Universidade do Minho

Escola de Engenharia

Departamento de Informática

Tema

EXPLORAÇÃO DE PARALELISMO MASSIVO EM ALGORITMOS EVOLUCIONÁRIOS

Identificação

Aluno - Tiago Augusto Simões Martins (PG25306)
E-mail - tiago_martins12@hotmail.com

Orientador - João Luís Ferreira Sobral

CO-Orientador - Miguel Francisco de Almeida Pereira da Rocha

Abril 2016

Resumo

Esta dissertação está centrada na paralelização massiva da biblioteca *Java Evolutionary Computation Library*, *JECoLi*, que se foca no desenvolvimento de meta-heurísticas de otimização(*e.g.* Algoritmos Evolucionários (AEs)) na linguagem *Java*. Os AEs são um paradigma da Computação Evolucionária (CE) utilizados para resolver problemas complexos através de um método iterativo que evolui um conjunto de soluções (população) tendo em conta os princípios da teoria de evolução por seleção natural apresentada por Charles Darwin.

Estes algoritmos estão divididos em duas categorias, AEs não estruturados e AEs estruturados. Os AEs não estruturados são caracterizados por uma população centralizada onde existe apenas um conjunto de soluções ao qual é aplicado o processo evolutivo. Por outro lado, os AEs estruturados contêm várias populações onde os processos evolutivos são conduzidos de forma independente, embora existindo troca de informação.

Os algoritmos de ambas as categorias podem ser paralelizados de diferentes maneiras. Nesta dissertação, foram implementadas quatro versões paralelas da plataforma *JECoLi* de forma o menos invasiva possível, tendo em conta modelos paralelos já formulados: um modelo de paralelismo global; um modelo de ilhas em ambiente de memória partilhada; um modelo de ilhas em ambiente de memória distribuída; e um modelo híbrido.

Estas implementações paralelas foram executadas no cluster *Services and Advanced Research Computing with HTC/HPC clusters (SeARCH)* utilizando o máximo de recursos computacionais possíveis de modo a realizar uma posterior análise dos resultados obtidos. Foram utilizados dois casos de estudo reais para validar as implementações paralelas, um problema de otimização de um bioprocesso de fermentação *fed-batch* e outro de otimização dos pesos de um protocolo de encaminhamento (*OSPF*).

Cada uma das implementações paralelas foi testada nos dois casos de estudo, aplicando o máximo de paralelismo possível tendo em conta as limitações de cada caso de estudo, dos modelos paralelos e dos recursos disponíveis.

Com estes testes concluí-se uma boa escalabilidade destes algoritmos, onde se destacam as implementações relativas ao modelo de ilhas em memória distribuída e ao modelo híbrido. Contudo, algumas configurações que originam maiores ganhos foram descartadas pois não produzem valores de aptidão aceitáveis.

Abstract

This dissertation is centered in the massive parallelization of the *Java Evolutionary Computation Library*, *JECoLi*, which focuses on the development of meta-heuristics optimizations (*e.g.* Evolutionary Algorithms (EAs)) in the Java programming language. The EAs are a paradigm of Evolutionary Computation (EC) used to solve complex problems through an iterative method that evolves a set of solutions (population) taking into account the principles of the theory of evolution by natural selection by Charles Darwin.

These algorithms are divided into two categories, unstructured EAs and structured EAs. The unstructured AEs are characterized by a centralized population where there is only one set of solutions for which the iterative method is applied. On the other hand, structured AEs contain several independent populations where evolutionary processes are applied, although there exchange information.

The algorithms of both categories can be parallelized in different ways. In this dissertation, it was implemented four parallel versions of the platform *JECoLi* in a less invasive way, taking into account parallel models already formulated: a global parallelism model; a island model in shared memory environment; a island model in distributed memory environment; and a hybrid model.

These parallel implementations were executed in the cluster *SeARCH* using the maximum computing resources in order to perform a further analysis of the results obtained. Two case studies were used to validate the parallel implementations, a bioprocess optimization problem of *fed-batch* fermentation and other weights optimization problem of a routing protocol (*OSPF*).

Each of the parallel implementations were tested in the two case studies, applying the maximum parallelism taking into account the limitations of each case studies, parallel models and available resources.

With these tests can be concluded a good scalability of these algorithms, which highlights the implementations on the island model in distributed memory and hybrid model. However, some settings that give greater gains were discarded because they produce no acceptable fitness values.

Índice

Resumo i

Abstract iii

Lista de Figuras ix

Lista de Tabelas xv

1 Introdução 1
- 1.1 Contextualização 1
- 1.2 Objetivos 3
- 1.3 Estrutura do Documento 4

2 Estado da Arte 7
- 2.1 Computação Evolucionária 7
  - 2.1.1 Origem 7
  - 2.1.2 Algoritmos Evolucionários 8
  - 2.1.3 Estruturação da População e Paralelismo nos AEs 13
- 2.2 Plataforma *JECoLi* 20
  - 2.2.1 ParJECoLi 22

3 Desenvolvimento 25
- 3.1 Implementação de meta-heurísticas na plataforma JECoLi 25
- 3.2 Paralelismo Global 27
- 3.3 Modelo de Ilhas 31
- 3.4 Modelo Híbrido 40

4 Resultados 43
- 4.1 Casos de Estudo 43
  - 4.1.1 Fermentação *fed-batch* 43
  - 4.1.2 *Open Shortest Path First* 43
- 4.2 Ambiente Experimental 44
- 4.3 Testes de Desempenho : Fermentação *fed-batch* 45
  - 4.3.1 Análise do Caso de Estudo 45
  - 4.3.2 Paralelismo Global 48
  - 4.3.3 Modelo de Ilhas em Memória Partilhada 50
  - 4.3.4 Modelo de Ilhas em Memória Distribuída 59

4.3.5 Modelo Híbrido 65
4.4 Testes de Desempenho : *OSPF* 68
4.4.1 Análise do Caso de Estudo 68
4.4.2 Paralelismo Global 73
4.4.3 Modelo de Ilhas em Memória Partilhada 75
4.4.4 Modelo de Ilhas em Memória Distribuída 76
4.4.5 Modelo Híbrido 79

5 Conclusão **85**
5.1 Síntese 85
5.2 Limitações 88
5.3 Trabalho Futuro 89

Referências Bibliográficas **91**

Anexos A : Ambiente Experimental Máquinas *NUMA* **97**

Anexos B : Resultados *OSPF* **99**

Lista de Figuras

Figura 1: Passos de Evolução nas Gerações de um AE. 10
Figura 2: Operadores Genéticos: Cruzamento e Mutação. 11
Figura 3: Seleção por roleta. 12
Figura 4: Seleção por torneio. 13
Figura 5: Diversos modelos de populações estruturadas e as suas características. . 14
Figura 6: a) modelo celular - *fine grained* ; b) modelo de ilhas - *coarse grained*. . 16
Figura 7: Estrutura de classes da JECoLi. 21
Figura 8: Arquitetura dos Módulos de Aspeto da *ParJECoLi*. 22
Figura 9: Exemplo de diagrama de sequência da execução de AEs na plataforma *JECoLi*. 26
Figura 10: Exemplo de diagrama de sequência da execução de AEs na plataforma *JECoLi* usando o modelo de Paralelismo Global. 29
Figura 11: Exemplo de diagrama de sequência da execução de AEs na plataforma *JECoLi* usando o modelo de ilhas. 34
Figura 12: Exemplo de diagrama de sequência da execução de AEs na plataforma *JECoLi* usando o modelo híbrido. 41
Figura 13: Resultados dos valores de ganho das execuções do modelo de paralelismo global no caso de estudo fermentação *fed-batch*. 48
Figura 14: Resultados dos valores de aptidão das execuções do modelo de paralelismo global no caso de estudo fermentação *fed-batch*. 49
Figura 15: Resultados da eficiencia do paralelismo das execuções do modelo de paralelismo global no caso de estudo fermentação *fed-batch*. 50
Figura 16: Resultados dos valores de ganho da abordagem de dividir gerações do modelo de ilhas em memória partilhada utilizando diferentes configurações e diferentes números de ilhas no caso de estudo fermentação *fed-batch*. . 52
Figura 17: Resultados dos valores de aptidão da abordagem de dividir gerações do modelo de ilhas em memória partilhada utilizando diferentes configurações e diferentes números de ilhas no caso de estudo fermentação *fed-batch*. 52
Figura 18: Resultados dos valores de ganho da abordagem de dividir indivíduos do modelo de ilhas em memória partilhada utilizando diferentes configurações e diferentes números de ilhas no caso de estudo fermentação *fed-batch*. 53

Figura 19: Resultados dos valores de aptidão da abordagem de dividir indivíduos do modelo de ilhas em memória partilhada utilizando diferentes configurações e diferentes números de ilhas no caso de estudo fermentação *fed-batch*. 53
Figura 20: Resultados dos valores de ganho das execuções ao modelo de ilhas síncrono em memória partilhada utilizando diferentes intervalos de migração e diferentes números de ilhas no caso de estudo fermentação *fed-batch*. . . 55
Figura 21: Resultados dos valores de aptidão das execuções ao modelo de ilhas síncrono em memória partilhada utilizando diferentes intervalos de migração e diferentes números de ilhas no caso de estudo fermentação *fed-batch*. 55
Figura 22: Resultados dos valores de ganho das execuções ao modelo de ilhas assíncrono em memória partilhada utilizando diferentes intervalos de migração e diferentes números de ilhas no caso de estudo fermentação *fed-batch*. . . 56
Figura 23: Resultados dos valores de aptidão das execuções ao modelo de ilhas assíncrono em memória partilhada utilizando diferentes intervalos de migração e diferentes números de ilhas no caso de estudo fermentação *fed-batch*. 56
Figura 24: Resultados dos valores de ganho das execuções ao modelo de ilhas assíncrono em memória partilhada utilizando diferentes intervalos de migração, percentagem de indivíduos a migrar e números de ilhas no caso de estudo fermentação *fed-batch*. 58
Figura 25: Resultados dos valores de aptidão das execuções ao modelo de ilhas assíncrono em memória partilhada utilizando diferentes intervalos de migração, percentagem de indivíduos a migrar e números de ilhas no caso de estudo fermentação *fed-batch*. 58
Figura 26: Resultados dos valores da eficiência do paralelismo para a melhor execução do modelo de ilhas assíncrono em memória partilhada com migração em cada 10 gerações no caso de estudo fermentação *fed-batch*. . 59
Figura 27: Resultados dos valores de ganho das execuções ao modelo de ilhas em memória distribuída com uma máquina 662 utilizando diferentes intervalos de migração no caso de estudo fermentação *fed-batch*. 60
Figura 28: Resultados dos valores de aptidão das execuções ao modelo de ilhas em memória distribuída com uma máquina 662 utilizando diferentes intervalos de migração no caso de estudo fermentação *fed-batch*. 61

Figura 29: Resultados dos valores de ganho das execuções ao modelo de ilhas em memória distribuída síncrono com duas máquinas 662 utilizando diferentes intervalos de migração no caso de estudo fermentação *fed-batch*. . 62
Figura 30: Resultados dos valores de aptidão das execuções ao modelo de ilhas em memória distribuída síncrono com duas máquinas 662 utilizando diferentes intervalos de migração no caso de estudo fermentação *fed-batch*. 63
Figura 31: Resultados dos valores de ganho das execuções ao modelo de ilhas em memória distribuída assíncrono com duas máquinas 662 utilizando diferentes intervalos de migração no caso de estudo fermentação *fed-batch*. . 63
Figura 32: Resultados dos valores de aptidão das execuções ao modelo de ilhas em memória distribuída assíncrono com duas máquinas 662 utilizando diferentes intervalos de migração no caso de estudo fermentação *fed-batch*. 64
Figura 33: Resultados dos valores da eficiência do paralelismo para a melhor execução do modelo de ilhas em memória distribuída com migração síncrona em cada 10 gerações no caso de estudo fermentação *fed-batch*. 65
Figura 34: Resultados dos valores de ganho das execuções ao modelo híbrido com diferentes números de máquinas 641 utilizando uma ilha por máquina com 16 e 32 *threads* do paralelismo global no caso de estudo fermentação *fed-batch*. 66
Figura 35: Resultados dos valores de ganho das execuções ao modelo híbrido com diferentes números de máquinas 641 utilizando uma ilha por máquina com 16 e 32 *threads* do paralelismo global no caso de estudo fermentação *fed-batch*. 67
Figura 36: Resultados dos valores de eficiência do paralelismo das execuções ao modelo híbrido utilizando diferentes números de máquinas 641 utilizando uma ilha por máquina com 16 e 32 *threads* do paralelismo global no caso de estudo fermentação *fed-batch*. 68
Figura 37: Resultados dos valores de ganho das execuções do modelo de paralelismo global no caso de estudo *OSPF* com o tamanho do problema de 30 nós. 73
Figura 38: Resultados dos valores de aptidão das execuções do modelo de paralelismo global no caso de estudo *OSPF* com o tamanho do problema de 30 nós. 73
Figura 39: Resultados da eficiência do paralelismo das execuções do modelo de paralelismo global no caso de estudo *OSPF* com o tamanho do problema de 30 nós. 74

Figura 40: Resultados dos valores de ganho das execuções ao modelo de ilhas síncrono em memória partilhada utilizando diferentes intervalos de migração e diferentes números de ilhas no caso de estudo *OSPF* com o tamanho do problema de 30 nós. 75
Figura 41: Resultados dos valores de aptidão das execuções ao modelo de ilhas síncrono em memória partilhada utilizando diferentes intervalos de migração e diferentes números de ilhas no caso de estudo *OSPF* com o tamanho do problema de 30 nós. 76
Figura 42: Resultados da eficiência do paralelismo das execuções ao modelo de ilhas síncrono em memória partilhada no caso de estudo *OSPF* com o tamanho do problema de 30 nós. 76
Figura 43: Resultados dos valores de ganho das execuções ao modelo de ilhas em memória distribuída com uma máquina 662 utilizando diferentes intervalos de migração no caso de estudo *OSPF* com o tamanho do problema de 30 nós. 77
Figura 44: Resultados dos valores de ganho das execuções ao modelo de ilhas em memória distribuída síncrono com duas máquinas 662 utilizando diferentes intervalos de migração no caso de estudo *OSPF* com o tamanho do problema de 30 nós. 78
Figura 45: Resultados da eficiência do paralelismo das execuções ao modelo de ilhas em memória distribuída síncrono com duas máquinas 662 no caso de estudo *OSPF* com o tamanho do problema de 30 nós. 78
Figura 46: Resultados dos valores de ganho das execuções ao modelo híbrido com diferentes números de máquinas 641 utilizando uma ilha por máquina com 16 e 32 *threads* do paralelismo global no caso de estudo *OSPF* para tamanhos do problema com 30, 50 e 80 nós. 79
Figura 47: Resultados dos valores de aptidão das execuções ao modelo híbrido com diferentes números de máquinas 641 utilizando uma ilha por máquina com 16 e 32 *threads* do paralelismo global no caso de estudo *OSPF* para o tamanho do problema com 30 nós. 80
Figura 48: Resultados dos valores de aptidão das execuções ao modelo híbrido com diferentes números de máquinas 641 utilizando uma ilha por máquina com 16 e 32 *threads* do paralelismo global no caso de estudo *OSPF* para o tamanho do problema com 50 nós. 81

Figura 49: Resultados dos valores de aptidão das execuções ao modelo híbrido com diferentes números de máquinas 641 utilizando uma ilha por máquina com 16 e 32 *threads* do paralelismo global no caso de estudo *OSPF* para o tamanho do problema com 80 nós. 81
Figura 50: Resultados dos valores de eficiência do paralelismo das execuções ao modelo híbrido com diferentes números de máquinas 641 utilizando uma ilha por máquina com 16 e 32 *threads* do paralelismo global no caso de estudo *OSPF* para o tamanho do problema com 80 nós. 82
Figura 51: Resultado do comando lstopo para as máquinas NUMA compute 662. . 97
Figura 52: Resultado do comando lstopo para as máquinas NUMA compute 641. . 98
Figura 53: Resultados dos valores de aptidão das execuções ao modelo de ilhas em memória distribuída com uma máquina 662 utilizando diferentes intervalos de migração no caso de estudo *OSPF* com o tamanho do problema de 30 nós. 99
Figura 54: Resultados dos valores de aptidão das execuções ao modelo de ilhas em memória distribuída síncrono com duas máquinas 662 utilizando diferentes intervalos de migração no caso de estudo *OSPF* com o tamanho do problema de 30 nós. 99

Lista de Tabelas

Tabela 1: Especificação das máquinas do *SeARCH* utilizadas para a realização dos testes. 44
Tabela 2: Mediana de doze execuções da versão sequencial do problema de fermentação relativos aos tempos de execução totais (segundos), tempos de execução na avaliação (segundos) e valores de aptidão. 46
Tabela 3: Resultados das medições de memória aos objetos *FermProcess* e *Island*. 47
Tabela 4: Tabela com os valores de ganho e valores de aptidão resultantes das execuções do modelo de paralelismo global no caso de estudo de fermentação *fed-batch*. 48
Tabela 5: Nome dos ficheiros de configuração, o número de nós e o número de ligações para cada tamanho do problema OSPF abordado. 69
Tabela 6: Mediana de doze execuções da versão sequencial do problema do protocolo *OSPF* com 30 nós numa máquina 662, relativos aos tempos de execução totais (segundos), tempos de execução na avaliação (segundos) e valores de aptidão. 70
Tabela 7: Mediana de doze execuções da versão sequencial do problema do protocolo *OSPF* para diferentes tamanhos do problema numa máquina 641, relativos aos tempos de execução totais (segundos), tempos de execução na avaliação (segundos) e valores de aptidão. 71
Tabela 8: Resultados das medições de memória aos objetos *OSPF* e *Island* para os vários tamanhos do problema. 71
Tabela 9: Vantagens e desvantagens do uso do modelo de paralelismo global e modelo de ilhas. 85

Acrónimos

AEPs Algoritmos Evolucionários Paralelos

AEs Algoritmos Evolucionários

AGs Algoritmos Genéticos

CE Computação Evolucionária

EEs Estratégias Evolucionárias

JECoLi Java Evolutionary Computation Library

MIMD Multiple Instruction Multiple Data

MPI Message Passing Interface

NP Non-deterministic Polynomial time

NUMA Non-Uniform Memory Acess

OSPF Open Shortest Path First

ParJECoLi Parallel Java Evolutionary Computation Library

PE Programação Evolucionária

POA Programação Orientada ao Aspeto

SeARCH Services and Advanced Research Computing with HTC/HPC clusters

SIMD Single Instruction Multiple Data

UMA Uniform Memory Acess

1 Introdução

1.1 Contextualização

A Computação Evolucionária (CE) é uma área da computação que visa a resolução de problemas complexos como problemas de procura, de otimização numérica e combinatória e de aprendizagem máquina. Estes problemas são tipicamente complexos pois os algoritmos requerem elevado poder computacional, podendo ser problemas do tipo *Non-deterministic Polynomial time (NP).*

Um exemplo deste tipo de problemas é o famoso problema do caixeiro-viajante que se resume na procura do melhor caminho que passe por todas as cidades cujas ligações são representadas por um dado grafo visitando apenas uma vez cada cidade, começando e acabando numa determinada cidade. A complexidade deste problema é elevada pois o tempo de resolução deste problema e o número de soluções possíveis aumentam exponencialmente com o aumento do número de cidades, sendo considerado um problema do tipo NP.

A CE consegue resolver problemas complexos propondo vários tipos de algoritmos, dos quais se destacam Algoritmos Evolucionários (AEs). Os AEs incluem vários tipos de algoritmos com técnicas semelhantes, sendo as suas principais diferenças em detalhes de implementação e relativas à natureza dos problemas a serem tratados.

Os AEs são algoritmos iterativos de otimização baseados na teoria de evolução apresentada por Darwin, com o objetivo da procura de melhores soluções possíveis para um dado problema. Todos os algoritmos dentro da ramificação dos AEs usam o mesmo princípio de existência de uma população. Esta população simboliza o conjunto de soluções que irão pertencer ao AE em determinado estágio da sua evolução.

Vários modelos de AEs foram formulados, levando ao aparecimento de AEs espacialmente estruturados que se caracterizam pela existência de várias populações que trocam informação e que conseguem por vezes uma convergência mais rápida na qualidade de soluções, contrariamente aos AEs não estruturados que possuem apenas uma população.

Os Algoritmos Evolucionários Paralelos (AEPs) correspondem a implementações computacionais paralelas dos AEs. Estas implementações podem ser de AEs espacialmente estruturados ou sobre o modelo centralizado de população única (não estruturado).

Uma abordagem paralela popular nos AEs estruturados denominada de modelo de ilhas, considera várias populações (AEs independentes) a computar em simultâneo, trocando soluções entre estas consoante a topologia e os parâmetros de migração estabelecidos. Outra abordagem paralela comum realizada sobre AEs não estruturados tem o nome de modelo de paralelismo global, sendo o paralelismo focado ao nível da avaliação das soluções. Adicionalmente, é possível implementar modelos híbridos através dos modelos mencionados, aplicando

o modelo de ilhas de forma a dividir as populações e em cada uma destas populações aplicar novamente um dos dois modelos (modelo de ilhas ou modelo de paralelismo global).

Com o aparecimento das tecnologias *multicore*, a sua constante evolução e crescimento do número de processadores por máquina e com o paralelismo ao nível da *thread*/processo, a evolução do desempenho dos processadores e o baixo custo de instalação de *clusters* computacionais levou a um fácil acesso a grandes capacidades de processamento. Nestas plataformas, o paralelismo pode ser em ambientes de memória partilhada ou de memória distribuída.

As diferenças destes ambientes reside na forma como os processadores acedem à memória. A memória partilhada pode ser utilizada numa máquina computacional tanto em arquiteturas *Uniform Memory Acess (UMA)*, como *Non-Uniform Memory Acess (NUMA)*. As máquinas de arquitetura *UMA* são as máquinas mais comuns encontradas no mercado pelo que podem possuir um ou mais processadores ligados a uma memória principal. As máquinas com arquitetura *NUMA* por outro lado podem conter dois ou mais *sockets* com um número considerável de processadores dentro da mesma máquina onde cada *socket* irá conter processadores ligados a um banco de memória privado.

Por outro lado, o ambiente de memória distribuída caracteriza-se por várias máquinas, cada uma com a seu tipo de arquitetura. Para executar programas paralelos neste tipo de ambiente de memória distribuída são utilizadas geralmente bibliotecas que implementam protocolos de comunicação entre as diversas máquinas como é o caso da biblioteca *Message Passing Interface (MPI)*.

A biblioteca MPI foi desenvolvida com o objetivo de fornecer uma definição para processos paralelos em memória distribuída sendo utilizada geralmente em *clusters* computacionais. A comunicação entre processos é realizada através de mensagens de grupo ou mensagens ponto-a-ponto, implementando também outras funcionalidades (*e.g.* barreiras).

A *Java Evolutionary Computation Library (JECoLi)*, é uma *framework* que implementa algoritmos de otimização de meta-heurísticas (AEs), usando Java como linguagem de programação. A *JECoLi* visa características como flexibilidade, extensibilidade, confiabilidade e adaptabilidade. Uma implementação paralela desta *framework*, *Parallel Java Evolutionary Computation Library (ParJECoLi)*, foi desenvolvida anteriormente no âmbito de uma dissertação de mestrado.

1.2 Objetivos

Os objetivos estabelecidos para esta dissertação terão como base o trabalho realizado na versão paralela da *JECoLi*, *ParJECoLi*.

O trabalho anteriormente realizado focou-se na exploração da Programação Orientada ao Aspeto (POA) para implementar paralelismo na plataforma base de modo a modularizar diferentes tipos de paralelismo. Foram implementados quatro modelos paralelos: um modelo de paralelismo global; um modelo de ilhas em memória partilhada; um modelo de ilhas em memória distribuída; e um modelo híbrido. Os testes realizados neste trabalho utilizaram poucos recursos computacionais com arquitetura UMA, não existindo testes sobre o modelo híbrido.

Com o aumento do número de *cores* nas máquinas *multicore* e tendo em conta a arquitetura NUMA, vários pontos deste trabalho não favorecem uma boa análise destas implementações paralelas na plataforma *JECoLi* e no seu estudo de escalabilidade face ao paralelismo. As plataformas computacionais atuais, mais concretamente máquinas computacionais de arquitetura NUMA, contêm um elevado poder de processamento através de uma quantidade considerável de unidades de processamento utilizando uma hierarquia de memória complexa, podendo existir diferentes números de *sockets* interligados cada uma com um banco de memória privado.

Com esta tese pretende-se desenvolver uma versão paralela da *JECoLi* mais adequada às plataformas computacionais atuais, nomeadamente:

- Implementar os quatro modelos mencionados sem a utilização de aspetos na *JECoLi*, dando mais ênfase ao modelo de ilhas e ao modelo híbrido;
- Estes modelos deverão ser implementados de forma menos invasiva possível ao código base da plataforma *JECoLi*;
- Pretende-se realizar testes a estes modelos com o máximo de recursos computacionais possíveis;
- Os testes realizados deverão ter em consideração a obtenção dos melhores resultados possíveis tanto a nível de desempenho(tempo de execução) como na qualidade das soluções nos casos de estudo utilizados: um problema de otimização de um bioprocesso de fermentação *fed-batch* e um problema de otimização de um protocolo de encaminhamento em redes.
- Por fim, os resultados obtidos deverão ser analisados de modo a realizar uma comparação entre os diversos modelos.

1.3 Estrutura do Documento

Este documento está estruturado em cinco capítulos.

O primeiro capítulo contém contextualização do tema da dissertação, os objetivos estabelecidos e a estrutura do documento.

No segundo capítulo será apresentado o estado da arte, incluindo a CE e os AEs, os tipos de AEs estruturados e não estruturados, os AEPs existentes, as características e estrutura da plataforma *JECoLi* e uma visão global da versão paralela realizada anteriormente (*ParJECoLi*).

O terceiro capítulo apresenta o desenvolvimento realizado ao longo desta dissertação, mais concretamente os modelos paralelos implementados sobre a *JECoLi*: modelo de paralelismo global, modelo de ilhas e modelo híbrido.

No quarto capítulo serão apresentados os casos de estudo utilizados para testar os modelos paralelos implementados com problemas reais, o ambiente experimental e os testes de desempenho realizados.

O último capítulo contém uma síntese do trabalho realizado assim como as conclusões recolhidas ao longo do percurso de desenvolvimento e testes, as limitações encontradas e o trabalho que poderá ser feito no futuro.

2 Estado da Arte

2.1 Computação Evolucionária

A Computação Evolucionária é uma área de investigação ainda em expansão que se tem revelado útil em diversos campos científicos nas últimas décadas. Os Algoritmos Evolucionários (AEs) são métodos de procura estocásticos inspirados na teoria da evolução natural de Charles Darwin. Estes são capazes de fornecer soluções de alta qualidade (soluções geralmente muito próximas do ótimo global) para problemas de grande complexidade (*e.g.* NP-completos), considerados intratáveis por métodos tradicionais de otimização.

2.1.1 Origem

Charles Darwin desenvolveu a teoria de seleção natural e convenceu a comunidade científica relativamente à evolução como um facto no final do século XIX[1]. Esta teoria pode ser comparada a um método de otimização, visto que se trata de um processo biológico que determina as espécies de seres vivos com melhor adaptação num dado meio ambiente.

Darwin realizou um estudo comparativo em diferentes regiões entre espécies de animais vivos bem adaptados e animais em vias de extinção. Através deste estudo conclui que seres vivos com características biológicas diferentes estão sujeitos a uma concorrência constante entre si num meio com recursos naturais limitados e que somente os mais adaptados às condições ambientais poderiam sobreviver.

Desta forma, afirma também que estes indivíduos mais fortes/adaptados possuem melhores características. Numa determinada população de indivíduos as suas características diferem, sendo muita da variação nas populações hereditária. Assim, as características biológicas passam por um processo dinâmico em que os indivíduos com melhores características tendem a aumentar a sua população, permanecendo as melhores características. A este processo dá-se o nome de seleção natural.

Nos anos 1950, foram propostas as primeiras ideias relativas aos AEs que se baseavam na seleção natural de Darwin. Até o fim da década de 1970 várias variantes e princípios dos AEs foram propostos por investigadores como Rechenberg, Holland, Schwefel e Fogel, realizando diversas publicações. Principalmente a partir destes foram formados diferentes tipos de AEs, dos quais se destacam:

Programação Evolucionária (PE) por Lawrence Fogel inicialmente em 1960[2][3];

Algoritmos Genéticos (AGs) por John Holland em meados de 1970, tornando-se a vertente mais popular entre os AEs[4][5];

Estratégias Evolucionárias (EEs) por Ingo Rechenberg[6][7] e Hans Schwefel[8][9][10] principalmente na década de 1970 com base na técnica *Hill Climbing*.

Outros investigadores como Box[11], Friedman[12], Bledsoe[13], Bremermann[14] e Reed et al[15] contribuíram significativamente nesta área nas décadas de 1950 e 1960, inspirados na CE desenvolvendo algoritmos independentes para problemas específicos (*e.g.* otimização de funções).

Devido à inexistência de plataformas com poder computacional para a implementação destes algoritmos, todo o trabalho referido (até ao fim da década de 1970) foi principalmente teórico. Por sua vez, este facto levou ao desconhecimento dos AEs na comunidade científica. A partir desta data, com o aumento do poder computacional verifica-se um crescimento no uso destes algoritmos e o desenvolvimento de código implementando estes métodos.

2.1.2 Algoritmos Evolucionários

Os AEs são algoritmos iterativos de otimização que se focam na procura de uma solução ótima para um dado problema. Os AEs podem ser modelados facilmente o que lhes permite adaptarem-se a diversos tipos de problema, independentemente das características da função de avaliação de soluções. A CE baseia-se nos princípios e conceitos da seleção natural de forma a introduzir analogias no contexto dos seus algoritmos.

O principal objetivo dos AEs passa por aplicar a ideia de evolução a uma dada população (conjunto de soluções que representam uma amostra do domínio de soluções possíveis para o problema em causa) que contém possíveis indivíduos (soluções) para um problema, através de operadores genéticos (*e.g.* cruzamento e mutação).

Cada indivíduo possui uma codificação ao qual chamamos de cromossoma(estrutura de dados) que irá representar uma dada solução para o problema. No caso dos AGs, estes cromossomas são codificados geralmente em vetores de valores binários onde cada bit corresponde a um gene. Através dos operadores genéticos, os cromossomas podem evoluir (alterar-se) ao longo das sucessivas gerações (iterações).

```
t:=0
inicializar[P(t)];
avaliar[P(t)];
ENQUANTO nao condicao_paragem EXECUTAR
    s := selecao[P(t)]
    P'(t) := aplicar_operadores[s];
    avaliar[P'(t)];
    P(t+1) := selecao[P'(t),P(t)];
    t := t + 1;
FIM ENQUANTO
```

Listagem 1: Pseudocódigo de um AE.

