UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE CIÊNCIA DA COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ABORDAGENS BASEADAS EM AUTÔMATOS CELULARES SÍNCRONOS PARA O ESCALONAMENTO ESTÁTICO DE TAREFAS EM MULTIPROCESSADORES

MURILLO GUIMARÃES CARNEIRO

Uberlândia - Minas Gerais
2012

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE CIÊNCIA DA COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

MURILLO GUIMARÃES CARNEIRO

ABORDAGENS BASEADAS EM AUTÔMATOS CELULARES SÍNCRONOS PARA O ESCALONAMENTO ESTÁTICO DE TAREFAS EM MULTIPROCESSADORES

Dissertação de Mestrado apresentada à Faculdade de Ciência da Computação da Universidade Federal de Uberlândia, Minas Gerais, como parte dos requisitos exigidos para obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Inteligência Artificial.

Orientadora:
Prof[a]. Dr[a]. Gina Maira Barbosa de Oliveira

Uberlândia, Minas Gerais
2012

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE CIÊNCIA DA COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Os abaixo assinados, por meio deste, certificam que leram e recomendam para a Faculdade de Ciência da Computação a aceitação da dissertação intitulada "**Abordagens Baseadas em Autômatos Celulares Síncronos para o Escalonamento Estático de Tarefas em Multiprocessadores**" por **Murillo Guimarães Carneiro** como parte dos requisitos exigidos para a obtenção do título de **Mestre em Ciência da Computação**.

Uberlândia, 28 de Fevereiro de 2012

Orientadora:

Profª. Drª. Gina Maira Barbosa de Oliveira
Universidade Federal de Uberlândia

Banca Examinadora:

Profª. Drª. Rita Maria da Silva Julia
Universidade de Federal de Uberlândia

Prof. Dr. Heitor Silvério Lopes
Universidade Tecnológica Federal do Paraná - UTFPR

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE CIÊNCIA DA COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Data: Fevereiro de 2012

Autor:	**Murillo Guimarães Carneiro**
Título:	**Abordagens Baseadas em Autômatos Celulares Síncronos para o Escalonamento Estático de Tarefas em Multiprocessadores**
Faculdade:	**Faculdade de Ciência da Computação**
Grau:	**Mestrado**

_____

Autor

Dedicatória

Aos meus amados pais, Sérgio e Carmelinda.
Ao meu amigo e irmão, Arthur.

Agradecimentos

Concluir o mestrado é um momento de muita alegria, mas é também momento de reconhecer as pessoas que me auxiliaram, tanto pessoalmente quanto academicamente, nessa tarefa. Apesar de tantas pessoas importantes, agradecer a uma só delas é necessário.

Assim, agradeço à Deus:

- Por sua companhia fiel em todos os momentos da minha vida e por me permitir viver momentos tão felizes quanto este.
- Pelos pais maravilhosos que me deu: Sérgio e Carmelinda, os quais me proporcionaram um lar repleto de amor e carinho, uma vida recheada de momentos felizes e me ensinaram o que há de mais importante nesse mundo: honestidade e humildade. Eles são meus exemplos de força, de perseverança e de superação. Pai e mãe, o apoio e a compreensão de vocês durante esses últimos anos foram cruciais para que eu pudesse concluir essa etapa. Sou eternamente grato à Deus por tê-los ao meu lado.
- Pelo meu irmão e grande companheiro Arthur, cuja bondade e cumplicidade sempre me inspiraram a fazer coisas boas e a ser motivo de alegria na vida das pessoas.
- Pela minha namorada, Luana, cujo apoio e carinho não me faltaram durante todo o mestrado e que tornou os meus dias em Uberlândia mais felizes.
- Por toda minha família, a qual sempre me proporcionou momentos de muita alegria e descontração. Em especial, agradeço à Deus pelo meu avô José e minha avó Celina. Quero agradecer a Deus também por todo o tempo que pude passar com minha avó Geracina e meu avô Samuel, que certamente olham por mim do céu.
- Pela minha orientadora, Prof. Gina M. B. Oliveira, cujo conhecimento, competência e simpatia sempre me motivaram no desenvolvimento deste trabalho. Sou grato à Deus pela paciência, confiança e incentivo dela às inovações apresentadas nesta pesquisa.
- Pelos grandes amigos do Ministério Universidades Renovadas, especialmente do GOU DOMINUS (uhull!), os quais a amizade, a atenção, o carinho e a alegria fizeram com que se tornassem minha segunda família.
- Pelos bons amigos que fiz no mestrado, na universidade e em Uberlândia e pelos velhos amigos de Ipameri-GO que, mesmo com a distância, estão sempre torcendo por mim.

- Pelos professores Carlos R. Lopes e Márcia A. Fernandes, que acrescentaram bastante à minha formação e pelos quais tenho um grande respeito. Agradeço ainda pelos demais professores que pude conhecer, por todo apoio recebido da UFU, CAPES e CNPq, e também pelo amigo e secretário do PPGCC, Erisvaldo.
- Pelos membros da banca de defesa, Prof. Heitor e Prof^a. Rita, que contribuíram para o enriquecimento deste trabalho.

"Comece fazendo o que é necessário, depois o que é possível, e de repente, você estará fazendo o impossível."

(São Francisco de Assis)

Resumo

O problema de escalonamento estático de tarefas computacionais (PEET) em uma arquitetura multiprocessada consiste em alocar tarefas que compõem um programa paralelo entre os nós de uma arquitetura com múltiplos processadores. Uma solução ótima de uma instância do PEET é tal que as restrições de precedência entre as tarefas sejam atendidas e o tempo total de execução - ou *makespan* - é minimizado. O problema é NP-Completo, mesmo limitado ao caso mais simples: um sistema paralelo com apenas dois processadores. Diante disso, métodos heurísticos, tais como HLFET (*Highest Level First with Estimated Time*) e MCP (*Modified Critical Path*), e meta-heurísticas, tais como algoritmos genéticos (AG) e *simulated annealing* (SA) têm sido frequentemente empregados na tentativa de encontrar boas soluções para o problema. Contudo, eles não apresentam qualquer habilidade para extrair conhecimento do processo de escalonamento de uma aplicação paralela e precisam reiniciar o processo a cada nova instância.

Neste contexto, o escalonamento baseado em autômatos celulares (ACs) tem se mostrado promissor, uma vez que tem por objetivo a extração do conhecimento sobre o processo de escalonamento de um programa paralelo e sua consequente reutilização em novas instâncias. Contudo, algumas características desejáveis ainda não foram exploradas com sucesso pelos modelos existentes na literatura: (i) paralelismo massivo intrínseco aos ACs, (ii) uso de um número arbitrário de processadores, (iii) reuso eficiente do conhecimento extraído sobre outras aplicações paralelas, entre outras. Desse modo, novas abordagens baseadas em ACs para o PEET foram investigadas. Essas abordagens foram avaliadas sistematicamente em relação a outras abordagens anteriores de escalonamento baseadas em AC e também a métodos heurísticos e meta-heurísticos.

Dentre as contribuições da pesquisa estão três novos modelos de escalonadores (EACS, EACS-H e EACS-HV) relacionados à extração, avaliação e reuso do conhecimento; uma nova estratégia para inicializar os reticulados dos ACs baseada em uma heurística determinística; uma nova estratégia para análise do comportamento dinâmico das regras do AC (penalização da regra) em tempo de execução e dois novos modelos de vizinhança de estrutura simples, porém com capacidade de extração de informações da aplicação paralela (V_{pl}-$c1$ e V_{pl}-$c2$) e uso de um número arbitrário de processadores. Como resultado dos experimentos, foi possível garantir o emprego bem sucedido do paralelismo nos ACs, além de um bom desempenho na fase de aprendizagem, onde os valores encontrados pelas abordagens estiveram próximos daqueles tomados por referência e foram superiores a heurísticas e algumas meta-heurísticas. Ainda, testes estatísticos foram realizados e comprovaram a superioridade das novas abordagens em relação a outros trabalhos baseados em AC. O reuso do conhecimento também foi avaliado e se mostrou competitivo nas novas abordagens. Em suma, as abordagens desenvolvidas mostram que o escalonamento baseado em AC possui grande potencial para o escalonamento eficiente de tarefas em sistemas multiprocessados. Logo, este potencial pode ser melhor explorado quando os modelos propostos trabalham diretamente sobre arquiteturas de *hardware* paralelo.

Palavras chave: autômato celular, escalonamento estático de tarefas, algoritmos evolutivos, heurísticas, meta-heurísticas

Abstract

The Static Task Scheduling Problem (STSP) in multiprocessors aims to allocate a set of computational tasks that compose a parallel application in the nodes of a multiprocessor architecture. An optimal solution for an instance of STSP is such that the precedence constraints are satisfied and the runtime - or makespan - is minimized. The problem is NP-Complete, even limited to the simplest case: a parallel system with only two processors. Approaches proposed to solve it typically use heuristics, such as HLFET (*Highest Level First with Estimated Time*) and MCP (*Modified Critical Path*), or metaheurístics, such as genetic algorithms (GA) and *simulated annealing* (SA) in an attempt to find good results for the problem. However, in these approaches, a computational effort is used to solve an instance of the problem and when a new instance is presented to the algorithm, the process needs to start again from scratch.

In this context, the cellular automata-based scheduling is a promising approach because its main feature is the extraction of knowledge while scheduling an application and its subsequent reuse in other instances. However some desirable features in CA-based scheduling such as: (i) massive parallelism inherent to CA, (ii) usage of an arbitrary number of processors, (iii) effective reuse of the knowledge extracted from parallel applications and others, have not been successfully exploited by previous models in the literature.

This dissertation investigates new CA-based approaches for STSP. They were evaluated systematically and compared to previous approaches of CA-based scheduling and heuristic and metaheuristic methods.

Some contributions of this research are three new models of schedulers (SCAS, SCAS-H and SCAS-HV) related to extraction, evaluation and reuse of knowledge, a new strategy to initialize CA lattices determined by a deterministic heuristic, a new strategy to analyze the dynamic behavior of CA rules (rule penalization) at runtime, and two new neighborhood models (V_{pl}-$c1$ e V_{pl}-$c2$) with simple structure, but able to extract information from parallel applications and use an arbitrary number of processors. The results showed it is possible to ensure the successful employment of parallelism in CAs as well as a good performance in the learning phase, in which the values found by new models are close to those of genetic algorithms (taken by reference) and superior than those obtained by heuristics and some metaheuristics. Furthermore, statistical tests were performed and proved the superiority of the new approaches in relation to other CA-based scheduling models. The reuse of knowledge was also evaluated and its performance was proved competitive in the new approaches. Finally, the methods developed showed the scheduling based on CA has great potential to efficiently schedule tasks in multiprocessor systems. This potential can be better exploited when the proposed models work directly on parallel hardware architectures.

Keywords: cellular automata, task static scheduling, evolutionary algorithms, heuristics, metaheuristics

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Lista de Abreviaturas e Siglas

AC	Autômato Celular
AG	Algoritmo Genético
b-level	*bottom level* ou nível inferior
CI	Configuração Inicial
DHLFET	Deterministic HLFET
EACS	Escalonador baseado em AC com atualização Síncrona
EACS-H	EACS com inicialização por Heurística
EACS-HV	EACS-H e Vizinhança pseudo-linear
GAD	Grafo Acíclico Direcionado
G_P	Grafo de Programa
HLFET	*Highest Level First with Estimated Time*
PEET	Problema de Escalonamento Estático de Tarefas
SA	*Simulated Annealing*
t-level	*top level* ou nível superior
V_s	número de processadores do sistema

Capítulo 1

Introdução

1.1 Contexto

O uso simultâneo de recursos computacionais tem sido uma das principais alternativas para suprir a demanda de aplicações cada vez mais intensivas. Tem sido frequente a exploração de ambientes multiprocessados ou multicomputadores por cientistas e empresas na tentativa de resolver problemas complexos. Entretanto, para que esses ambientes sejam efetivamente aproveitados é importante que seja possível a divisão da aplicação em tarefas independentes e que essas tarefas sejam alocadas nos nós de processamento de forma a tirar o máximo proveito dos mesmos. Diante disso, o escalonamento eficiente das tarefas tem um papel fundamental nas arquiteturas multiprocessadas.

De modo geral, o escalonamento é um processo de tomada de decisão, que envolve recursos e tarefas, na busca pela otimização de um ou mais objetivos. Os recursos podem ser máquinas em uma oficina, unidades de processamento em um ambiente computacional e assim por diante, enquanto que as tarefas podem ser operações em um processo de produção, execuções de um programa de computador, entre outros [41]. Assim, tem-se que o problema de escalonamento pode ser explorado sobre vários aspectos tais como o problema de escalonamento de produção, de empregados e de tarefas computacionais.

O problema de escalonamento de tarefas computacionais em uma arquitetura multiprocessada consiste em alocar tarefas que compõem um programa paralelo aos nós de uma arquitetura com múltiplos processadores. No caso do Problema de Escalonamento Estático de Tarefas (PEET) investigado nessa dissertação, todas as informações sobre as tarefas são conhecidas a priori. Uma solução ótima de uma instância do PEET é tal que as restrições de precedência entre as tarefas são atendidas e o tempo total de execução - ou *makespan* - é minimizado. O problema é conhecido por ser NP-Completo em sua forma geral [55] e por isso tem sido um grande desafio para muitos pesquisadores.

Vários métodos heurísticos têm sido empregados na tentativa de encontrar boas soluções para o problema, tais como HLFET (*Highest Level First with Estimated Time*), ISH

(*Insertion Scheduling Heuristic*) e MCP (*Modified Critical Path*) [26, 29]. Além destes, metaheurísticas tais como, *simulated annealing* (SA), algoritmos genéticos (AG) e redes neurais artificiais (RNA) também têm sido propostas para o problema [25, 30, 43]. É importante notar que o desenvolvimento e utilização dessas e outras técnicas de busca inspiradas na natureza, abre novas possibilidades para se obter soluções de qualidade para determinados problemas onde o alcance destas soluções, através de métodos sequenciais e determinísticos, é inviável. Além disso, há vários outros motivos que levam os pesquisadores à procura de alternativas aos algoritmos exatos, como eficiência, desempenho, rápida adaptação de soluções a novas instâncias, dentre outros. Em se tratando de um problema complexo como o PEET é interessante perceber que muitos algoritmos já foram propostos. Contudo a maioria deles não tem a capacidade de extrair conhecimento do processo de escalonamento de uma aplicação e precisam começar do zero a cada nova instância.

Resultados apresentados em [49, 53, 57] apontaram o uso promissor de abordagens baseadas em autômatos celulares (AC) para o PEET. É válido destacar que os ACs têm sido empregados com sucesso nos mais diversos campos de pesquisa da computação, tais como criptografia [65], simulação de sistemas complexos e de vida artificial [50], entre outros. Eles são sistemas dinâmicos discretos (tempo, espaço e estados) e possuem como uma de suas principais características a capacidade de emergir um comportamento global a partir de interações entre unidades locais. Um AC é composto de um reticulado e uma regra de transição. A regra de transição é aplicada sobre os estados das células do reticulado por um número finito de passos. O modo de atualização dos estados das células do reticulado mais investigado é o síncrono ou paralelo, uma vez que ele permite explorar o paralelismo intrínseco dos ACs. A vizinhança de uma célula está diretamente relacionada à regra de transição e é comumente determinada através de um raio R.

Um dos principais problemas relacionado ao uso dos autômatos celulares é que o espaço formado pelas regras de transição, que representam as possíveis soluções de um problema, é tipicamente de alta cardinalidade. Entretanto, trabalhos anteriores abriram novas possibilidades ao apresentar a aplicação da computação evolutiva para a busca de regras de ACs [18, 50]. Uma das técnicas evolutivas utilizadas nessa busca são os algoritmos genéticos (AG). Os AGs são algoritmos probabilísticos que fornecem um mecanismo de busca paralela e adaptativa baseado no princípio da sobrevivência dos indivíduos mais aptos e na reprodução, inspirados no princípio Darwiniano de seleção natural e na genética [23, 24]. Alguns trabalhos existentes na literatura mostraram que, combinados, os autômatos celulares [61] e os algoritmos genéticos [23] podem ser efetivamente usados para projetar algoritmos paralelos e distribuídos para resolver problemas complexos, tais como: classificação de densidade, sincronização [18, 35, 36, 39] e escalonamento [46].

O primeiro modelo de escalonador de tarefas baseado em autômato celular foi apresentado em [46]. Porém, outras investigações já foram realizadas [47–49, 51–53, 56, 57], sempre

motivadas pelas principais características do escalonador baseado em AC: a extração e o reuso do conhecimento.

1.2 Objetivos

O objetivo geral deste trabalho é a investigação de novas abordagens envolvendo autômatos celulares para o escalonamento de tarefas em sistemas multiprocessados. O objetivo principal é projetar algoritmos aptos a extrair conhecimento sobre o processo de escalonamento de uma determinada aplicação paralela a fim de reusá-lo no escalonamento de outras.

