6,978	10,599	13,703	4,117	10,689	0,639

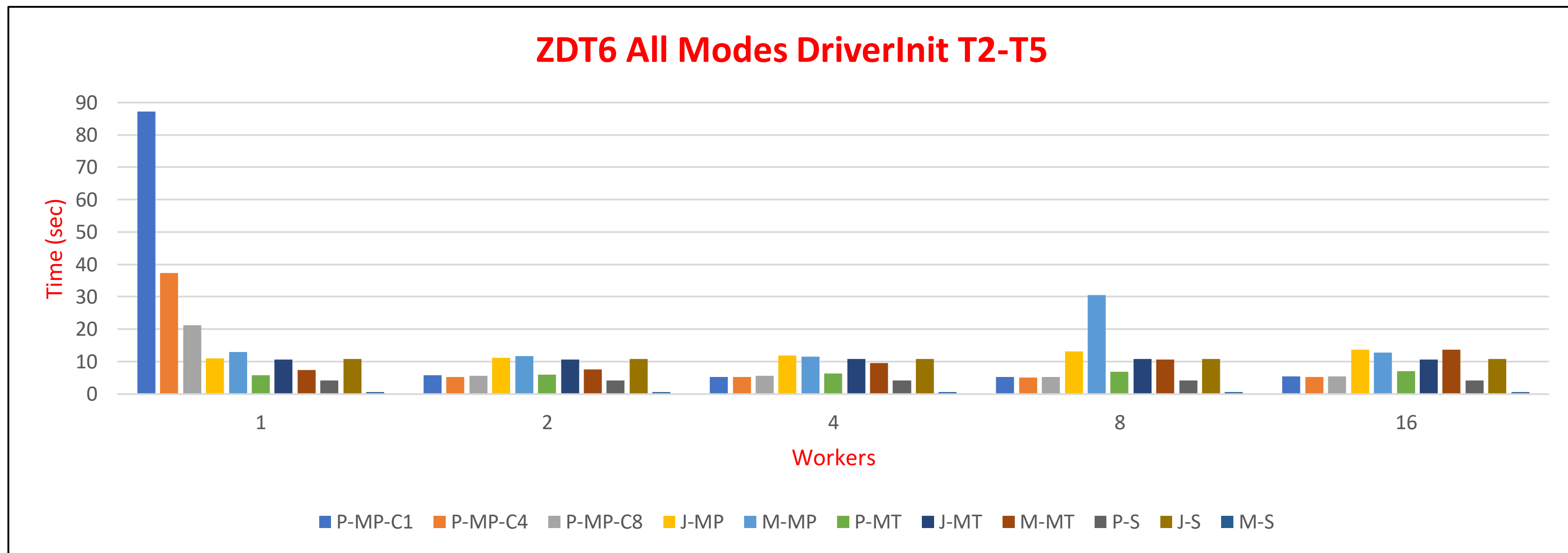


Figura 68 - Benchmarks ZDT6 DriverInit T2-T5

Tabela 68 - Performance Comparison - All Modes

	Sequential				Multiprocessing				Multithreading			
Func.	Rank.	Lang.	Time	Ratio	Rank.	Lang.	Time	Ratio	Rank.	Lang.	Time	Ratio
Styrene	1º	Julia	5675,5	1,01	1º	P-16-C1	473,3	1,13	1º	P-16	497,88	11,35
	2º	Python	5714,0		2º	M-16-DMSInit	536,86		2º	J-16	5652,44	
ZDT1	1º	Matlab	3,6	5,20	1º	P-16-C8	20,77	1,71	1º	M-1-DMSInit	14,4	1,49
	2º	Python	18,6		2º	J-1-DMSInit	35,41		2º	P-1	21,39	
ZDT2	1º	Matlab	3,2	6,01	1º	P-4-C8	20,70	1,75	1º	M-1-DMSInit	14,7	1,57
	2º	Python	19,1		2º	J-1-DMSInit	36,23		2º	P-1	23,16	
ZDT3	1º	Matlab	0,9	6,94	1º	P-16-C8	6,88	3,22	1º	M-1-DMSInit	5,5	1,42
	2º	Python	6,5		2º	M-1-DMSInit	22,2		2º	P-1	7,78	
ZDT4	1º	Matlab	3,1	4,89	1º	P-4-C4	16,8	2,02	1º	P-1	17,18	1,16
	2º	Python	15,1		2º	J-1-DMSInit	34,01		2º	M-1-DMSInit	19,89	
ZDT6	1º	Matlab	0,7	5,55	1º	P-8-C4	5,1	4,00	1º	P-1	5,68	1,53
	2º	Python	4,1		2º	J-1-DMSInit	20,4		2º	M-1-DMSInit	8,70	

B.1.5. Estudo ZDT All Modes

- **ZDT1**- Estudo comparativo de eficiência das linguagens para DMSInit com ambas as médias, considerando o melhor chunksize de Python:

  - Matlab-Sequencial é a execução mais eficiente em todos os *pool sizes*;
  - Matlab-*Multiprocessing* é a execução menos eficiente em todos os *pool sizes* excepto para *pool* de 2 *workers,* para o qual Julia-M*ultiprocessing* é o menos eficiente;

- De entre todos os *pool sizes*, Matlab-Sequencial é a execução mais eficiente e Matlab-*Multiprocessing* com *pool* de 16 *workers* é a execução menos eficiente.

- ZDT1- Principais diferenças para os resultados acima descritos quando consideramos todos os chunksizes de Python:
  - Python-*Multiprocessing*-chunksize=1 é a execução menos eficiente para os *pools* de 1 e 2 *workers*;
  - De entre todos os *pool sizes*, Python-*Multiprocessing*-chunksize=1 com *pool* de 1 *worker* é a execução menos eficiente;

- ZDT1- Diferenças de resultados obtidos com DriverInit:
  - Considerando o melhor chunksize de Python**:** Para T1-T5, Julia-*multithreading* é o menos eficiente para *pools* de 1 e 4 *workers* e Julia-Sequencial é o menos eficiente para *pool* de 2 *workers.* Para T2-T5, Python-*Multiprocessing* é o menos eficiente para *pool* de 1 *worker*, Julia-*Multiprocessing* é o menos eficiente para *pool* de 2 *workers* e Python-*Multithreading* é o menos eficiente para *pool* de 4 *workers*;
  - Considerando todos os chunksizes de Python: Para T1-T5, Julia-*Multithreading* é o menos eficiente para *pools* de 4 *workers.* Para T2-T5, Python-*Multithreading* é o menos eficiente para *pool* de 4 *workers* ;

- **ZDT2**- Estudo comparativo de eficiência das linguagens para DMSInit com ambas as médias, considerando o melhor chunksize de Python:
  - Matlab-Sequencial é a execução mais eficiente em todos os *pool sizes;*
  - Matlab-*Multiprocessing* é a execução menos eficiente em todos os *pool sizes* excepto para *pool* de 2 *workers,* para o qual Julia-M*ultiprocessing* é o menos eficiente;
  - De entre todos os *pool sizes*, Matlab-Sequencial é a execução mais eficiente e Matlab-*Multiprocessing* com *pool* de 16 *workers* é a execução menos eficiente;

- ZDT2- Principais diferenças para os resultados acima descritos quando consideramos todos os chunksizes de Python:
  - Python-*Multiprocessing*-chunksize=1 é a execução menos eficiente para *pools* de 1 e 2 *workers*;
  - De entre todos os *pool sizes*, Python-*Multiprocessing*-chunksize=1 com *pool* de 1 *worker* é a execução menos eficiente;
- ZDT2- Diferenças de resultados obtidos com DriverInit:
  - Considerando o melhor chunksize de Python**:** Para T1-T5, Python-*Multiprocessing* é o menos eficiente para pool de 1 worker e Julia-*Multithreading* é o menos eficiente para *pools* de 2 e 4 *workers.* Para T2-T5, Python-*Multiprocessing* é o menos eficiente para *pool* de 1 *worker*, Julia-*Multiprocessing* é o menos eficiente para *pool* de 2 *workers* e Python-*Multithreading* é o menos eficiente para *pool* de 4 *workers*;
  - Considerando todos os chunksizes de Python: Para T1-T5, Julia-*Multithreading* é o menos eficiente para *pool* de 4 *workers.* Para T2-T5, Python-*Multithreading* é o menos eficiente para *pool* de 4 *workers*;
- **ZDT3**- Estudo comparativo de eficiência das linguagens para DMSInit com ambas as médias, considerando o melhor chunksize de Python:
  - Matlab-Sequencial é a execução mais eficiente em todos os *pool sizes*;
  - Julia-*Multiprocessing* é a execução menos eficiente para todos os *pool sizes*;
  - De entre todos os *pool sizes*, Matlab-sequencial é a execução mais eficiente e Julia-*Multiprocessing* com *pool* com 16 *workers* é a execução menos eficiente;
- ZDT3- Principais diferenças para os resultados acima descritos quando consideramos todos os chunksizes de Python:
  - Python-*Multiprocessing*-chunksize=1 é a execução menos eficiente para os *pools* de 1 *worker*;
  - De entre todos os *pool sizes*, Python-*Multiprocessing*-chunksize=1 com *pool* de 1 *worker* é a execução menos eficiente;