O pseudocódigo, na *Listagem 1*, representa de forma sintetizada os passos principais de um AE. O algoritmo começa com uma população P que é inicializada/populada com indivíduos codificados aleatoriamente, sendo a variável t o número da geração inicializada com o valor de zero. Em seguida, todos os indivíduos são avaliados através da função de avaliação sendo-lhes atribuído um valor numérico, o seu valor de aptidão, antes de entrar no ciclo principal do algoritmo. Em seguida, começa a execução do ciclo que é executado até o caso de paragem (*condicao_paragem*) ser atingido. Esta condição de paragem difere com o problema em questão e é atingida normalmente com base numa análise dos valores de aptidão dos indivíduos da população, ou num número máximo de iterações pré-definido.

As rotinas presentes no ciclo são as seguintes: (i) são selecionados da população atual (P) os indivíduos que vão sofrer operadores genéticos; (ii) são aplicados operadores genéticos sobre os indivíduos selecionados (s), criando uma nova população (P'); (iii) os indivíduos da nova população, P', são avaliados, sendo-lhes atribuído um valor de aptidão; (iv) é realizada uma seleção dos indivíduos das duas populações, P e P', para formarem a população da próxima geração (t+1).

Codificação

A codificação nos AEs representa a forma como as soluções são estruturadas e representadas no sistema. Cada solução é definida pelo seu cromossoma, que é constituído por um conjunto de parâmetros denominados de genes. Os cromossomas podem ser representados computacionalmente em diversas estruturas de dados como: listas, vetores, árvores, matrizes, tabelas de *hash*. Nestas estruturas, os genes podem ser de diversos tipos de dados

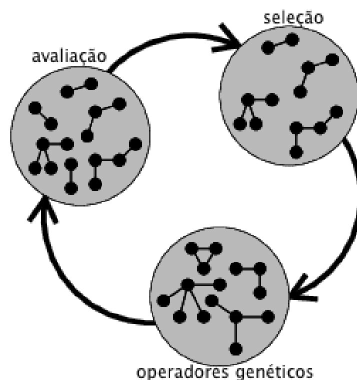

Figura 1: Passos de Evolução nas Gerações de um AE.

como: binários, inteiros, vírgula flutuante, carateres, booleanos, *strings*. A codificação de uma solução é inerente ao problema e é, de facto, um fator importante na eficiência do AE.

Depois de definida a codificação dos indivíduos da população inicial num AE, desenvolve-se o processo de evolução da população através dos passos realizados em cada geração como mostra a *Figura 1*: operadores genéticos, avaliação e seleção.

Operadores Genéticos

Existe normalmente apenas uma combinação possível de genes para representar um determinado cromossoma e consequentemente uma solução. Desta forma, a alteração de um gene ou a alteração de uma sequência de genes num dado cromossoma irá representar outra solução para o problema. Esta nova solução poderá aproximar-se ou não da solução ótima do problema em questão. Com isto em mente, os AEs utilizam técnicas que consistem na alteração de cromossomas, com o nome de operadores genéticos, para permitir a evolução das soluções numa população.

O objetivo principal dos operadores genéticos é criar diversidade na população atual num AE e dependem sempre da codificação envolvida. Existem muitas técnicas e variações de operadores genéticos, onde os mais usuais são o cruzamento e a mutação.

Os operadores de cruzamento envolvem dois ou mais indivíduos da população atual (pais) e recorrem à recombinação/cruzamento de genes entre eles para criar novos indivíduos da próxima geração (descendentes/filhos). Existem diversas maneiras de implementar esta técnica, pois existem muitas formas de recombinar dois ou mais cromossomas. Alguns exemplos destas técnicas são o cruzamento de ponto único, cruzamento de dois pontos e o cruzamento uniforme[16].

Na *Figura 2* pode-se observar o cruzamento entre os indivíduos A e B e a mutação do Indivíduo B, sendo os seus cromossomas do tipo vetor e os genes de vírgula flutuante. No cruzamento entre os dois indivíduos são gerados os seus descendentes C e D que possuem genes diferentes de ambos os pais. Com isto, esta técnica permite que os genes com melhores propriedades permaneçam após a seleção, tentando melhorar o conjunto das soluções presentes na geração em causa. Algumas das desvantagens do uso deste operador genético são o facto de não permitir muita diversidade ao longo das gerações, permanecendo os mesmos genes dos progenitores, e também por poder levar a soluções em ótimos locais não desejados.

Por outro lado, o operador de mutação resolve estes problemas com a alteração de valores nos genes dos indivíduos de forma aleatória (*Figura 2*). Desta forma, este operador garante a diversidade genética numa população, permitindo a procura de novas soluções e não deixando o AE ficar preso em ótimos locais. Assim, o operador de cruzamento juntamente com o operador de mutação permitem a procura de uma solução ótima numa dada população. Na maior parte dos casos, a mutação é realizada sobre uma pequena percentagem da população.

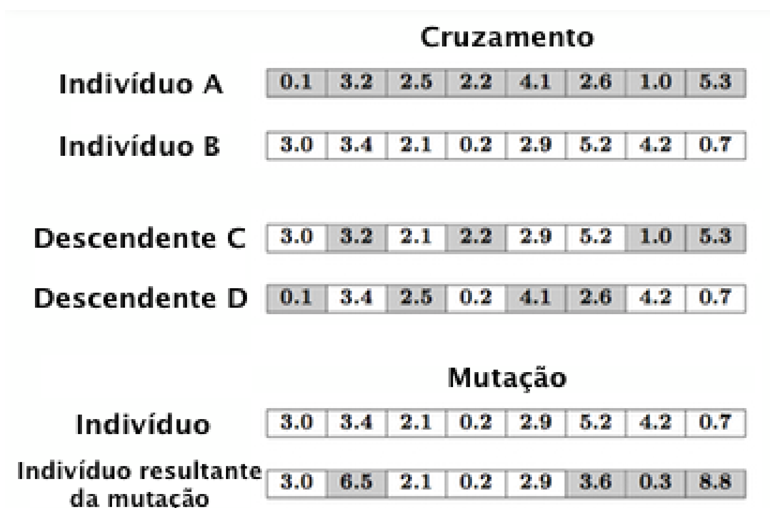

Figura 2: Operadores Genéticos: Cruzamento e Mutação.

Avaliação

Em todas as gerações, cada indivíduo da população atual (incluindo os novos indivíduos criados a partir de operadores genéticos) é avaliado por uma função de avaliação, a qual lhe atribui um valor de aptidão (valor numérico) consoante a qualidade da solução. Esta função de avaliação ou de aptidão é uma função objetivo que verifica a qualidade da solução para

um dado problema, dependendo desta forma sempre do problema em questão. Em muitos casos, esta avaliação corresponde à rotina mais pesada a nível de computação num AE.

Seleção

Seguidamente, é realizado um método de seleção onde, tal como na seleção natural, os indivíduos com melhores valores de aptidão têm maior probabilidade de permanecerem na próxima geração. Depois de avaliados, os indivíduos são organizados por ordem de aptidão, através dos dados recolhidos da função de avaliação. A seleção é realizada em duas situações, sendo a primeira na seleção dos progenitores aos quais são aplicados operadores genéticos e na segunda na seleção dos indivíduos que irão permanecer na próxima geração. O objetivo da seleção é então filtrar indivíduos de um dado conjunto, onde se espera que permaneçam os indivíduos mais aptos para o problema, com maior probabilidade de atingir a solução ótima.

Existem três técnicas principais de seleção: seleção por roleta[17], seleção por torneio e o elitismo. A primeira técnica consiste num mecanismo semelhante ao jogo da roleta. São escolhidos indivíduos ao acaso para pertencerem à próxima população, onde a probabilidade de serem escolhidos é diretamente proporcional ao seu valor de aptidão. Desta forma, os indivíduos com pior aptidão têm menor probabilidade de serem escolhidos do que os indivíduos com maior aptidão (*Figura 3*).

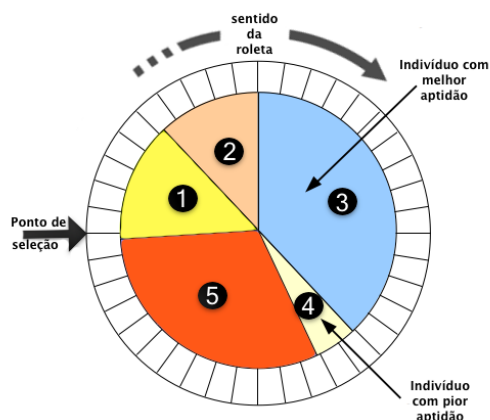

Figura 3: Seleção por roleta.

Na seleção por torneio, os indivíduos da população atual são divididos em grupos (geralmente de dois ou três indivíduos) aleatoriamente que irão "competir" entre si, onde apenas os mais aptos de cada grupo serão selecionados para a população futura (*Figura 4*).

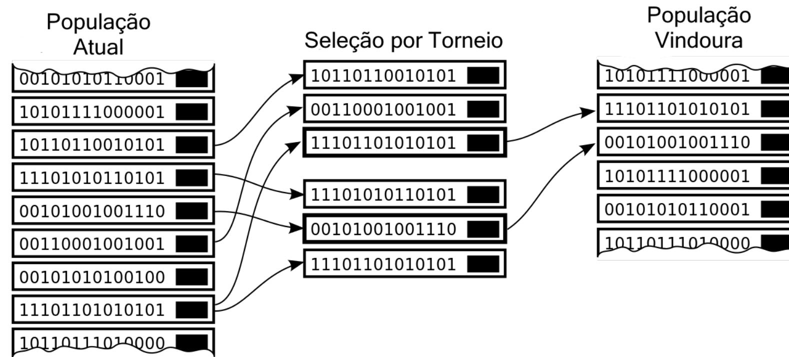

Figura 4: Seleção por torneio.

O elitismo consiste em preservar o grupo de indivíduos com mais aptidão na população atual, de modo a que estes não passem pelo processo de seleção. Como existe a probabilidade de excluir indivíduos com um elevado valor de aptidão num processo de seleção, esta técnica é utilizada para preservar os mais aptos. O número de indivíduos que passam para a geração seguinte corresponde ao grau de elitismo atribuído ao AE.

Ramificação dos AEs

Apenas no ano de 1990, foi unificado o tema da CE, como classe superior das três variações mencionadas anteriormente (PEs, AGs e EEs) assim como as outras técnicas baseadas na perceção evolucionária na resolução de problemas. Estas sub-áreas ou variações dos AEs são bastantes semelhantes na maneira como abordam os problemas na forma evolutiva, onde as suas maiores diferenças encontram-se na forma de codificam os indivíduos, na sua avaliação e seleção para as futuras gerações e os operadores genéticos usados. Ao longo desta dissertação, serão apenas considerados e utilizados os AEs, descartando as outras ramificações existentes.

2.1.3 Estruturação da População e Paralelismo nos AEs

Na subsecção anterior foram abordados os conceitos básicos inerentes a um AE. Nesta subsecção serão descritos os principais tipos de AEs (não estruturados e estruturados) e os seus modelos, assim como uma abordagem à aplicação de paralelismo nestes modelos.

Na CE a estrutura das populações mais usual é denominada de centralizada, representando o modelo padrão para as implementações sequenciais dos AEs. Neste tipo de modelo não estruturado, existe uma única população de indivíduos onde a seleção é realizada tendo em conta todos os indivíduos da população e qualquer indivíduo pode "acasalar" (operador

de cruzamento) com qualquer outro da mesma geração. Este modelo é referido como centralizado ou *panmictic* visto que todas as rotinas de evolução efetuam-se sobre a mesma população.

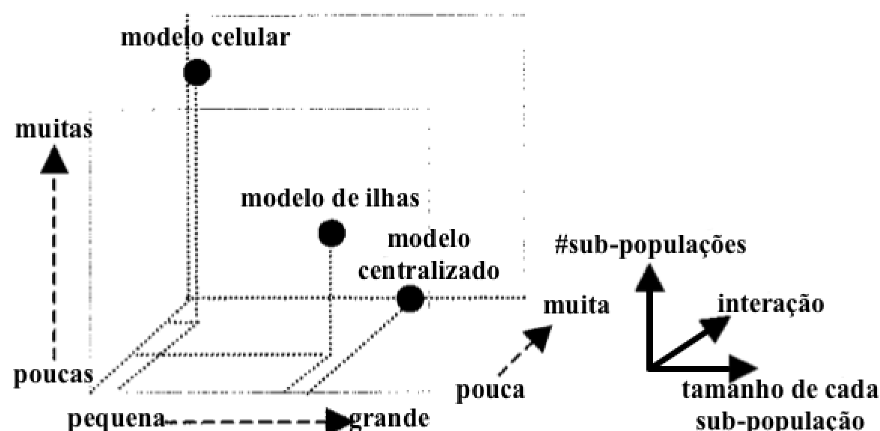

Figura 5: Diversos modelos de populações estruturadas e as suas características.

Por outro lado, os AEs estruturados são modelos descentralizados onde num dado problema a população é dividida em sub-populações isoladas sendo a interação entre indivíduos mais restrita. Estes modelos são de certa forma comparáveis ao mundo atual em que vivemos, onde as sub-populações de indivíduos podem ser vistas como cidades/países que vão comunicando periodicamente entre si em prol da evolução como um todo. Os principais modelos dentro dos AEs estruturados são o modelo de ilhas e o modelo celular. A *Figura 5* representa os três modelos mencionados num gráfico em três dimensões baseado no número de sub-populações, número de indivíduos por sub-população e a respetiva interação entre os indivíduos.

Quanto ao paralelismo, existem três modelos principais de AEPs: paralelismo global (modelo centralizado), celular e ilhas. É importante salientar que qualquer um destes tipos de AEs (estruturados e não estruturados) pode ser implementado sequencialmente, embora no modelo de ilhas e modelo celular seja frequente a utilização de paralelismo devido à sua estruturação.

Ascensão dos Algoritmos Evolucionários Paralelos

Os AEs são algoritmos que vão otimizando um conjunto de soluções ao longo de sucessivas iterações, onde se espera que quanto maior o número iterações melhor será a aproximação à solução ótima. Com isto, é-lhe associado um custo computacional elevado, necessitando de grandes recursos computacionais para a obtenção de boas soluções, principalmente quando se trata de um AE com indivíduos de estruturas complexas e/ou em grandes populações.

Em 2002, foi realizado um trabalho por Enrique Alba e Marco Tomassini[18] com o objetivo de centralizar a informação durante vários anos espalhada em inúmeros documentos relativa aos AEPs no campo dos AEs.

Desde o início da formulação dos AE, vários autores como Holland[4] e Bossert[19], introduziram algumas noções sobre a introdução do paralelismo e a possibilidade de estruturação dos AEs, entre a década de 1960 e 1970. Devido ao difícil acesso a sistemas de computação eficientes, estas ideias permaneceram numa vertente mais teórica durante estas datas.

Mais tarde, na década de 1980, vários estudos e contribuições de inúmeros autores fizeram com que aparecessem as primeiras versões paralelas com AEs estruturados. Em 1987, Tanese[20] e Cohoon[21] implementaram uma arquitetura paralela em hipercubo, onde as várias sub-populações eram colocadas em nós do hipercubo. Outros marcos importantes foram realizados nas diferentes ramificações dos AEs, como a implementação do primeiro modelo distribuído nas EEs por Duncan[22] em 1991 e em 1989 um modelo de multi-populações em GAs dinâmicos por Pettey e Leuze[23].

Uma das grandes vantagens no desenvolvimento de AEPs é o facto de grande parte das operações dos AEs poderem ser implementadas em paralelo. Para além disso, os AEPs têm a vantagem de possuírem um desempenho numérico a nível de valores de aptidão das soluções superior aos AEs sequenciais e que podem resultar assim em algoritmos mais rápidos na convergência para o resultado ótimo do problema[24][25]. As implementações onde estes factos se verificam com clareza são na utilização de AEs estruturados como o modelo de ilhas e o modelo celular.

Curiosamente e acrescentando, mesmo usando os modelos estruturados sem o uso de paralelismo, existem autores que conseguiram melhores resultados comparativamente aos modelos tradicionais como o modelo centralizado, onde se verificaram convergências mais rápidas da solução próxima da ótima[26]. Com a evolução das arquiteturas computacionais o uso de AEPs aumentou, tendo hoje como modelo de referência o modelo de ilhas.

Paralelismo Global

A função de avaliação aplicada a todos os indivíduos em cada geração é frequentemente a operação mais pesada na computação de um AE. A partir desta observação, nasce o modelo de paralelismo global[27][28][29] que consiste num modelo centralizado onde este processo de avaliação de indivíduos é realizado em paralelo. Devido a este facto, este modelo por vezes é referido como modelo "master-slave" onde, no decorrer da avaliação de indivíduos, existe um processador principal, denominado de "master", que distribui tarefas de avaliação pelos restantes processadores, "slaves". No fim de cada tarefa (avaliação de um conjunto de indivíduos) por parte dos processadores secundários, "slaves", os seus valores numéricos resultantes da avaliação são retornados ao processador principal para proceder às restantes operações de cada geração.

Este processo de avaliação normalmente não contém dependências na sua execução, pois a avaliação do indivíduo é independente do resto da população. Assim, esta rotina é paralelizada, enquanto os restantes procedimentos num AE são processados sequencialmente, permanecendo assim as características do AE inalteradas. A paralelização desta rotina geralmente não precisa de balanceamento de carga pelos processadores, pelo facto de ter o mesmo número de operações para qualquer avaliação de indivíduo.

Algoritmos Evolucionários Estruturados - Celular e Distribuído

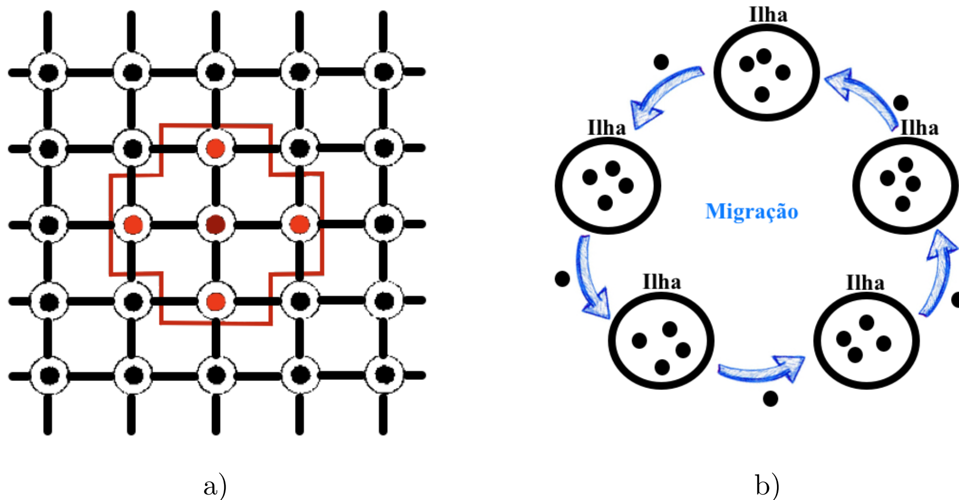

Figura 6: a) modelo celular - *fine grained* ; b) modelo de ilhas - *coarse grained.*

Os AEs estruturados mais conhecidos são o modelo celular e o modelo de ilhas. No caso do modelo celular (*Figura 6, a)*), também conhecido como AE de difusão ou grão fino (*fine grained*)[30], a população original é divida em pequenas sub-populações, geralmente de um ou dois indivíduos, dispostas numa grelha unidimensional ou bidimensional. Este algoritmo paralelo era muito usado nos supercomputadores *Single Instruction Multiple Data (SIMD)* típicos das décadas de 1970 e 1980. Estes supercomputadores vetoriais contêm inúmeras unidades de processamento que executam a mesma instrução simultaneamente. A cada unidade de processamento é atribuída uma sub-população do modelo celular.

Por este facto, a população inicial é dividida pelas unidades de processamento onde cada indivíduo apenas pode interagir com os seus vizinhos, ou seja, só poderá "acasalar" e competir com um grupo de indivíduos restrito. As vantagens desta implementação passam pela sua topologia e pela sua seleção descentralizada. É caracterizada também pela otimização local numa diversidade espalhada pela grelha, onde a existência de pequenas vizinhanças sobrepostas ajuda a explorar o espaço de procura de soluções. Este modelo promove uma progressiva difusão de boas soluções para o problema em causa ao longo da grelha.

Este modelo também pode ser implementado em arquiteturas *Multiple Instruction Multiple Data (MIMD)* com memória distribuída, embora não ofereça as mesmas vantagens devido à sua diferente topologia/estrutura. O declínio das máquinas SIMD e, consequentemente, o apogeu das máquinas MIMD (tipos de máquinas mais comuns na atualidade) juntamente com a tecnologia *multi-core* levou ao desuso deste modelo. Outro fator que contribuiu para este desfecho foi a flexibilidade do modelo de ilhas comparativamente a este modelo, bem como a facilidade de implementação.

O modelo de ilhas ou AE distribuído[31][32][33] caracteriza-se pela distribuição da população total em várias sub-populações (ilhas), sendo periodicamente migrados indivíduos de umas sob-populações para as outras (*Figura 6, b)*). O processo de migração ocorre normalmente após a avaliação dos indivíduos. Devido a estas sub-populações serem relativamente grandes, comparando ao modelo celular, estas implementações são conhecidas também como *coarse grained* ou grão grosso. Este modelo foi idealizado primeiramente por Wrigth[34] em 1943, onde matematicamente concluiu que o particionamento da população inicial tinha melhores probabilidades para uma melhor exploração da solução próxima da ótima através da diversidade entre as sub-populações.

Comparando ao modelo centralizado, onde qualquer indivíduo pode interagir com qualquer outro, este modelo difere ao restringir essa interação apenas com indivíduos da mesma ilha/sub-população. Desta forma, previne a convergência para soluções prematuras no caso de ótimos locais, pois cada ilha irá abordar uma região diferente do espectro total de soluções do problema.

Esta granularidade grossa é suportada por máquinas com arquiteturas MIMD populares na atualidade, quando se usa o modelo de ilhas paralelo. Estas máquinas possuem diversos processadores que executam instruções independentes assincronamente. Desta forma, cada processador pode executar instruções diferentes relativamente aos restantes processadores. O modelo de ilhas paralelo pode utilizar estas máquinas em ambientes de memória partilhada e memória distribuída.

Para este modelo em específico, normalmente é atribuído a cada processador uma única ilha em que o peso associado à comunicação entre os processos paralelos recai na migração de indivíduos. Um modelo de ilhas contém diversas configurações possíveis, sendo assim, flexível e ao mesmo tempo complexo. Nestas configurações devem-se ajustar parâmetros como o tamanho das sub-populações (ilhas), tipo de topologia, percentagem de indivíduos a migrar, intervalo de migrações, diferentes alternativas para a aplicação de operadores genéticos (*e.g.* taxas de mutações) e alternativas de seleção em cada ilha (tanto na seleção dos indivíduos que vão migrar como na seleção dos indivíduos que vão ser substituídos).

Relativamente à topologia das ilhas, esta está relacionada à conectividade entre as ilhas. O grau de cada ilha é definido pelo número de ilhas conectadas a essa ilha, ou seja, representa o número de vizinhos com que essa ilha pode trocar indivíduos. Existem diferentes tipos de topologias: anel; hipercubo; matriz; todos para todos. O tipo de topologia é importante no sentido que representa a comunicação que será feita entre as ilhas e consequentemente reflete-se no tempo de execução do algoritmo onde o custo de comunicação está envolvido.

Quanto à migração, é importante saber de quanto em quanto tempo é que esta ocorre (intervalo de migrações), qual a quantidade de indivíduos envolvidos nessa migração (percentagem de indivíduos a migrar) e quais os indivíduos que devem ser selecionados para migrar e quais os indivíduos que devem ser substituídos após a migração. A percentagem de indivíduos a migrar ou taxa de migração é um fator importante neste AE estruturado e depende diretamente do tamanho das ilhas e da topologia. Normalmente, quanto maior for o tamanho das populações nas ilhas maior será o seu processamento, embora seja possível analisar soluções mais abrangentes no espaço do problema evitando convergências para ótimos locais.

Outro parâmetro importante é o intervalo de migrações que define de quantas em quantas gerações irá haver migração. Assim, este parâmetro define a quantidade de comunicação que será feita entre as ilhas ao longo da execução, podendo a migração ser síncrona ou assíncrona. Na primeira, a migração é feita por todas as ilhas ao mesmo tempo (mesma geração) e na segunda, cada ilha decide quando deve efetuar a migração. Quanto à quantidade de indivíduos envolvidos numa migração, ao qual se chama taxa de migração ou percentagem de indivíduos a migrar, Cantú Paz[35] e Tanese[36] realizaram estudos dos quais provaram que quanto

maior for esta taxa, maior é a rapidez de alcançar uma solução ótima e que existe uma tendência de obter melhores soluções.

Na seleção e substituição de indivíduos durante a migração existem várias alternativas. Na seleção podem ser escolhidos os melhores indivíduos ou indivíduos aleatoriamente, e na substituição os indivíduos que vêm por migração podem ser trocados pelos piores indivíduos dessa população ou podem ser trocados por indivíduos aleatórios. Novamente, Cantú Paz[37], desenvolveu uma experiência na qual testa estas diferentes possibilidades na qual concluiu que a implementação que seleciona os melhores indivíduos e os substituiu pelos piores indivíduos obteve uma convergência significativamente mais rápida.

Implementações Híbridas

Tendo em conta os tipos de paralelismo mencionados, vários autores como Bianchini e Browm (1993)[38], Gruau (1994)[39], Cantú Paz (1998)[40], Herrera el al (1998)[41] seguiram a teoria de Davis[42] que em 1991 afirma que a combinação destas estratégias paralelas e estruturadas poderia trazer benefícios. Estas implementações são chamadas de implementações híbridas. Davis afirma também que se estes algoritmos forem bem implementados deverão apenas trazer vantagens aos resultados dos AEPs.

Seguidamente, serão dados alguns exemplos de implementações realizadas pelos autores:

Bianchini e Browm - 1993 Implementaram um modelo celular onde a avaliação do indivíduo era realizada com paralelismo global. Obtiveram resultados com a mesma qualidade de solução com melhor desempenho[38];

Gruau - 1994 Implementou o modelo de ilhas onde sobre cada ilha foi implementada uma grelha 2D com o modelo celular. Obteve bons resultados para o trabalho que estava a realizar sobre treino de uma rede neuronal[39];

Cantú Paz - 1998 Implementou assim como Gruau, um modelo de ilhas, onde em cada ilha atua um modelo celular[40];

Outra implementação usual nestes modelos híbridos é o modelo de ilhas em dois níveis onde o nível superior tem taxas de migração baixas e o nível inferior tem taxas de migração elevadas e topologias com um grau elevado nas ilhas. Desta forma, promove-se a diversidade no nível superior, onde o nível inferior é responsável pela convergência da solução ótima.

2.2 Plataforma *JECoLi*

A *Java Evolutionary Computation Library*(*JECoLi*)[43][44][45] é uma *framework* desenvolvida por grupos de pesquisa da Universidade do Minho na área dos AEs e outras meta-heurísticas. A *JECoLi* pretende ser uma plataforma de software adaptável, flexível, extensível e modular, que suporta dois tipos de objetivos principais: (i) desenvolver componentes de outras aplicações e sistemas usando os algoritmos de otimização disponíveis; (ii) permitir a rápida avaliação(*benchmarking*) de abordagens distintas em tarefas de otimização específicas.

As principais funcionalidades da *JECoLi* podem ser resumidas nos seguintes pontos:

- A plataforma implementa um grande conjunto de meta-heurísticas e variantes, nomeadamente AEs, Evolução Diferencial, Programação Genética, *Simulated Annealing* e abordagens multi-objetivo, entre outros.
- Suporta uma ampla configuração de cada algoritmo em termos de esquema de codificação da solução (binários, inteiros, reais, permutações, conjuntos ou árvores), operadores de reprodução e seleção, critérios de terminação e outros.
- Fornece suporte para desenvolver abordagens híbridas, através da combinação de algoritmos genéricos e de problemas específicos.
- É altamente flexível, permitindo que os componentes disponíveis sejam facilmente organizados e configurados de diversas maneiras. Um acoplamento é conseguido entre os algoritmos de otimização e problemas que permitem a integração fácil da biblioteca. Novos problemas são adicionados pela definição de uma única classe.
- Fornece uma plataforma baseada em componentes, com um conjunto de módulos reutilizáveis que podem ser usados por criadores de *software* para criar aplicações e permite a integração de novos componentes.
- É extensível: uma série de interfaces de programação são fornecidas para ativar a extensão da plataforma com algoritmos de grande relevância, representações, reprodução ou operadores de seleção, critérios de terminação, entre outros.
- O código é 100% Java, de código aberto(*open source*), lançado sob a licença GPL, juntamente com uma extensa documentação.

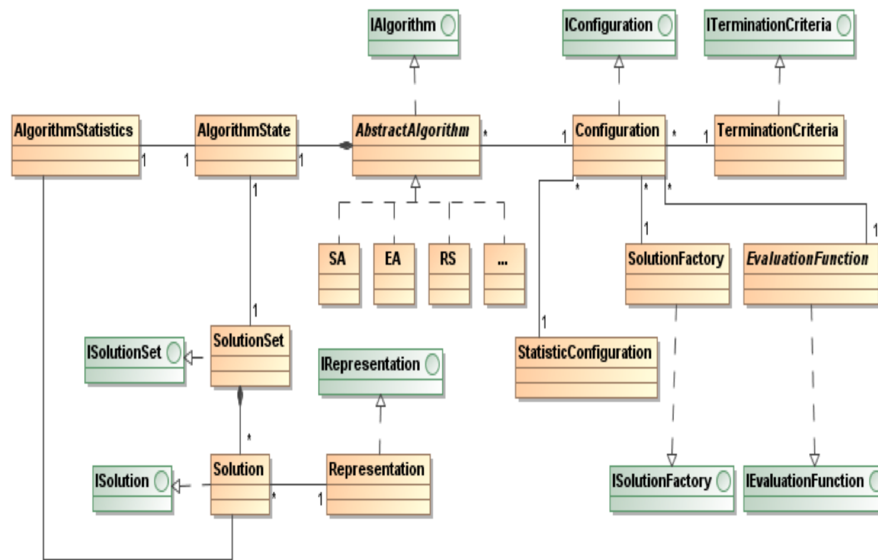

Figura 7: Estrutura de classes da JECoLi.

A *Figura 7* mostra as classes disponíveis na plataforma. Relativamente à sua implementação a biblioteca contém as principais entidades (classes):

AbstractAlgorithm representa a abstração dos métodos de otimização.

Termination Criteria implementa as condições de paragem dos algoritmos.

Evaluation Function representa as classes que lidam com a codificação dos genomas (conjunto de genes que representam o indivíduo, comparado a cromossoma) em soluções para o problema e o processo de atribuição dos valores numéricos de aptidão. Este componente é dependente do domínio do problema.