Dessa forma, é fulcral que os modelos propostos neste trabalho abordem características importantes que não foram bem exploradas nos trabalhos existentes na literatura ou ainda o desenvolvimento de novas estruturas com o intuito de obter melhor desempenho, tanto no aspecto computacional, quanto em relação à solução encontrada. Desse modo, abaixo são listadas algumas metas específicas que nortearam o desenvolvimento dos novos modelos em busca de atender o objetivo geral:

- Modo de atualização paralelo ou síncrono: uma das mais notáveis características dos AC é o seu paralelismo intrínseco ao utilizar o modo síncrono de atualização das células. Contudo, trabalhos anteriores após avaliarem o uso da atualização paralela, optaram pelo uso do modo sequencial por este ter apresentado melhor desempenho [53].
- Modelo de vizinhança linear: a maioria dos trabalhos anteriores investigou vizinhanças não lineares e complexas por retornarem resultados superiores à vizinhança linear "padrão" dos ACs. Essa dissertação buscou desenvolver uma estrutura de vizinhança linear simples e rápida ao invés de um modelo não linear complexo e com limitações. Para isso, foi preciso formular novos conceitos de vizinhança linear, considerando-se as relações entre as tarefas no grafo.
- Extensão para arquiteturas com número arbitrário de processadores: é importante investigar o comportamento dos modelos baseados em AC para sistemas com mais de dois processadores. Poucos trabalhos relacionados utilizaram um número de nós maior que 2 e os resultados obtidos por eles não foram satisfatórios.
- Modelo de avaliação do conhecimento: desenvolver novos métodos para direcionar, avaliar e utilizar o conhecimento assimilado durante o escalonamento de modo a obter um melhor desempenho nessa etapa e também no reuso.
- Comparação com métodos exatos e heurísticos: não foi encontrada em trabalhos anteriores qualquer análise ou comparação utilizando heurísticas conhecidas para o PEET, reproduções de trabalhos relacionados, ou mesmo métodos exatos.

1.3 Contribuições

Algumas das contribuições alcançadas através das investigações realizadas neste trabalho são apresentadas a seguir:

- estudo aprofundado e incorporação de conceitos, formulações técnicas e heurísticas relacionadas ao PEET no escalonamento baseado em AC;
- possibilidade de uso do modo de atualização paralelo das células com desempenho semelhante aos modelos sequenciais apresentados na literatura, através de um modelo de escalonador desenvolvido, denominado EACS;
- desenvolvimento de dois novos modelos (EACS-H e EACS-HV) relacionados à extração, avaliação e reuso do conhecimento no escalonamento baseado em AC e aptos a lidarem, de modo superior aos trabalhos da literatura, com a complexidade computacional referente ao aumento do número de processadores e o reuso em instâncias distintas do problema;
- detecção de uma falha relacionada ao uso da vizinhança linear na abordagem dos modelos anteriores. Construção de dois novos métodos de vizinhança, V_{pl}-$c1$ e V_{pl}-$c2$, como alternativa para resolver o problema;
- construção de uma nova estratégia de avaliação das regras do AC durante a execução do escalonador, de forma a direcionar a busca evolutiva a encontrar regras com comportamento dinâmico estável.

1.4 Estrutura da Dissertação

A estrutura dos demais capítulos deste trabalho foi organizada da seguinte forma:

- No Capítulo 2 são apresentados os principais conceitos, algumas taxonomias e diversas heurísticas envolvendo o problema de escalonamento estático de tarefas (PEET).
- No Capítulo 3 há alguns conceitos fundamentais relacionados aos algoritmos genéticos e aos autômatos celulares.
- No Capítulo 4 são mostrados conceitos e abordagens relacionados ao escalonamento baseados em autômatos celulares. Assim, o intuito neste capítulo é apresentar os principais trabalhos anteriores de modo a melhor definir a contribuição desta dissertação em relação ao estado da arte da pesquisa.
- No Capítulo 5 destacam-se as principais contribuições deste trabalho. São apresentadas as abordagens investigadas para alcançar os objetivos deste trabalho, além dos resultados obtidos nos experimentos.

- No Capítulo 6 são apresentadas as principais conclusões sobre as abordagens desenvolvidas e também apresentadas sugestões para trabalhos futuros.

Capítulo 2

Escalonamento Estático de Tarefas em Multiprocessadores

O problema de escalonar tarefas computacionais em uma arquitetura multiprocessada, mesmo limitado ao caso mais simples com dois processadores, é conhecido por ser NP-Completo [21]. Tal problema consiste em alocar as tarefas que compõem um programa paralelo aos nós do sistema de tal modo que as restrições de precedência entre as tarefas sejam atendidas. Uma solução de escalonamento é ótima se ela minimiza o tempo total de execução ou *makespan*, ao mesmo tempo que atende as restrições entre as tarefas [10].

Neste capítulo são apresentados os principais conceitos, algumas taxonomias e diversas heurísticas empregadas para o problema de escalonamento estático de tarefas (PEET). Assim, na Seção 2.1 destacam-se as principais taxonomias relacionadas ao escalonamento de tarefas e na Seção 2.2 tem-se a formulação do problema de escalonamento estático de tarefas em multiprocessadores. A Seção 2.3 traz informações importantes sobre a representação e geração de instâncias para o problema. Posteriormente, na Seção 2.4 descrevem-se, respectivamente, os tipos de heurísticas aplicadas ao PEET e os principais algoritmos de construção empregados no problema.

2.1 PEET

A Figura 2.1 ilustra a adaptação da taxonomia proposta em [13], onde inicialmente o problema de escalonamento de tarefas pode ser classificado levando em consideração o número de processadores disponíveis na plataforma distribuída. O escalonamento é dito local quando a designação das tarefas de um programa paralelo é feita a um único processador, enquanto que é denominado global quando essas tarefas são atribuídas a dois ou mais processadores.

De acordo com tal taxonomia, tem-se que o escalonamento do tipo global pode ser dividido em dinâmico e estático. Ele é denominado estático quando as decisões de alo-

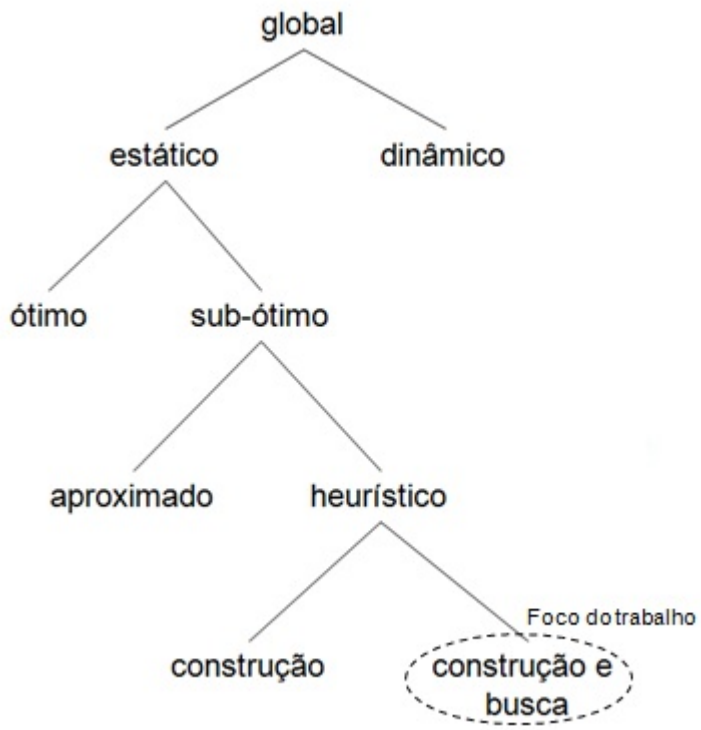


Figura 2.1: Taxonomia apresentada em [13] para o escalonamento de tarefas.

cação são realizadas antes da execução do programa e dinâmico quando essas decisões acontecem durante a execução do mesmo, uma vez que as informações relacionadas aos custos de computação e comunicação podem não ser totalmente conhecidas em tempo de compilação.

Por sua vez, o escalonamento estático pode ser dividido em ótimo e sub-ótimo, contudo são limitados os problemas onde se consegue alcançar um escalonamento ótimo em tempo polinomial para qualquer tamanho do problema [13]. Assim, temos o escalonamento sub-ótimo que é dividido em aproximado, quando se consegue estabelecer um limite de pior caso, e heurístico.

Em [6], tem-se que os métodos heurísticos relacionados ao escalonamento estático podem ser divididos em heurísticas de construção e heurísticas de construção e busca, conforme apresentado na Figura 2.1. Uma das principais diferenças entre as duas classes de heurísticas está no fato de que a primeira constrói a cada passo um único escalonamento como resposta para uma determinada entrada enquanto que a segunda considera um conjunto de escalonamentos como possíveis soluções para o problema.

Em [28], tem-se que o problema de escalonar um conjunto de tarefas para um conjunto de processadores pode ser dividido em duas categorias: "*job scheduling*" e "escalonamento e mapeamento", conforme exibido na Figura 2.2. A primeira está relacionada a tarefas independentes entre si a serem escalonadas em tempo de execução entre os processadores de um sistema de computação distribuído para otimizar todo desempenho do sistema.

É importante ressaltar que aqui entende-se por tarefas independentes, aquelas que podem ser executadas em qualquer ordem, sem restrições de precedências entre elas. Já a segunda requer a alocação de múltiplas tarefas autônomas, interagindo a partir de um programa paralelo único para minimizar o tempo de conclusão no sistema de computação paralelo. Estas tarefas se relacionam através de relações de precedência, isto é, embora autônomas podem depender da execução de outras tarefas para serem iniciadas, tal como será apresentado na Seção 2.2.

Dessa forma, quando as informações desse programa paralelo (custos de computação e comunicação das tarefas, por exemplo) são totalmente conhecidas em tempo de compilação, o problema é dito estático (PEET), caso contrário é denominado dinâmico.

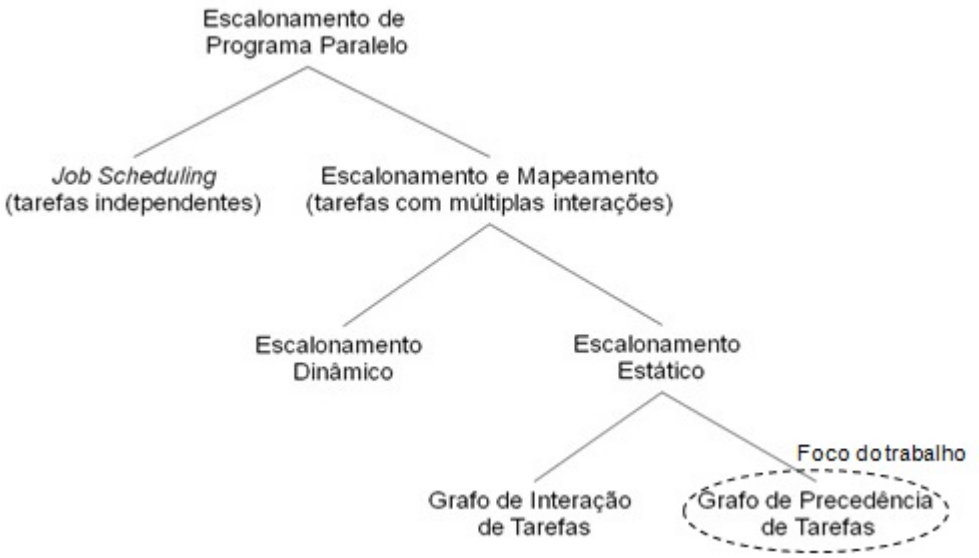


Figura 2.2: Taxonomia simplificada para o escalonamento de programas paralelos [28].

Duas representações distintas de programa paralelo têm sido consideradas extensivamente no contexto de escalonamento estático: GIT (Grafo de Interação de Tarefas) e GAD (Grafo Acíclico Direcionado, também denominado grafo de precedência de tarefas). GIT é normalmente usado no escalonamento estático de processos de comunicação fracamente acoplados (todas as tarefas são consideradas simultaneamente e executadas independentemente não havendo dependência de execução temporal) para um sistema distribuído. Já o GAD é comumente usado no escalonamento estático de um programa paralelo com tarefas fortemente acopladas em multiprocessadores [29].

Considerando-se as taxonomias apresentadas em [13] e [28], o foco deste trabalho é o escalonamento heurístico a partir de grafos de precedência de tarefas.

2.2 Formulação do PEET

No PEET, um programa paralelo pode ser representado por um GAD definido pela seguinte tupla $G_P = (V, E, W, C)$, onde $V = \{t_1, \ldots, t_N\}$ denota o conjunto de N tarefas do grafo; $E = \{e_{i,j} \mid t_i, t_j \in V\}$ representa o conjunto de arestas de comunicação, também denominadas restrições de precedência; $W = \{w_1, \ldots, w_n\}$ representa o conjunto de tempo de execução das tarefas, onde para cada tarefa $t \in V$ é associado um custo computacional $w(t) \in W$ referente ao custo de execução da mesma em qualquer processador da arquitetura; e $C = \{c_{i,j} \mid e_{i,j} \in E\}$ denota o conjunto de custos de comunicação das arestas, isto é, para cada aresta $e_{i,j} \in E$ é associado um custo de comunicação $c_{i,j} \in C$ relacionado ao custo de transferência de dados entre as tarefas t_i e t_j quando são executadas em processadores distintos. Ou seja, o custo de comunicação representa o tempo que a tarefa t_j deve aguardar para iniciar a sua execução, após o término da tarefa t_i. Satisfazendo essas condições G_P é denominado grafo de precedência de tarefas ou, simplesmente, grafo de programa. Na Figura 2.3 tem-se um exemplo de um grafo de programa denominado *gp9*, que representa um conjunto de 9 tarefas. É importante perceber que: (i) cada tarefa é associada a um nó do grafo (cada nó é identificado pelo número de ordem da tarefa a qual representa), (ii) à esquerda de cada nó tem-se o custo computacional da tarefa (quantas unidades de tempo são gastas na execução daquela tarefa, independente do processador) e (iii) em cada aresta há um valor relacionado ao custo de comunicação entre aquelas tarefas. Considerando, por exemplo, a tarefa 6, sabe-se que seu custo computacional é igual a 4, isto é, serão necessárias 4 unidades de tempo para executá-la. Além disso, caso as tarefas 0 e 6 estejam alocadas em processadores diferentes, terminada a tarefa 0, a tarefa 6 aguardará, pelo menos, 10 unidades de tempo para iniciar a sua execução, tempo este correspondente ao atraso na transmissão dos dados obtidos na tarefa 0.

Toda tarefa é autônoma, sendo assim é uma unidade indivísivel de computação, a qual pode ser uma indicação de atribuição, uma subrotina, entre outras. É importante esclarecer também que o conjunto de arestas E define as relações de precedência entre elas. Assim, uma tarefa não pode ser executada a menos que todos os seus predecessores tenham completado suas execuções e todos os dados relevantes estejam disponíveis. Preempção de tarefas e execuções redundantes não são permitidas na versão do problema considerado neste trabalho [45].

Um sistema multiprocessado por sua vez, pode ser representado por um grafo não ponderado e não direcionado $G_s = (V_s, E_s)$, denominado grafo de sistema. V_s é o conjunto de P processadores do grafo de sistema que representa os processadores de um computador paralelo de arquitetura MIMD (*Multiple Instruction Multiple Data*) com suas respectivas memórias locais. E_s é o conjunto de arestas representando canais bi-direcionais entre processadores e definindo a topologia do sistema multiprocessador. Nesse modelo assume-se

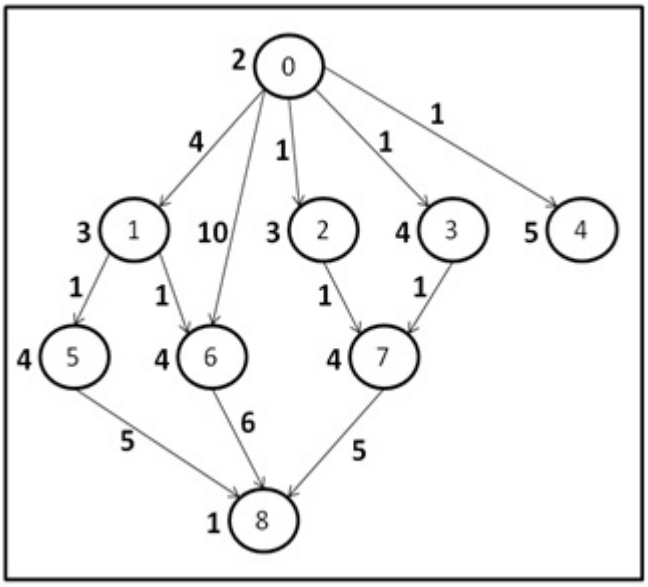


Figura 2.3: Exemplo de um grafo de programa com 9 tarefas (*gp9*).

também que todos os processadores tem o mesmo poder computacional e que as comunicações entre os canais não consomem qualquer tempo adicional do processador além do próprio tempo de comunicação entre as tarefas, já especificado no grafo.

Para apresentar uma solução encontrada no escalonamento de um grafo de programa é comum fazer uso de um gráfico de Gantt. Tal ferramenta é capaz de representar a alocação das tarefas nos processadores e o tempo em que cada uma delas começa e termina sua execução. A Figura 2.4 traz uma solução ótima para o *gp9* alocado em uma arquitetura com 4 processadores idênticos, expressa na forma de um gráfico de Gantt. Na figura, é possível identificar o *makespan*, isto é, a diferença entre o tempo de início e o tempo final de escalonamento da aplicação, que é igual a 16.