- ZDT3- Diferenças de resultados obtidos com DriverInit:
  - Considerando o melhor chunksize de Python**:** Para ambas as médias, Julia-*Multithreading* é o menos eficiente para *pools* de 1, 2, 4 e 8 *workers*;
  - Considerando todos os chunksizes de Python: Para T1-T5, Julia-*Multithreading* é o menos eficiente para *pools* de 2, 4 e 8 *workers.* Para T2-T5, Python-*Multiprocessing*-chunksize=1 é o menos eficiente para *pool* de 2 *workers*;
- **ZDT4**- Estudo comparativo de eficiência das linguagens para DMSInit com ambas as médias, considerando o melhor chunksize de Python:
  - Matlab-Sequencial é a execução mais eficiente em todos os *pool sizes*;
  - Matlab-*Multiprocessing* é a execução menos eficiente em todos os *pool sizes* excepto para *pool* de 1 *workers,* para o qual Python-*Multiprocessing* é o menos eficiente;
  - De entre todos os *pool sizes*, Matlab-Sequencial é a execução mais eficiente e Matlab-*Multiprocessing* com *pool* de 16 *workers* é a execução menos eficiente.
- ZDT4- Principais diferenças para os resultados acima descritos quando consideramos todos os chunksizes de Python:
  - De entre todos os *pool sizes*, Python-*Multiprocessing*-chunksize=1 com *pool* de 1 *worker* é a execução menos eficiente;
- ZDT4- Diferenças de resultados obtidos com DriverInit:
  - Considerando todos os chunksizes de Python: Para ambas as médias, Python-*Multiprocessing*-chunksize=1 é o menos eficiente para *pools* de 2 *workers.*;

- **ZDT6**- Estudo comparativo de eficiência das linguagens para DMSInit com ambas as médias, considerando o melhor chunksize de Python:
  - Matlab-Sequencial é a execução mais eficiente em todos os *pool sizes*;
  - Matlab-*Multiprocessing* é a execução menos eficiente em todos os *pool sizes* excepto para *pool* de 16 *workers,* para o qual Julia-*Multiprocessing* é o menos eficiente;
  - De entre todos os *pool sizes*, Matlab-Sequencial é a execução mais eficiente e Julia-*Multiprocessing* com *pool* de 16 *workers* é a execução menos eficiente;
- ZDT6- Principais diferenças para os resultados acima descritos quando consideramos todos os chunksizes de Python:
  - Python-*Multiprocessing*-chunksize=1 é a execução menos eficiente para *pool* de 1 *worker*;
  - De entre todos os *pool sizes*, Python-*Multiprocessing*-chunksize=1 com *pool* com 1 *worker* é a execução menos eficiente;
- ZDT6- Diferenças de resultados obtidos com DriverInit:
  - Considerando o melhor chunksize de Python**:** Para T1-T5, Python-*multiprocessing* é o menos eficiente para *pool* de 1 *worker*, Julia-sequencial é o menos eficiente para pool de 2 workers e Julia-*multithreading* é o menos eficiente para *pool* de 4 *workers.* Para T2-T5, Python-*multiprocessing* é o menos eficiente para *pool* de 1 *worker*, Julia-*multiprocessing* é o menos eficiente para *pool* de 4 *workers* e Matlab-*multithreading* é o menos eficiente para *pool* de 16 *workers*;
  - Considerando todos os chunksizes de Python: Para T1-T5, Julia-sequencial é o menos eficiente para pool de 2 workers e Julia-*multithreading* é o menos eficiente para *pool* de 4 *workers.* Para T2-T5, Julia-*multiprocessing* é o menos eficiente para *pool* de 4 *workers* e Matlab-*multithreading* é o menos eficiente para *pool* de 16 *workers*;

B.2. Microbenchmarks

B.2.1. Python Multiprocessing Chunksizes

Notação utilizada:

ChunkβColφ β = chunksize: 1, 4 ou 8.

φ = número de colunas da matriz: 20, 100 ou 500.

Exemplo : Chunk4Col500 corresponde a chunksize 4 e matriz com 500 colunas.

Tabela 69 - Resultados Microbenchmarks: Python Multiprocessing - Chunksizes

Python Multiprocessing - Chunksizes									
Workers	Chunk1Col20	Chunk1Col100	Chunk1Col500	Chunk4Col20	Chunk4Col100	Chunk4Col500	Chunk8Col20	Chunk8Col100	Chunk8Col500
1	0,05322	0,09388	0,14722	0,01821	0,02612	0,05336	0,02032	0,01898	0,05851
2	0,00188	0,00196	0,00145	0,00147	0,00159	0,00196	0,00162	0,00161	0,00186
4	0,00084	0,00094	0,00096	0,00089	0,00098	0,00103	0,00088	0,00096	0,00100
8	0,00084	0,00092	0,00105	0,00079	0,00083	0,00102	0,00077	0,00083	0,00102
16	0,00094	0,00090	0,00104	0,00088	0,00098	0,00102	0,00089	0,00099	0,00102

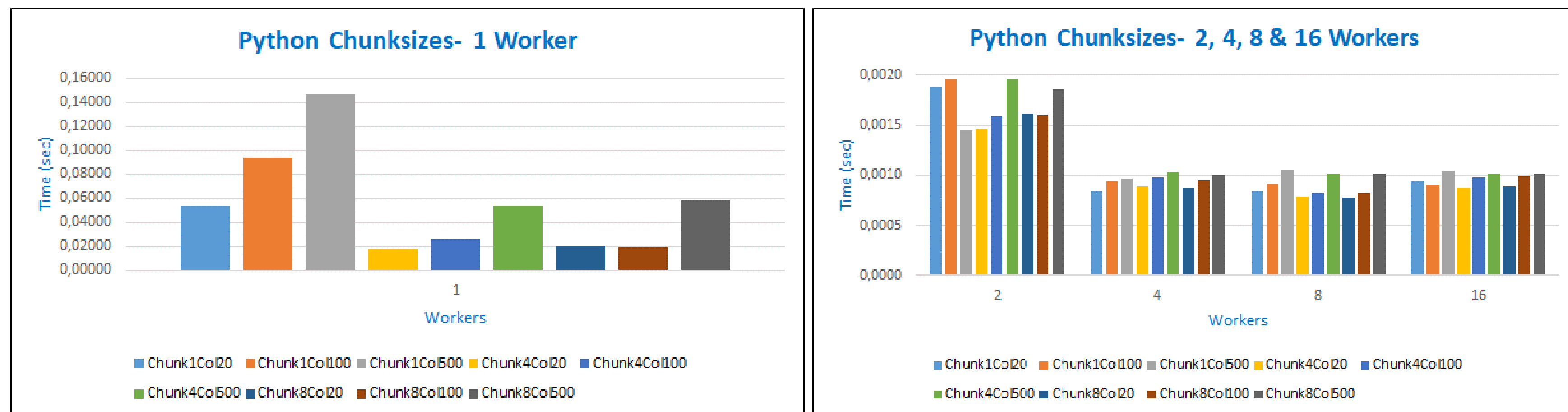


Figura 69 - Microbenchmarks Python Multiprocessing - Chunksizes

B.2.2. Comparação Microbenchmarks

Notação utilizada:

α-φ-Cβ-Colλ α = linguagem de programação: P(ython), J(ulia) ou M(atlab).

φ = modo: MP para Multiprocessing, MT para Multiththreading e S para Sequential.

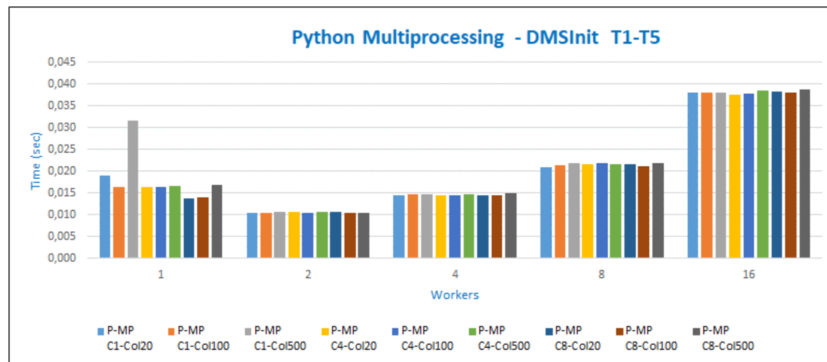
β = chunksize (apenas aplicável no caso de Python com Multiprocessing): 1, 4 ou 8.