Configuration contém todas as informações necessárias para executar os algoritmos, como informações partilhadas por todos os algoritmos (*e.g. Termination Criteria* ou *Evaluation Function*) e configurações específicas do algoritmo (*e.g.* operadores de seleção ou parâmetros de recombinação em AEs).

SolutionSet* e *Solution cada solução é composta por um genoma. Este genoma é codificado numa representação específica (*SolutionSet*) e num conjunto de valores de aptidão (*Solution*).

Representation as classes que implementam a representação das soluções.

SolutionFactory usado para criar e copiar soluções.

Para cada uma destas entidades, foram criadas interfaces como se pode observar na *Figura 7*, a verde. Estas foram utilizadas para caracterizar o comportamento das entidades, onde as classes representam implementações de instâncias particulares.

2.2.1 ParJECoLi

A versão paralela da plataforma *JECoLi*, *ParJECoLi*[46][47][48], foi desenvolvida utilizando *AspectJ*[49], organizado em camadas de paralelismo. *AspectJ* é uma linguagem de Programação Orientada ao Aspeto (POA) para a linguagem Java cujo objetivo principal é aumentar a modularidade do código de aplicações separando e encapsulando diferentes preocupações/funcionalidades[50]. Estas camadas de paralelismo aplicadas são módulos ou aspetos que implementam um conjunto de modelos de paralelismo[51] no domínio dos AEPs apresentados anteriormente nesta secção. A adição destas camadas/módulos na plataforma original cria assim diferentes versões da mesma. A plataforma *ParJECoLi* é definida por:

- Possui a versão sequencial da *JECoLi*;
- Um conjunto de camadas(aspetos) que abrangem vários padrões para explorar paralelismo neste domínio;
- Uma interface gráfica em GUI, que tem como objetivo compor os módulos com a plataforma de base e que resulta numa plataforma que incorpora capacidades de processamento paralelo, com possibilidade flexível de se adaptar ao algoritmo desejado.

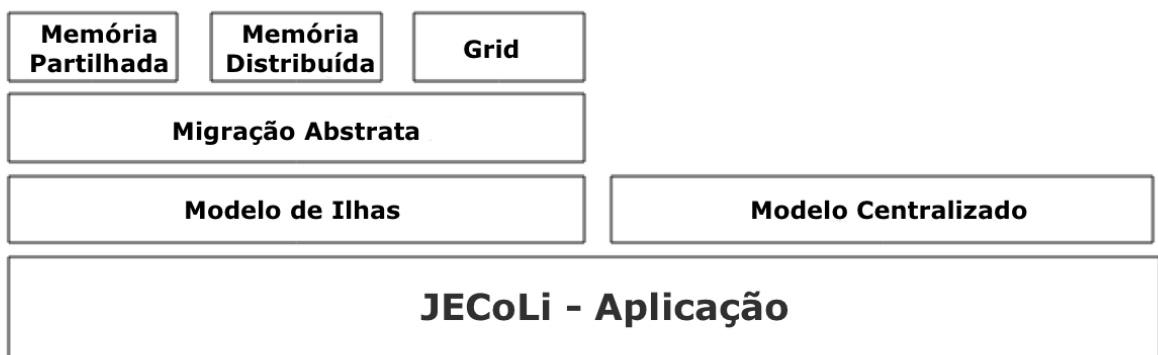

Figura 8: Arquitetura dos Módulos de Aspeto da *ParJECoLi*.

Vários módulos de paralelismo foram desenvolvidos para incorporar capacidades de processamento paralelo na plataforma *JECoLi*. Cada módulo fornece um padrão de exploração de paralelismo ou um mapeamento à plataforma. Os módulos desenvolvidos são organizados por nível de abstração. A *Figura 8* mostra essa organização de abstração: módulos independentes da plataforma e módulos dependentes da plataforma. Cada módulo de paralelismo é constituído por um ou vários aspetos, implementados em *AspectJ*.

Modelo de Ilhas e *Modelo Centralizado*(Modelo de Paralelismo Global) são os módulos de aspeto principais que não dependem diretamente da plataforma *JECoLi* e implementam as abordagens de exploração de paralelismo (AEPs) descritas anteriormente. A ordem com que estes módulos são aplicados à aplicação da *JECoLi* não é importante, uma vez que estes módulos são mutuamente exclusivos, isto é, afetam diferentes módulos ou secções da aplicação. Desta forma, ambos os módulos podem ser aplicados à plataforma de maneira a criar uma implementação híbrida: onde em cada ilha a avaliação será realizada em paralelo.

O Modelo de Paralelismo Global não escala bem para sistemas de memória distribuída devido à sobrecarga de comunicação ao enviar e receber conjuntos de soluções a/por avaliar, sendo um modelo centralizado. Por outro lado, o *Modelo de Ilhas* exige migração periódica de soluções entre ilhas. Este modelo normalmente adapta-se bem para ambientes de memória partilhada e memória distribuída embora as suas implementações diferam em função da plataforma subjacente (dependendo da plataforma pode utilizar tecnologias como *Java threads*, *MPI* ou *Grid middleware*).

O módulo de *Migração Abstrata* corresponde à introdução de novas funcionalidades comuns na migração de soluções entre ilhas que são necessárias para a implementação do Modelo de Ilhas paralelo. É importante reforçar que o módulo *Migração Abstrata* depende do módulo *Modelo de Ilhas*. Este módulo, *Migração Abstrata*, não afeta a execução da aplicação ou o seu comportamento mas é responsável por estabelecer uma interface que fornece funcionalidades básicas para que os outros módulos(*Memória Partilhada*, *Memória Distribuída* e *Grid*) possam ser aplicados.

Não foram realizados testes ao Modelo Híbrido e os restantes modelos foram testados com poucos recursos computacionais(sendo utilizados um máximo de dezasseis processadores na dissertação e posteriormente com vinte e quatro processadores num dos artigos mencionados).

3 Desenvolvimento

Esta secção irá debruçar-se no desenvolvimento realizado ao longo da dissertação apresentando as estratégias de implementação das versões paralelas da plataforma *JECoLi.* Para tal, será apresentado primeiramente como se implementa uma meta-heurística na *JECoLi* e em seguida as decisões tomadas para implementar as versões paralelas com base no modelo de paralelismo global, no modelo de ilhas e no modelo híbrido.

3.1 Implementação de meta-heurísticas na plataforma JECoLi

Nesta subsecção será explicado como são implementadas meta-heurísticas na plataforma em causa. A plataforma *JECoLi* foi construida de forma a ser facilmente utilizável tendo em conta as inúmeras funcionalidades que possui. Para isto são usadas interfaces sobre as componentes principais destes algoritmos (*Figura 7*). Estas interfaces são implementadas geralmente por classes abstratas que contêm os métodos principais de cada componente. Este tipo de organização do código Java favorece a possibilidade de fácil extensão de novos componentes ou da alteração dos existentes.

Esta plataforma implementa inúmeros algoritmos de otimização e procura de soluções já referidos tais como AEs, Evolução Diferencial, *Simulated Annealing*, abordagens multi-objetivo, entre outros. Cada um destes algoritmos contém uma classe que estende a classe *AbstractAlgorithm* e para cada um existe uma classe de configuração específica. Nesta dissertação, apenas os AEs serão utilizados para realizar testes com versões paralelas.

A *Figura 9* representa um exemplo da execução de AEs na *JECoLi* através de um diagrama de sequência. Geralmente, o programa começa por receber parâmetros relativos ao problema em questão. Em seguida, cria uma instância da configuração do AE que irá conter os parâmetros e características que o algoritmo precisa para a execução, tais como:

Função de Avaliação relativa ao problema em causa;

SolutionFactory para gerar as soluções/indivíduos do problema;

Tamanho da População ;

Critério de Terminação que define quando é que a execução deve terminar;

Parâmetros de Recombinação como o valor de elitismo e o número de descendentes por geração;

Operadores de Seleção ;

Conjunto de Operadores de Reprodução e a respetiva percentagem de utilização.

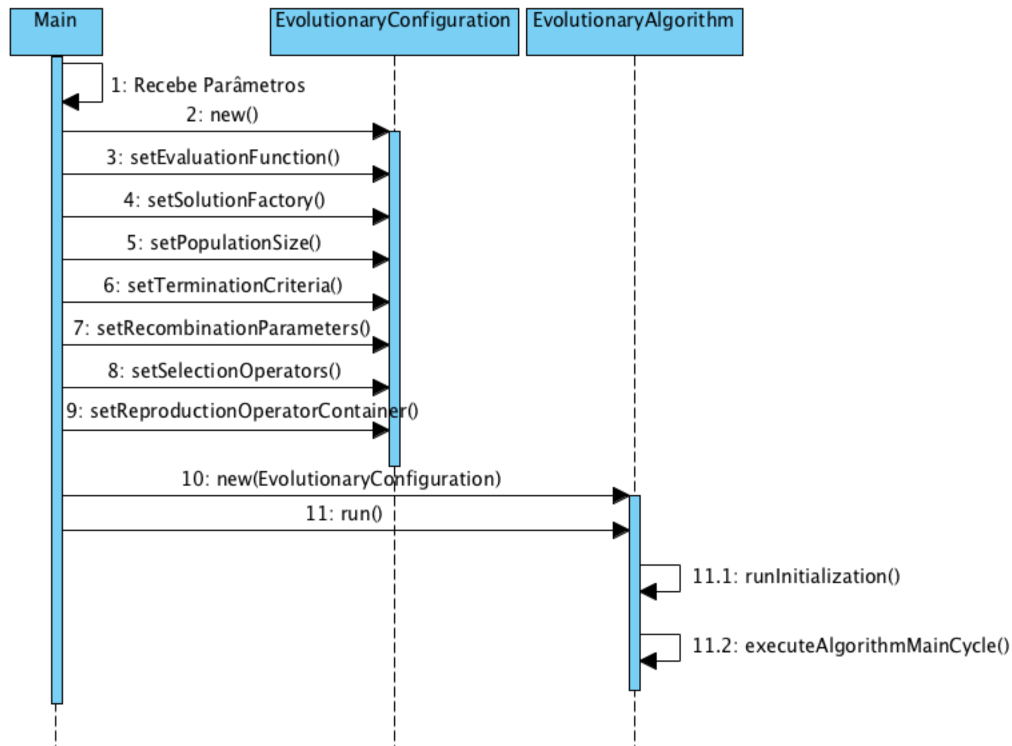

Figura 9: Exemplo de diagrama de sequência da execução de AEs na plataforma *JECoLi*.

Após esta fase de configuração que irá determinar o rumo da execução, cria-se uma instância do AE dando-lhe como parâmetro a instância de configuração, e de seguida é chamado o método *run*. O método *run* irá executar o AE tendo em conta o problema em questão, retornando o resultado. Começa por chamar o método *runInitialization* que irá criar uma instância da classe *AlgorithmState*, criar o conjunto de soluções iniciais (avaliando-as posteriormente) e atualizar a instância do estado criada com esta população.

Seguidamente, é executado o método *executeAlgorithmMainCycle* sendo este o método que representa a execução do AE. Este método lê os parâmetros da configuração estabelecida e começa o ciclo iterativo tendo em conta o critério de paragem/terminação. Muito resumidamente, em cada iteração do ciclo, são selecionados os indivíduos a aplicar os operadores genéticos de reprodução, aplicam-se estes operadores de forma a gerar um conjunto de soluções consoante os parâmetros da configuração, e por fim realiza-se uma vez mais a seleção de forma a substituir parte dos indivíduos atuais pelos novos gerados nessa iteração.

3.2 Paralelismo Global

Nesta subsecção, é exposta a primeira implementação paralela realizada, tendo em conta o modelo de paralelismo global já apresentado, assim como as alterações necessárias na plataforma *JECoLi*.

O modelo de paralelismo global é geralmente denominado por modelo "*master-slave*" onde o "*master*" vai ser o processo/*thread* principal que vai executar o AE e nas partes computacionalmente mais pesadas irá distribuir esse trabalho por outros processos/*threads*, "*slaves*". Os caso de estudo abordados nesta dissertação gastam maioritariamente o seu tempo na avaliação de indivíduos/soluções, por isso esta foi a parte do código que foi paralelizada tendo em conta este modelo.

O primeiro objetivo deste trabalho seria implementar versões paralelas da *JECoLi* de forma menos invasiva possível, ou seja, tentando minimizar o número de alterações ao código base. Assim sendo, foram consideradas várias alternativas até à implementação final, tanto nas estratégias de paralelismo como na tentativa de minimização do número de alterações ao código base.

Sendo o objetivo final paralelizar a avaliação dos indivíduos, existem apenas duas situações distintas onde a avaliação é efetuada: na inicialização do AE após a criação da população inicial; e em cada geração após a criação dos novos indivíduos (através dos operadores genéticos). Analisando o código base da *JECoLi*, o método chamado nestas situações que trata da avaliação dos indivíduos pertence à classe respetiva da função de avaliação. Com isto, a implementação paralela mais simples seria criar um novo método para tratar da avaliação dos indivíduos, que consiste no seguinte:

1. Cria-se o número de *threads* desejado;

2. Divide-se o número de indivíduos pelo número de *threads*;

3. Executa-se o conjunto de *threads* com a tarefa de avaliar o conjunto de indivíduos recebidos;

4. Destrói-se o conjunto de *threads*.

Esta foi a primeira estratégia implementada de forma a haver uma adaptação do funcionamento da plataforma *JECoLi*, assim como uma análise do que teria de ser obrigatoriamente alterado no código base da *JECoLi*. Esta estratégia serve então como teste introdutório à plataforma, levando à otimização de partes do código base de modo a obter uma versão paralela da *JECoLi*.

Foram criadas duas classes externas à *JECoLi* para implementar esta estratégia de forma a separar o código paralelo do código base da *JECoLi*. A primeira classe, *Parallel_Evaluation*,

é responsável por criar as *threads* necessárias, dividir o número de indivíduos pelo número de *threads* e executar o conjunto de *threads* com a tarefa de avaliar os indivíduos, destruindo as *threads* no fim deste processo. A segunda classe, *Evaluation_Task*, é uma classe *Runnable* que é inicializada com uma instância da função de avaliação e com o conjunto de indivíduos a avaliar, sendo o seu método *run* um ciclo para a avaliação de um conjunto de indivíduos.

Tendo em conta que criar e destruir *threads* em cada geração/iteração do ciclo principal do AE tem um certo peso no tempo total da execução desta implementação, foi considerada outra alternativa com a utilização de *Executors*. A classe *java.util.concurrent.Executors* da API do Java(a partir do Java Platform SE 7) implementa serviços semelhantes a uma *thread pool* de forma configurável, ou seja, é capaz de criar um conjunto de *threads* gerindo a sua execução. A interface *ExecutorService* representa um mecanismo de execução assíncrona capaz de executar tarefas em "background" com a possibilidade de saber quando é que cada tarefa termina através de uma interface *Future* com o método *get*.

Com o estudo destas componentes de concorrência do Java, modificou-se ligeiramente a estratégia anterior de forma a criar outra classe (*Parallel_Evaluation_Pool*) com estas componentes. Esta nova estratégia comporta-se de forma diferente, pois no momento de inicialização do AE é criada uma instância *ExecutorService* através do método *newFixedThreadPool* da classe *Executors* que representa a *thread pool*. É neste método que reside a grande diferença em relação à estratégia anterior, pois este método cria uma *thread pool* com um número de *threads* fixo, reutilizando as *threads* disponíveis já criadas anteriormente. No final da execução do AE, é necessário chamar o método *shutdown* de forma a destruir as *threads* criadas.

Apesar desta instância *ExecutorService* conseguir executar tarefas assíncronas, essa não é a funcionalidade pretendida pois cada iteração/geração é dependente da anterior. Isto obriga as tarefas de avaliação de indivíduos a serem síncronas, esperando que todas as *threads* finalizem a avaliação de modos a proceder com a execução do AE em cada geração. Para tal efeito, usou-se o método *submit* da interface *ExecutorService* que escolhe uma *thread* disponível para executar uma tarefa de avaliação, retornando uma instância da interface *Future* que permite monitorizar a finalização da tarefa. A classe *Parallel_Evaluation_Pool* tem um método principal, *parallel*, que divide o número de indivíduos pelo número de *threads*, cria um número de tarefas *Evaluation_Task* consoante esta divisão, usa o método *submit* para cada uma destas tarefas, e por fim espera que todas as tarefas terminem através da instância *Future* retornada para cada uma delas.

Tendo em consideração a implementação de meta-heurísticas na *JECoLi*, de forma a implementar esta estratégia, poucas alterações no código são necessárias para a execução de qualquer problema usando esta implementação do modelo de Paralelismo Global.

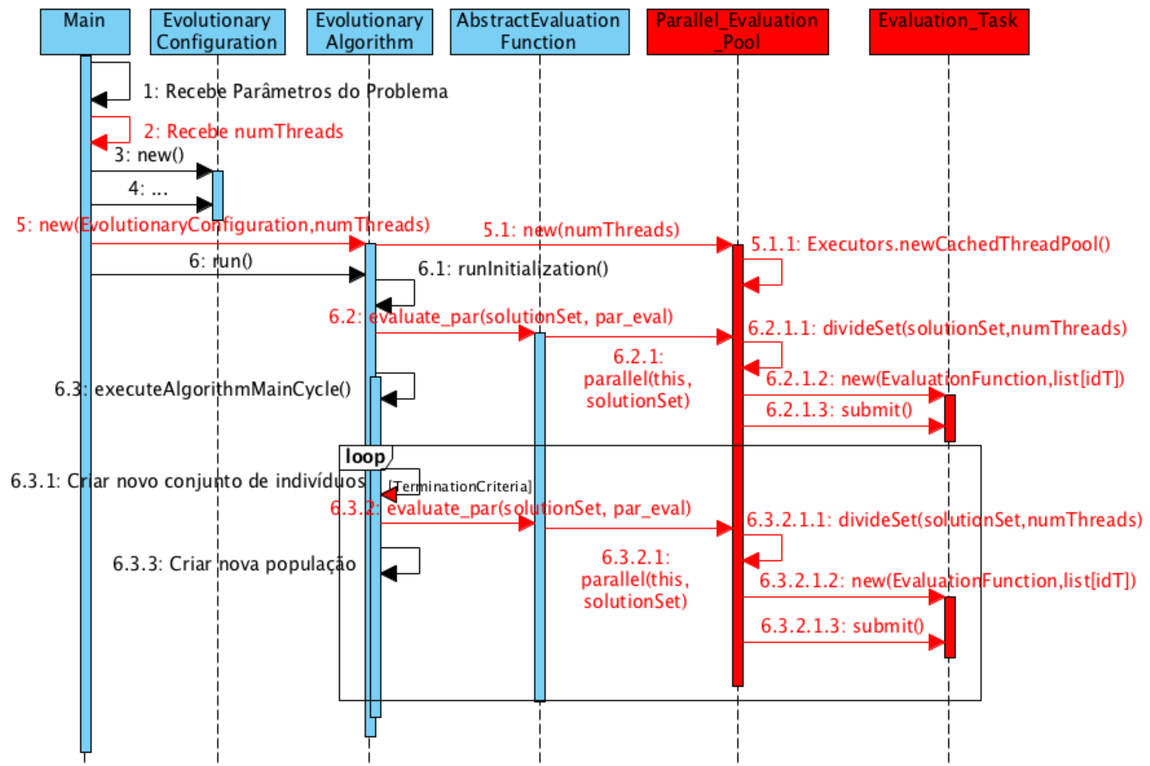

Figura 10: Exemplo de diagrama de sequência da execução de AEs na plataforma *JECoLi* usando o modelo de Paralelismo Global.

A *Figura 10* representa o diagrama de sequência semelhante ao da *Figura 9* para a execução da implementação paralela em causa. A vermelho estão as principais alterações ao código na criação de meta-heurísticas para um determinado problema, assim como as novas classes mencionadas anteriormente: *Parallel_Evaluation_Pool* e *Evaluation_Task*.

Relativamente às alterações ao código na criação de meta-heurísticas para um determinado problema na plataforma *JECoLi*, inicialmente o programa terá de receber o número de *threads* pretendidas para futuramente serem criadas no modelo de Paralelismo Global. De forma a facilitar o uso desta implementação paralela, o parâmetro de número de *threads* é obrigatório, caso seja igual a um o código será executado sequencialmente.

Em seguida, na configuração poderão ser necessárias algumas alterações dependendo do problema em causa. Existem problemas cuja função de avaliação não é *thread safe*, isto é, pode não possibilitar o uso de concorrência em determinadas partes do código, que implica uma reformulação da função de avaliação. Este último caso irá ser abordado na secção dos resultados neste relatório.

As alterações do código base da *JECoLi* foram realizadas nas seguintes classes.

AbstractAlgorithm Uma variável pública de classe *Parallel_Evaluation_Pool* foi inserida (inicializada com o valor de *null*) e um novo construtor semelhante ao existente foi inserido recebendo também um inteiro relativo ao número de *threads* a utilizar. Neste novo construtor era inicializada a instância *Parallel_Evaluation_Pool* que recebe o número de *threads*, criando assim a *thread pool* desejada (passos 5., 5.1. e 5.1.1. da *Figura 10*).

IEvaluationFunction O método *evaluate_par* foi adicionado que recebe o conjunto de indivíduos a ser avaliado e uma instância da classe *Parallel_Evaluation_Pool*.

AbstractEvaluationFunction Esta classe abstrata implementa a interface *IEvaluationFunction*, logo irá implementar o método *evaluate_par*. Este método verifica se o número de indivíduos é maior ou igual ao número de *threads*, caso a condição seja verdadeira invoca o método *parallel* da instância *Parallel_Evaluation_Pool* dando como parâmetros a própria instância de classe e o conjunto de indivíduos, caso seja falsa irá realizar a avaliação sequencialmente (este caso foi inserido apenas para os casos em que a população do AE é bastante pequena face à paralelização pretendida).

EvolutionaryAlgorithm Visto que esta classe estende a abstrata *AbstractAlgorithm*, foi adicionado um novo construtor que recebe a configuração e o número de *threads* redirecionando a inicialização para o construtor da classe abstrata.

Em todas as situações de avaliação de indivíduos, onde é chamado o método *evaluate*, foi colocada uma condição que verifica se a instância *Parallel_Evaluation_Pool* foi inicializada ou não. Caso tenha sido inicializada, é chamado o método *evaluate_par* dando como parâmetros o conjunto de indivíduos e a instância *Parallel_Evaluation_Pool*(passos 6.2. e 6.3.2. da *Figura 10*), caso contrário é chamado o método *evaluate*.

No passo 6.3.1. da *Figura 10*, no método *selectReproductionIndividuals*, verificou-se um problema perante a implementação deste modelo (Paralelismo Global). Este método, no código base da *JECoLi*, continha um ciclo onde a cada iteração eram gerados dois indivíduos de cada vez e avaliados logo de seguida, impedindo o uso de mais de duas *threads* na implementação paralela (um individuo para cada *thread*). Assim sendo, reestruturou-se o código original de forma a agregar todos os indivíduos criados através de operadores genéticos numa estrutura (*SolutionSet* pertencente à *JECoLi*) para permitir que a avaliação seja realizada no fim do ciclo, a todos os descendentes dessa geração.

3.3 Modelo de Ilhas

Serão apresentadas nesta subsecção duas versões paralelas do modelo de ilhas implementadas sobre a plataforma *JECoLi*, uma versão em memória partilhada e outra em memória distribuída, apresentando também as alterações necessárias à plataforma e as estratégias utilizadas nas implementações. A versão em memória partilhada usa *Java Threads* e a versão em memória distribuída usa *Java MPI*.

Estas implementações irão ter uma estratégia completamente diferente da implementação do modelo de paralelismo global visto que em vez de paralelizar uma parte do AE (*e.g.* paralelizar a avaliação de indivíduos) irão paralelizar todo o algoritmo AE, e com isto terão outro tipo de preocupações. O modelo de ilhas é um AE estruturado que tem vários AEs (ilhas) a executar dividindo assim o seu processamento. O processamento ou tempo de execução num AE é determinado pelo número de gerações/iterações e pelo fator principal responsável pelo tempo de execução em cada geração: o número total de indivíduos. Assim, existem duas formas de dividir o processamento de um AE sequencial, dividindo o número de gerações pelo número de ilhas (caso esse seja o critério de paragem do AE) ou dividindo o número de indivíduos pelo número de ilhas pretendido. A implementação efetuada tem em consideração estas duas formas de divisão do processamento pelas várias ilhas.

Como foi já discutido na secção anterior, o modelo de ilhas é bastante configurável e complexo, onde diversos parâmetros devem ser considerados tais como o tamanho das subpopulações (número de indivíduos), o tipo de topologia a utilizar, a percentagem de indivíduos a migrar, intervalo de migrações e alternativas de seleção em cada ilha de indivíduos para migrar e para serem substituídos. Todas estas preocupações são consideradas nas implementações paralelas, tendo em conta que o mínimo de alterações foram realizadas no código base da plataforma *JECoLi*. Para tal, foram criadas diversas classes externas para implementar este modelo, reduzindo alterações na *JECoLi*. Estas classes foram implementadas com a intenção de serem utilizadas da mesma forma que as classes da *JECoLi*, como uma API.

De seguida, são explicadas as classes externas criadas para a implementação do modelo de ilhas:

IslandModelParams Serve para definir os parâmetros inerentes ao modelo de ilhas, tais como o número de ilhas, o número total de indivíduos, o número total de gerações, o tipo de divisão de processamento (gerações ou indivíduos), se existirá migração, o intervalo de migração, o número de indivíduos a migrar por migração, o tipo de seleção nos indivíduos que vão migrar, o tipo de seleção nos indivíduos que irão ser substituídos, se o problema é de maximização ou não, se o modelo de ilhas será sequencial ou paralelo e se a execução paralela será feita usando *Java Threads* ou *Java MPI*.

IMTopology Tem como objetivo definir a estrutura da topologia a ser utilizada no modelo de ilhas através da criação de um grafo orientado. Um grafo é uma estrutura para modelar a relação entre objetos, ou seja, um conjunto de vértices que representam os objetos sendo as suas ligações representadas pelas arestas (neste caso, sendo orientado significa que cada aresta irá ter um vértice como origem e um vértice como destino). Neste caso específico, os vértices serão as ilhas e as arestas representam a vizinhança com a qual cada ilha migra indivíduos.

Para atingir este objetivo foi utilizada a biblioteca *libgraph*[52] que foi integrada no código base da *JECoLi* para facilitar a criação de grafos e o acesso aos seus componentes. Foram criados quatro tipo de topologias nesta classe: em anel; em anel bidirecional; em estrela; e totalmente conectada (todos para todos). A classe recebe o número de ilhas e o tipo desejado, criando assim um grafo que poderá ser consultado posteriormente.

MigrationBuffer Funciona como um *buffer* partilhado que representa as arestas do grafo orientado mencionado de maneira que irá conter um identificador do vértice de origem, um identificador do vértice de destino e uma lista de indivíduos para realizar a migração entre os vértices correspondentes (ilhas).

Migration Implementa a migração de indivíduos entre as ilhas dependendo da topologia escolhida assim como os parâmetros recebidos. Caso a migração seja desejada (o modelo de ilhas tem obrigatoriamente migração, mas em casos de teste pode ser desejado que não exista migração), uma instância *Migration* é criada com a informação do intervalo de migração, percentagem de indivíduos a migrar, se o problema é de maximização ou não e o tipo de seleção tanto nos indivíduos a migrar na ilha de origem como nos indivíduos a serem substituídos na ilha de destino. Os métodos principais desta classe implementam a migração de indivíduos em todas as ilhas quer seja utilizando memória partilhada quer seja em memória distribuída (utilizando *Java Threads* ou *Java MPI*). As estratégias de paralelismo e de implementação da migração serão discutidas posteriormente nesta secção.

O tipo de seleção de indivíduos é uma componente de grande importância dado que é um dos fatores que influencia diretamente a convergência de soluções ótimas do problema, pois vai de certo modo fazer uma filtragem às soluções atuais em cada migração. Com isto, foram implementados nesta classe quatro tipos de seleção de indivíduos a migrar nas ilhas de origem e dois tipos de seleção de indivíduos a substituir nas ilhas de destino. Os quatro tipos de seleção de indivíduos a migrar implementados foram a seleção dos melhores indivíduos, a seleção de indivíduos aleatoriamente, a seleção de

indivíduos através de seleção por torneio e a seleção de indivíduos através de seleção por *ranking*. Já os dois tipos de seleção de indivíduos a substituir implementados foram a seleção dos piores indivíduos e a seleção aleatória de indivíduos. Estes tipos de seleção são realizados consoante o tipo de problema, considerando se os problemas são de maximização ou minimização.

Island É a classe que representa a ilha e contém a instância do AE a executar, o seu identificador (número inteiro) relativamente à topologia, uma instância do *IAlgorithmResult* de forma a guardar o resultado, a lista de indivíduos atuais na ilha a cada momento, e por fim uma lista de *MigrationBuffer* de saída e uma lista de *MigrationBuffer* de entrada.

Esta classe contém dois métodos principais, o método *init*, que implementa o método de inicialização do algoritmo correspondente ao método *runInitialization* já apresentado, e o método *step*, que implementa todo o processo realizado em cada geração do algoritmo como sendo cada iteração do método *executeAlgorithmMainCycle* já apresentado.

IslandModel É a classe principal da implementação do modelo de ilhas que interliga todas as outras classes. A classe *IslandModel* contém todos os parâmetros estabelecidos através da classe *IslandModelParams*, uma lista de todas as ilhas do tipo de classe *Island*, a topologia escolhida e o seu grafo através de uma instância da classe *IMTopology*, uma instância da classe *Migration* para realizar a migração, o número de gerações pretendidas para a execução de cada ilha e por fim caso o modelo paralelo escolhido seja *Java Threads* irá ter um conjunto de *threads*.

Tendo em conta que existe um número considerável de configurações possíveis para este modelo dependendo da topologia e dos parâmetros recebidos, a inicialização desta classe é um pouco complexa. Inicialmente, nesta inicialização, tem-se em conta se o modelo vai ser executado sequencialmente ou em paralelo tendo em conta a utilização de *Java Threads* ou *Java MPI*. Depois é considerado o tipo de divisão de processamento em relação ao modelo centralizado, se irão dividir-se as gerações ou o número de indivíduos ajustando os parâmetros recebidos, dividindo respetivamente as gerações ou o número de indivíduos pelo número de ilhas. Posto isto, considerando o uso do modelo sequencial ou modelo paralelo com *Java Threads* ou *Java MPI* serão inicializadas todas as ilhas conforme cada caso, assim como, caso se pretenda a migração, inicializa-se a instância de migração conforme cada modelo e tipo de paralelismo escolhido. Para o caso em que a migração é efetuada, é lido o grafo da instância *IMTopology* de modo a atribuir identificadores a cada ilha, gerar todas as instâncias *MigrationBuffer* necessárias e atribui-las a cada ilha conforme as relações.