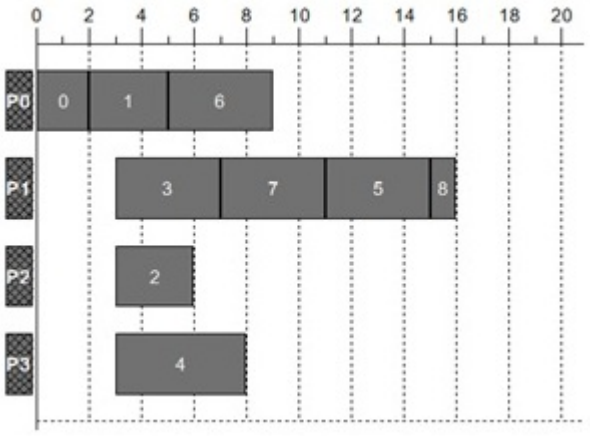


Figura 2.4: Gráfico de Gantt para uma solução ótima do *gp9*.

2.3 Grafos de programa utilizados

Esta seção tem por objetivo apresentar exemplos de grafos de programa utilizados na literatura e também destacar a importância do uso de grafos gerados aleatoriamente para análise de métodos de escalonamento.

2.3.1 Grafos em Trabalhos Correlatos

Conforme já foi apontado, há na literatura alguns grafos utilizados para avaliar as abordagens desenvolvidas para o escalonamento. Assim, é imprescindível utilizá-los também para analisar e comparar o desempenho dos modelos que serão propostos neste projeto.

O grafo de programa apresentado na Figura 2.5 é composto por 18 tarefas e representa a paralelização do algoritmo de eliminação Gaussiana [1] [15]. Destaca-se que todos os pesos do grafo de programa foram escalados pela divisão dos mesmos por 10.

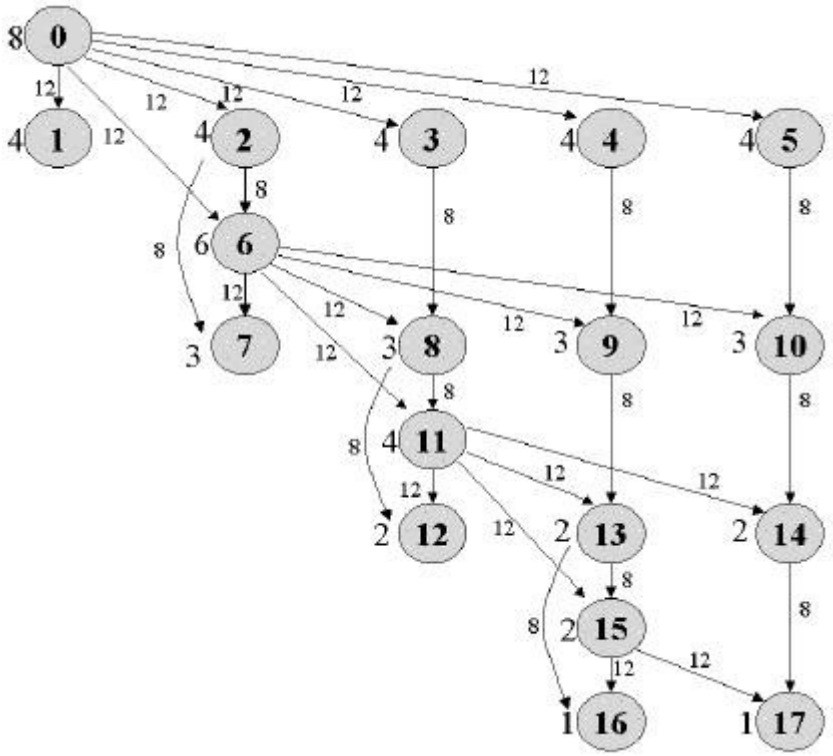


Figura 2.5: Grafo de programa *gauss18*.

Na Figura 2.6(a) também é apresentado um grafo de programa com 18 tarefas [19]. Ele foi denominado *g18*. O custo computacional de cada tarefa está impresso na figura e o custo de comunicação de todas as arestas é igual a 1. Já na Figura 2.6(b) tem-se o grafo de programa chamado *g40*. Ele possui 40 tarefas e o custo computacional de cada tarefa é

[1] o algoritmo de eliminação Gaussiana é bastante utilizado em álgebra linear para determinar soluções de um sistema de equações lineares. Formalmente, este algoritmo recebe um sistema arbitrário de equações lineares como entrada e retorna o seu vetor solução, se ele existe e é único.

igual a 4 enquanto que o custo de cada aresta é igual a 1. A Figura 2.6(c) mostra o grafo de programa de uma árvore binária denominado *outtree15*, que possui 15 tarefas. Em alguns experimentos também foram utilizados grafos que representam árvores binárias com 31, 63, 127, 255, 511 e 1023 tarefas. Os custos de computação e comunicação em todas elas é igual a 1.

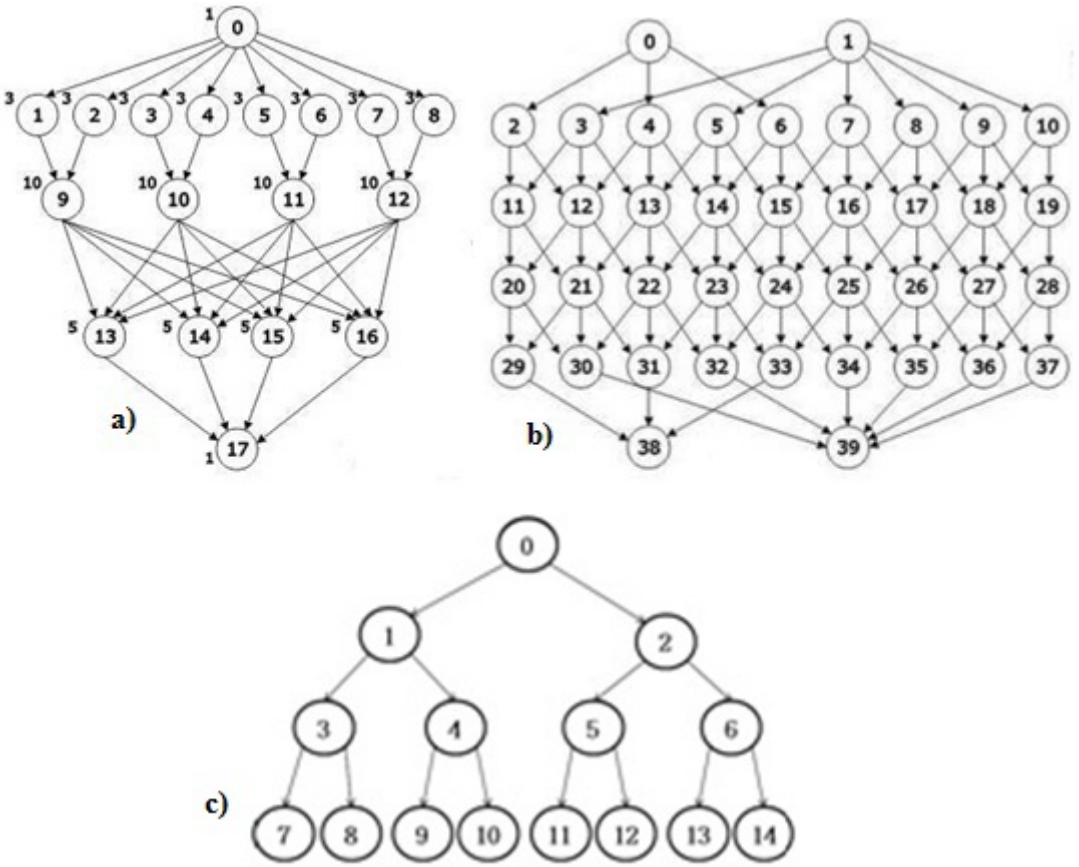


Figura 2.6: Grafos de programa: (a) *g18*; (b) *g40*; (c) *outtree15*.

2.3.2 Grafos Gerados Aleatoriamente

Alguns trabalhos na literatura têm destacado o grande número de algoritmos propostos para o problema de escalonamento, entretanto eles apontam também a falta de uma metodologia eficiente na avaliação desses modelos, de modo que a construção de novos modelos tem se tornado algo fácil diante de testes mal formulados, não padronizados ou ainda muito simples [13, 27, 28].

Diante disto, para a construção de um conjunto de testes realmente robusto e apto a avaliar coerentemente os modelos propostos neste trabalho, adotou-se além dos grafos encontrados na literatura, uma ferramenta desenvolvida em [17], denominada *DAG generation*, a qual tem por objetivo a criação de grafos de programas aleatórios em relação ao custo computacional e às restrições de precedência. A principal modificação realizada na

ferramenta diz respeito a possibilidade de utilizar uma representação de custos baseada em unidades computacionais de tempo para cada tarefa ao invés do tamanho de dados processados por tarefa. Através do gerador é possível criar grafos de programa utilizando diferentes parâmetros relacionados, por exemplo, ao grau de paralelismo entre as tarefas, à densidade do grafo, à regularidade da distribuição entre as tarefas nos níveis do GAD, entre vários outros.

2.4 Heurísticas para o PEET

Em geral, problemas de otimização têm como objetivo maximizar ou minimizar uma função definida sobre um certo domínio. Em problemas de otimização combinatória tem-se que o domínio é tipicamente finito sendo possível testar se um dado elemento pertence a esse domínio. Contudo, a idéia ingênua de testar todos os elementos deste domínio na busca pelo melhor mostra-se inviável na prática, mesmo considerando instâncias de tamanho moderado [12]. O PEET é considerado um desses problemas.

Assim, uma vez que algoritmos exatos não são capazes de encontrar soluções ótimas para quaisquer instâncias do PEET em tempo aceitável, é possível dividir as principais técnicas computacionais utilizadas no problema em heurísticas de construção e heurísticas de construção e busca, conforme apresentado na Seção 2.1.

As heurísticas de construção são caracterizadas por realizarem um único escalonamento baseado em uma série de atributos calculados diretamente de informações do GAD. Enquanto isso, as heurísticas de construção e busca são caracterizadas por trabalharem com um conjunto de possíveis escalonamentos para o GAD considerado. As heurísticas de construção podem ser divididas em vários grupos [6]:

- Escalonamento em lista: propõe basicamente a construção de uma lista de tarefas livres ordenadas de acordo com alguma prioridade;
- Aglomeração de tarefas: criam-se conjuntos ou agrupamentos de tarefas que devem ser executadas em um mesmo processador. O objetivo destas técnicas é minimizar o *makespam* da aplicação pela eliminação das comunicações existentes entre elas;
- Análise de caminho crítico: tentativa de diminuir o caminho crítico (caminho de maior custo de uma tarefa de entrada para uma tarefa de saída) buscando agrupar tarefas a ele pertencentes em um mesmo processador;
- Particionamento de grafos: divide-se o grafo em várias arestas com o objetivo de minimizar arestas que conectam vértices de diferentes partições;
- Replicação: propõe a redução do custo de comunicação entre processadores através da colocação de cópias da mesma tarefa em processadores distintos. É vantajosa quando se tem custos elevados de comunicação e tarefas com alto número de sucessores imediatos;

- *Lookahead*: visa escalonar o grafo de programa considerando tarefas que não estão prontas ou ainda atrasar o início de uma tarefa alocando-a a um processador que já possui algum de seus predecessores.

Como exemplos de heurísticas de construção e busca podem-se destacar técnicas tais como *Simulated Annealing*, Busca Tabu, Algoritmos Genéticos (AGs), entre outras metaheurísticas. A título de exemplo, apresenta-se, a seguir, a aplicação de um AG padrão para o PEET.

O AG parte de uma população inicial onde os indivíduos são representados por possíveis escalonamentos gerados aleatoriamente. Depois, essa população é avaliada em função do tempo de conclusão do escalonamento obtido para cada indivíduo. Posterior a essa avaliação, dá-se início ao processo evolutivo por um número determinado de gerações onde os indivíduos sofrem as operações genéticas de seleção, crossover e mutação, seguidos pelo processo de avaliação onde apenas os melhores indivíduos são mantidos para a geração seguinte. Ao contrário das heurísticas de construção que utilizam as informações do GAD para dirigir o escalonamento, o AG trabalha de forma a tentar melhorar a cada geração a qualidade do seu conjunto de possíveis soluções. Assim, é possível notar que o AG faz uma busca orientada dentro do espaço de possíveis soluções a fim de encontrar a distribuição e ordem das tarefas nos processadores que possibilite o menor custo de escalonamento. Contudo o seu processo de busca é dirigido exclusivamente pela sua estratégia evolutiva. Maiores detalhes sobre os Algoritmos Genéticos são apresentados no Capítulo 3. A seguir são detalhados alguns algoritmos de construção para o PEET.

Antes de mais nada, destaca-se que o intuito de estudar os algoritmos de construção é que alguns deles possuem estruturas de escalonamento muito simples, com bons resultados e que podem ser bem incorporados aos modelos de escalonador baseado em AC deste trabalho. Dessa forma, o objetivo desta seção é apresentar alguns aspectos importantes do estudo relacionado às principais heurísticas de construção, doravante também denominadas algoritmos clássicos, empregadas no PEET. Na literatura existem várias heurísticas de construção para o PEET. A maioria delas possui como características principais o baixo custo computacional e a utilização de diferentes atributos para definir a ordem de escalonamento das tarefas. Devido a esse grande número de heurísticas existentes, esta seção tem por objetivo fazer um levantamento sobre as mais utilizadas e conhecidas.

Em [27], os autores quantificaram a diferença entre cinco heurísticas através de um conjunto de métricas relacionadas à granularidade e outras informações. Além disso, os autores propõem um critério de classificação para GADs e enfatizam ainda o grande número de trabalhos na literatura propondo novos modelos para o escalonamento e a pequena quantidade de pesquisas destinadas a determinar a eficiência deles, a maioria delas utilizando GADs um tanto simples.

Em [28], foram escolhidos quinze algoritmos de escalonamento propostos na literatura e gerados vários conjuntos de teste a fim de elaborar um comparativo entre eles. Os

autores também destacaram a falta de padrões de teste para avaliar os algoritmos e o grande número de heurísticas que são propostas sem antes serem devidamente testadas. Uma outra contribuição deste trabalho foi a construção de uma taxonomia voltada exclusivamente para as heurísticas de construção para o PEET. Uma síntese dessa taxonomia é apresentada na Figura 2.7. Nessa taxonomia, diferentes características do problema foram consideradas tais como: custo de comunicação, replicação de tarefas, número de processadores (limitado ou ilimitado), topologia da rede (completamente ou arbitrariamente conectada), entre outras.

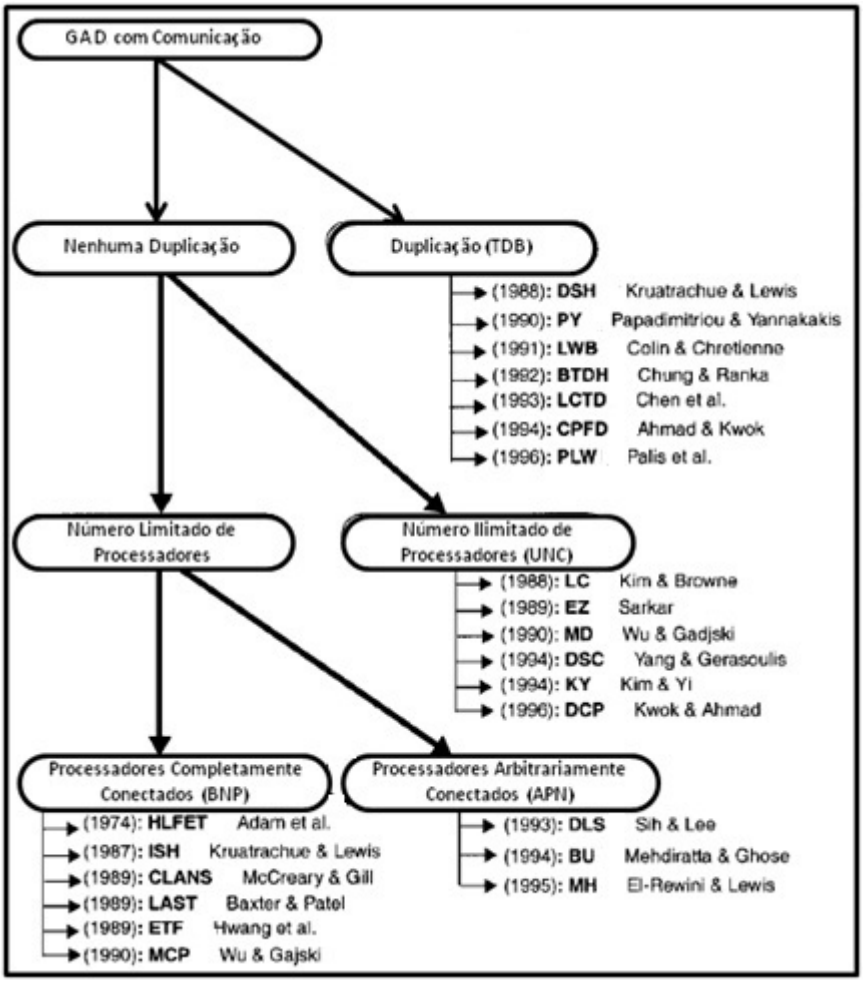


Figura 2.7: Taxonomia geral para o escalonamento estático em multiprocessadores [28].