λ = número de colunas da matriz: 20, 100 ou 500.

Exemplo: M-MT-Col100 corresponde à linguagem Matlab com Multithreading e matriz com 100 cols.

Tabela 70 - Resultados Microbenchmarks: Multiprocessing DMSInit T1-T5

Multiprocessing DMSInit T1-T5															
Workers	P-MP C1-Col20	P-MP C1-Col100	P-MP C1-Col500	P-MP C4-Col20	P-MP C4-Col100	P-MP C4-Col500	P-MP C8-Col20	P-MP C8-Col100	P-MP C8-Col500	J-MP Col20	J-MP Col100	J-MP Col500	M-MP Col20	M-MP Col100	M-MP Col500
1	0,01897	0,01642	0,03153	0,01646	0,01628	0,01672	0,01366	0,01398	0,01679	9,75322	9,04105	9,11204	19,54388	20,20816	19,73476
2	0,01047	0,01052	0,01059	0,01064	0,01052	0,01066	0,01060	0,01049	0,01052	11,28463	10,54621	10,65870	20,03883	20,29731	20,15663
4	0,01456	0,01464	0,01475	0,01457	0,01445	0,01477	0,01449	0,01457	0,01485	13,74556	13,67444	13,70942	21,38276	21,48098	21,53393
8	0,02085	0,02133	0,02190	0,02172	0,02174	0,02150	0,02160	0,02123	0,02184	19,18655	19,32641	19,25031	24,22639	24,32431	24,33743
16	0,03812	0,03806	0,03800	0,03762	0,03778	0,03841	0,03829	0,03806	0,03877	30,37159	30,39542	30,42803	31,35114	32,08776	32,10639



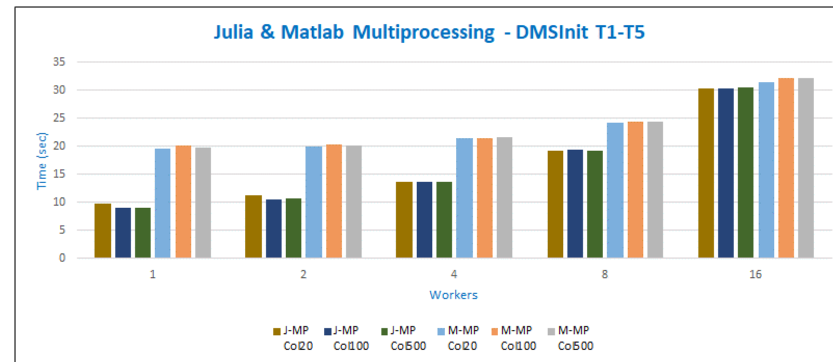
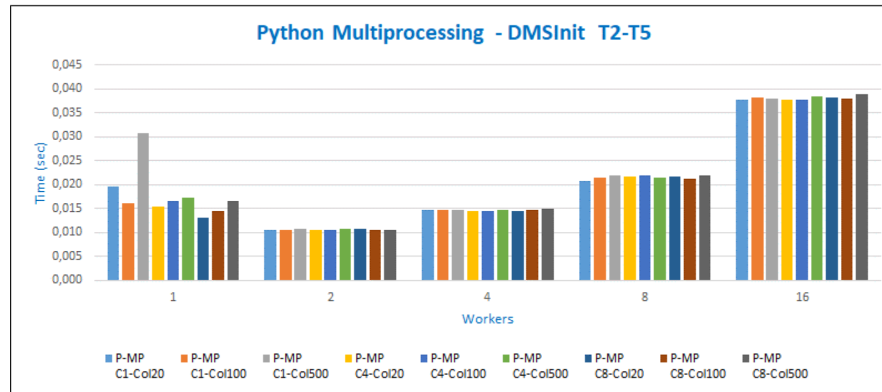


Figura 70 - Microbenchmarks Multiprocessing DMSInit T1-T5

Tabela 71 - Resultados Microbenchmarks: Multiprocessing DMSInit T2-T5

Multiprocessing DMSInit T2-T5															
Workers	P-MP C1-Col20	P-MP C1-Col100	P-MP C1-Col500	P-MP C4-Col20	P-MP C4-Col100	P-MP C4-Col500	P-MP C8-Col20	P-MP C8-Col100	P-MP C8-Col500	J-MP Col20	J-MP Col100	J-MP Col500	M-MP Col20	M-MP Col100	M-MP Col500
1	0,01956	0,01612	0,03081	0,01548	0,01656	0,01712	0,01310	0,01440	0,01657	9,76880	9,04565	9,08834	19,50833	20,16245	19,72654
2	0,01047	0,01058	0,01061	0,01058	0,01051	0,01065	0,01061	0,01046	0,01053	11,28392	10,54473	10,63448	20,07534	20,25710	20,11213
4	0,01464	0,01461	0,01475	0,01455	0,01451	0,01478	0,01448	0,01468	0,01481	13,72141	13,66097	13,68442	21,42278	21,48540	21,53157
8	0,02061	0,02136	0,02190	0,02173	0,02184	0,02145	0,02162	0,02109	0,02194	19,16776	19,31694	19,23467	24,25946	24,37142	24,34753
16	0,03771	0,03809	0,03784	0,03767	0,03775	0,03831	0,03819	0,03802	0,03879	30,10578	30,12318	30,04492	30,62136	31,02242	31,02968



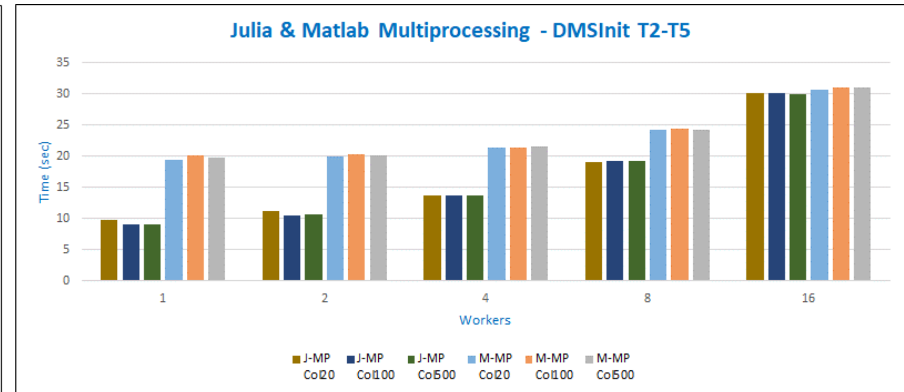
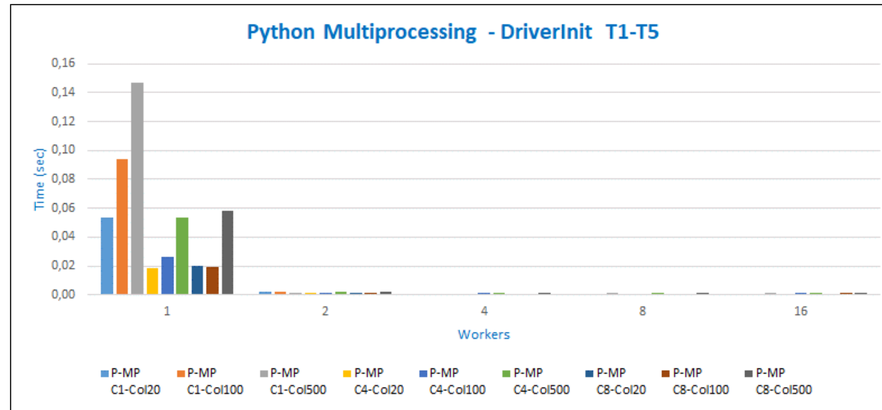