Dois métodos, *run* e *runMPI*, irão executar o modelo de ilhas conforme a inicialização efetuada. Adicionalmente, foram implementados mais dois métodos com o objetivo de consultar os resultados da execução do modelo de ilhas: *returnBest* que retorna a instância de *IAlgorithmResult* e imprime o melhor valor encontrado de todas as ilhas conforme o problema seja de maximização ou não; e *writeLastResults* que imprime várias informações de cada ilha, tais como o número de iterações, o número de avaliações de indivíduos, o tempo total de execução, o valor de aptidão mais alto, o valor de aptidão mais baixo, a média do valor de aptidão e o número de objetivos.

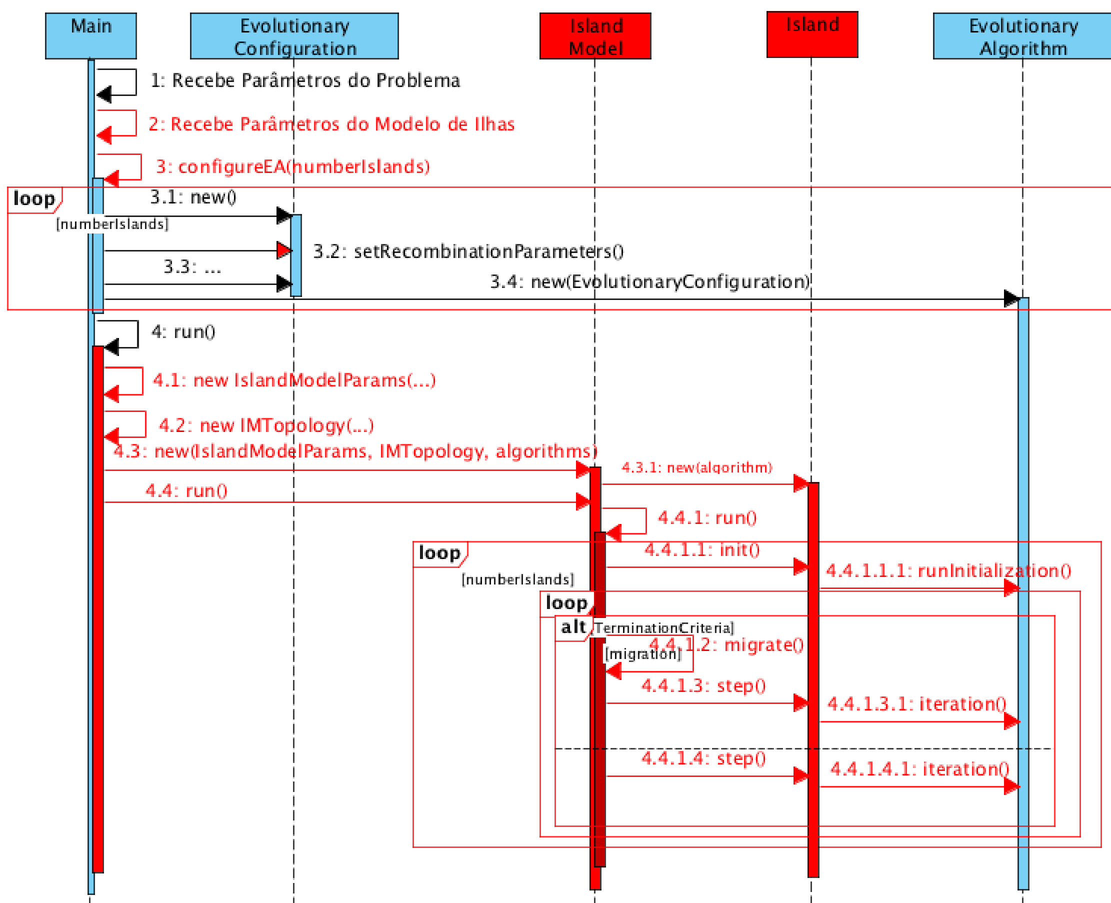

Figura 11: Exemplo de diagrama de sequência da execução de AEs na plataforma *JECoLi* usando o modelo de ilhas.

A *Figura 11* é apresentada para facilitar a compreensão da implementação do modelo de ilhas assim como para facilitar a construção de um problema que use este modelo jun-

tamente com a plataforma *JECoLi*. A vermelho estão as partes do código alteradas face à implementação de meta-heurísticas inicial, sendo as mudanças necessárias para usar a implementação do modelo de ilhas.

Numa fase inicial, recebem-se ou definem-se os parâmetros tanto do problema em causa como do modelo de ilhas. Em seguida, deverão realizar-se várias configurações dos AEs conforme o número de ilhas desejado, tendo em conta a divisão de processamento desejado (dividir gerações ou dividir indivíduos pelas ilhas) e posteriormente cada uma destas configurações deverá criar uma instância do AE, sendo estas guardadas numa estrutura. Nos casos de estudo analisados, foi necessário alterar o valor de elitismo e o valor de descendentes no passo 3.2 do diagrama devido a um ajuste na divisão de processamento, este será discutido na secção dos resultados.

Na fase de execução do algoritmo, deve-se criar uma instância *IslandModelParams* com os parâmetros desejados e uma instância *IMTopology* com o tipo de topologia pretendida e o número de ilhas (passos 4.1 e 4.2), com o objetivo de criar uma instância da classe principal do modelo de ilhas, *IslandModel*, que irá tratar da execução deste modelo (passo 4.3). Por sua vez, esta instância recebe também os AEs criados para poder inicializar o conjunto de ilhas e realizar a restante inicialização. Independentemente de executar sequencialmente ou em paralelo, todos os AEs das ilhas são inicializados (passos 4.4.1.1) e começam a execução dos seus algoritmos (*loop*). Durante o ciclo principal de cada AE, em todas as gerações é verificado o número da geração atual e é realizada migração entre todas as ilhas conforme o intervalo de migração estabelecido.

Quanto às alterações no código base da plataforma *JECoLi* apenas foi necessário mudar a declaração do método *iteration* da classe abstrata *AbstractAlgorithm* para a interface *IAlgorithm*, de maneira a poder aceder a este método no método *step* da classe *IslandModel*.

Modelo de Ilhas em Memória Partilhada

A primeira implementação paralela realizada utiliza *Java Threads*, de modo que este tipo de paralelismo apenas funciona para sistemas com memória partilhada. De modo a implementar uma versão paralela do modelo de ilhas, temos de considerar como será aplicado o paralelismo, verificar se existe concorrência e verificar qual a comunicação necessária entre *threads*/processos paralelos.

Foi criada uma classe denominada *IMThread* que estende o tipo *java.lang.Thread* e implementa a execução de um AE normal incluindo também a parte de migração do modelo de ilhas. Esta classe possui uma instância *Island*, um identificador da *thread*, uma variável com o objetivo de contar as iterações realizadas e uma instância *Migration* comum a todas as

threads (ilhas). Estas *threads* são criadas na classe principal, *IslandModel*, sendo inicializadas no construtor da mesma após a inicialização das instâncias ilhas e são executadas no método *run* da mesma.

Na implementação do modelo de ilhas apenas foi considerada a utilização do critério de terminação por número de gerações, onde é dado um número de gerações inicialmente pelo utilizador e consoante a divisão de processamento escolhida, cada ilha executa um número fixo de gerações.

Assim sendo, o método *run* desta classe (*IMThread*) irá invocar o método *init* da classe *Island* para inicializar a população e fazer a restante inicialização e entra num ciclo que tem como condição de paragem um número de gerações. Cada iteração deste ciclo representa então uma geração que é executada através do método *step* da classe *Island*. A cada iteração é verificado se é necessário realizar migração dependendo do intervalo de migração estabelecido.

Não existem dependências entre as execuções dos AEs pois cada AE terá a sua população, apenas é necessário ter alguns cuidados relativamente às funções de avaliação e/ou outras classes relativas ao problema em causa pois pode existir concorrência ou conflitos que são resolvidos facilmente se cada AE possuir as suas próprias instâncias. Com isto, a única preocupação quanto à concorrência será apenas na migração pelo que é a única parte deste modelo onde existe comunicação entre as ilhas com a troca de indivíduos.

A classe *Migration* é a responsável por tratar de toda a comunicação entre *threads* de forma que a migração seja efetuada. O método principal responsável por este processo tem o nome de *migrate* que recebe a lista de ilhas que irá tratar da migração, ou seja, usando este tipo de paralelismo, cada *thread* irá executar este método enviando apenas a sua ilha de modo a tratar da sua parte da migração (foi assim implementado pois para o caso sequencial a lista irá possuir todas as ilhas de forma a tratar da migração para todas as ilhas). Existem mais dois métodos importantes que correspondem às duas fases principais da migração: o método *putInBuffer* que recebe a ilha em questão e o estado *AlgorithmState* (contém os indivíduos na geração atual do AE) associado, de maneira a fazer a seleção dos indivíduos a migrar, criar uma cópia dos indivíduos selecionados e colocá-los em todos os *MigrationBuffer* de saída da ilha; e o método *getFromBuffer* que recebe igualmente a ilha em questão e o estado *AlgorithmState* associado, com o objetivo de percorrer todos os *MigrationBuffer* de entrada da ilha, retirar todos os indivíduos a migrar, aplicar a seleção na população atual da ilha para saber os "n" elementos a substituir e realizar a toca de indivíduos. O método *migrate* então chama inicialmente o método *putInBuffer* para fazer a seleção e colocar todos os indivíduos selecionados nos *buffers* partilhados e, de seguida, chama o método *getFromBuffer* para efetuar a substituição.

Após uma análise de todas as possibilidades de realizar esta migração com diferentes

topologias e com este modelo em *Java Threads*, implementou-se um tipo de migração síncrono e um tipo de migração assíncrono.

O tipo de migração síncrono tem uma "barreira" entre os métodos *putInBuffer* e *getFromBuffer* de maneira a que todas as *threads* têm de esperar nessa fase até que todas as *threads* completem o método *putInBuffer*, garantido que na segunda fase, no método *getFromBuffer*, todas as ilhas já têm os indivíduos que devem migrar nos seus *buffers*. Esta "barreira" é implementada utilizando uma instância *CyclicBarrier* que faz parte da classe *java.util.concurrent* e tem a característica de após ser inicializada pode ser reutilizada quando necessário.

O tipo de migração assíncrono não utiliza a "barreira" referida, permitindo a cada *thread* continuar a execução do seu AE sem esperar pelas restantes *threads*. Não existindo esta "barreira" significa que pode existir concorrência entre as diferentes *threads*, podendo uma *thread* estar a depositar indivíduos no *buffer* partilhado e outra a retirá-los simultaneamente. Com isto é implementado um *synchronized statement* focada na lista de indivíduos de modos a que apenas uma *thread* consiga aceder a esta lista de cada vez. Para este tipo de migração, o método *migrate* irá retornar um booleano que indica se a migração foi realizada inteiramente ou não, pois no caso em que pelo menos uma das trocas não foi realizada este booleano é retornado como falso. Este caso em que pelo menos uma das trocas não foi realizada acontece devido ao facto de no momento de migração, na troca de indivíduos, ainda não estarem disponíveis os indivíduos de uma ou mais ilhas vizinhas para a realização da migração. Para este caso, a *thread*/ilha em questão procede a execução do seu AE e em cada geração irá verificar se os indivíduos das ilhas vizinhas já se encontram disponíveis nos *buffers* partilhados de forma a realizar a troca de indivíduos e completar a migração caso os indivíduos estejam disponíveis.

Assim, este tipo de migração pode influenciar os princípios de migração pois no caso de existir *threads*/ilhas em gerações mais avançadas, estas irão substituir indivíduos menos aptos das *threads*/ilhas menos avançadas, assim como as *threads*/ilhas menos avançadas irão substituir indivíduos mais aptos. Assim, este modelo permite não limitar tanto a escalabilidade do paralelismo comparando ao tipo de migração síncrono que utiliza uma "barreira" global, embora possa alterar o comportamento da convergência de soluções.

Modelo de Ilhas em Memória Distribuída

Com o objetivo de implementar o modelo de ilhas em memória distribuída foi utilizado *MPI Java*, que em vez de utilizar *threads* utiliza processos. Enquanto as *threads* são criadas através de um processo sendo assim mais leves e partilhando os mesmos espaços de memória,

os processos podem correr em endereços separados, sendo independentes e permitindo então a execução do modelo de ilhas em memória distribuída. Foi utilizada a biblioteca open-mpi[53] juntamente com a interface para Java(*libmpiJava* versão 1.2.5[54]) de forma a usar as funcionalidades MPI na linguagem Java.

A utilização de *MPI Java* implica que os processos sejam independentes devendo existir comunicação através de mensagens de forma a trocar a informação necessária para executar os AEs em todos os processos tal como na troca de indivíduos na migração. Esta implementação começa então por inicializar os processos através do método *Init* do *MPI*, consoante o número de ilhas desejado. Cada processo inicializado irá possuir um identificador(*rank*) como forma de os distinguir e onde se assume que existe um processo principal com o valor do identificador a zero.

Todos os processos criados irão inicialmente interpretar os parâmetros recebidos (relativos ao problema em causa e relativos ao modelo de ilhas) e posteriormente realizar a configuração do seu AE correspondente. Após esta etapa estar concluída, o processo principal cria as instâncias relativas à topologia e aos parâmetros do modelo de ilhas, de forma a criar a instância principal do modelo de ilhas (*IslandModel*). A principal diferença entre esta implementação paralela e a implementação em memória partilhada é que a instância de *IslandModel* é criada apenas no processo principal, sendo este o processo responsável pela configuração do modelo de ilhas, recebendo um booleano que corresponde à utilização de *MPI* na migração.

Após a conclusão destas fases seguem-se a execuções dos AEs, mas caso a migração seja desejada é necessário que os restantes processos recebam a informação necessária. Para tal, foi criado um método *run_mpi_master* na classe *IslandModel* para ser executado apenas pelo processo principal que contém três fases. A primeira fase é realizada caso a migração seja desejada, que irá consistir no envio para cada processo secundário de duas instâncias relativas à topologia e à migração. A segunda fase corresponde à execução da sua ilha (incluindo as fases de migração). A última fase corresponde à receção de todas as ilhas dos processos secundários de forma a reunir os resultados de todas as execuções.

Por outro lado, para os processos secundários foi criada uma classe auxiliar, *MPISecondaryProcess*, para tratar da execução dos seus AEs. Depois da configuração dos seus AEs, os processos secundários criam uma instância de *MPISecondaryProcess* enviando o seu *rank* juntamente com o número de gerações dependendo da divisão de processamento escolhida. Em seguida, executam o método *run_mpi_secondary* que irá conter igualmente três fases. Na primeira fase, todos os processos secundários irão esperar pelas mensagens enviadas pelo processo principal com as instâncias da migração e da topologia. Após a receção destas instâncias, segue-se a segunda fase que será a execução do modelo de ilhas. Por fim, a ter-

ceira fase será posterior a esta execução em que cada processo secundário envia a instância da sua ilha para o processo principal.

A migração neste modelo de memória distribuída não utiliza as instâncias *MigrationBuffer* para a troca de indivíduos, sendo a troca realizada através de mensagens *MPI*. Para isto foram criados três métodos na classe *Migration*: *migrateMPI*, *sendSolutionsMPI* e *receiveSolutionsMPI*. Estes métodos implementam as mesmas funcionalidades que os métodos do modelo de memória partilhada exceto a forma como trocam indivíduos entre as ilhas. O método *sendSolutionsMPI* corresponde ao método *putInBuffers*, onde são selecionados os indivíduos a realizar a troca e enviados para as ilhas vizinhas utilizando o método *Isend* do *MPI* que realiza o envio de uma mensagem assíncrona, procedendo a sua execução. O método *receiveSolutionsMPI* corresponde ao método *getFromBuffers*, recebe os indivíduos de cada ilha vizinha através do método *Recv* do *MPI* e realiza a troca de indivíduos. Esta versão paralela implementa dois tipos de migração: migração síncrona e migração assíncrona.

O primeiro tipo de migração é um modelo síncrono mais flexível que o modelo síncrono da versão em memória partilhada, no sentido que este não utiliza uma "barreira" entre as duas fases da migração, ao que leva a que nem todas as ilhas tenham de esperar na geração estabelecida para a realização da migração. Neste tipo de migração, não existe uma "barreira" para todos os processos, em que cada ilha após receber os indivíduos das ilhas vizinhas pode continuar a sua execução. Isto é possível com a utilização do método *Isend* para enviar mensagens assíncronas e do método *Recv* do *MPI* que realiza uma espera até a que mensagem seja recebida.

Para o tipo de migração assíncrona, os indivíduos são enviados através do método *Isend* do *MPI*, onde a receção das mensagens que contêm os indivíduos da migração é flexível da mesma forma que o tipo de migração assíncrono do modelo em memória partilhada, no sentido que pode continuar a sua execução do AE mesmo no caso em que estas mensagens ainda não foram recebidas na geração pretendida. Na fase onde deverá ser realizada a receção destas mensagens durante a migração, é utilizado o método *Iprobe* do *MPI* que irá retornar o estado das mensagens a receber e através deste estado saber quais as mensagens é que estão prontas a serem recebidas. Na geração onde é suposto se realizar a migração, apenas as mensagens prontas a serem recebidas é que são aplicadas as trocas de indivíduos, continuando a execução do AE. Posteriormente, para cada ilha, se todas as mensagens não foram recebidas durante a fase de migração, em todas as gerações do AE seguintes é verificado através do método *Iprobe* se as mensagens por receber estão prontas a serem recebidas, de forma a aplicar a troca de indivíduos.

3.4 Modelo Híbrido

Como o título desta dissertação sugere, o objetivo principal passa por paralelizar massivamente os algoritmos em estudo, AEs, na plataforma *JECoLi*. Para atingir este objetivo é necessário implementar um modelo híbrido de forma a realizar testes com o máximo de paralelismo possível, utilizando mais processamento através da criação de *threads* e processos a executar em diferentes processadores físicos.

O modelo de ilhas tem limitações no paralelismo na medida em que divide o processamento de um AE, dividindo por gerações ou por indivíduos, onde utilizando um número elevado de ilhas iria prejudicar o bom funcionamento do AE pois haverá casos onde o número de gerações é insuficiente para atingir bons resultados e haverá casos onde o número de indivíduos por ilha é demasiado baixo para existir diversidade em cada ilha.

Por outro lado, o modelo de paralelismo global é geralmente aplicado em sistemas de memória partilhada pelo que o uso deste modelo centralizado em memória distribuída iria perder grande parte da sua performance na comunicação, que por sua vez teria de ser realizada em cada iteração/geração no envio de soluções/indivíduos para outros processadores para a posterior avaliação dessas soluções e no envio do resultado de volta para o processador principal responsável pela execução do AE.

Assim sendo, para a obtenção de melhores resultados, tanto a nível de tempo de execução como a nível de qualidade das soluções finais, implementou-se um modelo híbrido onde na camada exterior se tem o modelo de ilhas e na camada interior o modelo de paralelismo global. Com este modelo híbrido referido conseguem-se teoricamente diversas vantagens:

1. Com o modelo de ilhas consegue-se muitas vezes melhorar os valores de aptidão obtidos comparados aos da versão centralizada. Isto deve-se à diversidade entre as ilhas (devido ao isolamento de um número de soluções que apenas podem "interagir" entre si) e à migração que permite a troca duma percentagem de indivíduos definida onde os piores indivíduos poderão ser descartados sendo substituídos por melhores soluções;

2. Ao incluir o modelo de paralelismo global na fase de avaliação de indivíduos de cada ilha do modelo de ilhas, possibilita o uso de mais recursos computacionais em ambientes de memória distribuída, podendo ter ilhas em diferentes máquinas onde a fase computacionalmente mais exigente (*e.g.* avaliação de indivíduos) é por sua vez paralelizada utilizando um número de *threads*.

O desenvolvimento desta implementação foi bastante simples pois baseia-se na utilização das implementações já apresentadas que tiveram também em consideração no seu desenvolvimento a criação deste modelo híbrido. A *Figura 12* expõe, com algumas simplificações, as alterações necessárias para utilizar o modelo em causa.

A estrutura do código para executar o modelo híbrido é semelhante ao código utilizado para executar o modelo de ilhas. Tal como se sugere na figura, as ações indicadas a azul e a cor de laranja são as únicas alterações necessárias de modo a utilizar também o modelo de paralelismo global. Na ação a azul, o programa recebe não só os parâmetros do modelo de ilhas mas também o número de *threads* desejadas para realizar o modelo de paralelismo global em cada ilha.

Em seguida, após definidas as configurações de todas as ilhas, na criação dos AEs, deverá ser utilizado o construtor definido no modelo de paralelismo global de forma a utilizar este modelo passando como parâmetro o número de *threads* a realizar naquele AE. Posteriormente, é necessário realizar todos os passos mencionados na implementação do modelo de ilhas de forma a executar o modelo híbrido.

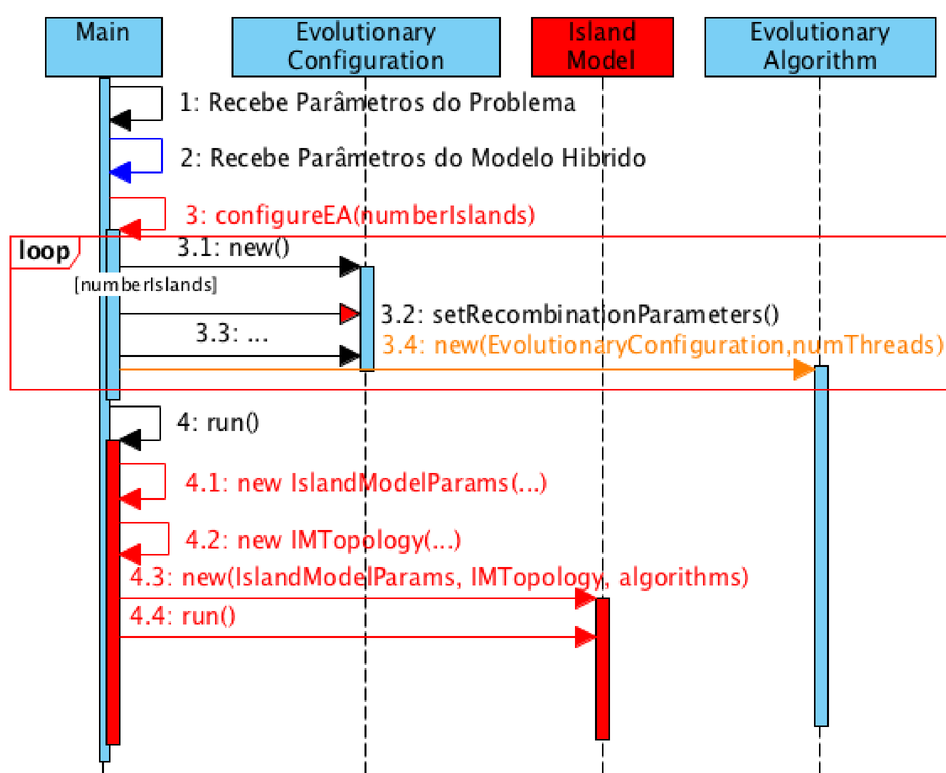

Figura 12: Exemplo de diagrama de sequência da execução de AEs na plataforma *JECoLi* usando o modelo híbrido.

4 Resultados

4.1 Casos de Estudo

As implementações realizadas nesta tese foram avaliadas com dois casos de estudo desenvolvidos anteriormente na plataforma *JECoLi*: um problema de otimização de bioprocessos de fermentações *fed-batch*[55] e um problema de otimização da qualidade de serviços de redes através da configuração dos protocolos de encaminhamento[56]. Em seguida ambos os casos de estudo serão apresentados.

4.1.1 Fermentação *fed-batch*

A fermentação *fed-batch* é uma técnica operacional utilizada em processos biotecnológicos onde vão sendo adicionados nutrientes ao biorreator durante todo o processo, levando a que o estado do sistema mude ao longo do tempo. O objetivo do uso de CE neste caso de estudo é controlar o estado ótimo do bioprocesso, encontrando a melhor taxa de alimentação de substrato possível para a cultura em causa, de modo a obter o máximo de produtividade com um custo mínimo.

A codificação dos indivíduos nos AEs consiste de um vetor de números reais, cada um codificando uma variável de entrada como uma sequência de valores de alimentação ao longo do tempo. Estes valores são fornecidos em pontos igualmente espaçados no tempo, enquanto os valores restantes são interpolados linearmente.

A função de avaliação é definida utilizando modelos matemáticos baseados em equações diferenciais ordinárias que representam o balanço das variáveis de estado. A avaliação dos indivíduos é realizada através de uma simulação deste modelo, tendo como entrada os valores da alimentação, usando um método de integração numérica. O valor de aptidão é calculado após a simulação, através dos valores calculados das variáveis de estado e de acordo com uma a função definida, medindo a produtividade do processo.

4.1.2 *Open Shortest Path First*

Open Shortest Path First (OSPF) é um protocolo de encaminhamento usado em infraestruturas de qualidade de serviço de redes de computadores como estratégia capaz de controlar o caminho seguido pelos dados que atravessam um dado domínio. O problema é tratado como um grafo, em que o administrador de rede atribui pesos para cada ligação da rede que são utilizados para calcular o melhor caminho de cada fonte para cada destino, através do algoritmo de *Dijkstra*[57]. Os resultados deste método são então utilizados para calcular as tabelas de encaminhamento em cada nó. Este protocolo é importante na medida

em que possibilita que os administradores de rede possam controlar o comportamento da rede através desta configuração de pesos, podendo medir o seu desempenho.

O objetivo da utilização de AEs neste problema é determinar os melhores pesos a serem usados pelo OSPF, que otimizam o congestionamento do tráfego da rede e ao mesmo tempo possam cumprir com requisitos específicos de atrasos. Cada indivíduo do AE é representado por um vetor de valores inteiros, onde cada valor(gene) corresponde ao peso de um link na rede(o número de genes equivale ao número máximo de ligações na rede).

A função de avaliação é definida por um modelo matemático que define funções de custo flexíveis que têm em conta várias medidas do comportamento da rede, tais como o congestionamento da rede e atrasos ponto-a-ponto. Sendo uma função de minimização, os valores de aptidão atribuídos às soluções do AE representam o custo da configuração estabelecida pela solução no sistema de rede face aos comportamentos referidos.

4.2 Ambiente Experimental

Os testes realizados às quatro implementações paralelas foram realizados no *cluster SeARCH* [58] do Departamento de Informática da Universidade do Minho, utilizando dois tipos de máquinas *NUMA*. As especificações destas máquinas encontram-se na *Tabela 1*.

Nome da Máquina	**compute-662**	**compute-641**
Modelo	Intel Xeon E5-2695 v2	Intel Xeon E5-2650 v2
# Sockets	2	2
# CPUs	24	16
# CPUs por Socket	12	8
Hyper-Threading	Sim	Sim
Frequência	2.4 GHz	2.6 GHz
Cache L1	32 KB + 32 KB	32 KB + 32 KB
Cache L2	256 KB	256 KB
Cache L3 Partilhada	30 MB	20 MB
RAM Total	64 GB	64 GB

Tabela 1: Especificação das máquinas do *SeARCH* utilizadas para a realização dos testes.

Ambas as máquinas são configuradas com frequência fixa, e possuem *hyper-threading*, o que significa que cada processador poderá executar duas *threads* em simultâneo. Nem sempre o uso desta tecnologia traz bons resultados, na medida em que a sua eficiência depende do código a executar.

Com o objetivo de compreender melhor a topologia das máquinas *NUMA* mencionadas

foi executado o comando *lstopo* do *Linux* tendo como resultado a *Figura 51* e a *Figura 52* em anexo.

Os códigos produzidos foram executados utilizando a versão *1.8.0* do Java, com as *flags* *-XX:+UseParallelGC -XX:+UseNUMA* de modo a possibilitar o uso de otimizações de desempenho para máquinas com a arquitetura *NUMA*.

4.3 Testes de Desempenho : Fermentação *fed-batch*

Nesta subsecção, são apresentados os testes realizados às implementações paralelas utilizando o caso de estudo de fermentação *fed-batch* fornecido, tendo em conta o ambiente experimental apresentado.

4.3.1 Análise do Caso de Estudo

O código fornecido relativo à fermentação *fed-batch* está dividido em três classes principais: *FermentationOptimization*, *FermentationEvaluation* e *FermProcess*.

A primeira classe, *FermentationOptimization*, representa a execução sequencial do problema na plataforma *JECoLi*, que recebe os parâmetros para este problema, realiza a configuração do algoritmo a utilizar (AE, Evolução Diferencial ou *Simulated Annealing*) e, por fim, executa o algoritmo consoante esta configuração.

A classe *FermentationEvaluation* implementa a função de avaliação deste problema de fermentação estendendo a classe *AbstractEvaluationFunction* da *JECoLi*, contendo uma instância *FermProcess* que é utilizada na avaliação dos indivíduos. A classe *FermProcess* irá simular o bioprocesso de fermentação *fed-batch* de maneira a atribuir um valor de aptidão a essa solução.

No modelo de paralelismo global são criadas várias *threads* com o propósito de dividir o número de indivíduos a avaliar em cada geração, realizando a avaliação em paralelo. De modo a paralelizar a execução deste problema com o modelo de paralelismo global foi necessário adaptar este código, pois a classe *FermProcess* não é *thread-safe* pois não permite que várias *threads* realizem simulações da fermentação em simultâneo.

Consequentemente, existem duas alternativas para a resolução desta situação: a primeira seria utilizar *locks* nas partes do código necessárias; a segunda seria cada *thread* possuir a sua instância de *FermProcess*, realizando independentemente a simulação dos seus indivíduos a avaliar.

A primeira alternativa foi descartada devido à complexidade das dependências no código desta classe. Por sua vez, a segunda alternativa foi implementada utilizando a estrutura *HashMap* do *java.util* em que a chave é um inteiro que corresponde ao identificador da *thread* criada pelo *Executor* e o valor corresponde a uma instância de *FermProcess*, cópia

da instância original. Esta alteração não foi necessária para a utilização do modelo de ilhas visto que este modelo paralelo executa AEs independentes.

O código deste caso de estudo implementa diferentes tipos de bioprocessos de fermentação, dois tipos de simulações e necessita de diferentes parâmetros para a execução do AE consoante o tipo de fermentação escolhido. Os testes realizados neste caso de estudo utilizam o tipo de fermentação com bactérias *E. coli* e a simulação numérica utilizando o método *Runge–Kutta*. Os restantes parâmetros são o valor de interpolação e o tempo final, para os quais foram utilizados os valores de 200 e 25, respetivamente.

Para este processo de fermentação em concreto, em [55], foram realizados diferentes testes com diferentes algoritmos (*e.g.* Evolução Diferencial, AE e *Fully Informed Particle Swarm*), sendo o melhor valor de aptidão recolhido utilizando AEs de aproximadamente 9,05.