Considerando ainda a Figura 2.7, é possível extrair dela um modelo para classificação das heurísticas em quatro grupos.

- Número limitado de processadores (BNP - *Bounded Number of Processors*), engloba um conjunto de heurísticas que trabalham com um número limitado de processadores. Em geral, possuem como uma de suas principais características o baixo custo computacional.

- Número ilimitado de agrupamentos (UNC - *Unbounded Number of Clusters*), também denominada *clustering*, está diretamente relacionada àqueles algoritmos que necessitam de um número ilimitado de processadores para realizar o escalonamento. As heurísticas dessa classe consideram cada nó como um *cluster* e usam mais processadores a fim de reduzir o tamanho do escalonamento.
- Duplicação de tarefas (TDB - *Task Duplication Based*) faz referência às heurísticas que utilizam a replicação ou duplicação de tarefas como meio de reduzir o tamanho do escalonamento. O raciocínio por trás dos algoritmos de escalonamento dessa classe é reduzir a "sobrecarga" de comunicação entre tarefas pela alocação redundante de algumas delas em vários processadores.
- Rede Arbitrária de Processadores (APN - *Arbitrary Processor Network*) está ligada aos algoritmos modelados para assumirem uma topologia de rede arbitrária. Assim, além de realizarem o escalonamento de tarefas, eles organizam também mensagens sobre os canais de comunicação da rede.

Levando em consideração a possibilidade de auxílio ou aplicação na proposta deste trabalho, enfatizou-se o estudo de três grupos: BNP, UNC e TDB.

Em [29], são descritas vinte e sete heurísticas de escalonamento e suas funcionalidades, além de uma avaliação acerca do desempenho e complexidade de cada uma delas. No artigo também são apresentadas algumas ferramentas de escalonamento e mapeamento tais como Hypertool, PYRROS e Parallax.

Em [26], foi realizado um estudo comparativo entre nove algoritmos de escalonamento, sendo três deles pertencentes às heurísticas clássicas: HLFET (*Hightest Level First with Estimated Time*), ISH (*Insertion Scheduling Heuristic*) e DSH (*Duplication Scheduling Heuristic*). Além destes, o estudo considerou também algumas heurísticas de construção e busca tais como Algoritmos Genéticos e *Simulated Annealing*, e apontou vantagens e desvantagens dos algoritmos.

Baseado nos trabalhos citados nesta seção, as seguintes heurísticas foram escolhidas para estudo mais aprofundado nesta dissertação: HLFET, ISH, MCP (*Modified Critical Path*), ETF (*Earliest Time First*), DLS (*Dynamic Level Scheduling*), EZ (*Edge Zeroing*), LC (*Linear Clustering*), DSC (*Dominant Sequence Cluster*), DCP (*Dynamic Critical Path*), DSH e CPFD (*Critical Path Fast Duplication*).

De fato, há um grande número de notações e conceitos associados às heurísticas de construção, uma vez que cada uma delas possui suas particularidades e consideram diferentes atributos no escalonamento. Diante disso, serão apresentadas algumas informações para uma melhor compreensão do funcionamento dos algoritmos apresentados nas próximas seções.

As heurísticas podem trabalhar com atributos estáticos ou dinâmicos. Os atributos dinâmicos são calculados a cada etapa do processo de escalonamento levando em conside-

ração as ações já realizadas. Os atributos estáticos, normalmente são calculados apenas uma vez e os valores encontrados são utilizados em todos os demais passos de escalonamento. Algoritmos que utilizam atributos estáticos ou dinâmicos para guiar o processo de escalonamento são classificados, respectivamente, como técnicas de prioridade estática ou dinâmica. Se, por um lado, as últimas refletem as condições do escalonamento em cada etapa, por outro, provocam um aumento em termos de desempenho computacional quando comparadas às primeiras.

Também é importante enfatizar que existem duas abordagens de escalonamento em relação aos espaços de tempo vazios. Espaços de tempo vazios (ETV) são espaços de tempo entre duas tarefas escalonadas que não foram preenchidos devido às relações de dependência da segunda tarefa. Na abordagem de não inserção, uma heurística sempre escalona a tarefa escolhida após a última tarefa escalonada no processador, não considerando ETVs, enquanto que na abordagem de inserção os ETVs são considerados no processo de escalonamento.

As heurísticas de construção realizam o escalonamento guiadas por informações obtidas do GAD através de um ou mais atributos. Logo, esta pesquisa também contemplou o estudo de vários atributos que são apresentados a seguir. Na Tabela 2.1 são mostrados os principais atributos envolvidos nas heurísticas de escalonamento pesquisadas [29]. Uma única heurística pode utilizar vários deles para dirigir o seu escalonamento.

Tabela 2.1: Atributos relacionados às principais heurísticas de construção.

Atributo	**Descrição**
w	custo de processamento de uma tarefa.
c	custo de comunicação entre duas tarefas.
sl	Nível estático de uma tarefa sem considerar custos de comunicação.
b-level	Nível inferior (*bottom level*) de uma tarefa.
D-blevel	Nível dinâmico de uma tarefa.
t-level	Nível superior (*top level*) de uma tarefa.
D-tlevel	Co-nível dinâmico de uma tarefa.
$ALAP$	Tempo de início mais cedo possível de uma tarefa.
CP	Caminho crítico do GAD.

Dentre os atributos apresentados na Tabela 2.1, o custo de processamento e de comunicação já foram apontados na Seção 2.3 e são obtidos diretamente da definição do GAD.

O *b-level* (*bottom-level*) ou nível inferior de uma tarefa i, consiste em encontrar o maior caminho, considerando custos computacionais e de comunicação, de i até uma tarefa de saída. O Algoritmo 1 apresenta um breve esboço de como é realizado o cálculo deste atributo.

Algoritmo 1 Algoritmo para cálculo do atributo *b-level*

```
1: Crie uma lista de nós N em ordem topológica reversa
2: for cada nó n_i em N do
3:    max = 0
4:    for cada filho n_y de n_i do
5:       if c(n_i, n_y) + blevel(n_y) > max then
6:          max = c(n_i, n_y) + blevel(n_y)
7:       end if
8:    end for
9:    blevel(n_i) = w(n_i) + max
10: end for
```

O nível dinâmico (*D-blevel*) é baseado no *b-level*. Entretanto, *D-blevel* é calculado durante o processo de escalonamento, considerando a alocação das tarefas nos processadores. Assim, o custo de comunicação ($c(n_i, n_y)$) é adicionado somente quando as tarefas estão alocadas em processadores distintos.

Outros três atributos da Tabela 2.1 relacionam-se ao *b-level*: o nível estático (sl), o caminho crítico (CP) e o $ALAP$. O algoritmo para cálculo de sl é praticamente o mesmo do *b-level*, exceto por não considerar custos de comunicação ($c(n_i, n_y)$). Por sua vez, o caminho crítico (CP) de um grafo, é idêntico ao caminho "percorrido" pela tarefa que apresenta maior valor de *b-level*, pois, de fato, CP representa o caminho de maior custo possível entre uma tarefa de entrada e uma tarefa de saída. Por fim, o cálculo do $ALAP$ é equivalente à subtração do tamanho do caminho crítico do grafo ($\|CP\|$), pelo *b-level* da tarefa i considerada, $ALAP[i] = \|CP\| - blevel(i)$.

O *t-level* (*top level*) ou nível superior de uma tarefa i, consiste em encontrar o maior caminho, considerando custos computacionais e de comunicação, de i (sem considerar o seu custo de processamento) até uma tarefa de entrada. O Algoritmo 2 apresenta uma descrição sobre as etapas para cálculo deste atributo. Em [56], o *t-level* é chamado de co-nível estático de uma tarefa.

Algoritmo 2 Algoritmo para cálculo do atributo *t-level*

```
1: Crie uma lista de nós N em ordem topológica
2: for cada nó n_i em N do
3:    max = 0
4:    for cada pai n_t de n_i do
5:       if tlevel(n_t) + w(n_t) + c(n_t, n_i) > max then
6:          max = tlevel(n_t) + w(n_t) + c(n_t, n_i)
7:       end if
8:    end for
9:    tlevel(n_i) = max
10: end for
```

A relação entre o co-nível dinâmico (*D-tlevel*) e o *t-level* é praticamente a mesma existente entre o *D-blevel* e o *b-level*. Assim, *D-tlevel* é calculado durante o processo de escalonamento, considerando a alocação das tarefas nos processadores sendo que o custo

de comunicação ($c(n_t, n_i)$) é adicionado apenas quando as duas tarefas (n_t e n_i) estão alocadas em processadores distintos.

No desenvolvimento desta dissertação, as heurísticas pertencentes ao grupo BNP, que consideram um número limitado de processadores, se mostraram mais relevantes ao nosso estudo. Por isso, nesta seção é feita uma revisão das principais heurísticas de construção pertencentes a esse grupo. Um resumo das heurísticas estudadas que pertencem aos grupos UNC e TDB é exibido no Apêndice A.

HLFET, ISH, MCP, ETF e DLS são heurísticas pertencentes ao grupo BNP. Como meio de fornecer um panorama geral sobre essas heurísticas é exibido na Tabela 2.2 um comparativo entre algumas de suas características. Observa-se que os atributos *TIC* (tempo de início mais cedo) e *DL* (*dynamic level*) não foram apresentados anteriormente, pois não são atributos do GAD, mas parâmetros derivados que surgem durante a execução do algoritmo.

Tabela 2.2: Comparativo entre heurísticas BNP.

Heurística	Prioridade	Abordagem	Atributos
HLFET	Estática	Não inserção	*sl*
ISH	Estática	Não inserção	*sl*
MCP	Estática	Inserção	*ALAP*, *sl*, *CP*
ETF	Dinâmica	Não inserção	*sl*, *TIC*
DLS	Dinâmica	Não inserção	*sl*, *DL*

O grafo de programa *gp9*, a ser escalonado pelas heurísticas, foi exibido na Figura 2.3. O sistema multiprocessado que será considerado nas próximas subseções é formado por três processadores. O tempo de conclusão ótimo (T) para este grafo é igual ao exibido na Figura 2.4: 16 unidades de tempo. Embora a figura citada faça uso de quatro processadores, é possível obter uma solução com mesmo *makespan* utilizando três processadores.

2.4.1 HLFET

O algoritmo HLFET (*Hightest Level First with Estimated Time*) [1] pertence às heurísticas de escalonamento em lista e é um dos algoritmos mais simples dessa classe. Na Figura 2.8 é apresentado um esboço do algoritmo. Note que o método é baseado em prioridade estática com uso do atributo nível estático (*sl*) e utiliza abordagem de não inserção, isto é, não considera espaços de tempo vazios (ETV) entre duas tarefas. A seguir descreve-se a aplicação da heurística HLFET sobre o grafo de programa *gp9* (Figura 2.3).

Inicialmente, para cada tarefa do grafo calcula-se o seu respectivo nível estático - *sl*. A Figura 2.9(a) apresenta os valores de *sl* encontrados para cada tarefa em *gp9*. Posterior ao cálculo do atributo, a cada passo do processo de construção do escalonamento, elabora-se uma lista de tarefas-prontas ordenada de modo decrescente em função de *sl* e escalona-se

HLFET (*Hightest Level First with Estimated Times*)	
Passo 0	Calcule o nível estático (sl) de cada tarefa.
Passo 1	Faça uma lista L com tarefas prontas em ordem decrescente de sl. Inicialmente, L contém somente os nós de entrada. Nós de mesmo sl são escolhidos aleatoriamente.
Passo 2	Enquanto todos os nós não são escalonados, faça (Passo 3-4).
Passo 3	Escalone o nó cabeça de L no processador que permita a sua execução mais cedo utilizando a abordagem de *não-inserção*.
Passo 4	Atualize L inserindo os novos nós prontos.

Figura 2.8: Principais passos do algoritmo HLFET.

a tarefa-cabeça dessa lista no processador do sistema que permite a sua execução mais cedo. A Figura 2.9(b) apresenta o processo de escalonamento executado pelo HLFET para o *gp9*. Na figura, a coluna L apresenta a lista de tarefas prontas no início de cada passo, ordenadas em ordem decrescente de *sl*. As colunas P0, P1 e P2 referentes à alocação atual, apresentam as tarefas já alocadas nos processadores no início de cada passo. A coluna Tarefa, apresenta a tarefa alocada a cada passo (cabeça da lista). As colunas P0, P1 e P2 relacionadas ao tempo de início, apresentam os tempos disponíveis para escalonamento em cada processador no início do passo (antes de alocar a nova tarefa no processador). Vale destacar que o processador onde a tarefa é alocada, possui o tempo de início marcado com *.

a)

Tarefa	0	1	2	3	4	5	6	7	8
sl	11	8	8	9	5	5	5	5	1

b)

Passo	L (tarefas prontas)	Alocação atual			Tarefa	Tempo de início		
		P0	P1	P2		P0	P1	P2
1	{0}	{}	{}	{}	0	0*	0	0
2	{3,2,1,4}	{0}	{}	{}	3	2*	3	3
3	{2,1,4}	{0,3}	{}	{}	2	6	3*	3
4	{1,4,7}	{0,3}	{2}	{}	1	6*	6	6
5	{4,7,5,6}	{0,3,1}	{2}	{}	4	9	6	3*
6	{7,5,6}	{0,3,1}	{2}	{4}	7	9	7*	8
7	{5,6}	{0,3,1}	{2,7}	{4}	5	9*	11	10
8	{6}	{0,3,1,5}	{2,7}	{4}	6	13	12*	12
9	{8}	{0,3,1,5}	{2,7,6}	{4}	8	22	18*	22

Figura 2.9: Escalonamento encontrado por HLFET para o *gp9* (a) nível estático (*sl*) das tarefas; (b) processo de escalonamento.

Na Figura 2.10, tem-se o resultado do escalonamento do grafo de programa *gp9* utilizando o algoritmo HLFET. No gráfico de *Gantt* são exibidas as disposições das tarefas

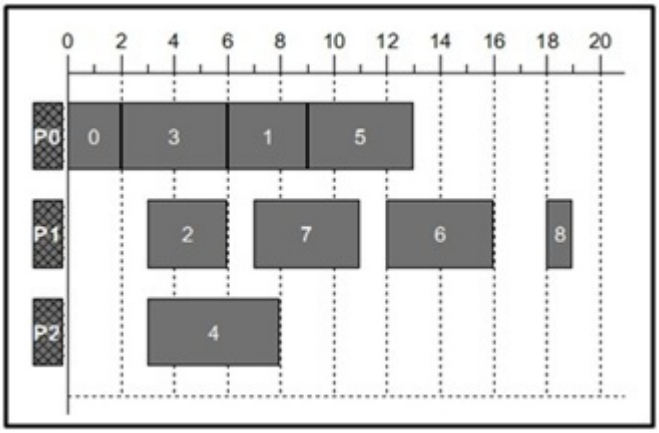


Figura 2.10: Representação do escalonamento final obtido pelo HLFET para o *gp9* em um gráfico de *Gantt*.

em relação aos processadores ao término do processo de escalonamento. O escalonamento encontrado pelo HLFET gastou 19 unidades de tempo (*makespan* = 19).

2.4.2 ISH

O algoritmo ISH (*Insertion Scheduling Heuristic*) trabalha com o conceito de ETV no escalonamento parcial, comumente denominados "buracos de escalonamento". O algoritmo tenta preencher esses buracos com o escalonamento de outras tarefas prontas e utiliza uma abordagem de não inserção. Além disso, o ISH é baseado em prioridade estática (*sl*) e seu esboço é apresentado na Figura 2.11. Um exemplo de aplicação do algoritmo é exibido na Figura 2.12 para o GAD *gp9*.

ISH (*Insertion Scheduling Heuristic*)	
Passo 0	Calcule o nível estático (sl) de cada nó.
Passo 1	Faça uma lista L com tarefas prontas em ordem descendente de sl. Inicialmente, L contém somente os nós de entrada. Nós de mesmo sl são escolhidos aleatoriamente.
Passo 2	Enquanto todos os nós não são escalonados, faça (Passo 3-5).
Passo 3	Escalone o nó cabeça de L no processador que permita a sua execução mais cedo utilizando a abordagem de não-inserção.
Passo 4	Se o escalonamento deste nó causa um espaço de tempo vazio(ETV), então encontre tantos nós quanto possível em L que podem ser escalonados no ETV, mas que não podem ser escalonado em tempo menor em outros processadores.
Passo 5	Atualize L inserindo os novos nós prontos.

Figura 2.11: Principais passos do algoritmo ISH.