Figura 71 - Microbenchmarks Multiprocessing DMSInit T2-T5

Tabela 72 - Resultados Microbenchmarks: Multiprocessing DriverInit T1-T5

Multiprocessing DriverInit T1-T5															
Workers	P-MP C1-Col20	P-MP C1-Col100	P-MP C1-Col500	P-MP C4-Col20	P-MP C4-Col100	P-MP C4-Col500	P-MP C8-Col20	P-MP C8-Col100	P-MP C8-Col500	J-MP Col20	J-MP Col100	J-MP Col500	M-MP Col20	M-MP Col100	M-MP Col500
1	0,05322	0,09388	0,14722	0,01821	0,02612	0,05336	0,02032	0,01898	0,05851	0,00867	0,00865	0,00892	0,02981	0,03061	0,04229
2	0,00188	0,00196	0,00145	0,00147	0,00159	0,00196	0,00162	0,00161	0,00186	0,00965	0,00977	0,01114	0,02873	0,03201	0,03788
4	0,00084	0,00094	0,00096	0,00089	0,00098	0,00103	0,00088	0,00096	0,00100	0,29615	0,29177	0,29523	0,03651	0,04130	0,04249
8	0,00084	0,00092	0,00105	0,00079	0,00083	0,00102	0,00077	0,00083	0,00102	0,01693	0,01589	0,01668	0,06234	0,07394	0,07836
16	0,00094	0,00090	0,00104	0,00088	0,00098	0,00102	0,00089	0,00099	0,00102	0,03870	0,03913	0,03883	0,06615	0,10227	0,12200



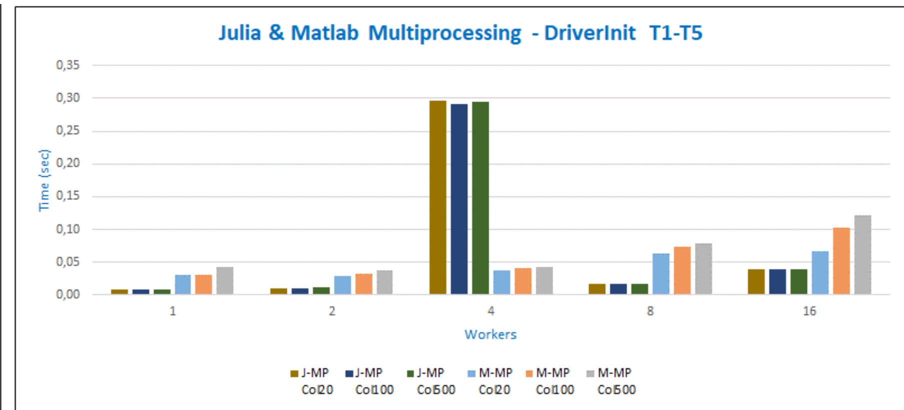
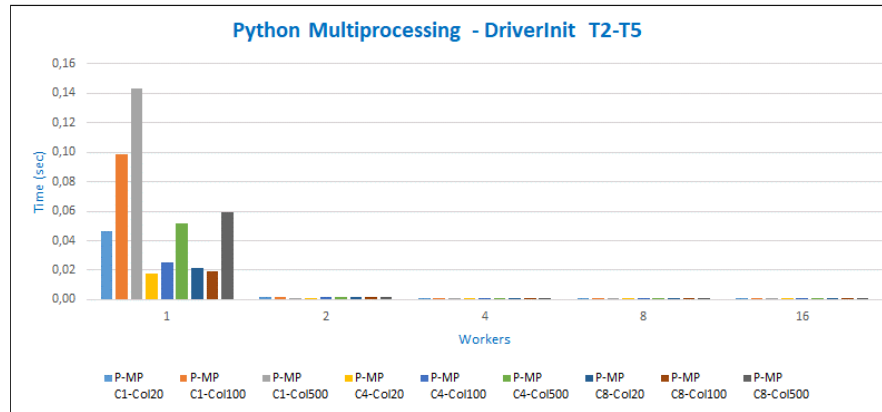


Figura 72 - Microbenchmarks Multiprocessing DriverInit T1-T5

Tabela 73 - Resultados Microbenchmarks: Multiprocessing DriverInit T2-T5

Multiprocessing DriverInit T2-T5															
Workers	P-MP C1-Col20	P-MP C1-Col100	P-MP C1-Col500	P-MP C4-Col20	P-MP C4-Col100	P-MP C4-Col500	P-MP C8-Col20	P-MP C8-Col100	P-MP C8-Col500	J-MP Col20	J-MP Col100	J-MP Col500	M-MP Col20	M-MP Col100	M-MP Col500
1	0,04682	0,09848	0,14324	0,01793	0,02533	0,05148	0,02134	0,01924	0,05906	0,00088	0,00084	0,00104	0,02781	0,02871	0,04007
2	0,00184	0,00190	0,00143	0,00141	0,00163	0,00208	0,00155	0,00175	0,00181	0,00182	0,00180	0,00270	0,02605	0,02928	0,03524
4	0,00085	0,00096	0,00096	0,00090	0,00097	0,00103	0,00089	0,00096	0,00100	0,29795	0,29354	0,29696	0,03358	0,03721	0,03990
8	0,00083	0,00094	0,00110	0,00078	0,00082	0,00101	0,00077	0,00083	0,00102	0,00894	0,00775	0,00813	0,06110	0,07025	0,07471
16	0,00090	0,00088	0,00105	0,00087	0,00098	0,00103	0,00089	0,00098	0,00101	0,01578	0,01539	0,01614	0,06046	0,09325	0,11158



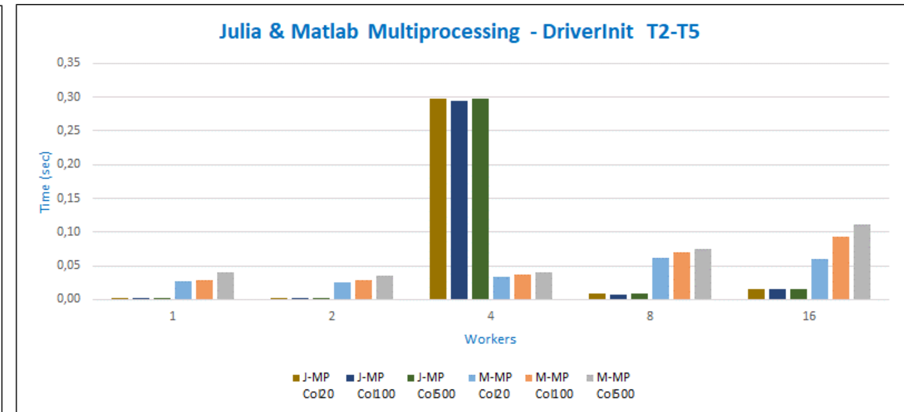
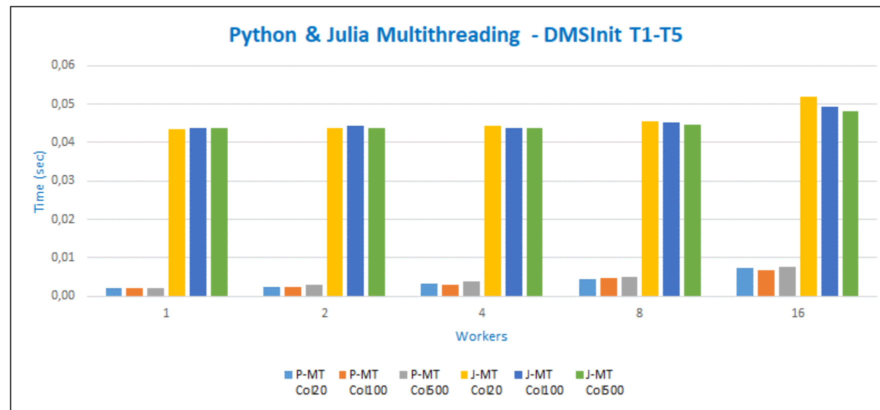


Figura 73 - Microbenchmarks Multiprocessing DriverInit T2-T5

Tabela 74 - Resultados Microbenchmarks: Multithreading DMSInit T1-T5

Multithreading DMSInit T1-T5									
Workers	P-MT Col20	P-MT Col100	P-MT Col500	J-MT Col20	J-MT Col100	J-MT Col500	M-MT Col20	M-MT Col100	M-MT Col500
1	0,00201	0,00211	0,00207	0,04356	0,04370	0,04378	1,25833	1,26793	1,26462
2	0,00250	0,00247	0,00289	0,04372	0,04430	0,04370	2,19529	2,30932	2,24040
4	0,00317	0,00306	0,00400	0,04443	0,04382	0,04363	4,13118	4,26259	4,21321
8	0,00433	0,00467	0,00501	0,04550	0,04524	0,04458	8,12246	8,31997	8,24157
16	0,00726	0,00679	0,00781	0,05192	0,04939	0,04801	16,35095	16,64718	16,57443