Os testes neste caso de estudo foram realizados utilizando AEs com 2000 gerações e 480 indivíduos, resultando em 480720 avaliações de indivíduos. De forma a fixar este número de avaliações para a implementação paralela do modelo de ilhas foi necessário alterar os parâmetros de recombinação colocando o valor de elitismo a um e o número de descendentes em cada geração igual a metade do número de indivíduos na população. Esta alteração foi necessária pois utilizando a divisão de processamento com a divisão dos indivíduos, verificou-se que o valor de elitismo tomava valores diferentes consoante o número de indivíduos da população e consequentemente alterava o número de descendentes que define o número de avaliações de indivíduos.

O número de indivíduos é adequado para os testes realizados em máquinas com 24 processadores (utilizando *hyper-threading* poderá utilizar 48 *threads*/processos) para testar o modelo de ilhas com o máximo de recursos computacionais possíveis, onde utilizando 48 *threads*/processos cada ilha ficará com 10 indivíduos na sua população.

Com a finalidade de realizar comparações entre a versão sequencial e as versões paralelas, a versão sequencial do problema de fermentação foi executada doze vezes, medindo o tempo de execução total, o tempo de execução gasto em avaliações de soluções e o melhor valor de aptidão obtido no final de cada execução.

Tempo de Execução na Avaliação(TEA)	2022,52
Tempo de Execução Total	2025,92
Valor de Aptidão	9,08

Tabela 2: Mediana de doze execuções da versão sequencial do problema de fermentação relativos aos tempos de execução totais (segundos), tempos de execução na avaliação (segundos) e valores de aptidão.

Os valores apresentados na *Tabela 2* são o resultado da mediana dos valores obtidos nas doze execuções. Conclui-se que a avaliação de indivíduos neste caso de estudo representa 99,83% do tempo de execução total, sendo a parte computacionalmente mais exigente na execução do AE.

A fim de realizar medições à memória utilizada pelas estruturas de dados utilizadas durante a execução dos modelos paralelos utilizou-se o código *open-source* de Dimitris Andreou[59]. Esta pequena *framework* tem duas componentes especializadas na travessia de objetos Java: a obtenção do total de objetos de cada tipo (*e.g. integer*, *double*) na travessia de um dado objeto; uma estimativa da memória utilizada por um objeto num determinado instante. Com esta ferramenta, usaram-se as duas componentes sobre os objetos de maior importância para as versões paralelas, *FermProcess* e *Island*.

Objeto	**Número de Indivíduos**	**Memória Usada**
FermProcess	-	354KB
Island	240 (duas ilhas)	576KB
	20 (vinte e quatro ilhas)	375KB

Tabela 3: Resultados das medições de memória aos objetos *FermProcess* e *Island*.

Na *Tabela 3* estão os resultados das medições de memória utilizando a segunda componente nos objetos *FermProcess* e *Island* obtidos durante a execução do modelo de paralelismo global (*FermProcess*) e duas execuções do modelo de ilhas (*Island*), uma execução com duas ilhas e outra execução com vinte e quatro ilhas. Realizaram-se duas execuções do modelo de ilhas, pois o objeto *Island* inclui todo o AE e a sua respetiva configuração, onde nas versões paralelas do modelo de ilhas irá conter diferentes números de indivíduos no caso da divisão do processamento através da divisão dos indivíduos. Com estes resultados, e tendo em conta o ambiente experimental apresentado, pode-se concluir que, em ambos os modelos, os dados utilizados cabem na *cache L3*, mesmo utilizando o máximo de *threads* por *socket* (9000KB com 24 objetos *Island* e 6000KB com 16 objetos *Island*).

As alterações necessárias foram realizadas na platorma *JECoLi* e uma nova classe foi implementada, *ParallelFermentationOptimization*, baseada na classe *FermentationOptimization* que consoante os parâmetros definidos executa uma das quatro versões paralelas ou a versão sequencial do problema.

Para cada configuração de testes realizadas no caso de estudo da fermentação foram realizadas seis execuções, sendo os valores apresentados ao longo desta subsecção o resultado da mediana das seis execuções.

4.3.2 Paralelismo Global

Neste problema de fermentação, a paralelização com o modelo de paralelismo global deverá representar uma boa solução para o problema visto que a maioria do tempo de execução é dedicado à avaliação de indivíduos.

#Threads	1	2	4	6	8	12	16	24	48
Ganho	1,00	1,98	3,83	5,65	7,54	10,71	13,39	14,50	21,56
Aptidão	9,07	9,07	9,08	9,07	9,00	9,07	9,10	9,10	9,07

Tabela 4: Tabela com os valores de ganho e valores de aptidão resultantes das execuções do modelo de paralelismo global no caso de estudo de fermentação *fed-batch*.

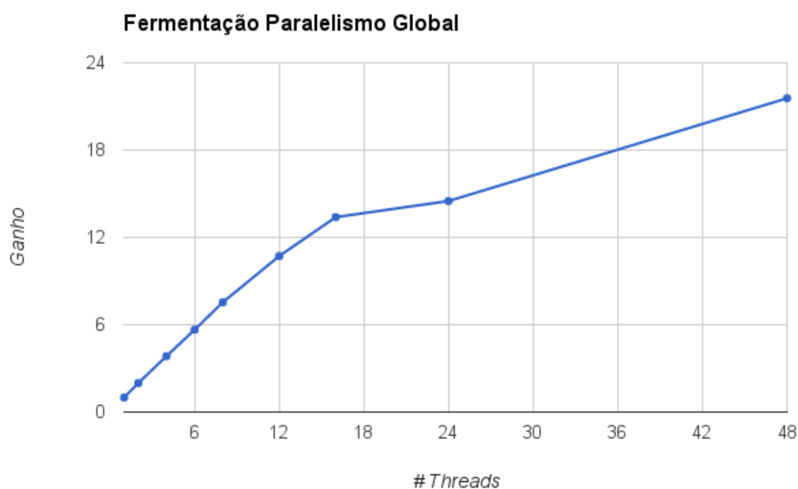

Figura 13: Resultados dos valores de ganho das execuções do modelo de paralelismo global no caso de estudo fermentação *fed-batch*.

Os gráficos da *Figura 13* e *Figura 14*, juntamente com a *Tabela 4*, contêm os resultados dos ganhos e valores de aptidão das execuções do modelo de paralelismo global usando 2, 4, 6, 8, 12, 16, 24 e 48 *threads* de execução nas máquinas *compute 662* com 24 unidades de processamento. Os valores de ganho são o resultado da divisão do tempo de execução total da versão sequencial (sem paralelismo) pelo tempo de execução total resultante das execuções paralelas pelo que são úteis para analisar a escalabilidade das versões paralelas.

Num caso ideal, os valores de ganho deveriam aproximar-se do número de *threads*/processos utilizados, caso não hajam limitações no sistema nem sobrecarga de paralelismo. Observando

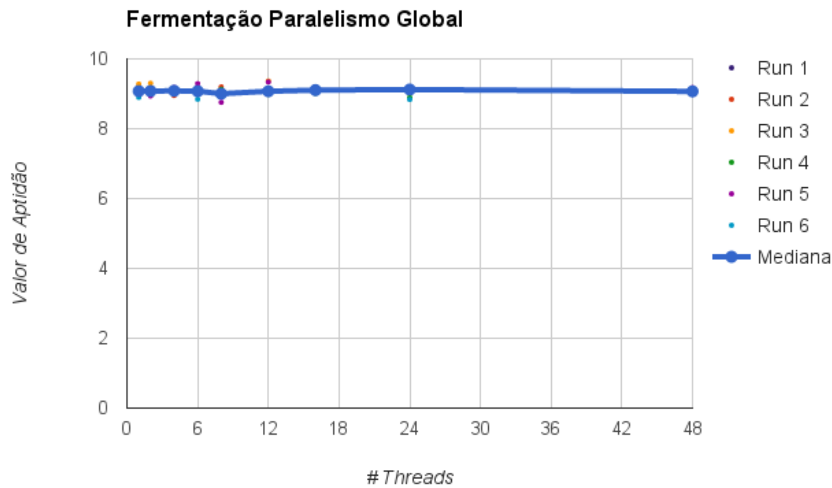

Figura 14: Resultados dos valores de aptidão das execuções do modelo de paralelismo global no caso de estudo fermentação *fed-batch*.

o gráfico da *Figura 13* pode-se concluir que os valores de ganho escalam bem até às 12 *threads*, enquanto que a partir das 16 *threads* os valores de ganho afastam-se dos valores ideais.

O gráfico da *Figura 14* mostra os resultados finais dos valores de aptidão das seis execuções realizadas para o conjunto de *threads* definidas, mostrando também o valor da mediana. Com estes valores pode-se concluir que os valores mantêm-se estáveis em comparação com a versão sequencial. Alguns valores de aptidão apresentados encontram-se desfasados dos restantes. Estes valores são normais porque os AEs são um processo estocástico.

Os valores na *Figura 15* são os resultados da eficiência do paralelismo, sendo esta uma percentagem que é calculada pela divisão do valor de ganho pelo número de *thread*. Estes valores são apresentados de modo a ter uma perceção da qualidade do paralelismo, pelo que se pode concluir que principalmente a partir das 16 *threads* a eficiência decresce significativamente.

As máquinas 662 possuem dois *sockets*, cada um com 12 unidades de processamento, pelo que até 12 *threads*, todas as *threads* criadas pelo *Executor* permanecem num só socket. Assumindo que os dados necessários (instâncias *FermProcess*) às *threads* residem na *cache L3* partilhada por todos os processadores do *socket*, os dados são facilmente acedidos.

Quando são utilizadas mais de 12 *threads*, as *threads* que são alocadas nos processadores do segundo *socket* (até 24 *threads*) são obrigadas em cada geração do AE a irem recolher os dados necessários à *cache L3* caso disponíveis, ou ao banco de memória do outro *socket*,

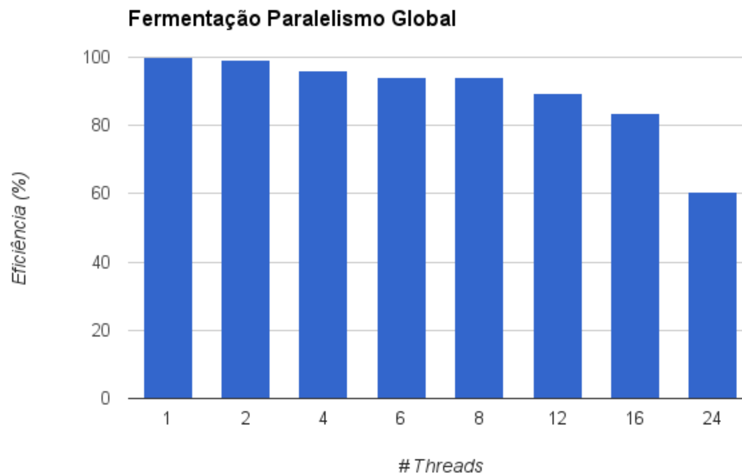

Figura 15: Resultados da eficiencia do paralelismo das execuções do modelo de paralelismo global no caso de estudo fermentação *fed-batch*.

podendo ser uma das razões para a menor escalabilidade.

Outra limitação nesta versão paralela é a dependência entre gerações, onde cada geração necessita da avaliação dos indivíduos da geração anterior. Isto faz com que exista uma "barreira" no final das avaliações dos indivíduos, onde a *thread* principal espera por todas as *threads* para prosseguir a execução.

Com o uso da tecnologia *hyper-threading* não é usual obter resultados lineares (geralmente o ganho obtido não é superior a 30%), normalmente porque em cada processador executam duas *threads* em simultâneo que irão partilhar os mesmos recursos. Com isto em consideração, os resultados com o uso de *hyper-threading* melhoram significativamente o desempenho da execução provavelmente porque atenua o efeito da arquitetura *NUMA*.

4.3.3 Modelo de Ilhas em Memória Partilhada

De maneira a realizar testes com as implementações paralelas do modelo de ilhas tem-se em consideração a topologia escolhida, os parâmetros de migração e o tipo de divisão de processamento, pois existem diversas configurações possíveis.

Um estudo relativo à topologia foi realizado por Cantú Paz e David Goldberg em [60], no qual concluem dois factos importantes: (i) todas as topologias com o mesmo grau (quantidade de ligações por ilha) resultam no mesmo tipo de convergência, ou seja, levam aos mesmos valores de aptidão; (ii) populações mais pequenas precisam de mais ligações para atingirem

bons valores de aptidão.

Dado o tipo de topologias implementadas nesta tese, os que têm maior grau são o tipo em anel bidirecional (grau 2) e o tipo totalmente conectado (grau n, sendo n o número de ilhas). No estudo referido, é mencionado que para o tipo totalmente conectado, na utilização de um número considerável de populações, o desempenho é afetado devido à comunicação associada.

Desta forma, e tendo o objetivo da obtenção do melhor desempenho computacional sem piorar os resultados de convergência e o objetivo de explorar estes algoritmos utilizando grandes recursos computacionais, é necessário uma configuração deste modelo que se adapte à utilização de um grande número de ilhas. Com isto em mente, todos os testes realizados utilizam uma topologia em anel bidirecional de forma a conseguir bons resultados de desempenho com o mínimo de comunicação entre as ilhas e resultados razoáveis nos valores de aptidão.

Outros parâmetros foram fixados para a execução dos testes neste modelo, sendo estes relativos à migração, mais especificamente na troca de indivíduos (seleção dos indivíduos a trocar e seleção dos indivíduos a serem substituídos). Mais uma vez, considerando as experiências de Cantú Paz em [37], a melhor configuração para obter melhor convergência será uma seleção estocástica que favoreça os melhores indivíduos a serem escolhidos para realizar a migração nas ilhas de origem, sendo substituídos na ilha destino os piores indivíduos. No caso de estudo em causa, foi escolhido o mesmo tipo de seleção estocástica utilizado para a seleção dos progenitores no ciclo principal do AE (seleção por torneio) como seleção dos indivíduos a migrar, sendo estes indivíduos substituídos pelos piores indivíduos.

Dividir Gerações ou Dividir Indivíduos

Duas abordagens foram consideradas de modo a fixar o número de avaliações: dividir gerações e dividir indivíduos. Na abordagem de dividir gerações permanece o mesmo número de indivíduos em todas as ilhas, dividindo o número de gerações inicial pelo número de ilhas. Na segunda abordagem permanece o mesmo número de gerações em todas as ilhas (2000 gerações) dividindo o total de indivíduos (480 indivíduos) pelo número de ilhas.

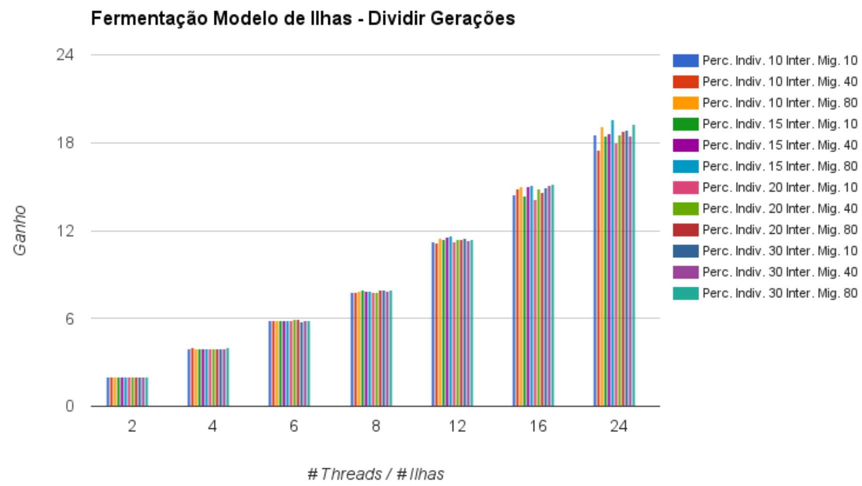

Figura 16: Resultados dos valores de ganho da abordagem de dividir gerações do modelo de ilhas em memória partilhada utilizando diferentes configurações e diferentes números de ilhas no caso de estudo fermentação *fed-batch*.

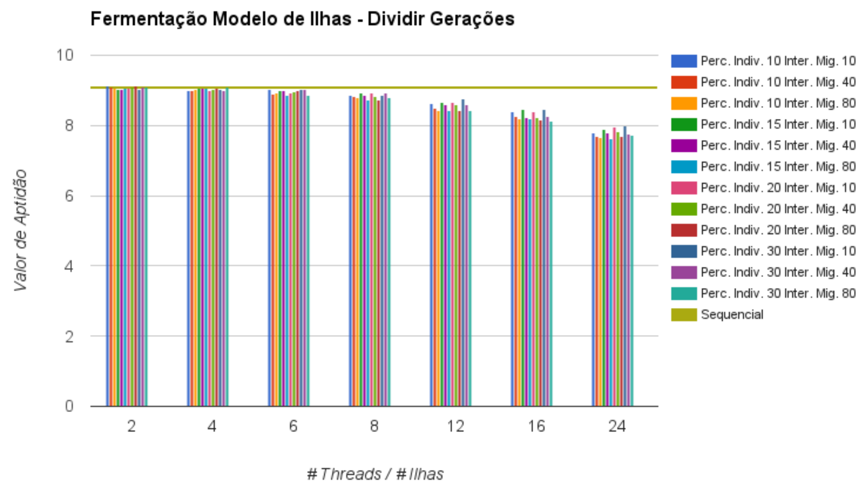

Figura 17: Resultados dos valores de aptidão da abordagem de dividir gerações do modelo de ilhas em memória partilhada utilizando diferentes configurações e diferentes números de ilhas no caso de estudo fermentação *fed-batch*.

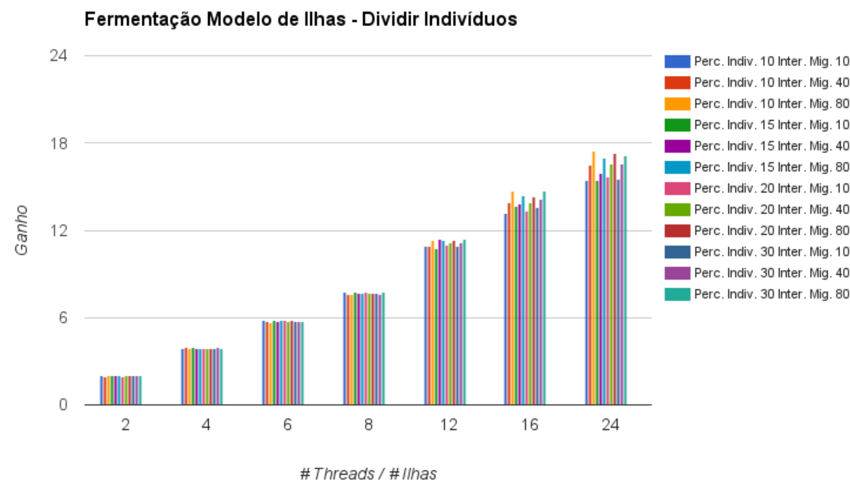

Figura 18: Resultados dos valores de ganho da abordagem de dividir indivíduos do modelo de ilhas em memória partilhada utilizando diferentes configurações e diferentes números de ilhas no caso de estudo fermentação *fed-batch*.

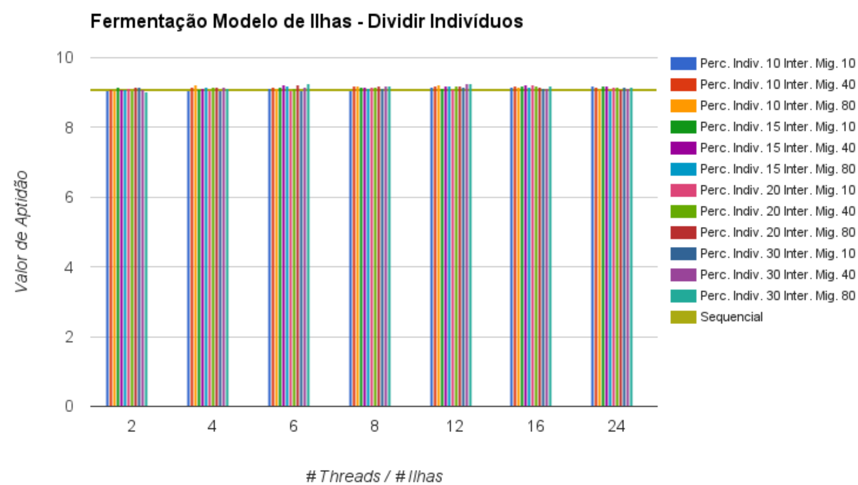

Figura 19: Resultados dos valores de aptidão da abordagem de dividir indivíduos do modelo de ilhas em memória partilhada utilizando diferentes configurações e diferentes números de ilhas no caso de estudo fermentação *fed-batch*.

Foram realizadas várias execuções destas abordagens na versão do modelo de ilhas em memória partilhada, utilizando uma máquina 662, com diferentes configurações tendo em conta os parâmetros de migração (intervalo de migração e percentagem de indivíduos a migrar). Para cada configuração foram usados diferentes intervalos de migração, sendo a migração realizada em cada 10, 40 e 80 gerações e diferentes percentagens de indivíduos a migrar de 10%, 15%, 20% e 30%. Estas configurações foram executadas utilizando diferentes números de ilhas: 2, 4, 6, 8, 12, 16 e 24.

Nos gráficos apresentados (*Figuras 16, 17, 18 e 19*) estão os resultados destas execuções, sendo os resultados que utilizam uma ilha relativos aos resultados obtidos na execução da versão sequencial apresentados anteriormente de forma a comparar a versão sequencial com a versão paralela.

Com estes resultados, pode-se concluir que na abordagem de dividir gerações obtêm-se melhores valores de ganho embora se obtenham piores valores de aptidão com o aumento do número de ilhas. Estes resultados devem-se à diminuição das gerações com o aumento do número de ilhas, onde o número de gerações é demasiado baixo para atingir os valores de aptidão desejados.

Desta forma, exclui-se esta abordagem pois a paralelização deve manter ou aumentar a aptidão, uma vez que esta é a medida do desempenho do AE no problema. Já a abordagem de dividir indivíduos obteve ligeiramente piores valores de ganho, conseguindo melhorar os valores de aptidão das execuções sequenciais. Assim, a partir destes resultados apenas será utilizada a abordagem de dividir indivíduos para a execução dos testes relativos ao modelo de ilhas.

Migração Síncrona ou migração Assíncrona

Para os dois tipos de migração na versão paralela em memória partilhada, síncrono e assíncrono, foram realizados testes utilizando 12, 24 e 48 ilhas de forma a analisar a escalabilidade deste modelo utilizando um e dois *sockets*, assim como a utilização de *hyper-threading*. Estes testes utilizam diferentes configurações com diferentes valores do intervalo de migração, migrações em cada 10, 20, 40, 60, 80, 160 e 320 gerações, fixando o valor de percentagem de indivíduos a migrar a 10%. Outro teste foi realizado sem a utilização de migração, com a finalidade de verificar o peso da comunicação neste modelo, assim como quantificar o impacto da migração na convergência das soluções.

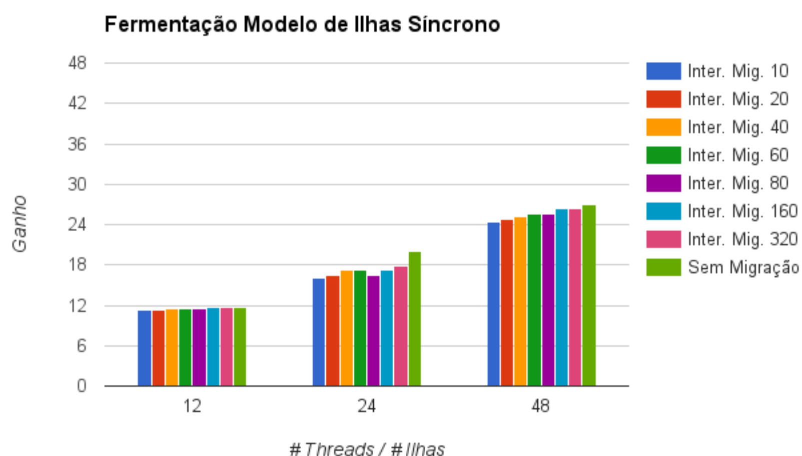

Figura 20: Resultados dos valores de ganho das execuções ao modelo de ilhas síncrono em memória partilhada utilizando diferentes intervalos de migração e diferentes números de ilhas no caso de estudo fermentação *fed-batch*.

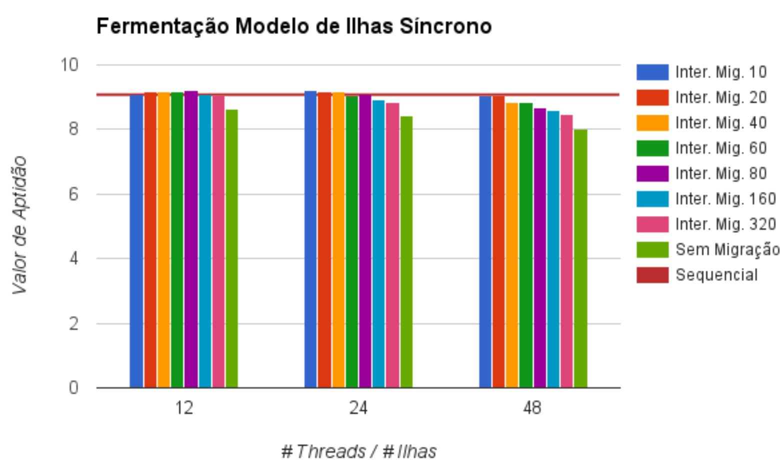

Figura 21: Resultados dos valores de aptidão das execuções ao modelo de ilhas síncrono em memória partilhada utilizando diferentes intervalos de migração e diferentes números de ilhas no caso de estudo fermentação *fed-batch*.

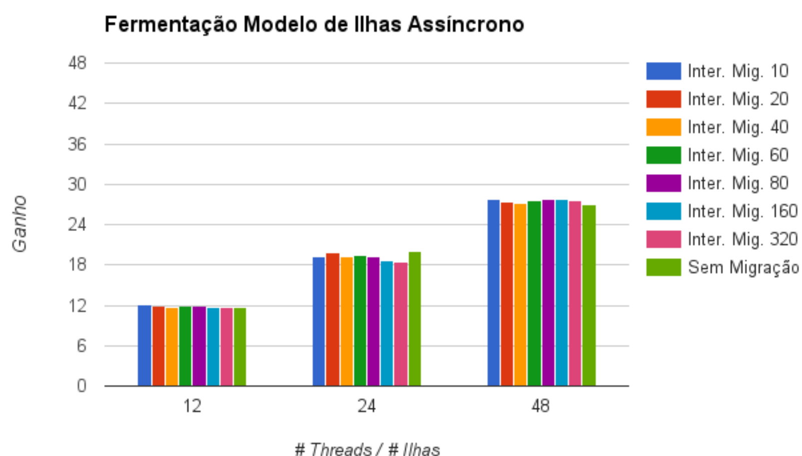

Figura 22: Resultados dos valores de ganho das execuções ao modelo de ilhas assíncrono em memória partilhada utilizando diferentes intervalos de migração e diferentes números de ilhas no caso de estudo fermentação *fed-batch*.

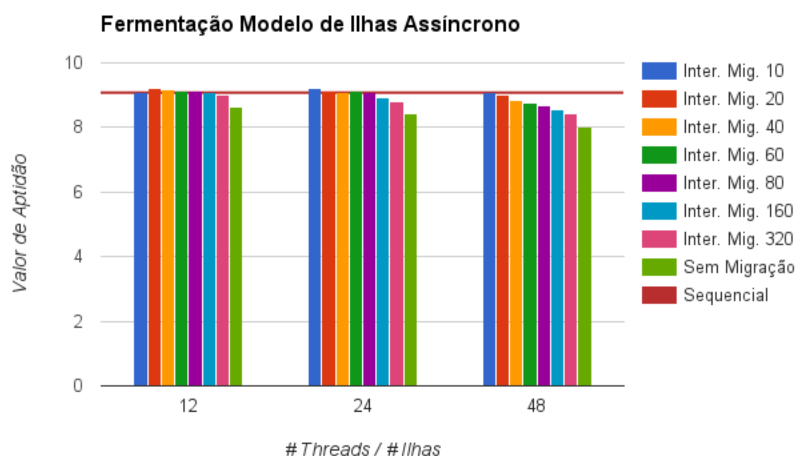

Figura 23: Resultados dos valores de aptidão das execuções ao modelo de ilhas assíncrono em memória partilhada utilizando diferentes intervalos de migração e diferentes números de ilhas no caso de estudo fermentação *fed-batch*.

Os gráficos apresentados (*Figura 20*, *Figura 21*, *Figura 22* e *Figura 23*), representam os resultados obtidos dos testes mencionados às duas versões do modelo de ilhas em memória partilhada, relativos aos valores de ganho e aptidão.

Analisando os valores de ganho, nos gráficos da *Figura 20* e da *Figura 22*, pode-se concluir que os valores têm comportamentos semelhantes aos do modelo de paralelismo global, onde utilizando 12 *threads* obtém-se um ganho linear, com 24 *threads* afasta-se do valor ideal e utilizando *hyper-threading* obtêm-se melhorias significativas. Na versão síncrona (*Figura 20*), verifica-se o impacto da comunicação que aumenta com a diminuição do intervalo de migração definido, penalizando os valores de ganho. Por outro lado, na versão assíncrona (*Figura 22*), o mesmo não se verifica conseguindo independentemente do intervalo de migração obter valores de ganho similares ao teste sem migração.

Os resultados dos valores de aptidão, nos gráficos da *Figura 21* e da *Figura 23*, mostram o quão importante é a migração para este modelo estruturado, onde o valor de aptidão mais baixo é relativo ao teste sem migração e diminuindo o intervalo de migração obtém-se melhores resultados. Isto quer dizer que quanto mais populações existirem menor serão as populações (número de indivíduos), levando a que a interação dos indivíduos de cada ilha seja mais restrita e evoluindo mais lentamente. Assim, com o aumento do número de ilhas é necessário aumentar a quantidade de migrações entre ilhas de forma a aumentar a interação e troca de indivíduos e obter valores de aptidão aceitáveis perante a versão sequencial do AE. Por outro lado, para um menor número de ilhas e consequentemente mais indivíduos por população, a convergência poderá ser penalizada aquando utilizados valores de intervalos de migração muito baixos, pois são necessárias mais gerações entre migração de forma a existir uma evolução/interação mais solida/abrangente em cada população.

Ao comparar os resultados do tipo de migração síncrono com o tipo de migração assíncrono, conclui-se que o tipo assíncrono consegue melhores valores de ganho, sendo os valores de aptidão recolhidos bastante similares. Com isto, o melhor resultado obtido foi da versão assíncrona aplicando migrações a ocorrer de 10 em 10 gerações em 48 ilhas, com um valor de ganho de 27,73 e um valor de aptidão de 9,08, superando os resultados obtidos no modelo de paralelismo global.