ISH inicialmente calcula o nível estático para cada tarefa do grafo, conforme mostrado na Figura 2.12(a). Assim como no HLFET, se duas ou mais tarefas apresentam mesmo

valor de *sl*, a escolha por uma delas é feita aleatoriamente. Uma lista de tarefas-prontas ordenada de modo decrescente em função de *sl* é então criada. ISH escalona a tarefa-cabeça dessa lista no processador do sistema que permite a sua execução mais cedo até que todas tarefas tenham sido escalonadas. A Figura 2.12(b) apresenta o processo de escalonamento executado por ISH para o *gp9*. Os campos L, Alocação Atual (P0, P1, P2), Tarefa e Tempo de Início (P0, P1, P2) seguem a mesma descrição feita na Figura 2.9(b). Na Figura 2.12(b), o campo ETV faz referência aos espaços de tempo vazios em um dado processador ocasionado por determinada tarefa recém escalonada. Quando há ETV na coluna "Passo" significa que a heurística obteve êxito em escalonar uma tarefa pronta em um "buraco de escalonamento" ocasionado no passo anterior. Para destacar a diferença em relação ao método anterior (HLFET) analisemos a Figura 2.12(b) a partir do passo 3, quando na alocação da tarefa 1 no processador P1 (escolhido aleatoriamente entre P0, P1 e P2, por terem o mesmo tempo de início), surge um ETV de 0 à 6 no processador P1. ISH então, verifica se alguma das tarefas em L pode preencher esse "buraco". Dessa forma, o algoritmo encontra a tarefa 2 que pode ser escalonada em P1 entre 3 e 6. Contudo, ela ainda não é alocada, pois ISH analisa se esta tarefa pode ser escalonada com tempo de início menor em algum outro processador. Caso não seja possível, a tarefa é escalonada, tal como acontece na figura.

a)

Tarefa	0	1	2	3	4	5	6	7	8
sl	11	8	8	9	5	5	5	5	1

b)

Passo	L (tarefas prontas)	Alocação atual			Tarefa	Tempo de início			ETV
		P0	P1	P2		P0	P1	P2	
1	{0}	{}	{}	{}	0	0*	0	0	-
2	{3,1,2,4}	{0}	{}	{}	3	2*	3	3	-
3	{1,2,4}	{0,3}	{}	{}	1	6	6*	6	[0..6]
ETV		{0,3}	{}	{}	2	-	3-6*	-	[0..3]
5	{4,7,5,6}	{0,3}	{2,1}	{}	4	6	9	3*	[0..3]
6	{7,5,6}	{0,3}	{2,1}	{4}	7	7*	9	8	[6..7]
7	{5,6}	{0,3,7}	{2,1}	{4}	5	11	9*	10	-
8	{6}	{0,3,7}	{2,1,5}	{4}	6	11*	13	12	-
9	{8}	{0,3,7,6}	{2,1,5}	{4}	8	18*	21	21	[15..18]

Figura 2.12: Escalonamento encontrado por ISH para o *gp9* (a) nível estático (*sl*) das tarefas; (b) processo de escalonamento.

Na Figura 2.13 é exibido o resultado do escalonamento do grafo de programa *gp9*, representado em um gráfico de *Gantt*, utilizando o algoritmo ISH. O custo do escalonamento (*makespan*) encontrado pela heurística foi de 19 unidades de tempo.

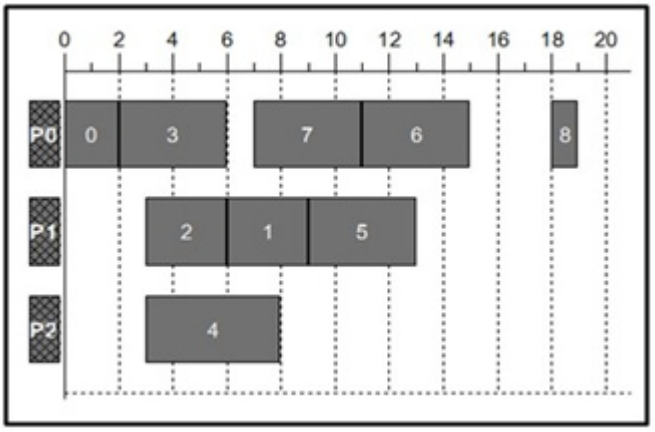


Figura 2.13: Representação do escalonamento final obtido pelo ISH para o *gp9* em um gráfico de *Gantt*.

2.4.3 MCP

Assim como na heurística ISH, o algoritmo MCP (*Modified Critical Path*) [69] também utiliza o conceito de espaços de tempo vazios. Contudo, o MCP olha para um espaço vazio de um dado nó enquanto que o ISH olha para um nó de um determinado espaço vazio. É importante destacar que o MCP utiliza a abordagem de inserção e atributo com prioridade estática ($ALAP$). A heurística é brevemente descrita na Figura 2.14.

MCP (Modified Critical Path)	
Passo 0	Calcule o ALAP de cada nó.
Passo 1	Ajuste os nós em ordem ascendente dos tempos ALAP e crie uma lista de nós *N* de acordo com esta ordem. Caso dois ou mais nós apresentem o mesmo tempo ALAP, escolhe-se o nó cujo filho apresenta menor tempo de ALAP.
Passo 2	Enquanto *N* não está vazia, faça (Passo 4-5).
Passo 3	Escalone o nó cabeça de *N* no processador que permita a sua execução mais cedo utilizando a abordagem de inserção.
Passo 4	Remova o nó de *N*.

Figura 2.14: Principais passos do algoritmo MCP.

Uma descrição da aplicação do MCP é feita a seguir. No primeiro passo do algoritmo, é calculado o tempo $ALAP$ de cada tarefa do grafo. É importante destacar que o caminho crítico para o *gp9* é igual a 23 (0->6->8). A Figura 2.15(a) apresenta os valores de $ALAP$ encontrados para cada tarefa em *gp9*. Uma lista de nós em ordem ascendente dos tempos $ALAP$ é então criada. MCP, em cada passo, escalona o nó cabeça da lista no processador que permite a sua execução mais cedo utilizando a abordagem de inserção (ou seja, verifica se a tarefa a ser alocada pode ocupar algum ETV). A Figura 2.15(b) apresenta o processo de escalonamento executado por MCP para o *gp9*. Na figura, o

escalonamento e atualização da lista de nós N é representado passo-a-passo em cada processador, através da alocação atual e do tempo de início das tarefas.

a)

Tarefa	0	1	2	3	4	5	6	7	8
ALAP	0	8	9	8	18	13	12	13	22

b)

Passo	N (lista de nós)	Alocação atual			Tarefa	Tempo de início		
		P0	P1	P2		P0	P1	P2
1	{0,1,3,2,6,7,5,4,8}	{}	{}	{}	0	0*	-	-
2	{1,3,2,6,7,5,4,8}	{0}	{}	{}	1	2*	6	-
3	{3,2,6,7,5,4,8}	{0,1}	{}	{}	3	5	3*	-
4	{2,6,7,5,4,8}	{0,1}	{3}	{}	2	5	7	3*
5	{6,7,5,4,8}	{0,1}	{3}	{2}	6	5*	12	12
6	{7,5,4,8}	{0,1,6}	{3}	{4}	7	9	7*	8
7	{5,4,8}	{0,1,6}	{3,7}	{4}	5	9	11	6*
8	{4,8}	{0,1,6}	{3,7}	{4,5}	4	9*	11	10
9	{8}	{0,1,6,4}	{3,7}	{4,5}	8	16	15*	16

Figura 2.15: Escalonamento encontrado por MCP para o *gp9* (a) *ALAP* das tarefas; (b) processo de escalonamento.

Na Figura 2.16 tem-se o resultado do escalonamento do grafo de programa *gp9*, representado em um gráfico de *Gantt*, utilizando o algoritmo MCP que encontrou um escalonamento com *makespan* igual a 16 unidades de tempo. No exemplo, não houve nenhum caso de aproveitamento de ETV. Mas, a cada passo, o algoritmo verifica se existe essa possibilidade.

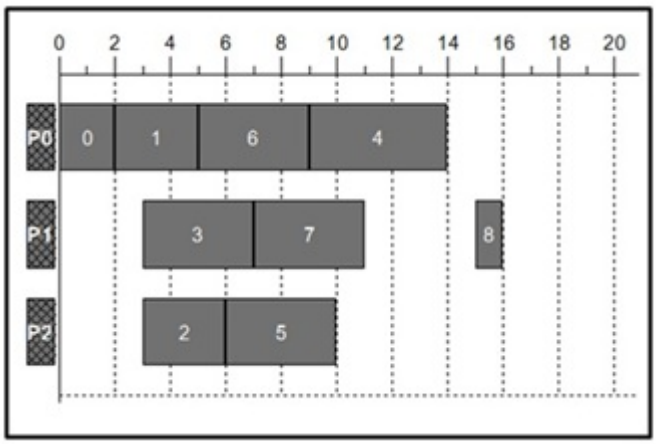


Figura 2.16: Representação do escalonamento final obtido pelo MCP para o *gp9* em um gráfico de *Gantt*.

2.4.4 ETF

O algoritmo ETF (*Earliest Time First*) calcula, em cada processador e a cada passo, o tempo de início mais cedo (*TIC*) para todas as tarefas prontas, e seleciona o par (tarefa, processador) com menor valor desse atributo. O algoritmo é apresentado na Figura 2.17. Vale destacar que essa heurística utiliza prioridade estática (*sl*) e abordagem de não inserção. A seguir tem-se um exemplo da aplicação do ETF.

ETF (Earliest Time First)	
Passo 0	Calcule o nível estático (sl) de cada nó.
Passo 1	Inicialmente, o conjunto de nós prontos *NP* inclui somente os nós de entrada.
Passo 2	Enquanto todos os nós não são escalonados, faça (Passo 3-4).
Passo 3	Calcule o tempo de início mais cedo em cada processador para cada nó em *NP*. Tome o par nó-processador que possui o menor tempo de início considerando a abordagem de não inserção. Em caso de tarefas com mesmo tempo de início, escolhe-se a tarefa de maior sl.
Passo 4	Adicione os novos nós prontos para o *NP*.

Figura 2.17: Principais passos do algoritmo ETF.

Assim como no HLFET e ISH, o primeiro passo de ETF é o cálculo de *sl* para cada tarefa do grafo de programa. Os valores obtidos foram exibidos na Figura 2.18(a). A partir daí, cria-se uma lista de tarefas-prontas para o escalonamento. A cada iteração, o algoritmo calcula o tempo de início mais cedo em cada processador para cada uma das tarefas nessa lista e escalona o par (tarefa, processador) que apresenta o menor tempo considerando a abordagem de não inserção. Ao final de cada iteração, a lista é atualizada. A Figura 2.18(b) apresenta o processo de escalonamento executado por ETF para o *gp9* passo-a-passo. Na figura, são consideradas a alocação das tarefas em cada passo do escalonamento e também o tempo de início de cada tarefa nos processadores do sistema. É importante destacar que no ETF, uma tarefa não é escalonada por ser a cabeça da lista, mas apenas por apresentar menor tempo de início que as demais tarefas, em algum processador.

O escalonamento do grafo de programa *gp9* pelo ETF é apresentado na Figura 2.19 através de um gráfico de *Gantt* e, conforme pode ser observado, possui tempo de conclusão (*makespan*) equivalente a 20 unidades de tempo.

2.4.5 DLS

Esta heurística calcula um atributo dinâmico (ou seja, considera a alocação atual em cada passo) derivado do atributo nível estático (*sl*). Esse atributo derivado é chamado

a)

Tarefa	0	1	2	3	4	5	6	7	8
sl	11	8	8	9	5	5	5	5	1

b)

Passo	NP (nós prontos)	Alocação atual			Tarefa	TIC	Tempo de início		
		P0	P1	P2			P0	P1	P2
1	{0}	{}	{}	{}	0	0	0*	0	0
2	{1,2,3,4}	{0}	{}	{}	1 2 **3** 4	2 2 **2** 2	2 2 2* 2	6 3 3 3	6 3 3 3
3	{1,2,4}	{0,3}	{}	{}	1 **2** 4	6 **3** 3	6 6 6	6 3* 3	6 3 3
4	{1,4,7}	{0,3}	{2}	{}	1 **4** 7	6 **3** 7	6 6 7	6 6 7	6 3* 7
5	{1,7}	{0,3}	{2}	{4}	**1** 7	**6** 7	6* 7	6 7	7 8
6	{5,6,7}	{0,3,1}	{2}	{4}	5 6 7	9 9 7	9 9 9	10 12 7*	10 12 8
7	{5,6}	{0,3,1}	{2,7}	{4}	5 **6**	9 **9**	9 9*	12 12	12 12
8	{5}	{0,3,1,6}	{2,7}	{4}	**5**	**10**	13	11	10*
9	{8}	{0,3,1,6}	{2,7}	{4,5}	**8**	**19**	19*	19	19

Figura 2.18: Escalonamento encontrado por ETF para o *gp9* (a) nível estático (*sl*) das tarefas; (b) processo de escalonamento.

pelos autores de nível dinâmico (*DL*), embora não seja o conceito mais usual de nível dinâmico empregado na literatura (*D-blevel*) [46]. *DL* é a diferença entre o *sl* de uma tarefa e seu tempo de início mais cedo (*TIC*) em um processador. Na Figura 2.20 tem-se um esboço do algoritmo. DLS (*Dynamic Level Scheduling*) utiliza prioridade dinâmica e abordagem de não inserção. Um exemplo da aplicação do algoritmo é descrito a seguir.

Inicialmente, obtem-se o nível estático de cada tarefa conforme mostrado na Figura 2.21(a). A etapa seguinte consiste em incluir as tarefas-prontas em uma lista. Posteriormente, a cada passo do escalonamento, o DLS calcula o *TIC* para todas as tarefas-prontas e seleciona para o escalonamento o par (tarefa, processador) que possui maior valor de *DL*. Ao final de cada passo, o algoritmo atualiza a lista de tarefas-prontas. A Figura 2.21(b) apresenta um esboço do processo de escalonamento executado por DLS para o *gp9*. Na figura, o campo *DL* faz referência ao nível dinâmico da tarefa escolhida (maior *DL* daquele passo). Assim, tal como acontece no ETF, uma tarefa não é escalonada por ser a cabeça da lista, mas apenas por apresentar maior *DL* que as demais tarefas prontas.

O resultado encontrado para o escalonamento do grafo de programa *gp9* pela heurística DLS é apresentado através de um gráfico de *Gantt* na Figura 2.22. O *makespan* foi igual

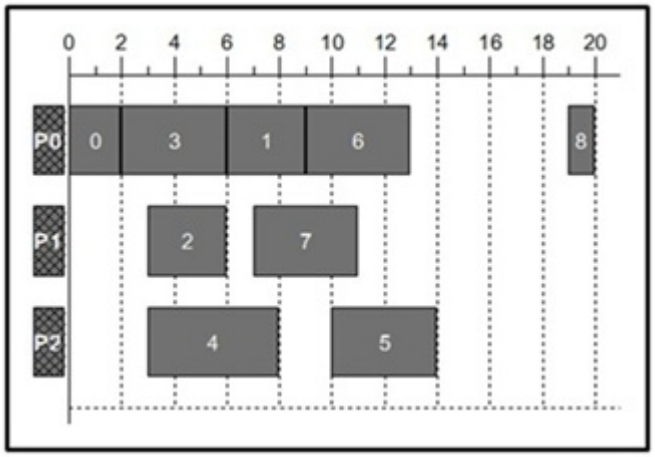


Figura 2.19: Representação do escalonamento final obtido pelo ETF para o *gp9* em um gráfico de *Gantt*.

DLS (Dynamic Level Scheduling)	
Passo 0	Calcule o sl de cada nó.
Passo 1	Inicialmente, o conjunto de nós prontos *NP* inclui somente os nós de entrada.
Passo 2	Enquanto todos os nós não são escalonados, faça (Passo 3-5).
Passo 3	Calcule o tempo de início mais cedo em cada processador para cada nó em *NP*. Calcule o DL (*Dynamic Level*) para cada par nó-processador pela subtração do tempo de início mais cedo pelo sl do nó.
Passo 4	Selecione o par nó-processador que apresenta maior DL. Escalone o nó para o processador correspondente.
Passo 5	Adicione os novos nós prontos para o *NP*.

Figura 2.20: Principais passos do algoritmo DLS.

a 19 unidades de tempo.

a)

Tarefa	0	1	2	3	4	5	6	7	8
sl	11	8	8	9	5	5	5	5	1

b)

Passo	NP (nós prontos)	Alocação atual			Tarefa	DL	Tempo de início		
		P0	P1	P2			P0	P1	P2
1	{0}	{}	{}	{}	0	11	0*	0	0
2	{1,2,3,4}	{0}	{}	{}	1 2 **3** 4	6 6 **7** 3	2 2 2* 2	6 3 3 3	6 3 3 3
3	{1,2,4}	{0,3}	{}	{}	1 **2** 4	2 **5** 2	6 6 6	6 3* 3	6 3 3
4	{1,4,7}	{0,3}	{2}	{}	**1** 4 7	**2** 2 -2	6* 6 7	6 6 7	6 3 7
5	{4,5,6,7}	{0,3,1}	{2}	{}	**4** 5 6 7	**2** -4 -4 -2	9 9 9 7	6 10 12 7	3* 10 12 8
6	{5,6,7}	{0,3,1}	{2}	{4}	5 6 7	-4 -4 **-2**	9 9 9	10 12 7*	10 12 8
7	{5,6}	{0,3,1}	{2,7}	{4}	**5** 6	**-4** -4	9* 9	11 12	10 12
8	{6}	{0,3,1,5}	{2,7}	{4}	**6**	-7	13	12*	12
9	{8}	{0,3,1,5}	{2,7,6}	{4}	**8**	**19**	22	18*	22

Figura 2.21: Escalonamento encontrado por DLS para o *gp9* (a) nível estático (*sl*) das tarefas; (b) processo de escalonamento.