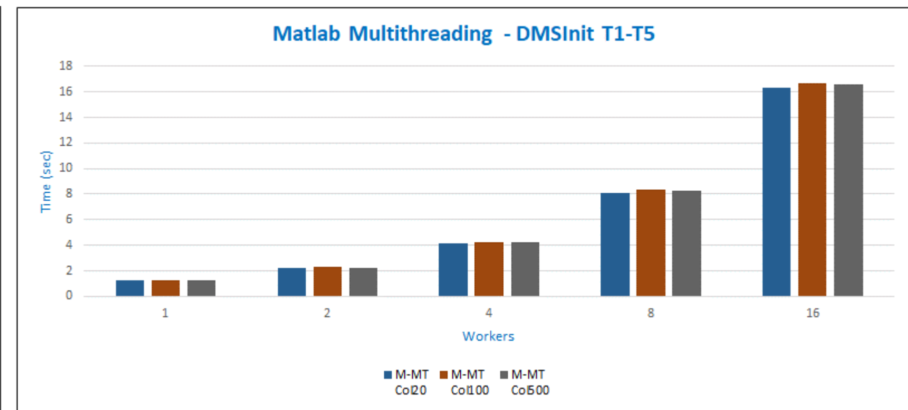
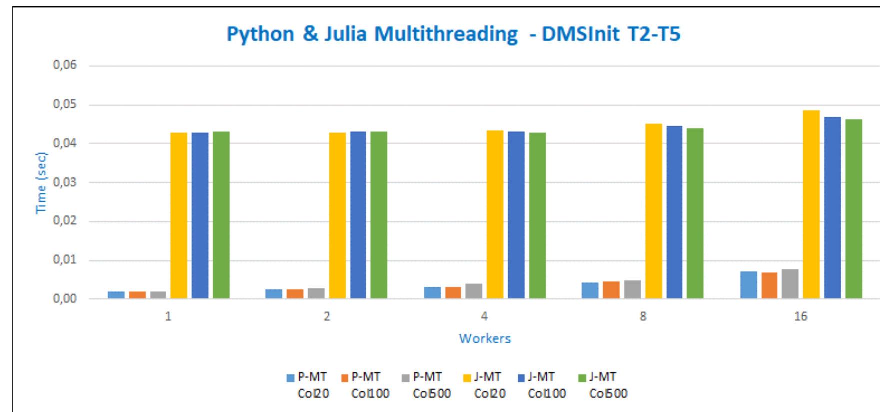


Figura 74 - Microbenchmarks Multithreading DMSInit T1-T5

Tabela 75 - Resultados Microbenchmarks: Multithreading DMSInit T2-T5

Multithreading DMSInit T2-T5									
Workers	P-MT Col20	P-MT Col100	P-MT Col500	J-MT Col20	J-MT Col100	J-MT Col500	M-MT Col20	M-MT Col100	M-MT Col500
1	0,00202	0,00211	0,00206	0,04294	0,04293	0,04310	1,26298	1,26385	1,26989
2	0,00250	0,00246	0,00290	0,04284	0,04308	0,04297	2,19149	2,31119	2,23230
4	0,00317	0,00306	0,00397	0,04347	0,04310	0,04284	4,11993	4,26153	4,20279
8	0,00432	0,00467	0,00500	0,04505	0,04441	0,04403	8,12113	8,30972	8,22784
16	0,00726	0,00681	0,00779	0,04863	0,04690	0,04635	16,24929	16,51744	16,42657



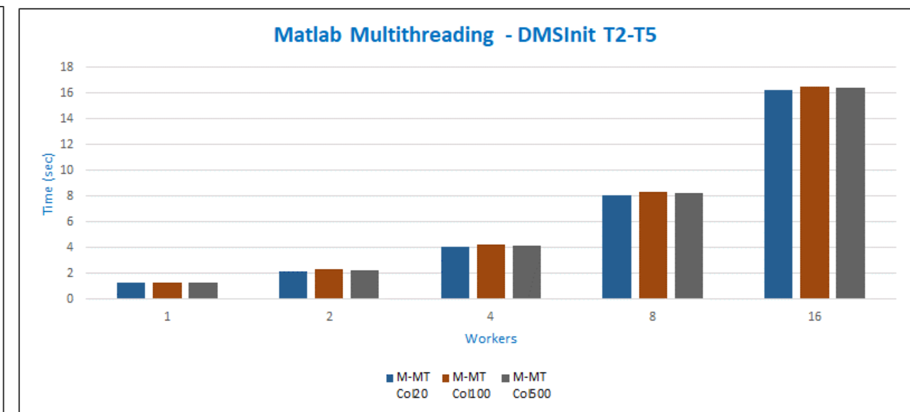
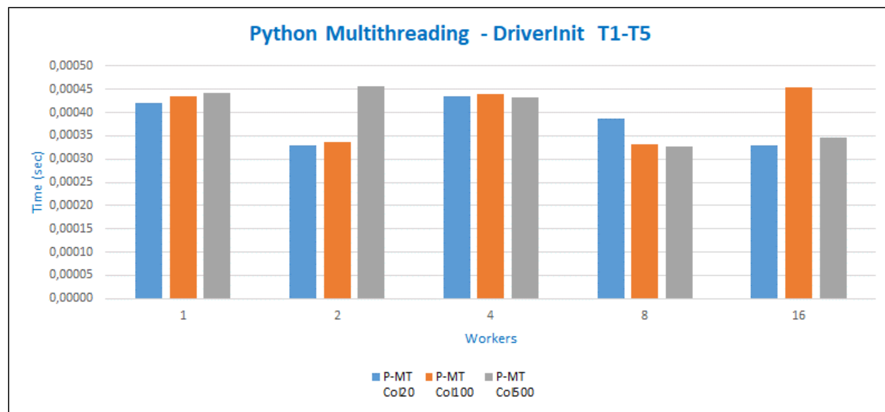


Figura 75 - Microbenchmarks Multithreading DMSInit T2-T5

Tabela 76 - Resultados Microbenchmarks: Multithreading DriverInit T1-T5

Multithreading DriverInit T1-T5									
Workers	P-MT Col20	P-MT Col100	P-MT Col500	J-MT Col20	J-MT Col100	J-MT Col500	M-MT Col20	M-MT Col100	M-MT Col500
1	0,00042	0,00044	0,00044	0,04356	0,04370	0,04378	0,01171	0,01365	0,02096
2	0,00033	0,00034	0,00046	0,04372	0,04430	0,04370	0,01445	0,01695	0,02062
4	0,00044	0,00044	0,00043	0,04443	0,04382	0,04363	0,02121	0,02504	0,02849
8	0,00039	0,00033	0,00033	0,04550	0,04524	0,04458	0,02886	0,05278	0,05476
16	0,00033	0,00046	0,00035	0,05192	0,04939	0,04801	0,04191	0,08435	0,11418



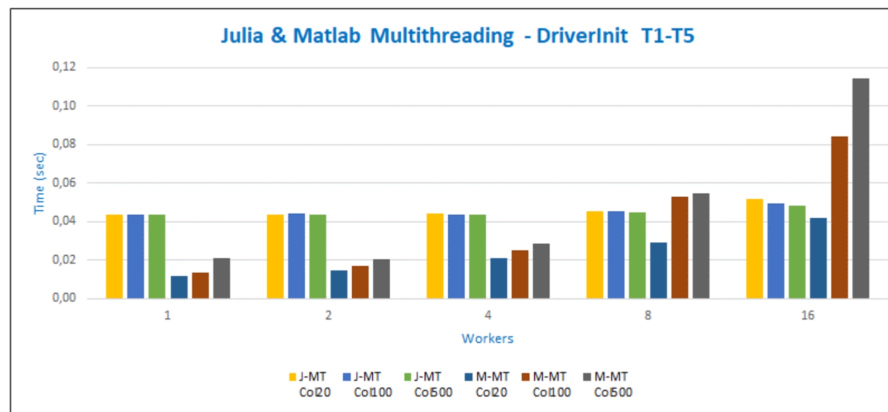
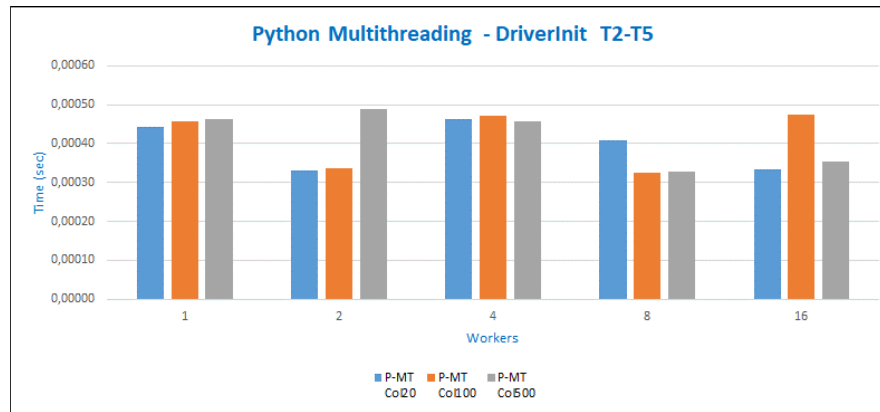