Face aos objetivos pretendidos com esta dissertação, realizou-se mais um conjunto de testes ao modelo de ilhas em memória partilhada com a versão assíncrona com o objetivo de verificar o impacto da percentagem de indivíduos a utilizar na migração. Para tal, realizaram-se testes com configurações diferentes, utilizando intervalos de migração a cada 10 e 20 gerações, e percentagens de indivíduos a migrar de 10%, 20%, 30%, 40%, 50% e 100%.

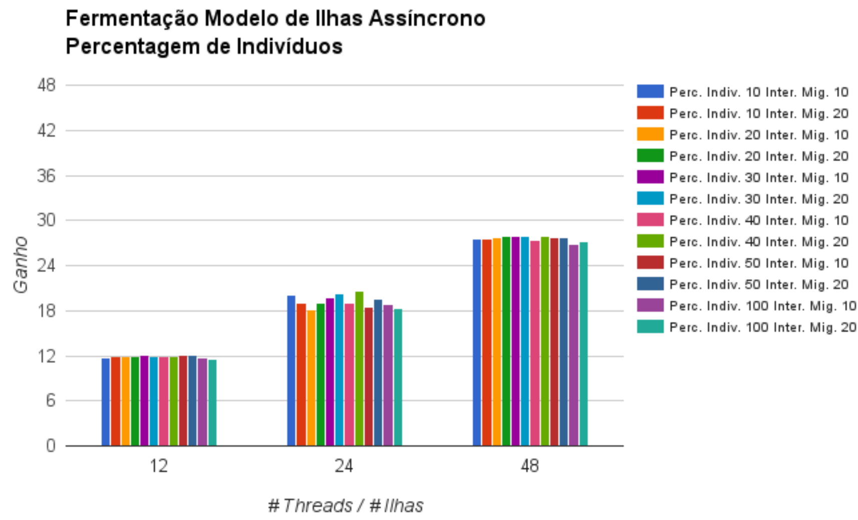

Figura 24: Resultados dos valores de ganho das execuções ao modelo de ilhas assíncrono em memória partilhada utilizando diferentes intervalos de migração, percentagem de indivíduos a migrar e números de ilhas no caso de estudo fermentação *fed-batch*.

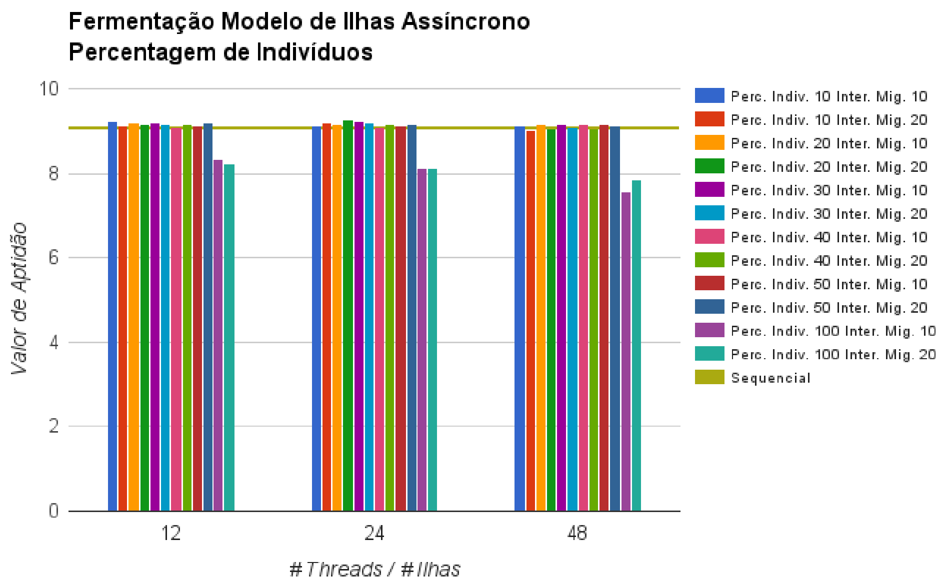

Figura 25: Resultados dos valores de aptidão das execuções ao modelo de ilhas assíncrono em memória partilhada utilizando diferentes intervalos de migração, percentagem de indivíduos a migrar e números de ilhas no caso de estudo fermentação *fed-batch*.

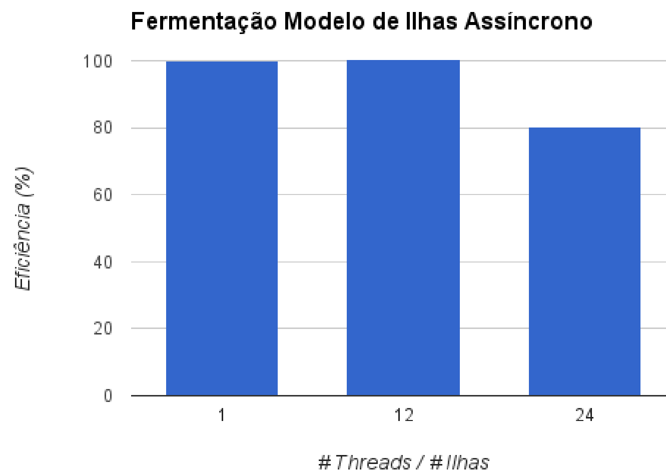

Figura 26: Resultados dos valores da eficiência do paralelismo para a melhor execução do modelo de ilhas assíncrono em memória partilhada com migração em cada 10 gerações no caso de estudo fermentação *fed-batch*.

Os gráficos da *Figura 24* e da *Figura 25* são a representação dos resultados dos testes mencionados. Observa-se pelos resultados dos valores de ganho que não há diferenças significativas com a alteração das percentagens de indivíduos a migrar. O mesmo se verifica nos valores de aptidão, exceto para o caso em que 100% dos indivíduos são migrados onde não existe tanta diversidade entre as ilhas, pois todos os indivíduos de cada ilha são substituídos por todos os indivíduos das ilhas vizinhas. Com isto, conclui-se que, para este caso de estudo, a alteração do parâmetro de percentagem de indivíduos na migração não é significativa para os valores de ganho e de aptidão.

Por fim, calcularam-se os valores da eficiência do paralelismo, apresentados na *Figura 26*, para a combinação com melhor resultado: modelo de ilhas assíncrono com 10% de indivíduos a migrar e migrações em cada 10 gerações.

4.3.4 Modelo de Ilhas em Memória Distribuída

Os testes realizados ao modelo de ilhas em memória distribuída têm a particularidade de distribuir o processamento por diversas máquinas, onde cada processo *MPI* cria uma instância do programa principal, possuindo portanto uma *Java Virtual Machine*(JVM) a executar o código Java deste modelo paralelo. Os testes a este modelo foram igualmente executados nas máquinas 662, utilizando uma e duas máquinas deste tipo. Exclui-se a pos-

sibilidade de utilizar mais máquinas devido à limitação do modelo de ilhas, pois utilizando uma população de 480 indivíduos, a divisão de processamento em mais de 48 ilhas (número de núcleos de processamento existentes em duas máquinas 662) resulta num número de indivíduos por população que não será aceitável para a obtenção de bons valores de aptidão.

Tendo em conta as conclusões dos resultados do modelo de ilhas em memória partilhada, os testes realizados ao modelo de ilhas em memória distribuída utilizam as mesmas configurações que foram fixadas anteriormente (abordagem dividir indivíduos, topologia em anel bidirecional, seleção por torneio nos indivíduos a migrar, seleção dos piores indivíduos a serem substituídos na migração e 10% de indivíduos como taxa de migração). Assim, as execuções desta versão paralela têm em conta apenas o intervalo de migração, existindo migração em cada 10, 20, 40, 60, 80, 160 e 320 gerações.

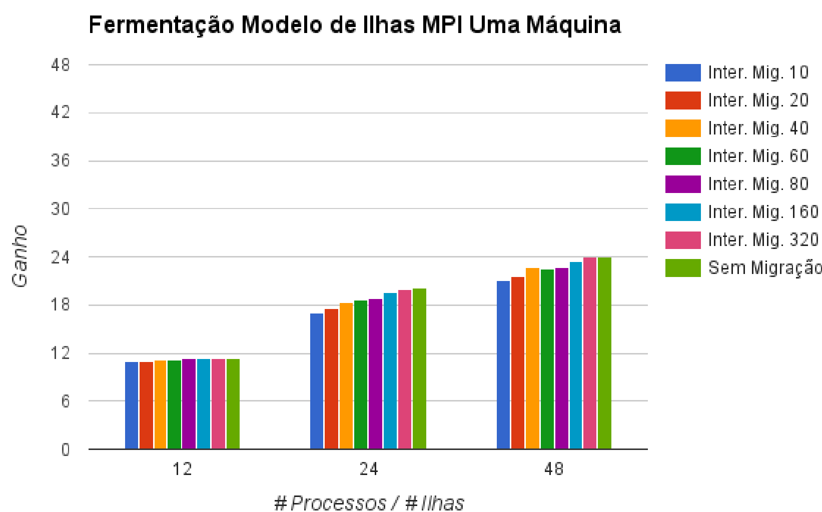

Figura 27: Resultados dos valores de ganho das execuções ao modelo de ilhas em memória distribuída com uma máquina 662 utilizando diferentes intervalos de migração no caso de estudo fermentação *fed-batch*.

Foram realizados testes utilizando uma máquina 662 com o tipo de migração síncrono, com o objetivo a comparar este modelo paralelo com os restantes apresentados tendo em conta as configurações descritas. Foi realizado também um teste sem migração para este modelo, estando os resultados dos valores de ganho e de aptidão apresentados na *Figura 27* e *Figura 28*, respetivamente.

Quanto aos valores de ganho, na utilização de 12 ilhas verifica-se valores lineares devido à utilização de um *socket* das máquinas 662. Os valores de ganho para 24 ilhas são semelhantes

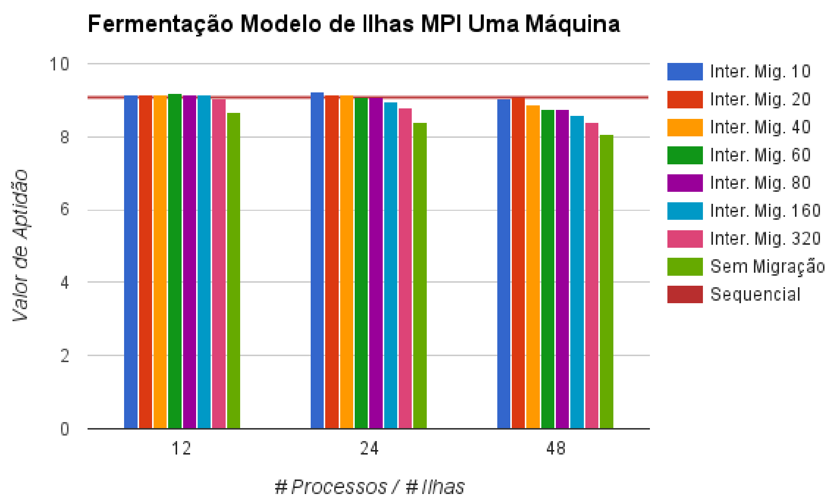

Figura 28: Resultados dos valores de aptidão das execuções ao modelo de ilhas em memória distribuída com uma máquina 662 utilizando diferentes intervalos de migração no caso de estudo fermentação *fed-batch*.

à versão paralela anterior, embora o mesmo não se verifica na utilização de *hyper-threading* alcançando um valor máximo de ganho de 24 no teste sem migração. Esta diferença no uso de *hyper-threading* (48 ilhas) é compreensível devido ao uso de processos, onde cada núcleo de processamento executa dois processos em simultâneo e cada processo executa o seu código Java através de uma *JVM*. Apesar desta versão paralela em memória distribuída ser uma versão síncrona mais flexível que a versão síncrona em memória partilhada, os valores de ganho revelaram-se mais baixos na utilização de uma máquina 662.

Relativamente aos valores de aptidão, observam-se bons resultados de modo semelhante à versão em memória partilhada. As diferenças do código paralelo relativamente à migração são relativas à maneira como os indivíduos são trocados entre as ilhas, não afetando os valores de aptidão finais.

No segundo conjunto de testes a este modelo paralelo, foram utilizadas duas máquinas 662 de maneira a analisar a escalabilidade de paralelismo utilizando núcleos físicos sem *hyper-threading* num máximo de 48 ilhas com os tipos de migração síncrona e assíncrona. Estas execuções têm o mesmo tipo de configurações do primeiro conjunto de testes, embora a distribuição dos processos pelas máquinas seja realizada de forma diferente. Isto é, metade do número de processos é executado na primeira máquina e a restante metade do número de processos é executada na segunda máquina.

Foi utilizada a forma de distribuição de processos por defeito do *MPI* de forma a que

primeiramente os processos preencham a primeira máquina e seguidamente a segunda máquina, minimizando a comunicação entre máquinas face à topologia escolhida (apenas dois processos de cada máquina comunicam entre máquinas). Para este conjunto de testes, a comunicação realizada pela biblioteca *MPI* para a comunicação entre máquinas utiliza ligações *ethernet*.

Os resultados apresentados na *Figura 29* e *Figura 30* são relativos aos valores de ganho e de aptidão com o tipo de migração síncrona. Os resultados dos valores de ganho mostraram-se superiores para um número de ilhas de 24 e 48, comparando ao primeiro conjunto de testes utilizando uma máquina 662. Para o caso de 24 ilhas, apenas um *socket* de cada máquina é utilizado, pelo que resulta em melhores tempos de execução, possivelmente devido à organização dos dados nos bancos de memória de cada máquina. Para o caso de 48 ilhas, os valores de ganho melhoram significativamente pois cada processo é executado num núcleo de processamento, sem a utilização de *hyper-threading*. Por fim, os resultados relativos aos valores de aptidão (*Figura 30*), possuem o mesmo padrão dos resultados apresentados no primeiro conjunto de testes.

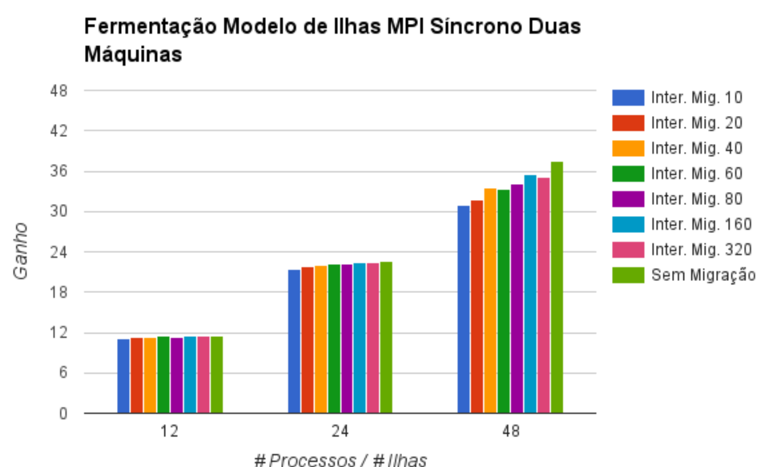

Figura 29: Resultados dos valores de ganho das execuções ao modelo de ilhas em memória distribuída síncrono com duas máquinas 662 utilizando diferentes intervalos de migração no caso de estudo fermentação *fed-batch*.

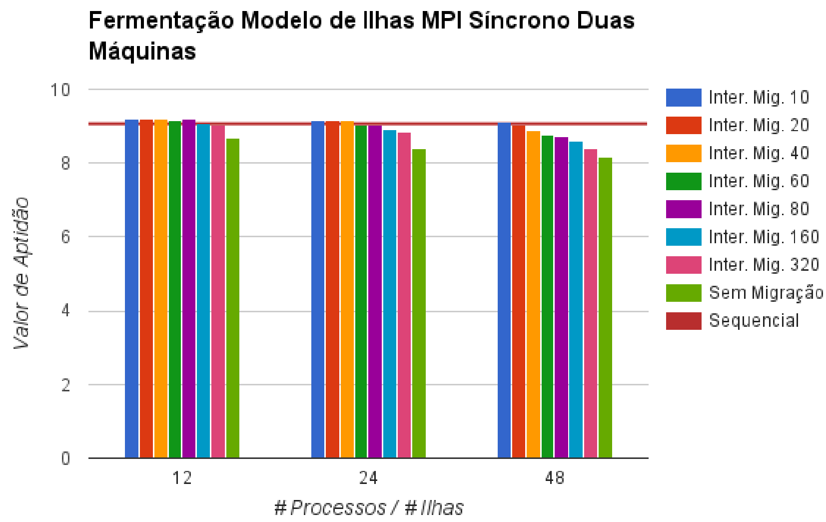

Figura 30: Resultados dos valores de aptidão das execuções ao modelo de ilhas em memória distribuída síncrono com duas máquinas 662 utilizando diferentes intervalos de migração no caso de estudo fermentação *fed-batch*.

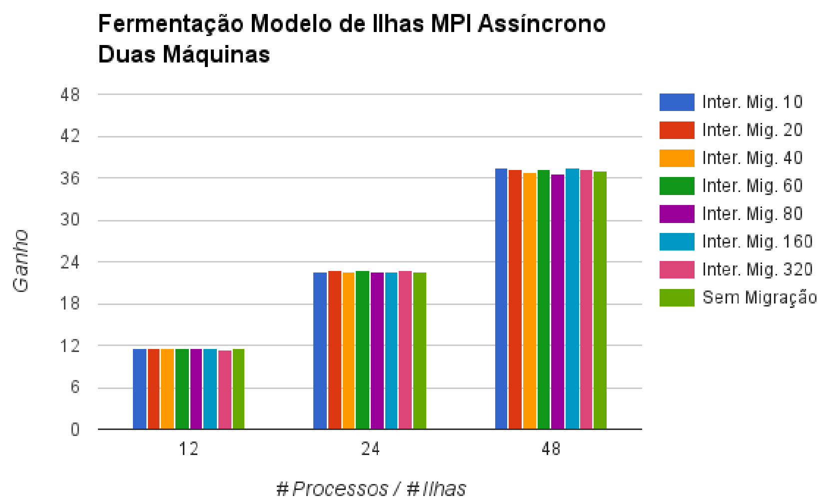

Figura 31: Resultados dos valores de ganho das execuções ao modelo de ilhas em memória distribuída assíncrono com duas máquinas 662 utilizando diferentes intervalos de migração no caso de estudo fermentação *fed-batch*.

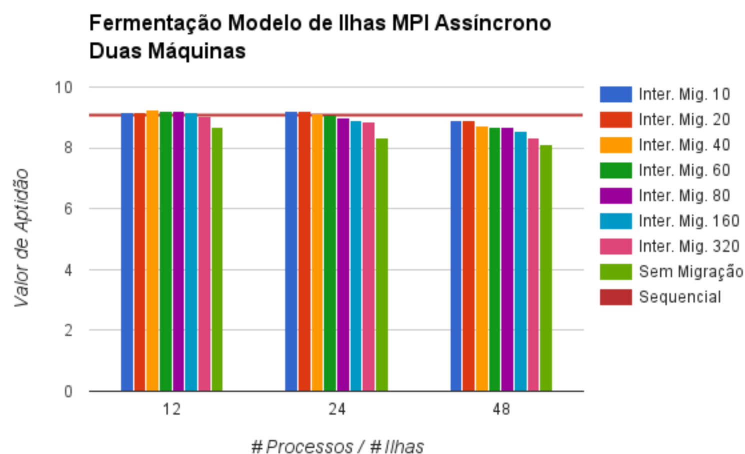

Figura 32: Resultados dos valores de aptidão das execuções ao modelo de ilhas em memória distribuída assíncrono com duas máquinas 662 utilizando diferentes intervalos de migração no caso de estudo fermentação *fed-batch*.

A *Figura 31* e *Figura 32* apresentam os resultados das execuções à versão assíncrona com duas máquinas 662. Observando os valores de ganho, *Figura 31*, verifica-se que estes aproximam-se dos valores de ganho do teste sem migração, pelo que se pode concluir que esta versão assíncrona consegue também suprimir o peso da comunicação, obtendo melhores resultados que a versão síncrona. Contrariamente, os valores de aptidão da *Figura 32* sugerem que a flexibilidade desta versão assíncrona tem uma penalização face a estes valores de aptidão, tal como se pode observar nos resultados das execuções com 48 ilhas. Com as semelhanças de implementação com o mesmo tipo de migração da versão paralela em memória partilhada, seria de esperar que os resultados fossem semelhantes embora tal não se verifique pois conceptualmente ambas são executadas em ambientes diferentes. Em ambas as implementações, os momentos (gerações) em que ocorrem as migrações são imprevisíveis, na medida que dependem do *hardware* e *software* onde são executadas. Nesta implementação em memória distribuída, existem migrações realizadas entre ilhas de diferentes máquinas pelo que pode existir maior latência associada na sua comunicação que é realizada através da biblioteca *MPI*, afetando a convergência das soluções.

Por fim, o melhor resultado recolhido desta versão paralela foi utilizando duas máquinas 662 e com o tipo de migração síncrona, com 48 ilhas e migrações em cada 10 gerações com um valor de ganho de 30,99 e um valor de aptidão de 9,11. Apesar dos bons resultados obtidos nos valores de ganho nos testes com tipo de migração assíncrono, não obtiveram bons resultados nos valores de aptidão, pelo que é preferível escolher a versão síncrona neste

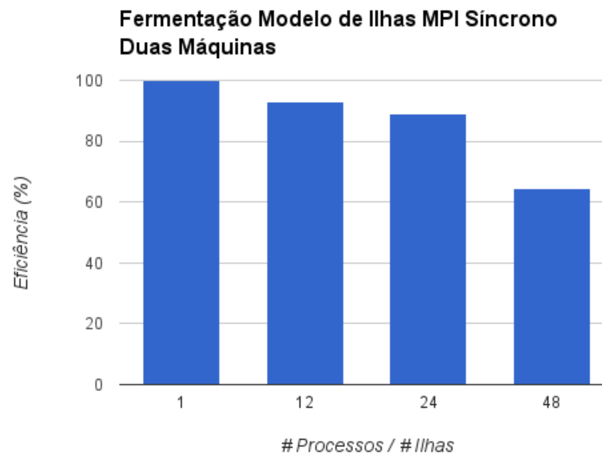

Figura 33: Resultados dos valores da eficiência do paralelismo para a melhor execução do modelo de ilhas em memória distribuída com migração síncrona em cada 10 gerações no caso de estudo fermentação *fed-batch*.

modelo em memória distribuída.

A *Figura 33* representa os valores de eficiência do paralelismo aos testes com duas máquinas e tipo de migração síncrono, onde se conclui que utilizando 24 ilhas distribuídas em duas máquinas se obtêm melhor desempenho (aproximadamente 88% de eficiência) e que utilizando 48 ilhas a eficiência baixa provavelmente devido ao uso completo dos dois sockets de cada máquina, reduzindo também o número de indivíduos e avaliações por ilha.

Um teste interessante para confirmar esta última suposição seria executar 48 processos utilizando quatro máquinas 662 de forma a utilizar apenas um socket em cada máquina. Este teste não foi realizado devido à impossibilidade de executar em quatro máquinas deste tipo.

4.3.5 Modelo Híbrido

Os testes relativos à implementação do modelo híbrido foram realizadas utilizando máquinas 641 para possibilitar testes num maior número de máquinas (8 máquinas). Tendo em conta que as máquinas 641 têm diferentes frequências de processador às máquinas 662, realizaram-se novamente doze execuções da versão sequencial nas máquinas 641 para possibilitar o cálculo dos valores de ganho neste modelo paralelo. Os resultados da mediana dos valores do tempo total de execução e de aptidão das doze execuções foram 1837,26 segundos e 9,08, respetivamente.

Foram realizados testes neste modelo utilizando 1, 2, 4 e 8 máquinas 641. Quanto à utilização do modelo de ilhas em memória distribuída como camada superior do modelo

híbrido, existe apenas um processo *MPI* por máquina utilizando 10% de indivíduos a migrar em cada 80 gerações numa topologia em anel bidirecional, com o tipo de migração síncrono. Para cada conjunto de testes são utilizadas 16 *threads* e 32 *threads* criadas na *thread pool* do modelo de paralelismo global em cada ilha (processo *MPI* / máquina 641), de forma a analisar a escalabilidade do modelo com e sem *hyper-threading*. Para o último conjunto de testes, utilizando 8 máquinas 641, apesar de se realizar testes com 32 *threads*, o número total de *threads* ultrapassa o número total de avaliações de indivíduos por geração, pelo que na realidade são utilizadas 30 *threads* em cada máquina que representa o máximo de paralelismo aplicável dadas as configurações (240 avaliações de indivíduos por geração e 240 *threads*).

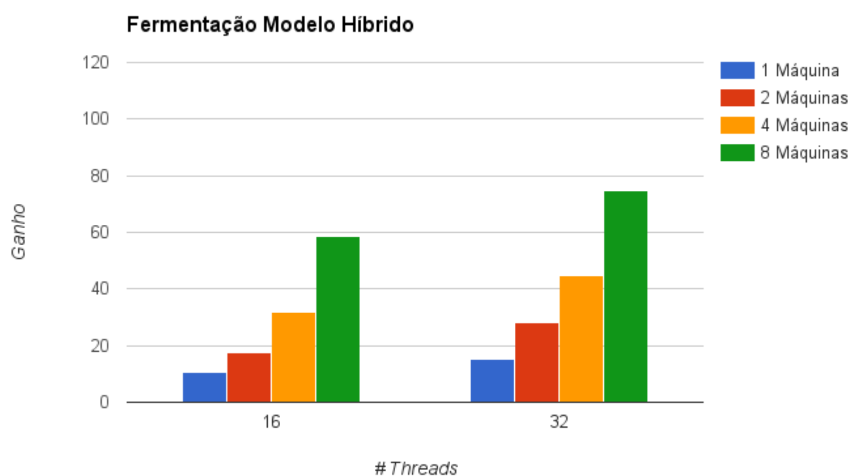

Figura 34: Resultados dos valores de ganho das execuções ao modelo híbrido com diferentes números de máquinas 641 utilizando uma ilha por máquina com 16 e 32 *threads* do paralelismo global no caso de estudo fermentação *fed-batch*.

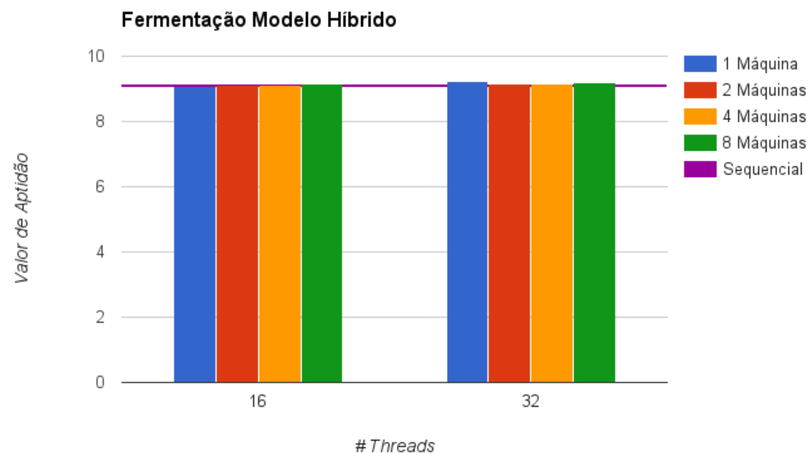

Figura 35: Resultados dos valores de ganho das execuções ao modelo híbrido com diferentes números de máquinas 641 utilizando uma ilha por máquina com 16 e 32 *threads* do paralelismo global no caso de estudo fermentação *fed-batch*.

Os gráficos da *Figura 34* e *Figura 35* apresentam estes resultados, nos quais a execução relativa a uma máquina é o resultado do uso exclusivo do modelo de paralelismo global. Observando o gráfico da *Figura 34*, verifica-se que os valores de ganho não são lineares relativamente à quantidade de *threads* e processos utilizados embora continue a escalar com o aumento do número de máquinas. Outra observação é o facto de, tal como esperado, se verificar parte do peso associado ao uso de *hyper-threading* neste modelo (*e.g.* as execuções com 16 *threads* com 2 máquinas 641 possuem o mesmo número de *threads* do paralelismo global comparadas às execuções com 32 *threads* com 1 máquina 641) .

Os resultados dos valores de aptidão, *Figura 35*, permanecem estáveis em comparação à versão sequencial pois a maior parte do paralelismo aplicado utiliza o modelo de paralelismo global que não influência estes valores de aptidão e o paralelismo por parte do modelo de ilhas é realizado com um número reduzido de ilhas conseguindo por vezes melhores valores de aptidão.

O melhor resultado obtido desta versão paralela é relativo às execuções em 8 máquinas com 32 *threads* por máquina, obtendo um valor de ganho de 74,95 e um valor de aptidão de 9,19. Com o objetivo de analisar a eficiência do paralelismo neste modelo, foram consideradas as execuções sem a utilização de *hyper-threading* (16 *threads*) relativas à utilização de 1, 2, 4 e 8 máquinas 641, assim como a execução que representa o máximo de paralelismo possível com 8 máquinas 641 e 32 *threads* do paralelismo global em cada uma destas máquinas. Com

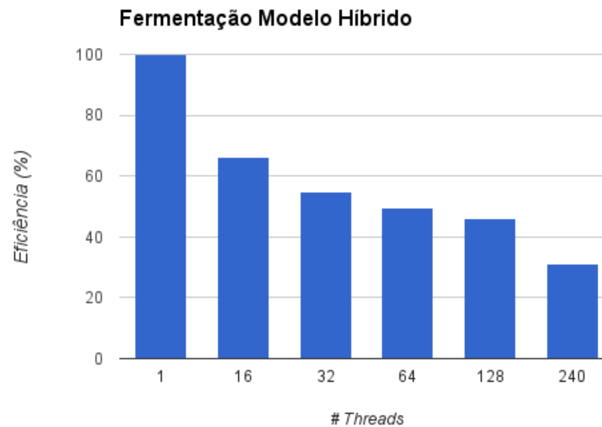

Figura 36: Resultados dos valores de eficiência do paralelismo das execuções ao modelo híbrido utilizando diferentes números de máquinas 641 utilizando uma ilha por máquina com 16 e 32 *threads* do paralelismo global no caso de estudo fermentação *fed-batch*.

os valores de eficiência apresentados na *Figura 36*, pode-se verificar uma redução linear destes valores que se justifica pelo uso do paralelismo global, tendo em conta a análise já realizada deste modelo, e pela diminuição do número de avaliações a realizar por cada *thread* em cada geração. Por exemplo, para o melhor caso obtido, apenas 30 das *threads* criadas realizam uma avaliação em cada geração, levando à diminuição de trabalho realizado por cada *thread*, onde os custos ao nível do tempo do paralelismo e acesso aos dados por parte dos processadores torna-se mais evidente. É de notar também que o valor de eficiência das 240 threads implica o uso de *hyper-threading* pelo que é de esperar que haja uma perda de eficiência associada.