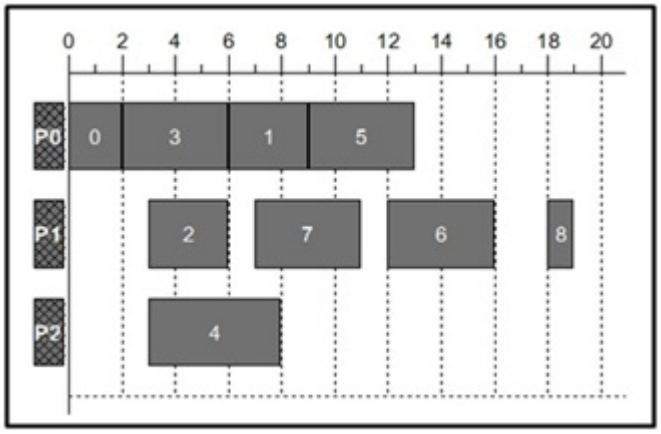


Figura 2.22: Representação do escalonamento final obtido pelo DLS para o *gp9* em um gráfico de *Gantt*.

Capítulo 3

Computação Bio-inspirada

Uma das características que tornam um problema díficil é a complexidade do seu espaço de busca. De fato, em muitos problemas difíceis, algoritmos exatos não são capazes de resolver instâncias consideráveis desse tipo de problema devido ao grande custo computacional dos mesmos. Nesse contexto, técnicas de computação bio-inspiradas têm sido largamente utilizadas em uma tentativa de resolver problemas nos quais não se conhecem algoritmos convencionais eficazes. Muitos desses problemas estão relacionados à otimização combinatória e são bem conhecidos na área de inteligência artificial (IA), tais como: Problema do Caixeiro Viajante [14], Problema do Roteamento de Veículos [8, 22] e o próprio Problema de Escalonamento de Tarefas [41].

A computação bio-inspirada é uma sub-área da computação natural que envolve o estudo de técnicas computacionais cujos mecanismos são inspirados na biologia. Alguns exemplos de técnicas muito conhecidas são as redes neurais artificiais (RNA) e os algoritmos genéticos (AGs), baseadas, respectivamente, no modelo biológico do cérebro humano e no princípio Darwiniano da seleção natural e na genética. Além dos AGs, outra técnica computacional utilizada no desenvolvimento deste trabalho são os autômatos celulares (ACs), originalmente propostos por Von Neumann e Ulam [58] para investigar a auto-reprodução em autômatos e posteriormente utilizados por John Conway [3] para simulação sistemas complexos de vida artificial.

Neste capítulo são apresentados os conceitos fundamentais relacionados aos algoritmos genéticos (Seção 3.1) e aos autômatos celulares (Seção 3.2).

3.1 Algoritmos Genéticos

Algoritmos genéticos (AG) são métodos computacionais estocásticos de busca e otimização guiados pela simulação dos mecanismos de seleção natural e genética [23, 24]. Os AGs utilizam uma estratégia de busca paralela, estruturada e de caráter estocástico. Diz-se que a estratégia é paralela em decorrência da sua organização em conjuntos de possíveis soluções e é estruturada devido às características pré-determinadas da forma como se dá a

busca. O caráter estocástico é devido à natureza probabílistica da busca, por exemplo, na geração inicial de um conjunto de possíveis soluções ou na probabilidade que uma dessas soluções passe por alguma modificação em outra etapa do processo evolutivo.

No AG, uma população de possíveis soluções para o problema em questão evolui de acordo com operadores probabilísticos concebidos a partir de metáforas biológicas, de modo que há uma tendência de que, em média, os indivíduos representem soluções cada vez melhores à medida que o processo evolutivo continua [54]. Dentre as metáforas utilizadas para descrever as estruturas do AG, destacam-se:

- Indivíduo: estrutura que representa uma possível solução para o problema. Pode ser composto de um ou mais cromossomos;
- Cromossomo: conjunto de genes que determinam as características do indivíduo. Normalmente composto por diversos genes;
- Gene: denominação dada a cada característica que compõe um cromossomo;
- População: nome utilizado para se referir ao conjunto de indivíduos (soluções) manipulados durante a busca;
- Aptidão: função responsável por avaliar a qualidade de cada indivíduo da população como solução em potencial para o problema;
- Seleção: procedimento responsável por selecionar os melhores indivíduos da população para serem submetidos aos operadores genéticos;
- Cruzamento: processo de troca de genes entre dois indivíduos selecionados;
- Mutação: possibilidade dos genes dos novos indivíduos (descendentes) sofrerem alguma alteração;
- Geração: iteração do processo de busca, sendo que a cada iteração uma nova população é gerada e avaliada.

O primeiro passo de um AG é a geração aleatória de uma população inicial e sua consequente avaliação de acordo com uma determinada função de aptidão. Posterior a isso, inicia-se um ciclo de iterações onde normalmente utiliza-se como critério de parada o número máximo de gerações. Enquanto tal condição de parada não for satisfeita, realiza-se a seleção e a aplicação dos operadores genéticos de cruzamento e mutação. É importante salientar que o intuito da seleção é encontrar bons indivíduos para as etapas seguintes. O cruzamento por sua vez, tem por objetivo, através da troca de material genético entre os indíviduos selecionados, gerar descendentes cujos genes herdem as boas características dos pais. Após esse procedimento, esses mesmos descendentes são submetidos a uma probabilidade de mutação, onde a estrutura genética deles pode ou não ser alterada. Após a mutação, faz-se a avaliação dessa população de descendentes e adota-se uma estratégia de re-inserção para obter a população final. Nesse passo, pode ser utilizada a re-inserção

baseada na aptidão, onde os melhores indivíduos dentre pais e filhos constituem a nova população e os demais são descartados. Terminada a re-inserção, uma nova geração é contada e o ciclo volta ao seu início. Ao atingir a condição de parada (fim do algoritmo), a solução encontrada é o melhor indivíduo da população. A Figura 3.1 apresenta um esquema desse processo evolutivo.

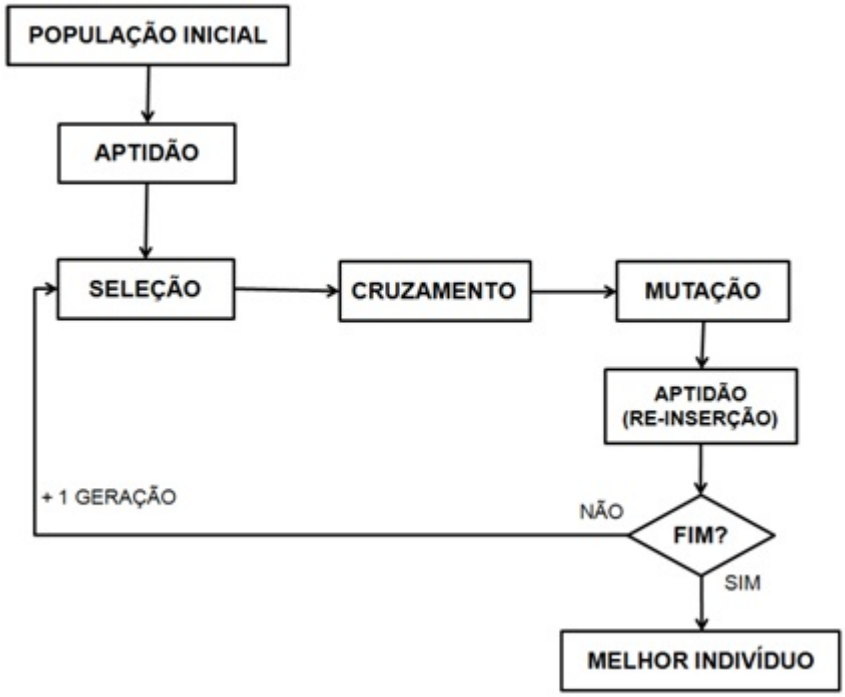


Figura 3.1: Esquema do processo evolutivo em um Algoritmo Genético (adaptado de [7]).

Considerando o problema abordado neste trabalho e apresentado no Capítulo 2, os próximos tópicos detalham os principais aspectos do projeto de um algoritmo genético e apresentam a construção passo-a-passo de um AG para o PEET.

3.1.1 Representação dos Indivíduos e População Inicial

Dois dos principais aspectos a serem considerados no uso de um AG dizem respeito às escolhas da representação cromossômica dos indivíduos e da função de avaliação. Note que apesar da importância dos operadores genéticos, estas etapas merecem maior atenção pois são completamente dependentes do problema considerado. Assim, a representação do indivíduo depende do tipo de problema e do que, essencialmente, se deseja manipular. Uma das codificações mais utilizadas é a binária uma vez que possui estrutura bastante simples e facilidade para manipulação dos cromossomos através dos operadores genéticos. Contudo, existem também representações baseadas em números reais, números inteiros, entre outras [40]. É até mesmo possível para alguns problemas a utilização de indivíduos com representações distintas [20].

Ao analisar o PEET, é possível representar os indivíduos como uma matriz do tipo inteiro, onde o número de colunas é dado pelo número de tarefas do grafo de programa considerado e o número de linhas é igual a 2, onde cada elemento da primeira linha representa uma tarefa específica enquanto a segunda linha refere-se ao nó em que a tarefa está alocada. A ordem das tarefas na primeira linha é que define a ordem delas nos processadores. Na Figura 3.2 tem-se um exemplo dessa representação, considerando o grafo de programa *gp9* (apresentado na Figura 2.3). A Figura 3.2 representa a seguinte solução de escalonamento de 9 tarefas (t0, t1, ..., t8) em 3 processadores (P0, P1, P2), que poderia ter sido gerada aleatoriamente na população inicial. Esse indivíduo é uma possível representação do seguinte escalonamento apresentado logo abaixo.:

Tarefa	0	1	3	2	6	7	5	8	4
Processador	0	0	1	2	0	1	2	1	0

Figura 3.2: Representação de um indivíduo no AG para o PEET.

- P0: t0->t1->t6->t4
- P1: t3->t7->t8
- P2: t2->t5

Definida a representação cromossômica, o primeiro passo do processo de execução do AG é a geração de uma população inicial de indivíduos, com características aleatórias. É muito importante que os indivíduos gerados representem possíveis soluções para o problema. Por exemplo, considerando o PEET, é importante que as tarefas em cada indivíduo estejam dispostas no vetor de modo a não desobedecer nenhuma das restrições de precedência do grafo de programa. Em alguns casos, também é possível incorporar heurísticas para direcionar a construção dos indivíduos, entretanto é importante ter cuidado, pois assim como ocorre na natureza, a variedade de indivíduos contribui para que ocorra a seleção natural.

Assim, no caso do PEET, a primeira linha de cada indivíduo é dada por uma permutação aleatória dos números correspondentes às tarefas (para o grafo de programa *gp9*, de 0 a 8) e a segunda linha é um sorteio aleatório de uma palavra ternária (no exemplo, utiliza-se a base três porque são três processadores). Depois da geração dessas duas linhas de forma aleatória, um procedimento verifica se as ordens de precedência dentro de cada processador são respeitadas. Se não forem, o procedimento corrige a ordem. Por exemplo, suponha que em P0 seja sorteada a ordem t1->t0->t4->t6. No caso do grafo de programa *gp9* (Figura 2.3), t1 não pode ser executada antes de t0 e o procedimento corrige a ordem sorteada para: t0->t1->t4->t6.

3.1.2 Avaliação

A avaliação dos indivíduos é uma etapa extremamente importante nos algoritmos genéticos, uma vez que é responsável por direcionar a busca evolutiva. A função de aptidão é específica para cada problema e a definição de uma função capaz de avaliar de forma eficiente os pontos do espaço de busca do problema é fundamental para o sucesso de qualquer AG.

Ao ser avaliado, cada indivíduo recebe um valor que o caracteriza em relação ao espaço de possíveis soluções para o problema e em relação aos demais indivíduos da população. Esse valor é denominado valor de aptidão do indivíduo e normalmente é utilizado também nos métodos de seleção a fim de que os melhores indivíduos tenham maior probabilidade de participarem do cruzamento e consequentemente gerar descendentes com partes do seu material genético.

Uma boa função de avaliação para o PEET deve considerar somente indivíduos válidos, ou seja, indivíduos onde as restrições de precedência do grafo de programa sejam respeitadas. A função de aptidão pode ser calculada através do escalonamento do indivíduo, ou seja, o tempo total de escalonamento - *makespam* - é atribuído como seu valor de aptidão. Nesse caso, é possível notar que o cálculo da aptidão deverá ser realizado apenas uma vez para cada indivíduo. Contudo existem problemas em que a aptidão é recalculada a cada geração, como por exemplo, na abordagem híbrida de escalonamento com AC e AG, que será apresentada no Capítulo 4.

3.1.3 Seleção

O processo de seleção é responsável por selecionar, a cada geração, pares de pais para o cruzamento. Esse procedimento é diretamente responsável por determinar a pressão seletiva do AG. Por pressão seletiva entende-se a tendência com que os melhores indivíduos são favorecidos no processo de seleção. Assim, é interessante perceber que se a pressão seletiva for demasiadamente baixa, o processo de convergência será lento e o AG, desnecessariamente, poderá tomar um caminho mais longo para encontrar uma boa solução. Entretanto, se essa pressão for bastante alta, provavelmente o AG convergirá prematuramente para um mínimo local. Dessa forma, o ideal é que o método de seleção preserve a diversidade da população, além de prover uma pressão seletiva adequada.

Alguns dos mecanismos mais utilizados na seleção são a roleta e o torneio simples. A roleta é um dos processos de seleção mais conhecidos [40]. A sua inspiração é a própria roleta presente nos jogos de azar. Assim, proporcionalmente à sua aptidão, cada indivíduo recebe casas na roleta. É comum, em problemas de maximização, realizar um somatório da aptidão de todos os indivíduos que consequentemente define o tamanho máximo T_{MAX} da roleta e a posição de cada indivíduo. Para a seleção de um indivíduo, a roleta é "girada", ou seja, sorteia-se, considerando T_{MAX}, um número aleatório que representa a

casa de algum dos indivíduos na roleta. Obviamente, os indivíduos com mais casas terão mais chances de serem sorteados.

Já na seleção por torneio, inicialmente define-se o tamanho do torneio T_{TOUR} que indica a quantidade de indivíduos a serem escolhidos, de forma aleatória e com mesma probabilidade, para o torneio. O indivíduo com maior aptidão, entre os escolhidos, é considerado vencedor do torneio e é selecionado para o cruzamento. Uma vantagem deste tipo de seleção é a sua simplicidade e seu baixo custo computacional. Assim, no modelo de AG para o PEET, o torneio simples foi empregado para seleção dos indivíduos.

3.1.4 Cruzamento

O cruzamento é a etapa onde ocorre a troca de material genético entre os pares de pais selecionados anteriormente, simulando o processo de troca de material genético que ocorre na natureza. Ele é um operador muito discutido devido à sua natureza rompedora, uma vez que pode separar informações importantes. Nos AGs, a sua aplicação é fundamental com o intuito de preservar um bom material genético e possibilitar a formação de outros ainda melhores. A frequência com que o cruzamento ocorre é controlada pelo parâmetro P_{cross}, probabilidade ou taxa de cruzamento, que determina a probabilidade de cada indivíduo ser submetido ao operador de cruzamento.

As abordagens mais conhecidas para a reprodução são o cruzamento de um ponto (ou ponto-simples) e o cruzamento multiponto (ou múltiplo), especialmente quando a representação do indivíduo é binária. As Figuras 3.3(a) e 3.3(b) apresentam, respectivamente, um exemplo de cruzamento de ponto-simples e multiponto. Contudo, é sempre importante lembrar que o tipo de cruzamento escolhido está diretamente relacionado à representação dos indivíduos, sendo possível encontrar vários outros modelos de cruzamento na literatura.

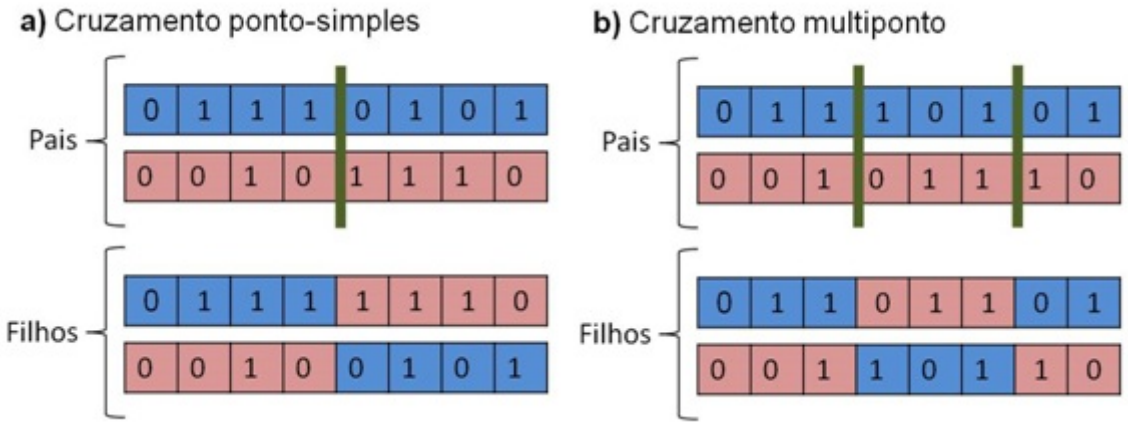


Figura 3.3: Cruzamento: (a) ponto-simples; (b) multiponto.