Figura 76 - Microbenchmarks Multithreading DriverInit T1-T5

Tabela 77 - Resultados Microbenchmarks: Multithreading DriverInit T2-T5

Multithreading DriverInit T2-T5									
Workers	P-MT Col20	P-MT Col100	P-MT Col500	J-MT Col20	J-MT Col100	J-MT Col500	M-MT Col20	M-MT Col100	M-MT Col500
1	0,00044	0,00046	0,00046	0,04294	0,04293	0,04310	0,01138	0,01347	0,02092
2	0,00033	0,00034	0,00049	0,04284	0,04308	0,04297	0,01432	0,01683	0,02042
4	0,00046	0,00047	0,00046	0,04347	0,04310	0,04284	0,02113	0,02465	0,02834
8	0,00041	0,00033	0,00033	0,04505	0,04441	0,04403	0,02811	0,05184	0,05303
16	0,00033	0,00047	0,00035	0,04863	0,04690	0,04635	0,03534	0,07780	0,10707



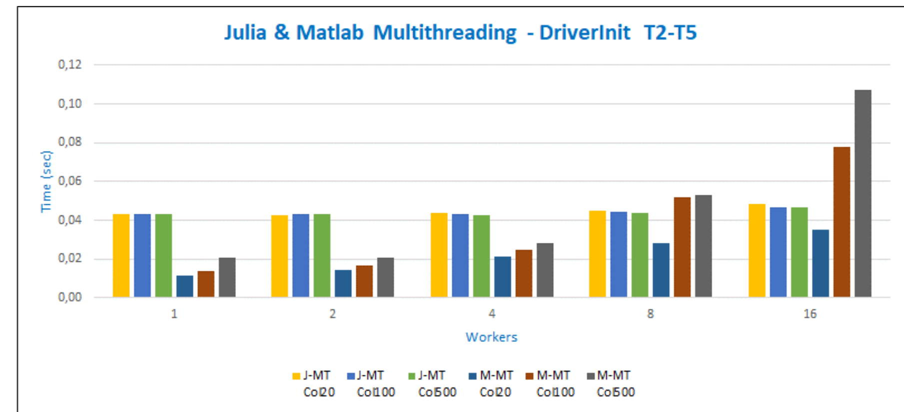


Figura 77 - Microbenchmarks Multithreading DriverInit T2-T5

Tabela 78 - Resultados Microbenchmarks: Sequential T1-T5

Sequential T1-T5								
P-S Col20	P-S Col100	P-S Col500	J-S Col20	J-S Col100	J-S Col500	M-S Col20	M-S Col100	M-S Col500
0,00004	0,00012	0,00063	0,00539	0,00567	0,00568	0,00035	0,00135	0,00598

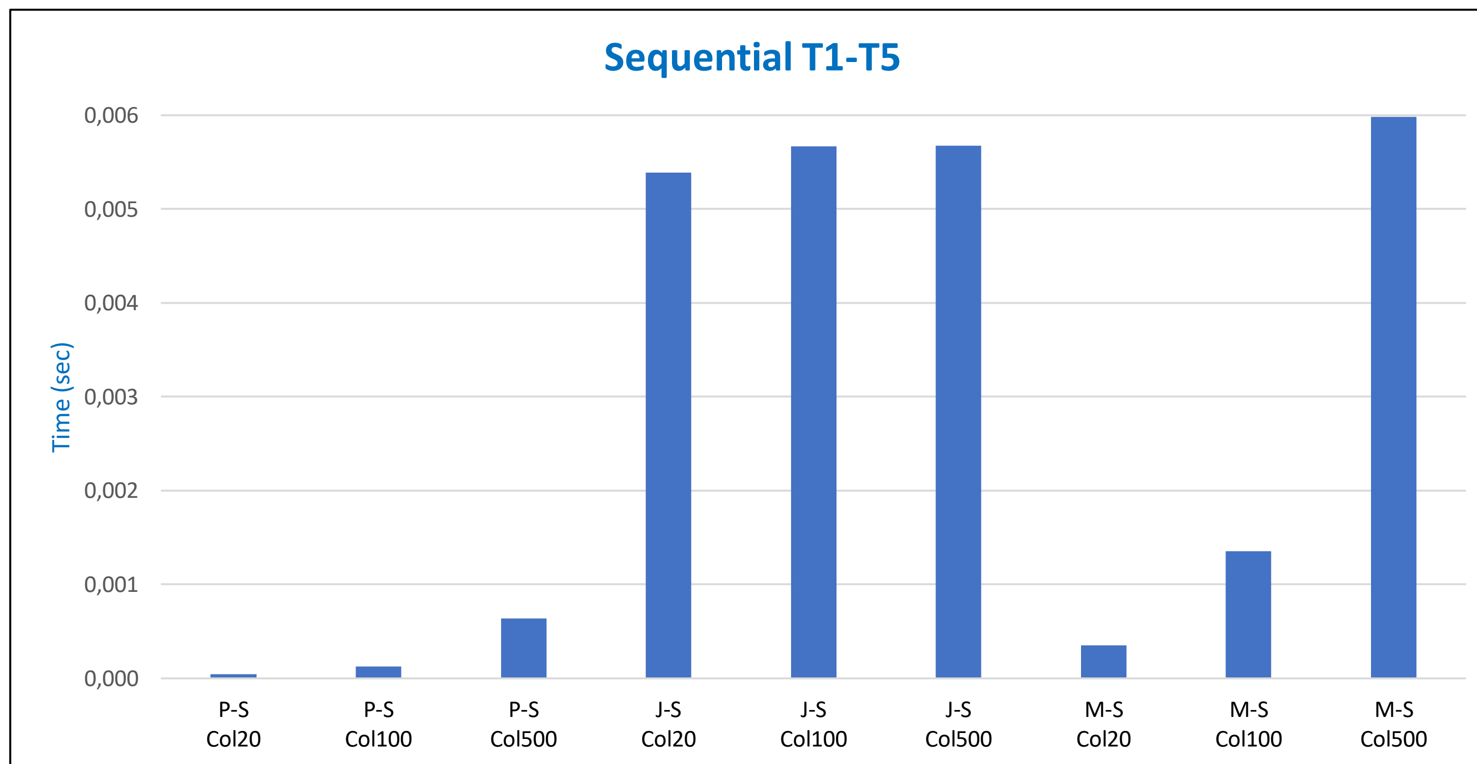


Figura 78 - Microbenchmarks Sequential T1-T5

Tabela 79 - Resultados Microbenchmarks: Sequential T2-T5

Sequential T2-T5								
P-S Col20	P-S Col100	P-S Col500	J-S Col20	J-S Col100	J-S Col500	M-S Col20	M-S Col100	M-S Col500
0,00004	0,00012	0,00064	0,00491	0,00512	0,00513	0,00034	0,00119	0,00571