4.4 Testes de Desempenho : *OSPF*

Nesta subsecção, são apresentados os testes realizados às implementações paralelas utilizando o caso de estudo *OSPF* fornecido, tendo em conta o ambiente experimental apresentado.

4.4.1 Análise do Caso de Estudo

O código relativo ao problema de otimização da qualidade de serviços de rede fornecido, está dividido em três classes principais semelhantes ao caso de estudo da fermentação:

EAOSPF representa a execução sequencial do problema na plataforma *JECoLi*, recebe os parâmetros relativos ao problema, realiza a configuração do AE e por fim executa-o.

OSPFEvaluationFunction implementa a função de avaliação do problema.

OSPF implementa a simulação para uma dada solução do problema, aplicando o algoritmo *Dijkstra* de cada fonte para cada destino para calcular os pesos para cada nó, atribuindo um valor de aptidão à solução conforme o custo da configuração estabelecida.

Para este caso de estudo, também existem dependências na utilização da mesma instância *OSPF* por diferentes *threads*, pelo que foi utilizada a mesma estratégia do caso anterior de atribuir uma instância *OSPF* a cada *thread* utilizando a estrutura *HashMap* do *java.util*.

Para a execução deste caso de estudo são necessários quatro parâmetros. O primeiro parâmetro consiste num nome que é associado a um conjunto de ficheiros de texto que contêm a informação da configuração da rede. O segundo parâmetro representa a média de congestão expectável em cada ligação da rede e o terceiro parâmetro representa o atraso expectável nas ligações da rede. O último parâmetro é uma percentagem relativa aos pesos que serão atribuídos ao custo por parte dos atrasos e por parte da congestão de forma a calcular os valores de aptidão das soluções.

Um aspeto a ter atenção neste caso de estudo é que se trata de um problema de minimização dos custos de congestão e/ou de atraso no sistema de rede, e que independentemente dos parâmetros recebidos o valor de aptidão normalizado ideal tem o valor de 1, onde em muitos dos casos pode não ser possível de alcançar tal valor[56].

Várias configurações foram fornecidas com diferentes tamanhos do problema, com números de nós e números de ligações bidirecionais diferentes. Na *Tabela 5* está a informação dos nomes dos ficheiros de configuração, o número de nós e o número de ligações para cada tamanho do problema.

Nome do Ficheiro	Número de nós	Número de ligações
isno_30_2	30	57
isno_50_4	50	190
isno_80_4	80	310

Tabela 5: Nome dos ficheiros de configuração, o número de nós e o número de ligações para cada tamanho do problema OSPF abordado.

Neste caso de estudo foram realizados dois conjuntos de testes. O primeiro conjunto de testes foi realizado com o objetivo de comparar os resultados do caso de estudo da fermentação com os resultados deste caso de estudo, realizaram-se testes ao modelo de paralelismo global e ao modelo de ilhas utilizando apenas a configuração de tamanho do problema mais pequena, com 30 nós. Para o segundo conjunto de testes, de forma a obter outras perspetivas sobre a

influência do tamanho do problema nos AEs e a sua escalabilidade, realizaram-se testes ao modelo híbrido com todas as configurações apresentadas.

Para o primeiro conjunto de testes, foi utilizado o nome *isno_30_2* como primeiro parâmetro. Para o segundo e terceiro parâmetros foram utilizados os valores de 0,2 e 3, respetivamente. Quanto ao quarto parâmetro foi utilizado um peso de 0,5, pelo que para o cálculo dos valores de aptidão será considerado 50% do custo de congestão e 50% do custo de atraso. No segundo conjunto de testes ao modelo híbrido utilizaram-se os mesmos parâmetros, sendo que para os restantes tamanhos do problema utilizaram-se diferentes ficheiros para o primeiro parâmetro conforme os nomes que a primeira coluna da *Tabela 5* apresenta.

Todos os testes realizados neste problema de otimização do protocolo *OSPF* utilizam, tal como no caso de estudo de fermentação, 480 indivíduos, em que para fixar o número de avaliações de indivíduos foi necessário realizar as mesmas alterações dos parâmetros de recombinação. Para os diferentes tamanhos do problema utilizou-se um número de gerações diferentes como critério de terminação, pelo que para o tamanho do problema com 30 nós utilizou-se 2000 gerações, para o de 50 nós utilizou-se 3000 gerações e para o de 80 nós utilizou-se 4000 gerações. Este número de gerações adaptou-se ao tamanho do problema de modo a dar o número de gerações necessárias para convergir em valores de aptidão aceitáveis.

Todos os valores apresentados nesta subsecção serão o resultado da mediana de 6 execuções, exceto para os valores da versão sequencial, sendo o resultado da mediana de 12 execuções.

Tempo de Execução na Avaliação(TEA)	657,49
Tempo de Execução Total	664,08
Valor de Aptidão	2,71

Tabela 6: Mediana de doze execuções da versão sequencial do problema do protocolo *OSPF* com 30 nós numa máquina 662, relativos aos tempos de execução totais (segundos), tempos de execução na avaliação (segundos) e valores de aptidão.

A *Tabela 6* representa as medianas de 12 execuções dos tempos de execução relativo às avaliações, dos tempos de execução totais e dos valores de aptidão para o tamanho do problema com 30 nós numa máquina 662 de forma a ter estes valores como referência para a versão sequencial no primeiro conjunto de testes. Com estes valores, conclui-se que 99% do tempo total é realizado na avaliação de indivíduos.

Para o segundo conjunto de testes, foram realizados os mesmos testes em máquinas 641 para todos os tamanhos do problema *OSPF*, representados na *Tabela 7*, de forma a ter referências das versões sequenciais para os testes ao modelo híbrido.

Tamanho do Problema	30	50	80
Tempo de Execução na Avaliação(TEA)	599,349	2837,405	11455,601
Tempo de Execução Total	606,046	2849,721	11475,091
Valor de Aptidão	2,705	2,690	12,669
Percentagem do TEA	98,9%	99,6%	99,8%

Tabela 7: Mediana de doze execuções da versão sequencial do problema do protocolo *OSPF* para diferentes tamanhos do problema numa máquina 641, relativos aos tempos de execução totais (segundos), tempos de execução na avaliação (segundos) e valores de aptidão.

<table>
<tr><th>Tamanho do Problema</th><th>Objeto</th><th>Número de Indivíduos</th><th>Memória Usada</th></tr>
<tr><td rowspan="3">30</td><td>OSPF</td><td>-</td><td>95KB</td></tr>
<tr><td rowspan="2">Island</td><td>240 (duas ilhas)</td><td>254KB</td></tr>
<tr><td>20 (vinte e quatro ilhas)</td><td>111KB</td></tr>
<tr><td rowspan="3">50</td><td>OSPF</td><td>-</td><td>251KB</td></tr>
<tr><td rowspan="2">Island</td><td>240 (duas ilhas)</td><td>673KB</td></tr>
<tr><td>20 (vinte e quatro ilhas)</td><td>291KB</td></tr>
<tr><td rowspan="3">80</td><td>OSPF</td><td>-</td><td>600KB</td></tr>
<tr><td rowspan="2">Island</td><td>240 (duas ilhas)</td><td>1225KB</td></tr>
<tr><td>20 (vinte e quatro ilhas)</td><td>661KB</td></tr>
</table>

Tabela 8: Resultados das medições de memória aos objetos *OSPF* e *Island* para os vários tamanhos do problema.

Foram realizadas medições à memória com o mesmo código de Dimitris Andreous[59], aos objetos relevantes para as versões paralelas neste caso de estudo, *OSPF* e *Island*, apresentados na *Tabela 8*. Analisando os resultados, conclui-se que aumentando o tamanho do problema aumenta-se significativamente a memória usada em ambos objetos.

A memória necessária para realizar a avaliação de indivíduos (*OSPF*), nos tamanhos do problema com 30 e 50 nós, provavelmente cabe na *cache L2* de cada processador (exceto para o uso de *hyper-threading* no tamanho do problema com 50 nós) enquanto que o tamanho do problema maior apenas cabe na *cache L3*. Com esta análise, espera-se que não haja problemas na escalabilidade do modelo de paralelismo global para os tamanhos do problema com 30 e 80 nós, ao contrário do tamanho do problema com 50 nós que com o uso de *hyper-threading* provavelmente aumentará os *miss rates* na *cache L2* pois terá de executar duas *threads* em simultâneo (dois objetos *OSPF*).

Para o modelo de ilhas, a situação é diferente, pois a memória necessária para uma instância *Island* depende do número de indivíduos. Com o tamanho do problema com 30 nós, para o caso de existirem duas ilhas (240 indivíduos por ilha) ou até na versão sequencial, os dados poderão não caber na *cache L2*, aumentando os seus *miss rates* nesta *cache* de memória. Por outro lado, aumentando o número de ilhas, o número de indivíduos por ilha irá diminuir, pelo que espera-se que no caso de vinte e quatro ilhas haja uma diminuição significativa dos *miss rates* na *cache L2*, podendo assim os dados necessários para a própria ilha e os dados necessários para cada avaliação caber na *cache L2*. Para os restantes tamanhos do problema, os dados apenas caberão inteiramente na *cache L3* pelo que se espera *miss rates* altas na *cache L2* para qualquer número de ilhas.

Tal como no caso de estudo anterior, uma nova classe foi implementada, *ParallelEAOSPF*, que consoante os parâmetros recebidos e outros definidos executa uma das quatro versões paralelas ou a versão sequencial deste caso de estudo. Nas restantes subsecções estarão primeiramente os resultados do primeiro conjunto de testes e em seguida os resultados do segundo conjunto de testes.

4.4.2 Paralelismo Global

Tal como no caso de estudo da fermentação, a maior parte da sua computação realizada no caso de estudo do protocolo *OSPF* é efetuada na avaliação de indivíduos, sendo uma boa solução o uso do modelo de paralelismo global focado na avaliação dos indivíduos.

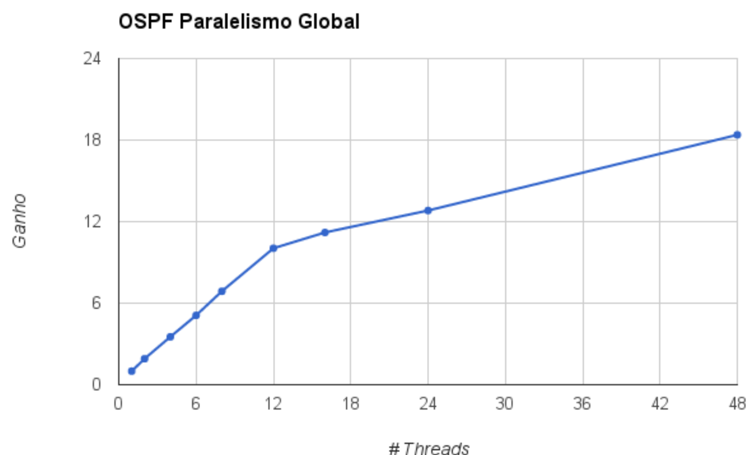

Figura 37: Resultados dos valores de ganho das execuções do modelo de paralelismo global no caso de estudo *OSPF* com o tamanho do problema de 30 nós.

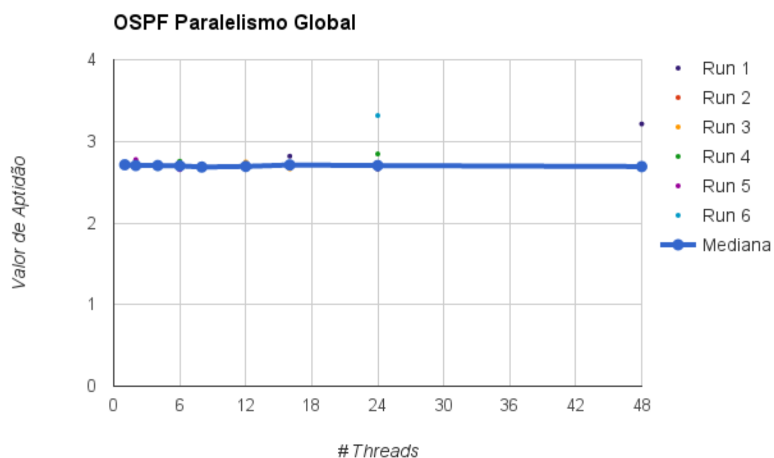

Figura 38: Resultados dos valores de aptidão das execuções do modelo de paralelismo global no caso de estudo *OSPF* com o tamanho do problema de 30 nós.

Realizaram-se testes a este modelo da mesma forma que os testes realizados no caso de estudo anterior, utilizando o tamanho do problema com 30 nós numa máquina 662, estando os seus resultados dos valores de ganho e de aptidão representados na *Figura 37* e *Figura 38*, respetivamente.

Observando os valores de ganho, concluí-se que estes valores aumentam com o aumento do número de *threads*. Comparativamente aos resultados do caso de estudo anterior (*Figura 13*), os resultados dos valores de ganho são inferiores neste caso de estudo, devendo-se provavelmente ao facto de utilizar um tamanho do problema pequeno. Analisando o tempo de execução total para este caso de estudo concluí-se que o tempo para cada avaliação de uma solução é bastante inferior ao caso de estudo anterior, pelo que é expectável que o uso de paralelismo neste tamanho do problema resulte numa eficiência inferior. Adicionalmente, a sincronização no fim das avaliações em cada geração neste modelo paralelo agrava estes resultados, na medida que ao existir pouca carga computacional por *thread*, quando são utilizados dois *sockets*, as *threads* que executam no primeiro *socket* serão mais rápidas a acabar a sua tarefa assumindo que os dados estão no primeiro *socket* e é necessária a transferência de parte destes dados para que as *threads* do segundo *socket* realizem a sua tarefa. Na *Figura 38*, confirma-se que este modelo de paralelismo global não influência os valores de aptidão.

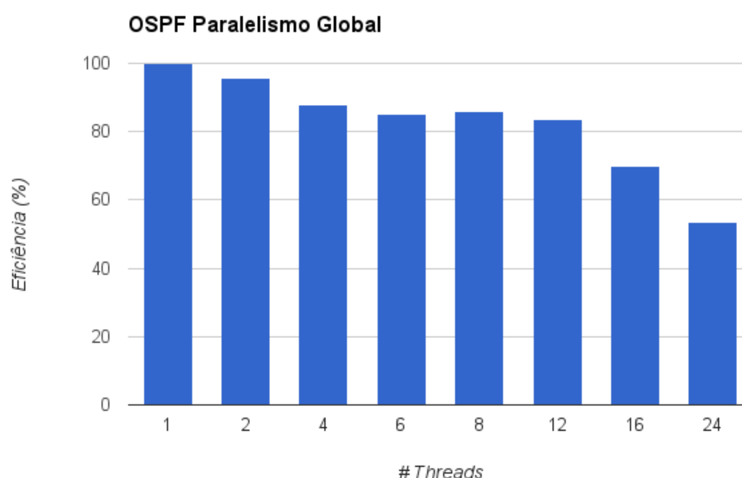

Figura 39: Resultados da eficiência do paralelismo das execuções do modelo de paralelismo global no caso de estudo *OSPF* com o tamanho do problema de 30 nós.

Na *Figura 39* estão os resultados da eficiência do paralelismo, onde se confirma a perda de eficiência face ao caso de estudo anterior (*Figura 15*), assim como se verifica uma perda significativa no uso de dois *sockets* da máquina *NUMA* para os casos com mais de 12 *threads*.

4.4.3 Modelo de Ilhas em Memória Partilhada

Para o modelo de ilhas, decidiu-se apenas realizar testes com o tipo de migração síncrono devido à imprevisibilidade do tipo assíncrono, de maneira a não afetar os valores de aptidão e simplificar a comparação dos resultados com o caso de estudo anterior.

Após as conclusões do caso de estudo anterior, para este modelo em memória partilhada, realizou-se apenas um conjunto de testes alterando o seu intervalo de migração, com os restantes parâmetros fixos nos mesmo moldes do caso de estudo anterior.

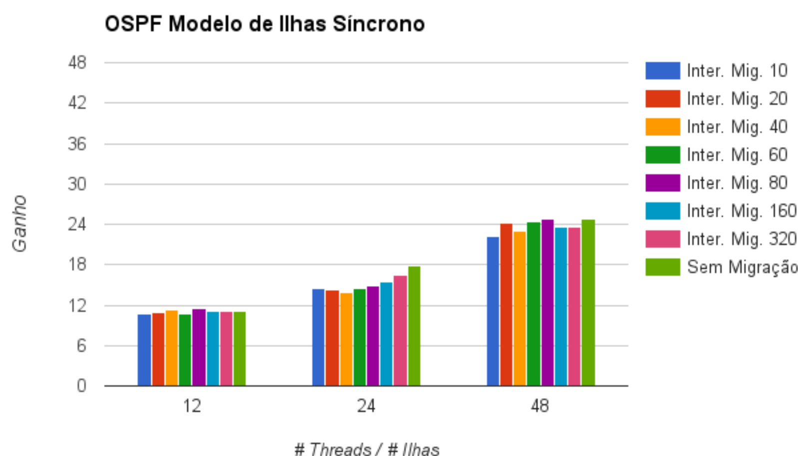

Figura 40: Resultados dos valores de ganho das execuções ao modelo de ilhas síncrono em memória partilhada utilizando diferentes intervalos de migração e diferentes números de ilhas no caso de estudo *OSPF* com o tamanho do problema de 30 nós.

Na *Figura 40* estão representados os valores de ganho resultantes das execuções. Comparativamente à *Figura 20*, os valores recolhidos são igualmente inferiores tal como no modelo de paralelismo global. Tendo em conta que este caso de estudo é de minimização, os resultados relativos aos valores de aptidão (*Figura 41*) revelaram-se mais estáveis que o caso de estudo da fermentação, pois para o caso de 48 ilhas os valores de aptidão são iguais ou inferiores ao valor da versão sequencial, exceto para o teste sem migração.

Foram calculados os valores da eficiência do paralelismo para este modelo paralelo utilizando os valores de ganho relativos à utilização de 10 gerações como intervalo de migração. Com a maior flexibilidade deste modelo paralelo quanto à sincronização de *threads* verifica-se melhores resultados que o modelo de paralelismo global, alcançando eficiências aceitáveis.

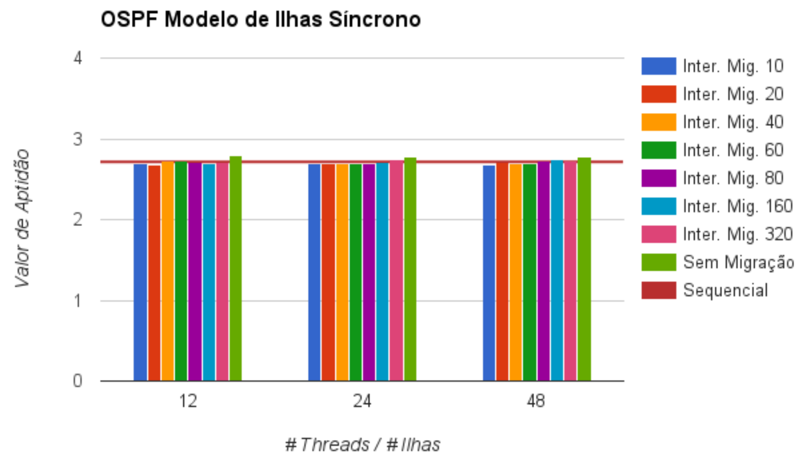

Figura 41: Resultados dos valores de aptidão das execuções ao modelo de ilhas síncrono em memória partilhada utilizando diferentes intervalos de migração e diferentes números de ilhas no caso de estudo *OSPF* com o tamanho do problema de 30 nós.

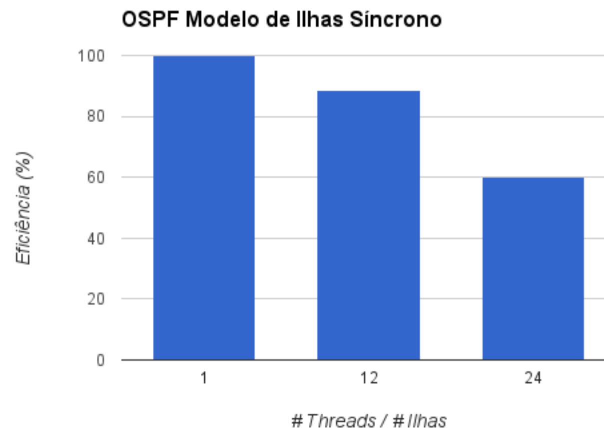

Figura 42: Resultados da eficiência do paralelismo das execuções ao modelo de ilhas síncrono em memória partilhada no caso de estudo *OSPF* com o tamanho do problema de 30 nós.

4.4.4 Modelo de Ilhas em Memória Distribuída

Nos testes ao modelo de ilhas em memória distribuída foi utilizado também o tipo de migração síncrono com o mesmo ambiente de testes, utilizando uma e duas máquinas 662.

Os valores de ganho dos testes executados numa máquina 662 estão representados na

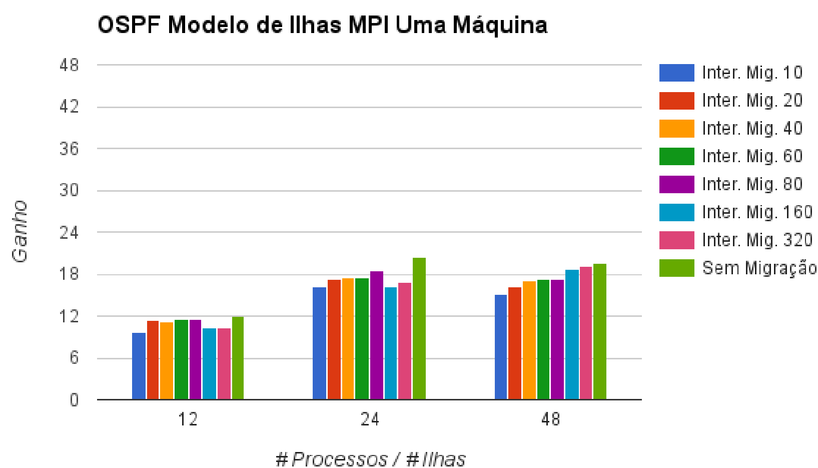

Figura 43: Resultados dos valores de ganho das execuções ao modelo de ilhas em memória distribuída com uma máquina 662 utilizando diferentes intervalos de migração no caso de estudo *OSPF* com o tamanho do problema de 30 nós.

Figura 43. Os resultados de 12 e 24 ilhas são semelhantes aos resultados deste modelo em memória partilhada, ao contrário dos resultados das 48 ilhas que são explicados pelo facto de estarem duas JVMs a executar em cada núcleo (CPU). Comparando estes valores ao do caso de estudo anterior (*Figura 27*), concluí-se que para o caso com o uso de *hyper-threading*, de 48 ilhas, os valores de ganho são inferiores, não existindo melhorias em comparação ao uso de 24 ilhas. Isto deve-se possivelmente ao facto mencionado das JVMs aliado ao facto da carga computacional por processo diminuir. Quanto aos valores de aptidão, estes permaneceram semelhantes ao modelo em memória partilhada, estando estes resultados na *Figura 53* do *Anexo B*.

A *Figura 44* contém os valores de ganho para os testes com duas máquinas 662, a qual se observa melhorias face aos testes com uma máquina, embora em comparação ao caso de estudo da fermentação (*Figura 29*) estes valores são bastante mais baixos devido à pouca carga computacional neste tamanho do problema. Com o uso de duas máquinas com memória distribuída, obtêm-se resultados mais estáveis para 12 e 24 ilhas pelo uso de apenas um *socket* em cada uma das máquinas, onde a comunicação na migração entre *sockets* e entre máquinas é só verificada no uso de 48 ilhas. Os resultados dos valores de aptidão são similares aos anteriores deste modelo e estão representados na *Figura 54* do *Anexo B*.

Para finalizar o primeiro conjunto de testes neste caso de estudo, realizou-se o cálculo da eficiência do paralelismo para o melhor resultado obtido, tendo em conta os valores de

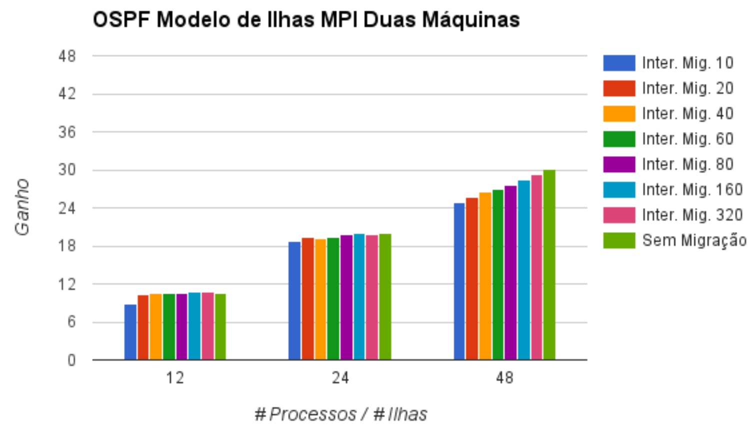

Figura 44: Resultados dos valores de ganho das execuções ao modelo de ilhas em memória distribuída síncrono com duas máquinas 662 utilizando diferentes intervalos de migração no caso de estudo *OSPF* com o tamanho do problema de 30 nós.

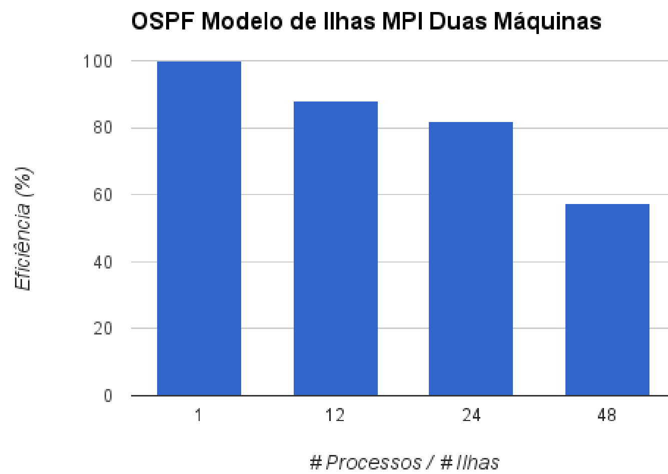

Figura 45: Resultados da eficiência do paralelismo das execuções ao modelo de ilhas em memória distribuída síncrono com duas máquinas 662 no caso de estudo *OSPF* com o tamanho do problema de 30 nós.

aptidão, com o intervalo de migração de 80 em 80 gerações dos testes com duas máquinas (*Figura 45*). Tendo em conta o tamanho do problema pequeno, os valores obtidos foram positivos, conseguindo uma eficiência de 58% com 48 ilhas.

4.4.5 Modelo Híbrido

No segundo conjunto de testes deste caso de estudo, foi utilizado o modelo híbrido de forma a comparar os valores de ganho utilizando diferentes tamanhos do problema *OSPF*. Para isto, realizaram-se os mesmo testes para os diferentes tamanhos do problema, com 480 indivíduos e diferentes números de gerações utilizando 1, 2, 4 e 8 máquinas 641, aplicando em cada máquina uma ilha do modelo de ilhas com intervalo de migração de 80 gerações e 16 e 32 *threads* por máquina correspondentes ao paralelismo global. Tal como no caso de estudo da fermentação, para os testes com uma máquina 641 foi aplicado apenas o modelo de paralelismo global, da mesma forma que as execuções com oito máquinas 641 e 32 *threads* por máquina são apenas usadas 30 *threads* por máquina devido à limitação do número de indivíduos.

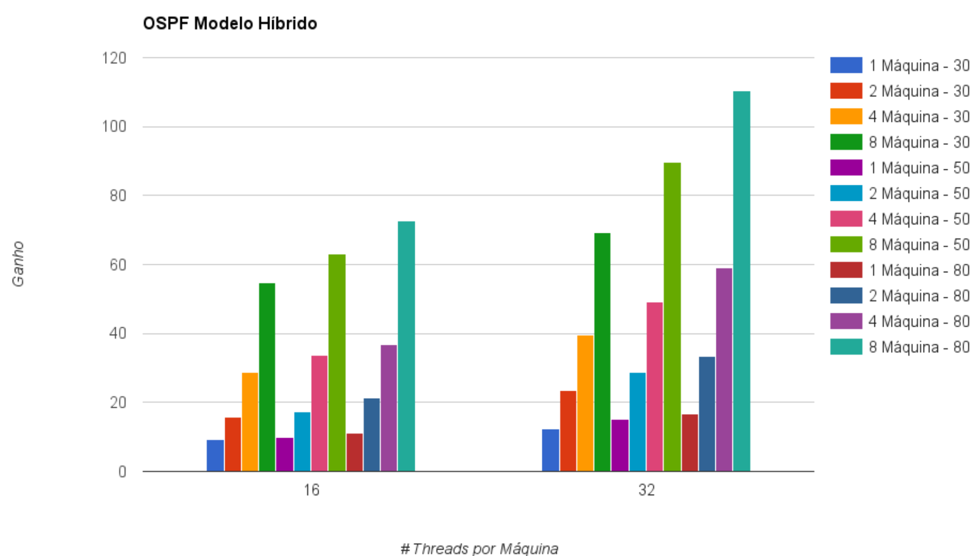

Figura 46: Resultados dos valores de ganho das execuções ao modelo híbrido com diferentes números de máquinas 641 utilizando uma ilha por máquina com 16 e 32 *threads* do paralelismo global no caso de estudo *OSPF* para tamanhos do problema com 30, 50 e 80 nós.

Na *Figura 46* estão representados os valores de ganho para os testes mencionados para os tamanhos do problema com 30, 50 e 80 nós. Com estes resultados pode-se concluir que consoante se aumenta o tamanho do problema aumenta-se a carga efetuada por *thread*/processo, levando a um aumento significativo dos valores de ganho. Assim, o melhor resultado obtido

neste caso de estudo foi utilizando oito máquinas 641 com 32 *threads* no tamanho do problema com 80 nós com um valor de ganho igual a 111, conseguindo então ser 111 vezes mais rápido que a versão sequencial. Para tamanhos do problema superiores aos usados, provavelmente atingiria-se o limite da escalabilidade principalmente devido a limitações da memória usada.