Ao considerar a representação escolhida para o PEET na Seção 3.1.1, optou-se por

utilizar um cruzamento denominado cíclico cujo exemplo é exibido na Figura 3.4. A vantagem desse tipo de cruzamento em relação aos citados anteriormente é que ele exclui a possibilidade de gerar indivíduos com tarefas repetidas. Assim, esse tipo de cruzamento é especialmente indicado para problemas de permutação.

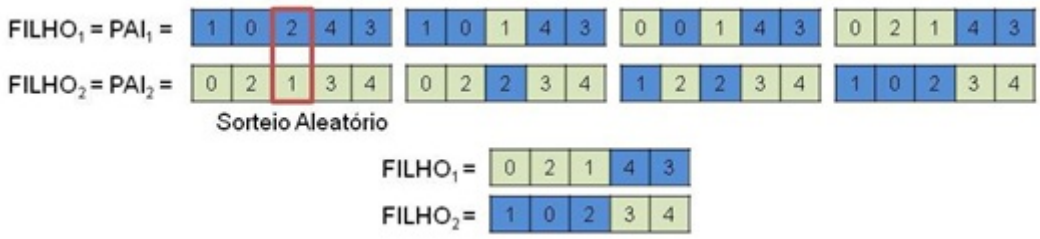


Figura 3.4: Cruzamento cíclico.

Conforme a Figura 3.4, inicialmente os filhos são cópias dos pais. No primeiro passo, uma posição é escolhida aleatoriamente no FILHO1 e há troca daquele gene com o FILHO2, iniciando o ciclo. O próximo gene a ser trocado é sempre aquele que pertence ao FILHO1 e apresenta o mesmo valor do gene recém-obtido do FILHO2. O ciclo (cruzamento) termina quando não há nenhum gene com o mesmo valor.

No caso do PEET, como o indivíduo é uma matriz de duas linhas, onde apenas a primeira linha representa uma permutação, o cruzamento cíclico é aplicado observando o ciclo para troca apenas pela linha das tarefas. As alocações nos processadores (2^a linha), acompanham as tarefas trocadas. A Figura 3.5 apresenta um exemplo de cruzamento de 2 soluções para o grafo de programa *gp9*.

Figura 3.5: Cruzamento cíclico para o PEET.

3.1.5 Mutação

A mutação tem como objetivo aumentar a diversidade de indivíduos na população. Ela consiste em alterações aleatórias na estrutura genética dos filhos gerados no cruzamento, sendo que estas alterações estão condicionadas à uma probabilidade P_M, denominada taxa de mutação. Esse operador genético, juntamente com o cruzamento, estimula a exploração de novos pontos do espaço de busca. É válido destacar ainda que a taxa P_M pode ser dada em razão do indivíduo ou do gene.

Os tipos mais comuns de mutação estão relacionados ao complemento do *bit*, quando consideramos representações binárias, e à permutação, quando há troca entre os valores de dois ou mais genes. Na Figura 3.6(a) é mostrado um exemplo da mutação do tipo complemento do *bit*. Já na Figura 3.6(b) tem-se um exemplo de mutação do tipo permutação que é a mesma adotada no AG para o PEET.

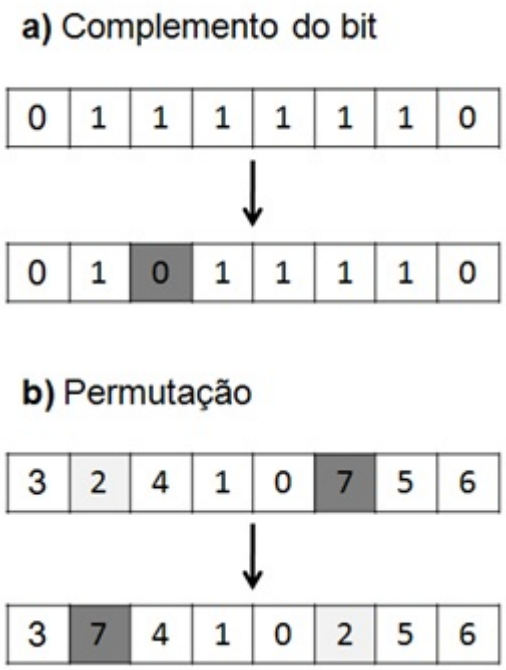


Figura 3.6: Mutação: (a) tipo complemento do *bit*; (b) tipo permutação.

No caso do PEET, as trocas são feitas baseadas na 1^a linha da solução (tarefas) e a 2^a linha (alocação) acompanha as trocas. A Figura 3.7 apresenta um exemplo de mutação em uma solução para o grafo de programa *gp9*. Além disso, na 2^a linha, uma mutação do tipo permutação é aplicada, onde o processador em que uma tarefa escolhida aleatoriamente está alocada pode ser alterado. O sorteio da posição a ser alterada na 2^a linha, é independente do sorteio das duas posições trocadas na 1^a linha.

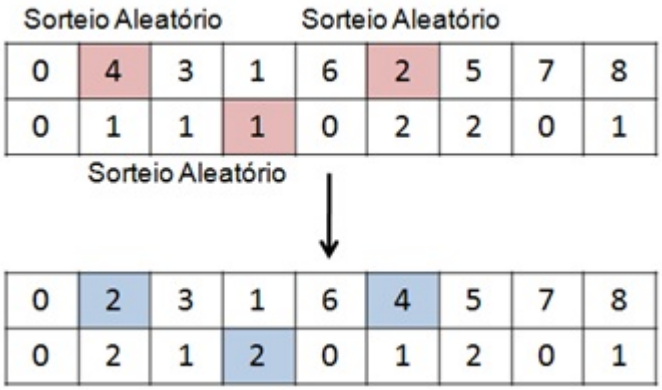


Figura 3.7: Exemplo de mutação do tipo permutação no AG para o PEET.

3.1.6 Re-inserção

Após a aplicação dos operadores genéticos, é preciso definir quais indivíduos dentre pais e filhos passarão para próxima geração do algoritmo, tal procedimento é denominado re-inserção. Alguns dos tipos de re-inserção mais utilizados são:

- Elistismo: um percentual dos melhores indivíduos é mantido para a próxima geração.
- Re-inserção uniforme: um método de seleção, tais como roleta ou torneio simples, é aplicado para definir os indivíduos da próxima geração.
- Re-inserção baseada na aptidão: os melhores indivíduos dentre pais e filhos são selecionados para a próxima geração.

Para o PEET, utilizou-se a re-inserção baseada na aptidão.

3.1.7 Fluxo Geral do Algoritmo Genético

O Algoritmo 3 apresenta o fluxo geral de um AG. Considerando a representação adotada para o PEET, os métodos utilizados pelo AG, denominado AG-1, foram torneio simples para seleção, cruzamento cíclico, mutação do tipo permutação e re-inserção baseada na aptidão. Maiores detalhes sobre os parâmetros utilizados em AG-1 podem ser vistos na Seção 5.1.2.

Algoritmo 3 Fluxo geral de um AG

1: Gere uma população inicial com T_{POP} indivíduos
2: Avalie T_{POP}
3: $t \Leftarrow 0$
4: **while** $t < numGeracoes$ **do**
5: Selecione pares de pais em T_{POP} através de um método de seleção (roleta, torneio, etc)
6: Gere T_{DESC} indivíduos através de um método de cruzamento (ponto-simples, cíclico, etc) entre os pares de pais
7: Submeta T_{DESC} à mutação (complemento do bit, permutação, etc) *troca do bit* com probabilidade P_M
8: Avalie T_{DESC}
9: Utilize um método de re-inserção (elitismo, uniforme, baseada na aptidão, etc)
10: $t \Leftarrow t + 1$
11: **end while**
12: **return** melhor indivíduo de T_{POP}

3.2 Autômatos Celulares

Os autômatos celulares foram originalmente desenvolvidos na década de 1940 por Ulam e von Neumann com o intuito de prover uma estrutura formal para investigar o comportamento de sistemas complexos [58]. Contudo, dois outros pesquisadores também se destacam por suas contribuições na área. John Conway, na década de 70, investigava se seria possível criar um autômato celular simples com computabilidade universal [3]. O resultado de sua pesquisa foi o *Game of Life*. Conway popularizou os ACs nos meios acadêmicos, pois foi o primeiro exemplo de AC relativamente simples que mostrou sua habilidade em produzir padrões complexos e estruturas que se assemelhavam a organismos artificiais. Contudo, os trabalhos de Wolfram [59–64] na década de 80, sobre o estudo do comportamento dinâmico dos ACs, se tornaram fonte de referência de todos os trabalhos posteriores nesta área. Uma de suas principais contribuições, que mudou o rumo das pesquisas na área, foi demonstrar que mesmo os modelos de ACs mais simples - unidimensionais, binários e com vizinhança formada por 3 células - seriam capazes de exibir padrões interessantes, com comportamento emergente [37].

ACs são sistemas dinâmicos com tempo, espaço e variáveis discretos [44]. Eles são compostos por um conjunto de componentes simples e idênticos (denominados células) que possuem conectividade local entre si e apresentam como uma de suas principais características a capacidade de emergir um comportamento global a partir de interações entre essas unidades locais.

Nas próximas seções apresentam-se os principais conceitos e estudos relacionados aos ACs, importantes para o entendimento deste trabalho.

3.2.1 Definições e Notações

Basicamente, um AC é um sistema composto por um espaço celular e uma função ou regra de transição de estados. O espaço celular é um reticulado de l células (componentes simples e idênticos que possuem conectividade local e condição de contorno) dispostas em um arranjo d-dimensional. Cada célula assume um estado de um conjunto finito de k estados possíveis a cada instante de tempo. A regra (ou função) de transição f por sua vez, é responsável por determinar o próximo estado de cada célula do AC a partir de seu estado atual e dos estados de suas vizinhas. Apesar da aparente simplicidade na definição das regras de transição, é interessante notar que a previsão da dinâmica de um AC a partir da análise exclusiva dessas regras é um problema indecidível [16].

A regra de transição determina o comportamento apresentado pelo AC durante a evolução temporal do reticulado que é o processo de aplicar a regra sobre o reticulado por um número determinado de passos de tempo t. Durante a evolução temporal, a atualização das células pode acontecer dos seguintes modos [49]:

- **Paralelo ou Síncrono**: onde todas as células do reticulado atualizam seus estados sincronamente em cada passo de tempo. De modo geral, essa é a forma mais usual de atualização na literatura dos ACs [44].
- **Sequencial ou Assíncrono**: em que apenas uma célula por vez atualiza o seu estado e esse novo estado é considerado na atualização das outras células. Diz-se sequencial porque a ordem em que cada célula é atualizada é dada pela sua posição no reticulado, da esquerda para a direita. Essa foi a forma de atualização mais investigada de ACs no contexto específico do problema de escalonamento.
- **Sequencial-Aleatório**: a atualização das células é semelhante ao modo sequencial porém a ordem de atualização das células é definida aleatoriamente.

Para cada célula i, chamada célula central, uma vizinhança de raio R é definida. Assim, o tamanho da vizinhança de cada célula i (que inclui a própria célula) é dado por: $m = 2R + 1$. É importante perceber que, se a célula i no tempo t apresenta estado s_i^t, então no tempo $t + 1$ o seu estado s_i^{t+1} dependerá apenas dos estados das células de sua vizinhança no tempo t, ou seja, será dado por: $s_i^{t+1} = f(s_{i-R}^t, \ldots, s_{i-1}^t, s_i^t, s_{i+1}^t, \ldots, s_{i+R}^t)$.

Como exemplo, na Figura 3.8(a) é exibido um autômato celular unidimensional ($d = 1$), binário ($k = 2$), com dez células ($l = 10$) e vizinhança de raio 1 ($m = 3$). Na Figura 3.8(b) tem-se a regra de transição (f) que apresenta o novo estado da célula central (q_i^{t+1}) para todas as configurações possíveis da vizinhança ($q_{i-R}^t, \ldots, q_i^t, \ldots, q_{i+R}^t$). Na Figura 3.8(c) é apresentada a evolução temporal do reticulado por 2 passos de tempo considerando-se o modo de atualização síncrono e com condição de contorno periódica (célula mais à esquerda está conectada à célula mais à direita, formando um anel), por dois passos de tempo. Na Figura 3.8(d) é apresentada a evolução temporal do reticulado

em modo sequencial e com condição de contorno nula (tanto a vizinhança à esquerda da 1^a célula quanto a vizinhança à direita da última célula são consideradas no estado 0) também por dois passos de tempo. Na figura, o primeiro passo de tempo foi representado de forma detalhada, apresentando a atualização sequencial de cada célula até se obter o novo reticulado. No segundo passo de tempo, apenas o resultado final do reticulado (após essa atualização sequencial) está representado.

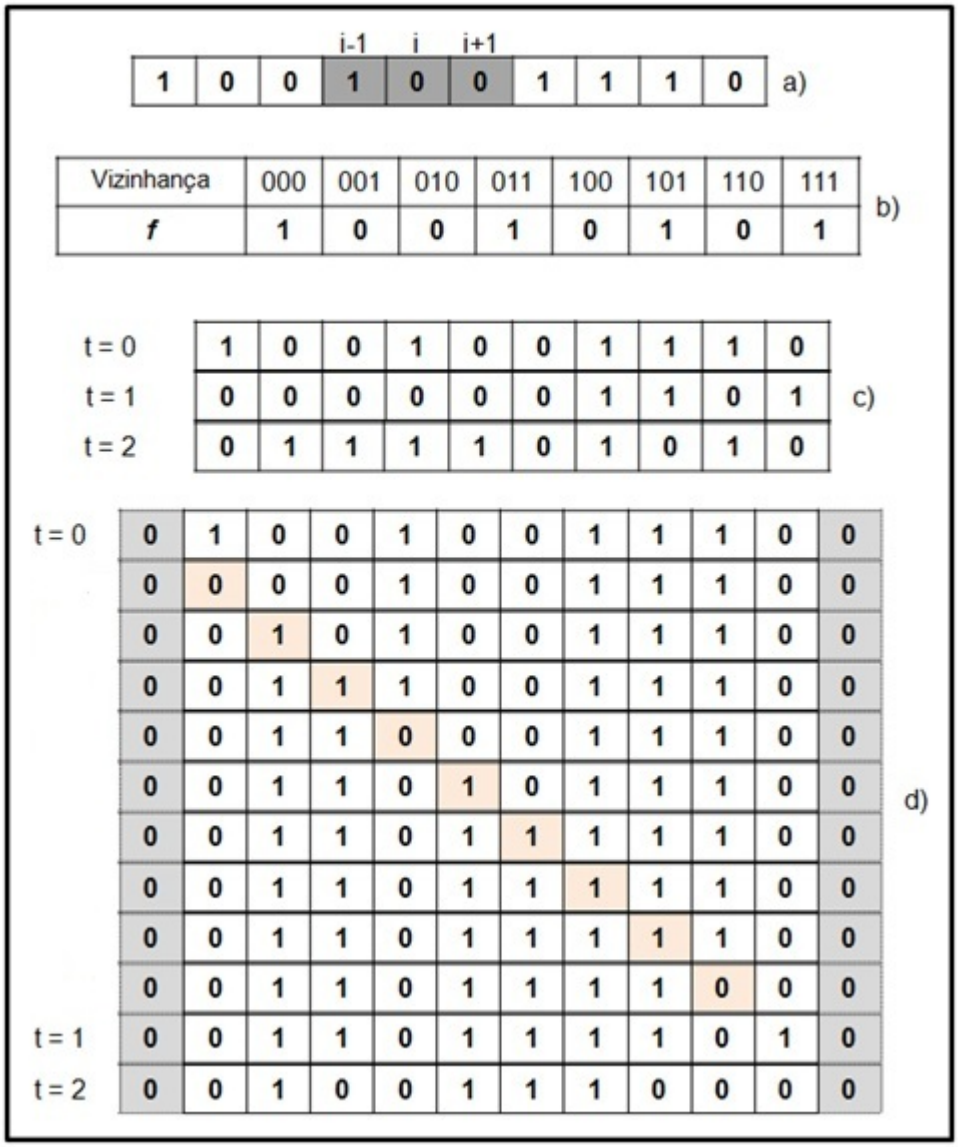


Figura 3.8: Exemplo de AC: (a) reticulado inicial; (b) regra de transição; (c) evolução temporal com modo de atualização síncrono; (d) evolução temporal com modo de atualização sequencial.

É interessante analisar também a cardinalidade do espaço de regras do autômato celular. De modo geral, quando considera-se um AC binário ($k = 2$), com raio $R = 1$ ($m = 3$) sabe-se que o tamanho da regra ($T_{rule} = k^m$) é igual a 8 e o número de possíveis regras ($N_{rules} = k^{T_{rule}}$) é igual a 256. Se $R = 2$ e mantendo-se $k = 2$ então $T_{rule} = 32$

e $N_{rules} = 2^{32}$. E se $k = 3$ com $R = 2$, então $T_{rule} = 243$ e $N_{rules} = 3^{243}$. Assim, a complexidade dos ACs em relação a T_{rule} e N_{rules} está diretamente ligada ao tamanho do raio R e ao número de possíveis estados k que as células podem assumir.