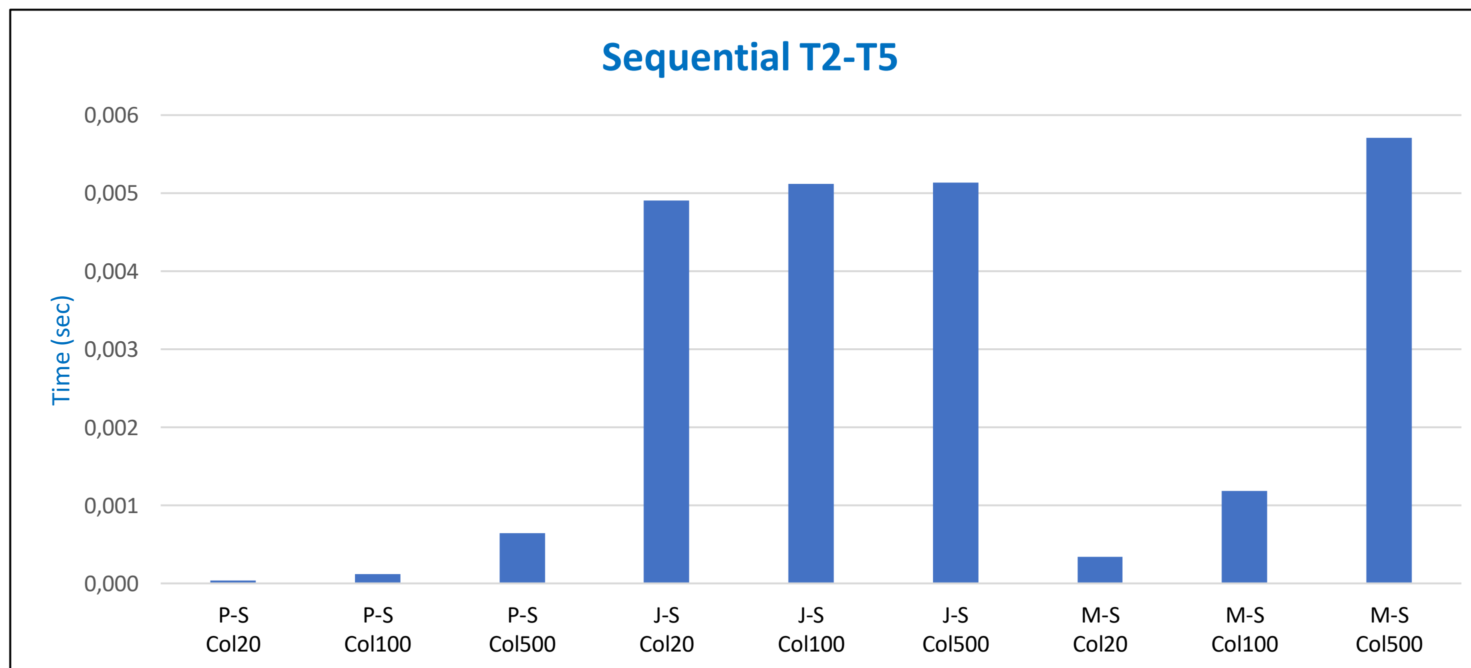


Figura 79 - Microbenchmarks Sequential T2-T5

B.2.3. Comparação DMSInit e DriverInit

Notação utilizada:

Multiprocessing

α-MP-Cβ-λ α = linguagem de programação: P(ython), J(ulia) ou M(atlab).

β = chunksize (apenas aplicável no caso de Python): 1, 4 ou 8.

λ = modo de inicialização: DMSInit ou DriverInit.

Exemplos: P-MP-C8-DriverInit corresponde à linguagem Python, com Multiprocessing, chunksize 8 e DriverInit.

J-MP-DMSInit corresponde à linguagem Julia, com Multiprocessing e DMSInit.

Multithreading

α-MT-λ-Colφ α = linguagem de programação: P(ython) ou M(atlab).

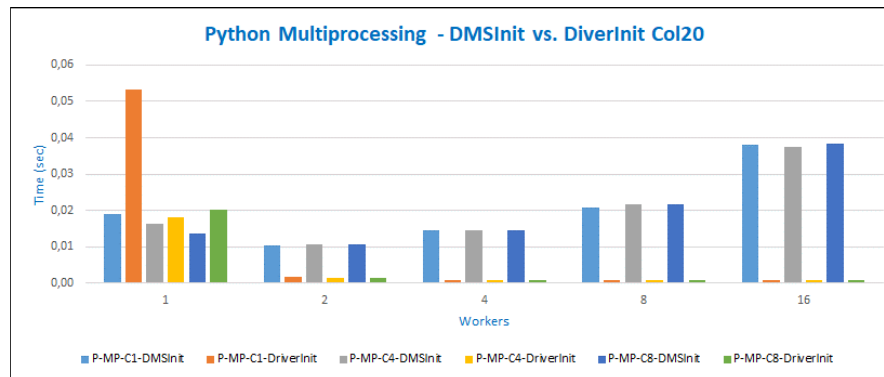
λ = modo de inicialização: DMS para DMSInit ou Driver para DriverInit.

φ = número de colunas da matriz: 20, 100 ou 500.

Exemplo: M-MT-DMS-Col20 corresponde à linguagem Matlab, com Multithreading, DMSInit e matriz com 20 colunas.

Tabela 80 - Resultados Microbenchmarks: Multiprocessing - DMSInit vs. DriverInit - 20 Cols.

Multiprocessing - DMSInit vs. DriverInit - 20 Cols.										
Workers	**P-MP-C1-DMSInit**	**P-MP-C1-DriverInit**	**P-MP-C4-DMSInit**	**P-MP-C4-DriverInit**	**P-MP-C8-DMSInit**	**P-MP-C8-DriverInit**	**J-MP-DMSInit**	**J-MP-DriverInit**	**M-MP-DMSInit**	**M-MP-DriverInit**
1	0,01897	0,05322	0,01646	0,01821	0,01366	0,02032	9,75322	0,00867	19,54388	0,02981
2	0,01047	0,00188	0,01064	0,00147	0,01060	0,00162	11,28463	0,00965	20,03883	0,02873
4	0,01456	0,00084	0,01457	0,00089	0,01449	0,00088	13,74556	0,29615	21,38276	0,03651
8	0,02085	0,00084	0,02172	0,00079	0,02160	0,00077	19,18655	0,01693	24,22639	0,06234
16	0,03812	0,00094	0,03762	0,00088	0,03829	0,00089	30,37159	0,03870	31,35114	0,06615



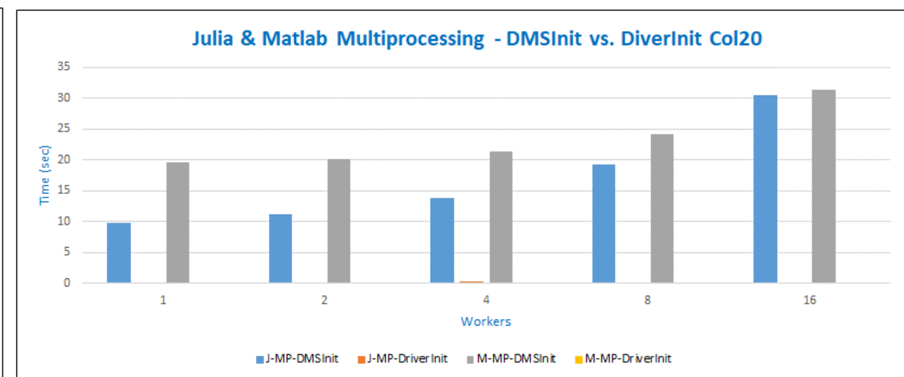
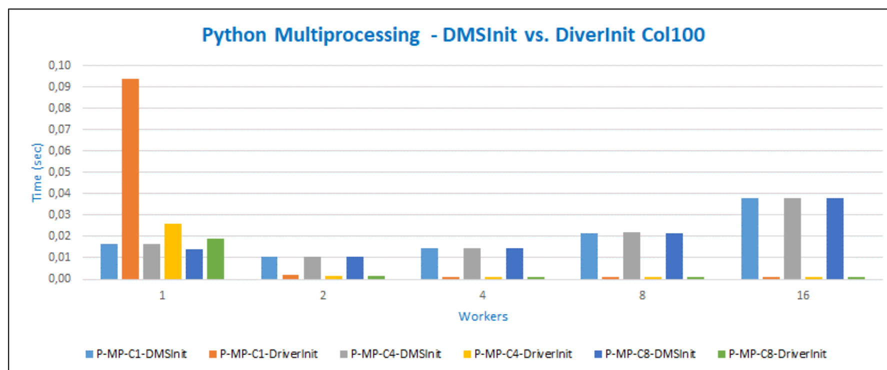


Figura 80 - Microbenchmarks Multiprocessing - DMSInit vs. DriverInit - 20 Cols.

Tabela 81 - Resultados Microbenchmarks: Multiprocessing - DMSInit vs. DriverInit - 100 Cols.