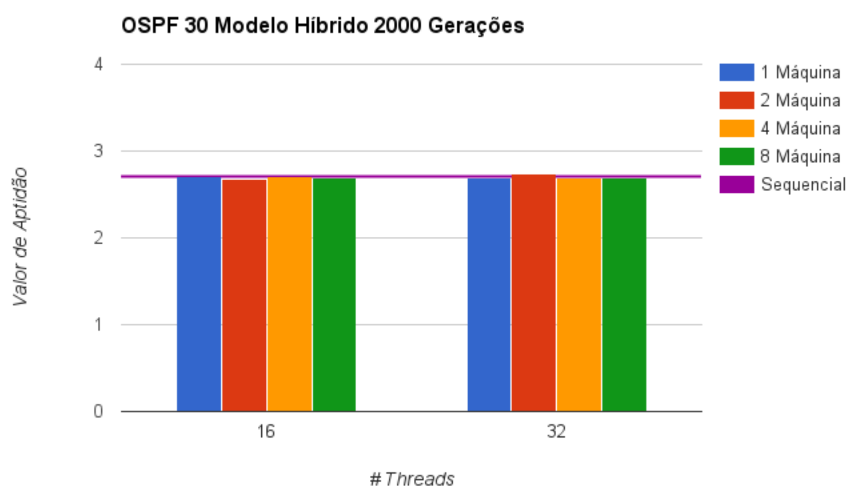

Figura 47: Resultados dos valores de aptidão das execuções ao modelo híbrido com diferentes números de máquinas 641 utilizando uma ilha por máquina com 16 e 32 *threads* do paralelismo global no caso de estudo *OSPF* para o tamanho do problema com 30 nós.

Na *Figura 47*, *Figura 48* e *Figura 49* estão os resultados dos valores de aptidão relativos às execuções dos mesmos testes para os tamanhos do problema com 30, 50 e 80 nós, respetivamente. Nos resultados obtidos para os três tamanhos do problema concluí-se que estes valores foram satisfatórios, na medida que os valores de aptidão destas execuções paralelas resultaram em valores similares à versão sequencial. Por outro lado, nos resultados para os tamanhos do problema com 50 e 80 nós (*Figura 48* e *Figura 49*), obtiveram-se valores mais irregulares devido à existência de uma maior variabilidade nos valores recolhidos para o calculo dos valores das medianas apresentadas.

Para finalizar, calcularam-se os valores da eficiência do paralelismo para o melhor caso obtido com o tamanho do problema com 80 nós, utilizando 1, 2, 4 e 8 máquinas com 16 *threads* incluindo o caso com *hyper-threading* para os testes com 8 máquinas que representa o caso com máximo de paralelismo possível, representados na *Figura 50*. Comparando aos resultados do caso de estudo da fermentação (*Figura 36*), obtiveram-se melhores resultados neste caso de estudo *OSPF* conseguindo para o caso com máximo de paralelismo possível

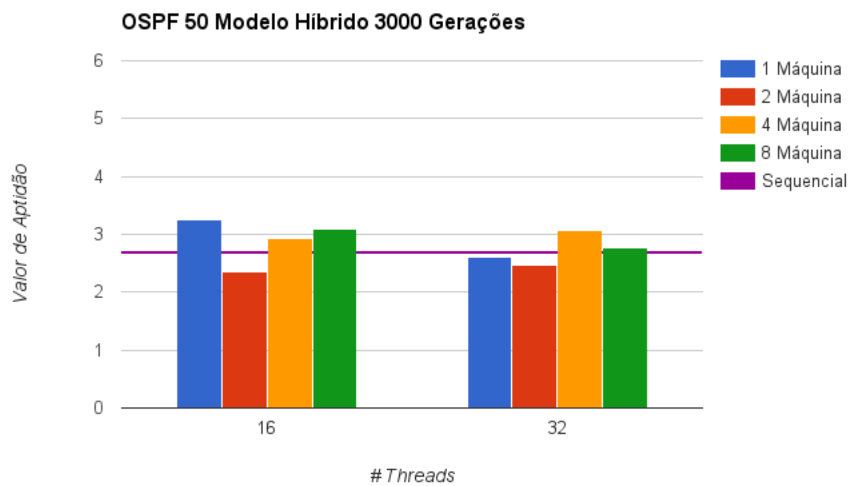

Figura 48: Resultados dos valores de aptidão das execuções ao modelo híbrido com diferentes números de máquinas 641 utilizando uma ilha por máquina com 16 e 32 *threads* do paralelismo global no caso de estudo *OSPF* para o tamanho do problema com 50 nós.

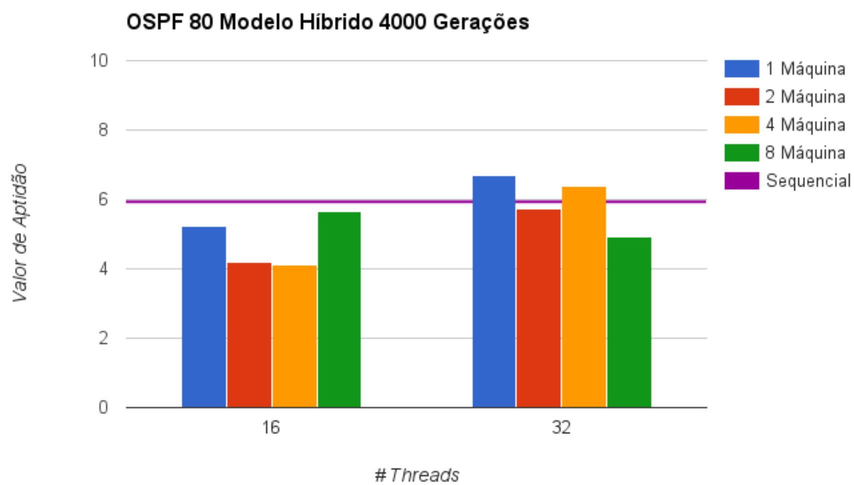

Figura 49: Resultados dos valores de aptidão das execuções ao modelo híbrido com diferentes números de máquinas 641 utilizando uma ilha por máquina com 16 e 32 *threads* do paralelismo global no caso de estudo *OSPF* para o tamanho do problema com 80 nós.

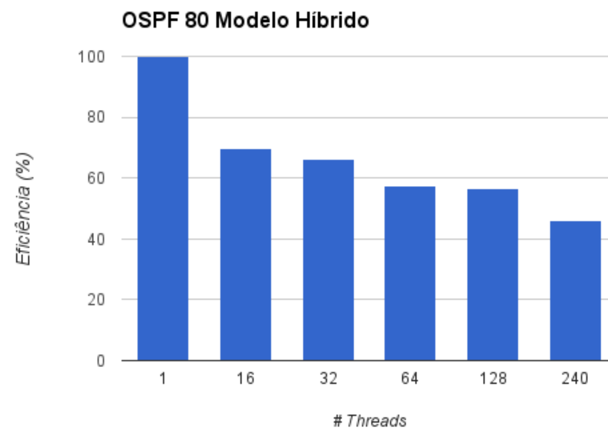

Figura 50: Resultados dos valores de eficiência do paralelismo das execuções ao modelo híbrido com diferentes números de máquinas 641 utilizando uma ilha por máquina com 16 e 32 *threads* do paralelismo global no caso de estudo *OSPF* para o tamanho do problema com 80 nós.

uma eficiência de 46%.

5 Conclusão

5.1 Síntese

Tendo em conta os objetivos propostos nesta dissertação, todos eles foram cumpridos e apresentados da forma o mais clara possível. As quatro implementações paralelas propostas foram realizadas sobre a plataforma *JECoLi* de forma menos invasiva possível. Estas implementações foram testadas e otimizadas de diversas formas até à obtenção dos resultados finais apresentados que utilizaram o máximo de recursos computacionais disponíveis, validando assim a sua correta implementação.

Com os resultados apresentados na secção anterior das quatro implementações paralelas, nos dois casos de estudos, pode-se concluir que na generalidade todas as implementações conseguem escalar em ambos os casos de estudo obtendo valores de aptidão aceitáveis e eficiências do paralelismo razoáveis.

Analisando as implementações paralelas realizadas sobre a plataforma *JECoLi* e os seus resultados, podem concluir-se as seguintes vantagens e desvantagens do modelo de paralelismo global e do modelo de ilhas:

Implementação	Vantagens	Desvantagens
Paralelismo Global	+ Configuração e implementação simples; + Paralelismo mais flexível (possui limite no número total de avaliações por geração).	− Implementação em memória partilhada; − Paraleliza apenas uma parte do AE; − Sincronização entre gerações.
Modelo de Ilhas	+ Implementação em memória distribuída; + Paraleliza todas as partes do AE; + Migração assíncrona possibilita mais flexibilidade na comunicação.	− Configuração e implementação complexa; − Paralelismo menos flexível (possui limite no número de indivíduos por ilha).

Tabela 9: Vantagens e desvantagens do uso do modelo de paralelismo global e modelo de ilhas.

Outro ponto importante a adicionar nestes modelos paralelos é a sua influência quanto à eficiência do algoritmo dada pelos valores de aptidão finais. O modelo de paralelismo global

não influência estes valores ao contrário do modelo de ilhas, que conforme a quantidade de interação entre os indivíduos definida pela configuração e parâmetros estabelecidos pode afetar positivamente ou negativamente nos valores de aptidão finais.

De maneira a obter bons resultados nos valores de aptidão com o modelo de ilhas utilizando um grande número de ilhas é necessário aumentar o grau de ligações na topologia e/ou aumentar o número de comunicação através do intervalo de migração, resultando numa penalização nos valores de ganho. Contudo, tal como os resultados apresentados sugerem, é possível obter valores de aptidão razoáveis a um custo de comunicação bastante baixo, embora não esquecendo que a convergência das soluções é dependente do problema a ser resolvido pelo AE.

Relativamente à implementação paralela do modelo de ilhas em memória distribuída em concreto, verifica-se outra desvantagem pelo uso da biblioteca *MPI* e da linguagem *Java* em simultâneo, que resulta numa sobrecarga adicional ao paralelismo onde cada processo *MPI* cria uma *JVM*, que por sua vez cria uma quantidade de *threads* considerável para gerir a execução. Isto resulta nos resultados observados utilizando uma máquina, onde se consegue significativamente melhores valores de ganho na implementação em memória partilhada.

Quanto ao tipo de migração assíncrono no modelo de ilhas pode-se concluir que é uma boa implementação para utilizar com um número de ilhas reduzido, pois com o aumento do número de ilhas e consequentemente o aumento da comunicação levam a uma alteração significativa do comportamento da convergência de soluções (*e.g.* modelo de ilhas em memória distribuída).

A implementação correspondente ao modelo híbrido revelou-se uma boa opção para os casos de estudo utilizados por conciliar vantagens dos dois modelos paralelos integrantes de forma a adicionar outras vantagens como permitir utilizar mais paralelismo ao aplicar o modelo de paralelismo global em diferentes ilhas em memória distribuída e conseguir melhorar os valores de aptidão com a utilização do modelo de ilhas. Por outro lado, a eficiência do paralelismo vai decrescendo consoante o número de *threads* e de processos aumentam devido à sincronização impostas dos dois modelos e pelo número de *threads* se aproximar do número de soluções a avaliar.

Adicionalmente, a eficiência do paralelismo aumenta consoante a carga computacional aumenta na avaliação dos indivíduos, tal como se verificou nos testes ao modelo híbrido com o segundo caso de estudo, alternando o tamanho do problema. Contudo, para tamanhos do problema superiores aos usados ou para outros problemas com uma carga computacional excessiva, provavelmente atingiria-se o limite da escalabilidade destas implementações, principalmente devido a limitações da memória usada.

Analisando as eficiências do paralelismo de todos os modelos implementados, o modelo

de ilhas é o que consegue melhores resultados, embora deva existir uma configuração cuidada deste modelo de maneira a que os valores de aptidão sejam aceitáveis. Em alternativa, caso existam recursos computacionais disponíveis e se pretender aplicar mais paralelismo, o modelo híbrido é uma boa opção para obter melhores soluções com mais rapidez de execução.

Por último, apesar destas implementações paralelas sobre a plataforma *JECoLi* se terem focado apenas nos AEs, o código paralelo facilmente se adapta a outro tipo de meta-heurísticas de otimização.

5.2 Limitações

Durante o desenvolvimento desta dissertação, a maior parte das limitações encontradas são relativas à compreensão dos resultados obtidos. O uso de ferramentas para a análise das execuções paralelas e dos recursos do sistema, como ferramentas de *profilling* e contadores de hardware (*e.g.* PAPI), seriam decisivas na compreensão dos resultados. O seu uso não foi possível devido à utilização da linguagem *Java* e à inexistência destas ferramentas para esta linguagem de programação.

A maior parte do tempo no desenvolvimento desta dissertação foi gasto na realização de testes às implementações paralelas e na sua análise. Com isto em consideração, a maior limitação encontrada nesta dissertação deveu-se à dificuldade de obter disponíveis os recursos computacionais necessários para os testes realizados.

5.3 Trabalho Futuro

Tendo em conta o trabalho apresentado, para um trabalho futuro poderia-se realizar os seguintes pontos:

- Obter um caso de estudo com mais carga computacional de modo a utilizar mais recursos computacionais e analisar a sua escalabilidade;
- Implementar um modelo híbrido, tendo como camadas superiores e inferiores o modelo de ilhas. Para este caso, seria necessário um aumento do número de indivíduos, assim como um caso de estudo com mais carga computacional;
- Implementar o modelo celular descentralizado de forma a ser executado em aceleradores (e.g. *GPUs*[61][62] e *MICs*) atuais, visto que este tipo de *hardware* possui dezenas/-centenas de núcleos, cada um executando um único indivíduo;
- Implementar um *script* que executa diferentes configurações do modelo de ilhas de forma a retornar a melhor configuração para o problema em causa, tendo em conta os valores de ganho e de aptidão obtidos.

Referências Bibliográficas

[1] Charles Darwin. *The origin of species*. 1809-1882.

[2] Owens A J Fogel L J and Walsh M J. *Artificial Intelligence through Simulated Evolution*. New York: wiley, USA, 1966.

[3] L.J. Fogel and G.H. Burgin. *Competitive goal-seeking through evolutionary programming*. Final Report, Contract AF 19(628)-5927, Air Force Cambridge Research Laboratories, 1969.

[4] Holland J H. *Adaptation in Natural and Artificial Systems*. PhD thesis, University of Michigan, Ann Arbor, 1975.

[5] Holland J H. *Adaptation in Natural and Artificial Systems*. MIT Press Second Edition, 1992.

[6] Rechenberg I. *Cybernetic Solution Path of an Experimental Problem. Ministry of Aviation*. Royal Aircraft Establishment (U.K.), 1965.

[7] Rechenberg I. *Evolutions Strategie : Optimierung technischer Systeme nach Prinzipien der biologischen Evolution (Stuttgart: Frommann-holzboog)*. 1973.

[8] Schwefel H-P. *Kybermetische Evolution als Strategie der expermentellen Forschung in der Stromungstechnik Diplomarbeit Hermann Fottinger Institut fur Stromungstechnik, technische Universitat, Berlin*. 1965.

[9] Schwefel H-P. *Evolutionsstrategie und numerische Optimierung*. PhD thesis, Technische Universität Berlin, 1975.

[10] Schwefel H-P. *Numerische Optimierung von Computer-Modellen mittels der Evolutionsstrategie.Basel: Birkhauser*. 1977.

[11] G. E. P. Box. *Evolutionary operation: A method for increasing industrial productivity*. Journal of the Royal Statistical Society C 6, no. 2: 81–101., 1957.

[12] G. J. Friedman. *Digital simulation of an evolutionary process*. General Systems Yearbook 4: 171–184., 1959.

[13] W. W. Bledsoe. *The use of biological concepts in the analytical study of systems*. ORSA-TIMS National Meeting, San Francisco., 1961.

[14] H. J. Bremermann. *Optimization through evolution and recombination.* In M. C. Yovits, G. T. Jacobi,and G. D. Goldstein, eds., SelfOrganizing Systems. Spartan Books., 1962.

[15] Toombs R. Reed, J. and N. A. Barricelli. *Simulation of biological evolution and machine learning.* Journal of Theoretical Biology 17: 319–342., 1967.

[16] Gilbert Syswerda. *Schedule optimization using genetic algorithms.* In L.Davis, editor, Handbook of Genetic Algorithms. Van Nostrand, 1991.

[17] J.E. Baker. *Reducing Bias and Inefficiency in the Selection Algorithm.* In J.Grenfenstette, editor, Proceedings of the Second International Confer- ence on Genetic Algorithms and their Applications., 1987.

[18] Enrique Alba and Marco Tomassini. Parallelism and evolutionary algorithms. *IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION*, 6(5):443–462, Outubro 2002.

[19] W. Bossert. *Mathematical optimization: Are there abstract limits on natural selection? in Mathematical Challenges to the Neo-Darwinian Interpretation of Evolution.* Philadelphia, PA: Wistar Inst. Press pp. 35–46., 1967.

[20] R. Tanese. *Parallel genetic algorithms for a hypercube.* Proc. 2nd Int. Conf. Genetic Algorithms p. 177., 1987.

[21] W. N. Martin J. P. Cohoon, S. U. Hegde and D. Richards. *Punctuated equilibria: A parallel genetic algorithm.* Proc. 2nd Int. Conf. Genetic Algorithms pp. 148–154., 1987.

[22] B. S. Duncan. *Parallel evolutionary programming.* Proc. 2nd Annu. Conf. Evolutionary Programming pp. 202–208., 1993.

[23] C. C. Pettey and M. R. Leuze. *A theoretical investigation of a parallel genetic algorithm.* Proc. 3rd Int. Conf. Genetic Algorithms pp. 398–405., 1989.

[24] *Efficient and Accurate Parallel Genetic Algorithms.* Norwell, MA: Kluwer., 2000.

[25] M. Schomisch H. Mühlenbein and J. Born. *The parallel genetic al- gorithm as function optimizer.* Parallel Comput., vol. 17, pp. 619–632., Sept. 1991.

[26] V. S. Gordon and D. Whitley. *Serial and parallel genetic algorithms as function optimizers.* Proc. 5th Int. Conf. Genetic Algorithms pp. 177–183., 1993.

[27] E. Alga and M.Tomassini. *Parallelism and evolutionary algorithms.* IEEE Trans.Evol. Comput. 6:443–462., 2002.

[28] E. Cantu-Paz. *A survey of parallel genetic algorithms.* Calculateurs Parallels, 10(2), Paris, Hermes., 1998.

[29] B.Massingill T.Mattson, B.Sabders. *Patterns for Parallel Progamming.* Addison-Wesley., 2004.

[30] A. Gupta A. Karypis G. Kumar, V. Grama. *Introduction to Parallel Computing, Design and Analysis of Algorithms.* The Benjamin/Cummings Publishing Company, Inc., 1994.

[31] MARUYAMTA. *Effective number of alleles in a subdivided population.* Theor. Pop. Biol. 1: 273-306., 1970.

[32] J. MAYNARSDM ITH. *Population size, polymorphism, and the rate of non-Darwinian evolution.* Amer. Nat. 104: 231-236., 1970.

[33] D. Whitley T. Starkweather and K. Mathias. *Optimization Using Distributed Genetic Algorithms.* Parallel Problem Solving, Springer Verlag., 1991.

[34] S. Wright. *Isolation by distance.* Genetics 28: 114-138., 1943.

[35] Cantú-Paz E. *Designing Efficient and Accurate Parallel Genetic Algorithms.* Technical Report No.99017, Illinois Genetic Algorithms Laboratory, UIUC, USA., July 1999.

[36] Tanese. *Distributed genetic algorithms.* in ICGA-3 pp. 434–439, 1989.

[37] Cantú-Paz E. *Using Markov chains to analyze a bounding case of parallel genetic algorithms.* In Koza, J. R., Banzhaf,W., Chellapilla, K., Deb, K., Dorigo, M., Fogel, D. B., Garzon, M. H.,Goldberg, D. E., Iba, H. and Riolo, R. L., editors, Genetic Programming 1998: Proceedings of the Third Annual Conference, pages 456–462, Morgan Kaufmann, San Francisco, California, 1998.

[38] B ROWN C. M. BIANCHINI R. *Parallel genetic algorithms on distributed-memory architectures.* In ATKINS S., WAGNER A. S., Eds., Transputer Research and Applications 6, p. 67–82, IOS Press (Amsterdam), 1993.

[39] GRUAU F. *Neural network synthesis using cellular encoding and the genetic algorithm.* Unpublished doctoral dissertation, L'Universite Claude Bernard-Lyon I, 1994.

[40] Cantú-Paz E. *A survey of parallel genetic algorithms.* Calculateurs Parallels, 10(2), Paris, Hermes, 1998.

[41] M. Lozano F. Herrera and C. Moraga. *Hybrid distributed real-coded genetic algorithms.* in Parallel Problem Solving from Nature, PPSN IV. ser. Lecture Notes in Computer Science, A. Eiben, T. Bäck, M. Schoe- nauer, and H.-P. Schwefel, Eds. Berlin, Germany: Springer-Verlag, vol. 1498, pp. 603–612, 1998.

[42] L.D. Davis. *Handbook of Genetic Algorithms.* VanNostrandReinhold,NewYork, 1991.

[43] Paulo Maia Pedro Evangelista and Miguel Rocha. *Implementing metaheuristic optimization algorithms with JECoLi.* Ninth International Conference on Intelligent Systems Design and Applications, Braga, Portugal, 2009.

[44] Emanuel Gonçalves Paulo Maia João Luis Sobral Pedro Evangelista, Jorge Pinho and Miguel Rocha. A software platform for evolutionary computation with pluggable parallelism and quality assurance. *Artificial Intelligence Applications and Innovations - EANN/AIAI'11*, 364:45–50, September 2011.

[45] Código *open-source* da plataforma *JECoLi.* `http://sourceforge.net/p/optflux/jecoli3/ci/master/tree/`.

[46] Jorge Henrique Martins de Pinho. Desenvolvimento de algoritmos evolucionário para ambientes grid com aplicações à optimização de sistemas biológicos. Master's thesis, Universidade do Minho, 11 2009.

[47] João Luis Sobral Jorge Pinho, Miguel Rocha. Pluggable parallelization of evolutionary algorithms applied to the optimization of biological processes. *18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing, Pisa, Italy*, pages 395–402, February 2010.

[48] Miguel Rocha Jorge Pinho, João Sobral. *Parallel Evolutionary Computation in Bioinformatics Applications.* Computer Methods and Programs in Biomedicine, 110(2), Braga, Portugal, May 2013.

[49] Joseph D. Gradecki and Nicholas Lesiecki. *Mastering AspectJ: Aspect-Oriented Programming in Java.* John Wiley & Sons, Inc., New York, NY, USA, 2003.

[50] J. Sobral B. Medeiros, R. Silva. *Parallelism Layers: Modular and Reusable Parallelism.* Braga, Portugal.

[51] J. Sobral. *Incrementally Developing Parallel Applications with AspectJ.* 20th IEEE International Parallel & Distributed Processing Symposium (IPDPS'06), Greece, Rhodes, April 2006.

[52] D.B. West. *Introduction to Graph Theory.* Prentice-Hall, Englewood Cliffs, NJ, 1995.

[53] OPEN MPI. A High Performance Message Passing Library. `http://www.open-mpi.org`.

[54] Glenn Judd Tony Skjellum Bryan Carpenter, Vladimir Getov and Geoffrey Fox. *MPJ: MPI-like Message Passing for Java. Concurrency: Practice and Experience.* Setembro 2000.

[55] Orlando Rocha Isabel Rocha Eugénio C. Ferreira Miguel Rocha, Rui Mendes. *Optimization of fed-batch fermentation processes with bio-inspired algorithms.* Expert Systems with Applications, Braga, Portugal, 2013.

[56] Paulo Cortez Miguel Rio Miguel Rocha, Pedro Sousa. *Quality of Service constrained routing optimization using Evolutionary Computation.* Applied Soft Computing, Braga, Portugal, 2009.

[57] E.W. Dijkstra. *A note on two problems in connexion with graphs.* Numerische Mathematik 1 (269–271), 1959.

[58] Services and Advanced Research Computing with HTC/HPC clusters. `http://www4.di.uminho.pt/search/pt/`.

[59] Ferramenta *Memory Measurer* de medição de memória alocada em objetos Java por Dimitris Andreou. `https://github.com/DimitrisAndreou/memory-measurer`.

[60] David E. Goldberg E. Cantú-Paz. *On the Scalability of Parallel Genetic Algorithms.* Massachusetts Institute of Technology, USA, 1999.

[61] Lucas de P. Veronese and Renato A. Krohling. Differential evolution algorithm on the gpu with c-cuda. In *IEEE Congress on Evolutionary Computation'10*, pages 1–7, 2010.

[62] El-Ghazali Talbi Thé Van Luong, Nouredine Melab. *GPU-based Island Model for Evolutionary Algorithms.* Genetic and Evolutionary Computation Conference (GECCO), Portland, United States, December 2009.

Anexos A : Ambiente Experimental Máquinas *NUMA*

Figura 51: Resultado do comando lstopo para as máquinas NUMA compute 662.

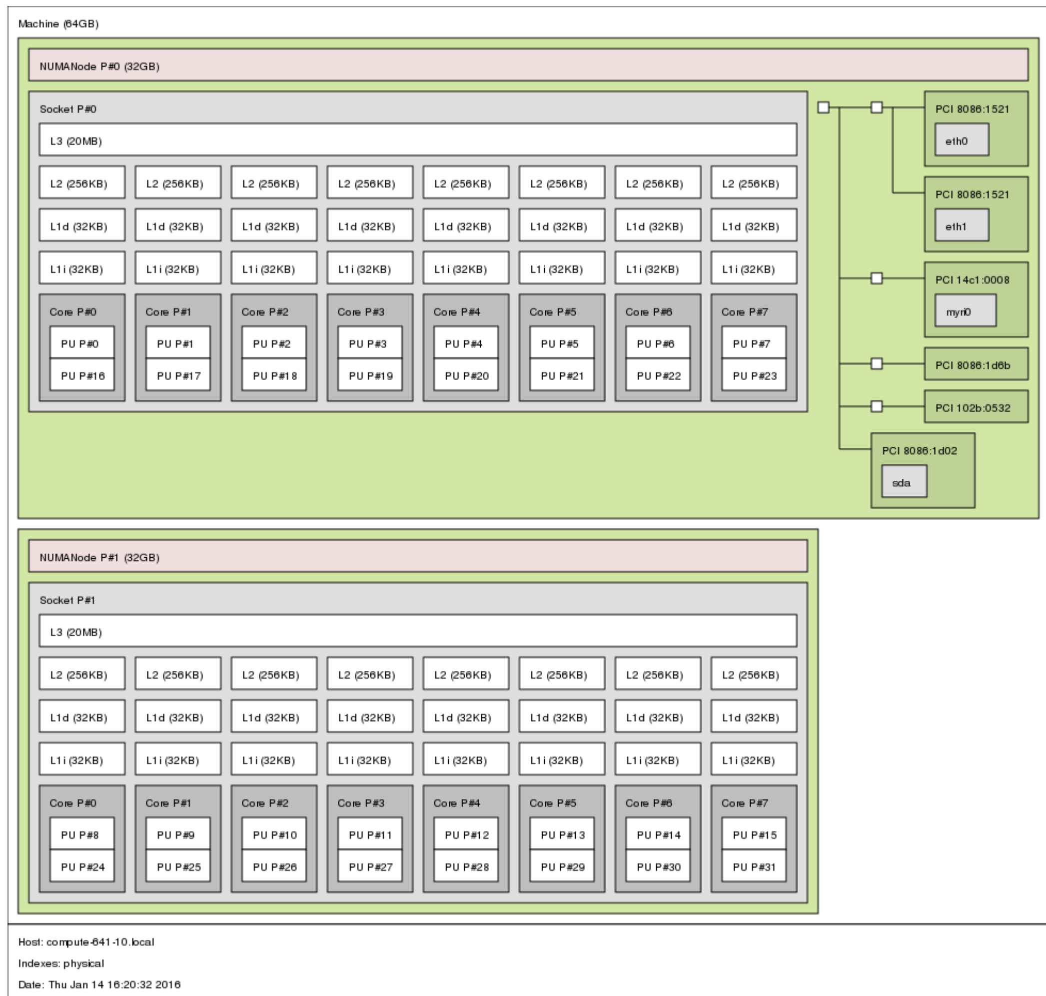

Figura 52: Resultado do comando lstopo para as máquinas NUMA compute 641.

Anexos B : Resultados *OSPF*

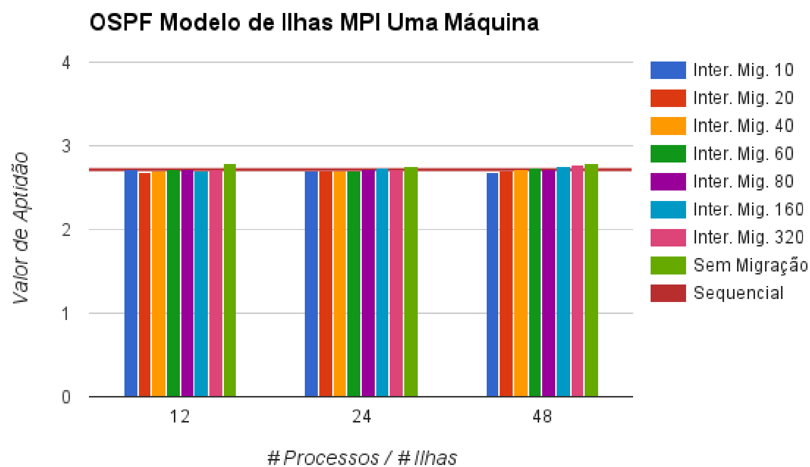

Figura 53: Resultados dos valores de aptidão das execuções ao modelo de ilhas em memória distribuída com uma máquina 662 utilizando diferentes intervalos de migração no caso de estudo *OSPF* com o tamanho do problema de 30 nós.

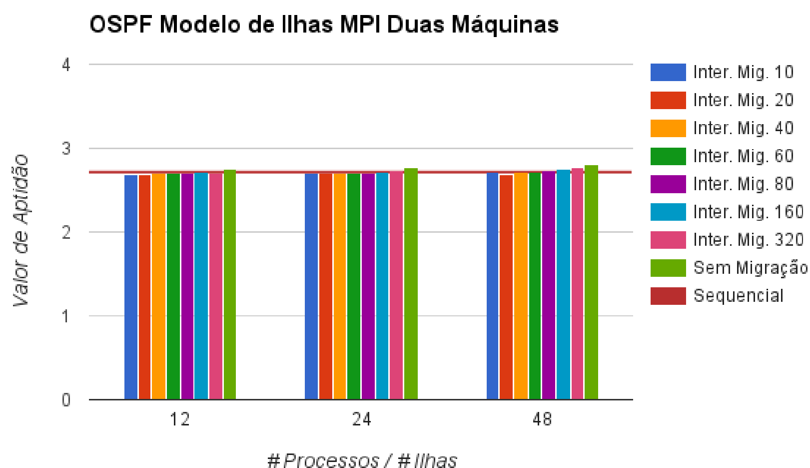

Figura 54: Resultados dos valores de aptidão das execuções ao modelo de ilhas em memória distribuída síncrono com duas máquinas 662 utilizando diferentes intervalos de migração no caso de estudo *OSPF* com o tamanho do problema de 30 nós.