Multiprocessing - DMSInit vs. DriverInit - 100 Cols.										
Workers	P-MP-C1-DMSInit	P-MP-C1-DriverInit	P-MP-C4-DMSInit	P-MP-C4-DriverInit	P-MP-C8-DMSInit	P-MP-C8-DriverInit	J-MP-DMSInit	J-MP-DriverInit	M-MP-DMSInit	M-MP-DriverInit
1	0,01642	0,09388	0,01628	0,02612	0,01398	0,01898	9,04105	0,00865	20,20816	0,03061
2	0,01052	0,00196	0,01052	0,00159	0,01049	0,00161	10,54621	0,00977	20,29731	0,03201
4	0,01464	0,00094	0,01445	0,00098	0,01457	0,00096	13,67444	0,29177	21,48098	0,04130
8	0,02133	0,00092	0,02174	0,00083	0,02123	0,00083	19,32641	0,01589	24,32431	0,07394
16	0,03806	0,00090	0,03778	0,00098	0,03806	0,00099	30,39542	0,03913	32,08776	0,10227



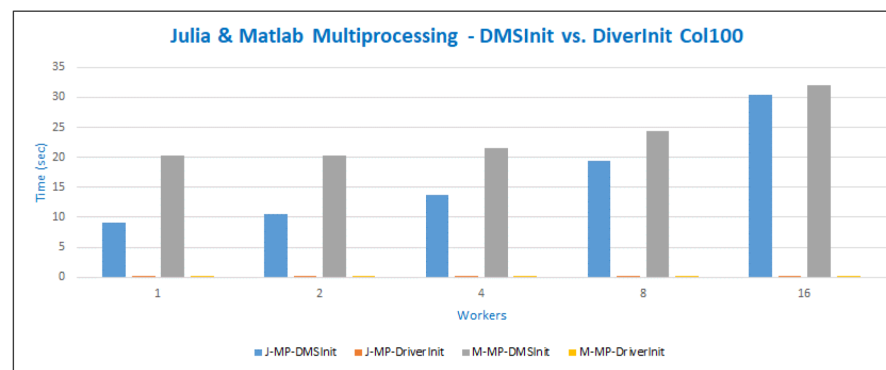
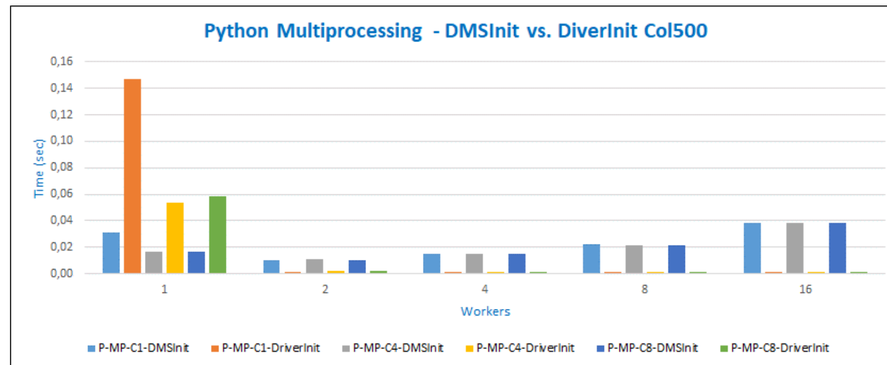


Figura 81 - Microbenchmarks Multiprocessing - DMSInit vs. DriverInit - 100 Cols.

Tabela 82 - Resultados Microbenchmarks: Multiprocessing - DMSInit vs. DriverInit - 500 Cols.

Multiprocessing - DMSInit vs. DriverInit - 500 Cols.										
Workers	**P-MP-C1-DMSInit**	**P-MP-C1-DriverInit**	**P-MP-C4-DMSInit**	**P-MP-C4-DriverInit**	**P-MP-C8-DMSInit**	**P-MP-C8-DriverInit**	**J-MP-DMSInit**	**J-MP-DriverInit**	**M-MP-DMSInit**	**M-MP-DriverInit**
1	0,03153	0,14722	0,01672	0,05336	0,01679	0,05851	9,11204	0,00892	19,73476	0,04229
2	0,01059	0,00145	0,01066	0,00196	0,01052	0,00186	10,65870	0,01114	20,15663	0,03788
4	0,01475	0,00096	0,01477	0,00103	0,01485	0,00100	13,70942	0,29523	21,53393	0,04249
8	0,02190	0,00105	0,02150	0,00102	0,02184	0,00102	19,25031	0,01668	24,33743	0,07836
16	0,03800	0,00104	0,03841	0,00102	0,03877	0,00102	30,42803	0,03883	32,10639	0,12200



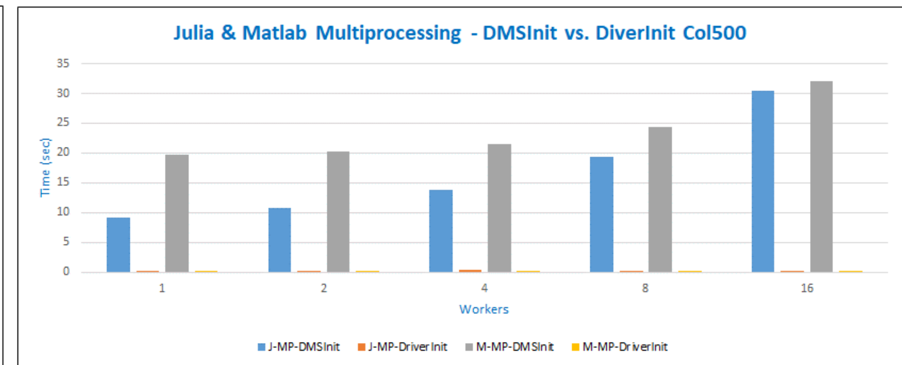
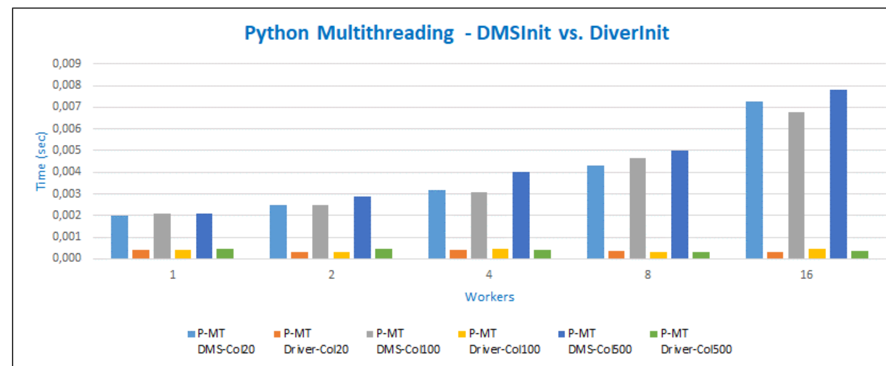


Figura 82 - Microbenchmarks Multiprocessing - DMSInit vs. DriverInit - 500 Cols.

Tabela 83 - Resultados Microbenchmarks: Multithreading - DMSInit vs. DriverInit

Multithreading - DMSInit vs. DriverInit												
Workers	P-MT DMS-Col20	P-MT Driver-Col20	P-MT DMS-Col100	P-MT Driver-Col100	P-MT DMS-Col500	P-MT Driver-Col500	M-MT DMS-Col20	M-MT Driver-Col20	M-MT DMS-Col100	M-MT Driver-Col100	M-MT DMS-Col500	M-MT Driver-Col500
1	0,00201	0,00042	0,00211	0,00044	0,00207	0,00044	1,25833	0,01171	1,26793	0,01365	1,26462	0,02096
2	0,00250	0,00033	0,00247	0,00034	0,00289	0,00046	2,19529	0,01445	2,30932	0,01695	2,24040	0,02062
4	0,00317	0,00044	0,00306	0,00044	0,00400	0,00043	4,13118	0,02121	4,26259	0,02504	4,21321	0,02849
8	0,00433	0,00039	0,00467	0,00033	0,00501	0,00033	8,12246	0,02886	8,31997	0,05278	8,24157	0,05476
16	0,00726	0,00033	0,00679	0,00046	0,00781	0,00035	16,35095	0,04191	16,64718	0,08435	16,57443	0,11418



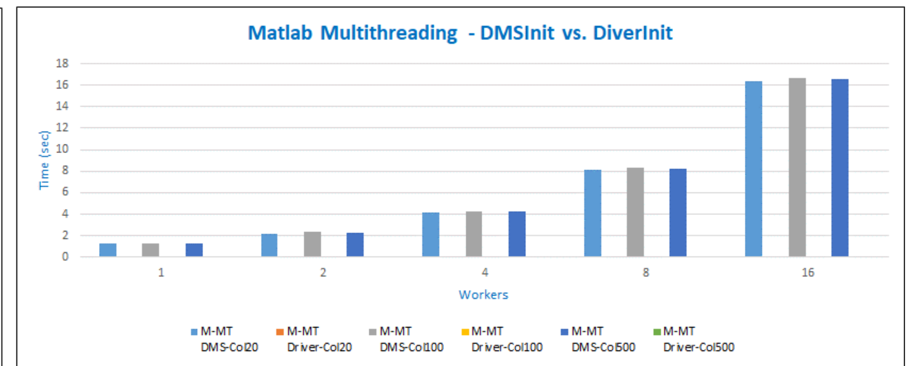


Figura 83 - Microbenchmarks Multithreading - DMSInit vs. DriverInit