3.2.2 Dinâmica dos Autômatos Celulares

Existe uma forte dependência da dinâmica do reticulado em relação à regra de transição. Tal relação tem como consequência principal a possibilidade de uma enorme variedade de comportamentos dinâmicos. Muitos métodos para análise da dinâmica em ACs têm sido pesquisados e até mesmo adaptados de outras áreas, como por exemplo a análise de bacias de atração, derivada do estudo de sistemas dinâmicos contínuos. Assim, autômatos celulares têm se consolidado como importantes objetos de estudo para a área de sistemas dinâmicos [37].

Existem diferentes formas de se analisar a dinâmica de um determinado AC, contudo uma das mais diretas é observar o comportamento dos diagramas de padrões espaço-temporais gerados por ele a partir de vários reticulados aleatórios. Outra maneira possível é a análise do espaço de estados [66–68]. Contudo, a análise da regra de transição associada ao AC, a qual define um espaço de regras que apresenta menor tamanho comparado ao espaço de padrões e ao espaço de estados, tem sido abordada por diversos estudos [4, 5, 31–34, 38].

Em seguida são apresentados alguns esquemas para classificação do comportamento dinâmico das regras.

Classificação Dinâmica

Há na literatura diferentes esquemas de classificação para o comportamento dinâmico de um AC. A classificação dinâmica mais conhecida foi proposta por Wolfram em [62] e divide os ACs em 4 classes.

- Classe 1: o AC evolui grande parte das configurações iniciais, após um número finito de passos de tempo, para a mesma configuração homogênea fixa (por exemplo, todas as células no estado 0).
- Classe 2: o AC converge a maioria das configurações iniciais, após um período transiente, para algum ponto-fixo heterogêneo ou algum ciclo periódico de configurações.
- Classe 3: quase todas as possíveis configurações iniciais resultam, após um período transiente, em padrões não periódicos, também denominados caóticos.
- Classe 4: algumas configurações iniciais resultam em estruturas localizadas complexas e por vezes bastante duradouras.

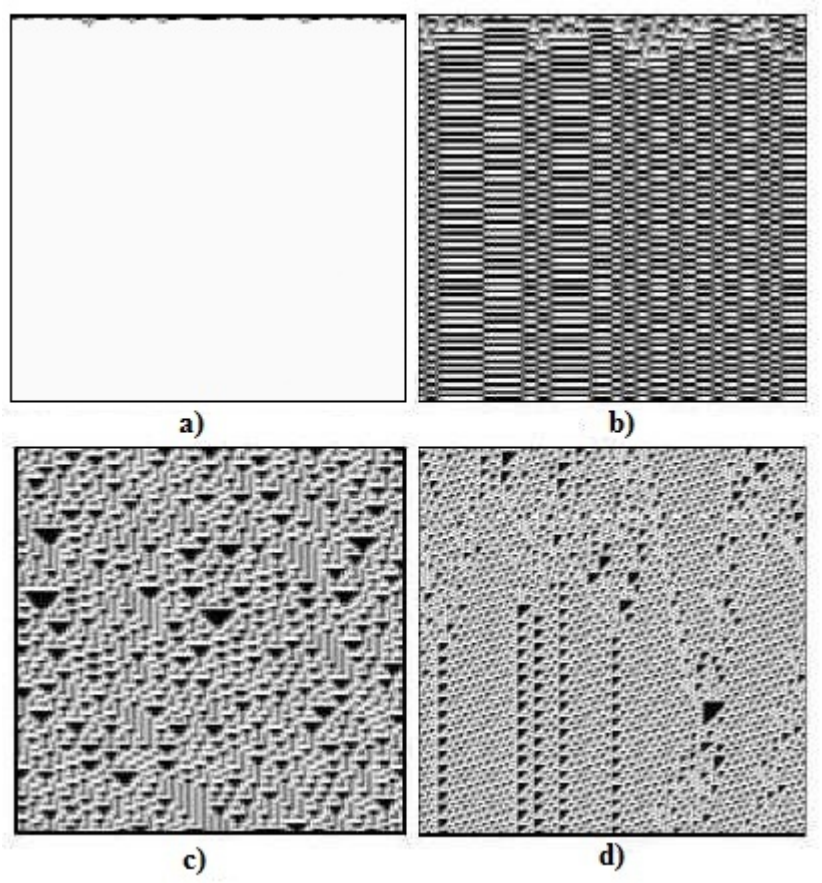


Figura 3.9: Exemplos de comportamento dinâmico dos ACs [37] de acordo com a classificação de Wolfram: (a) regra 0100100 (Classe 1); (b) regra 0100101 (Classe 2); (c) regra 0011110 (Classe 3); (d) regra 1101110 (Classe 4).

Na Figura 3.9 são apresentados exemplos de evolução de ACs unidimensionais binários de $R = 1$ que se enquadram nas quatro classes de comportamento dinâmico descritas anteriormente. A Figura 3.9(a) representa um comportamento referente à Classe 1, a Figura 3.9(b) representa um comportamento relacionado à Classe 2, a Figura 3.9(c) representa um comportamento dinâmico pertencente a Classe 3 e a Figura 3.9 (d) representa um comportamento dinâmico da Classe 4. Na figura, os *bits* de saída das regras são apresentados da vizinhança 000 à vizinhança 111.

Trabalhos posteriores tais como [31–34] propuseram uma série de refinamentos na classificação original [62]. Li e Packard desenvolveram um esquema de classificação que divide o espaço de regras em 6 classes [31]:

- Regras Nulas, em que a configuração limite é constituída apenas por um único estado;
- Regras Ponto Fixo, onde, excluindo-se as configurações formadas por um único estado (nulas), a configuração limite é invariante ao reaplicarmos a regra do AC (com um possível deslocamento espacial);

- Regras Ciclo Duplo, nas quais a configuração limite é invariante ao se reaplicar a regra do AC duas vezes, com um possível deslocamento;
- Regras Periódicas, onde a configuração limite é invariante à aplicação da Regra L (> 2) vezes, com o tamanho do ciclo L ou independente ou fracamente dependente do tamanho do sistema;
- Regras Caóticas, em que regras, caracterizadas pela grande divergência do comprimento do seu ciclo com o tamanho do sistema e pela instabilidade com respeito a perturbações, produzem dinâmicas não periódicas;
- Regras Complexas (ou à Beira do Caos), onde embora a dinâmica limite possa ser periódica, o intervalo de transição pode ser extremamente longo e tipicamente crescer bastante com o tamanho do sistema. Durante o transiente, estruturas complexas "surgem" no diagrama espaço-temporal.

Capítulo 4

Escalonamento Baseado em Autômatos Celulares

Apesar de algumas metaheurísticas tais como os AGs conseguirem, em muitos casos, apresentar boas soluções para o problema do escalonamento, estas apresentam um alto consumo de tempo e memória representado pelo custo de execução do escalonamento. Assim, no caso das metaheurísticas, a principal causa relacionada a este alto consumo é a necessidade do cálculo de uma função de custo durante várias iterações do algoritmo. Contudo, é interessante notar que este esforço computacional não pode ser "reaproveitado" em uma próxima execução do algoritmo, uma vez que é necessário que ele parta do zero para escalonar uma nova instância. De fato, a maioria dos algoritmos aplicados ao PEET não extraem, conservam ou reutilizam qualquer conhecimento sobre o escalonamento enquanto resolvem instâncias do problema [56]. Se este conhecimento fosse "armazenado" e utilizado posteriormente na resolução da própria instância escalonada ou de outras instâncias diferentes, poder-se-ia obter um ganho durante a resolução destas instâncias.

Outra característica destacável observada na literatura é que a maioria dos algoritmos de escalonamento são sequênciais, como as heurísticas de construção descritas na Seção 2.4. Uma nova área de pesquisa estuda algoritmos de escalonamento paralelos e distribuídos [2], baseados na aplicação de técnicas computacionais não convencionais [46]. Um exemplo pode ser visto através de resultados apontados em pesquisas recentes, os quais mostraram que, combinados, os ACs e os AGs podem ser efetivamente usados para projetar algoritmos de escalonamento paralelos e distribuídos [9, 10, 49, 53, 57].

Neste capítulo são apresentados os modelos publicados de algoritmos de escalonamento baseado em autômatos celulares. A idéia central desses modelos tem como objetivo a extração do conhecimento sobre o processo de escalonamento de um grafo de programa e sua consequente reutilização enquanto resolvendo novas instâncias do problema de escalonamento. Assim, o intuito das próximas seções é apresentar os principais trabalhos de modo a melhor definir a contribuição desta dissertação em relação ao estado da arte da pesquisa.

4.1 Conceitos e Definições

Na terminologia do problema de escalonamento, uma distinção é frequentemente realizada entre escalonamento, sequência e política de escalonamento [50]. O escalonamento, na maioria das vezes, se refere à uma possível alocação das tarefas dentro do sistema multiprocessado. A sequência normalmente corresponde à ordem em que as tarefas serão executadas em um dado processador. De modo geral, uma política de escalonamento define a sequência das tarefas em cada processador do sistema. Assim, enquanto o escalonamento distribui as tarefas entre os processadores, a política de escalonamento ordena estas tarefas dentro de cada processador, definindo a sequência das tarefas.

A Figura 4.1 mostra um esboço conceitual do sistema escalonador apresentado neste trabalho. As aplicações paralelas a serem escalonadas são representadas em grafos de programa (GAD) e as arquiteturas multiprocessadas em grafos de sistema, conforme apresentados na Seção 2.2.

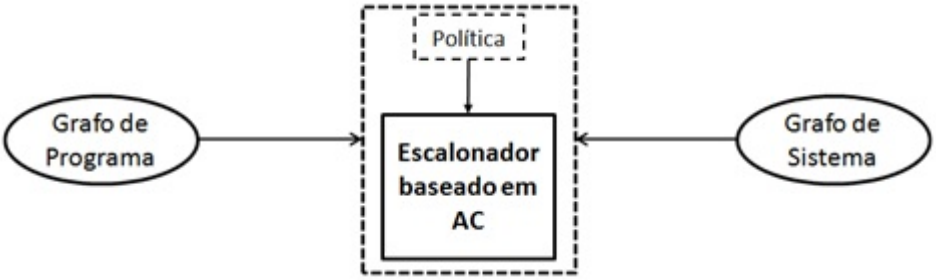


Figura 4.1: Esboço conceitual do escalonador baseado em AC.

Um exemplo de política de escalonamento frequentemente encontrada nos modelos de escalonamento baseado em AC é "a tarefa com maior nível dinâmico deve ser executada primeiro". Considerando o nível de uma tarefa como o maior custo entre ela e uma tarefa de saída do grafo, conforme mostrado na Seção 2.4, o nível de uma tarefa i pode ser recursivamente calculado por:

$$bl_i = \left\{ \begin{array}{l} w_i, \text{ se } i \text{ é uma tarefa saída;} \\ \max_{j \in successores(i)}(bl_j + c_{i,j}) + w_i, \text{ caso contrário.} \end{array} \right\}$$

Assim, o nível da tarefa é dinâmico quando é calculado considerando a alocação das tarefas nos processadores e o custo de comunicação é adicionado quando as tarefas estão alocadas em processadores distintos.

O elemento principal da arquitetura apresentada na Figura 4.1 é o escalonador baseado em AC. No modelo de escalonador baseado em AC, assume-se que cada célula do reticulado está associada com uma tarefa do grafo de programa. Assim, se um conjunto de tarefas do grafo de programa tem cardinalidade x, o reticulado do AC deve possuir x células. Além disso, considerando-se uma arquitetura composta por P processadores, o AC a ser

utilizado terá P possíveis estados por célula. Dessa forma, supondo um sistema com dois processadores (P0 e P1), tem-se que cada célula do AC pode assumir, em um instante de tempo t, o valor 0, indicando que a tarefa correspondente está alocada no processador P0, ou o valor 1, indicando que a tarefa está alocada no processador P1. Por exemplo, um grafo de programa que é composto por 4 tarefas, deve ser representado por um reticulado de 4 células e em uma configuração onde as tarefas 0 e 3 estão alocadas em P0 e as tarefas 1 e 2 em P1, o reticulado será 0110. A Figura 4.2 ilustra a idéia geral do escalonador baseado em AC. No estado inicial representado na figura, o reticulado é formado pela alocação aleatória das tarefas nos processadores do sistema. Posteriormente, a função de transição é aplicada sobre esse reticulado por um número determinado de passos de tempo até que se obtenha o estado final do reticulado que equivale à alocação final das tarefas nos processadores. Dessa forma, a regra de transição do AC é responsável por levar o reticulado do AC de uma configuração inicial correspondente a uma alocação inicial de tarefas à uma configuração final que corresponda a uma alocação adequada das tarefas, que após a aplicação da política de escalonamento, retorne um valor baixo de *makespan*, preferencialmente ótimo.

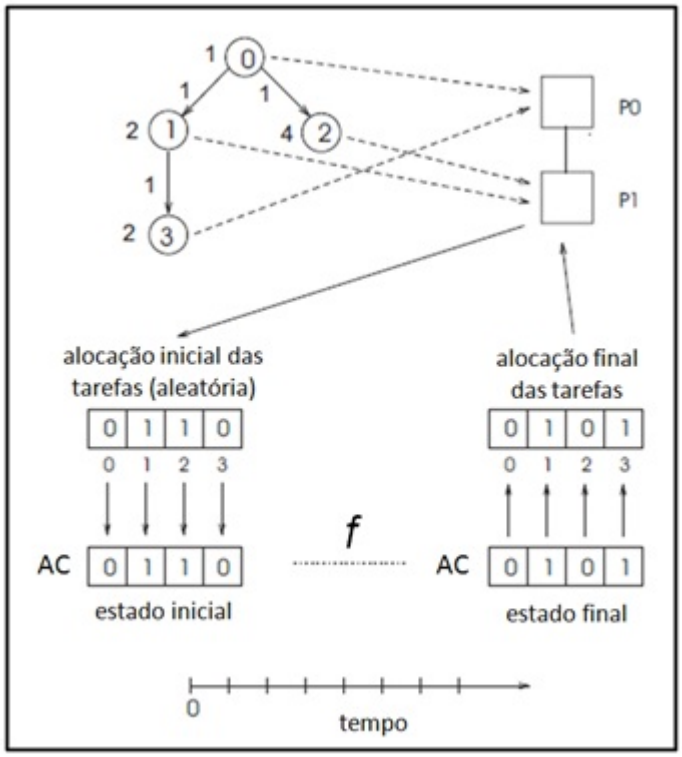


Figura 4.2: Visão geral do escalonamento baseado em AC.

4.2 Estado da Arte

Existem poucos trabalhos na literatura com o próposito de explorar o escalonamento baseado em ACs. Contudo, algumas abordagens merecem destaque e serão enfatizadas nesta seção.

No modelo proposto em [46] e utilizado também em [48, 49, 51, 52, 56], é apresentado um escalonador baseado em AC que opera em dois modos: aprendizagem e operação. No modo de aprendizagem, o escalonador faz uso de um AG para descobrir regras de ACs capazes de encontrar soluções ótimas (ou sub-ótimas) partindo-se de diferentes alocações aleatórias de um grafo de programa, ou seja, diferentes reticulados iniciais. Cabe ressaltar que o AG discutido aqui, nada tem a ver com o AG elaborado para o PEET, descrito no Capítulo 3. Aqui, o propósito do AG não é encontrar diretamente o escalonamento de um grafo de programa, mas sim encontrar uma regra de transição de AC capaz de fazê-lo. A população inicial desse AG é composta de possíveis regras de transição geradas aleatoriamente. A cada geração do AG, um conjunto de reticulados ou configurações iniciais (CI) é definido aleatoriamente. Assim, o método ou função de avaliação da população é a evolução temporal a partir de cada reticulado inicial dessa CI pelos indivíduos (regras de transição) por t passos de tempo. No tempo t, cada reticulado obtido é escalonado mediante uma política de escalonamento escolhida a priori. Considerando-se todos os reticulados iniciais do CI obtêm-se a média de custo de escalonamento para aquela regra. No modelo de AC, foram avaliados diferentes modos de atualização das células: sequencial, paralelo e sequencial-aleatório [46]. Vale destacar também que o modelo de AG nesses trabalhos utiliza uma estratégia elitista, onde as T_{elite} melhores regras são mantidas para a próxima geração, e que apenas a elite é considerada na seleção de pais para o *crossover*. A evolução temporal das células do reticulado, a partir do instante inicial, pode ser feita no modo síncrono ou sequencial, sendo este um parâmetro de entrada do escalonador. Os dois modos de atualização são avaliados em [46]. Um esboço deste modelo pode ser visto na Figura 4.3.

No modo de operação desse modelo, espera-se que, para uma dada alocação inicial qualquer de tarefas, as regras de AC sejam capazes de evoluir a configuração das células para um estado que represente uma alocação que minimize o *makespan*. É esperado também que as regras obtidas na fase de aprendizagem possam ser utilizadas com bom desempenho no escalonamento de outros grafos de programa.

No estudo apresentado em [53], tem-se um escalonador baseado em AC que opera em três modos: aprendizagem, operação e reutilização. Os dois primeiros modos são semelhantes aos propostos por [46] e utilizam atualização sequencial e síncrona. No modo de reutilização, as regras descobertas anteriormente são reusadas, com o apoio de um Sistema Imunológico Artificial (SIA), para resolver novas instâncias do problema. No modelo de SIA, as regras descobertas são consideradas anticorpos enquanto que novas

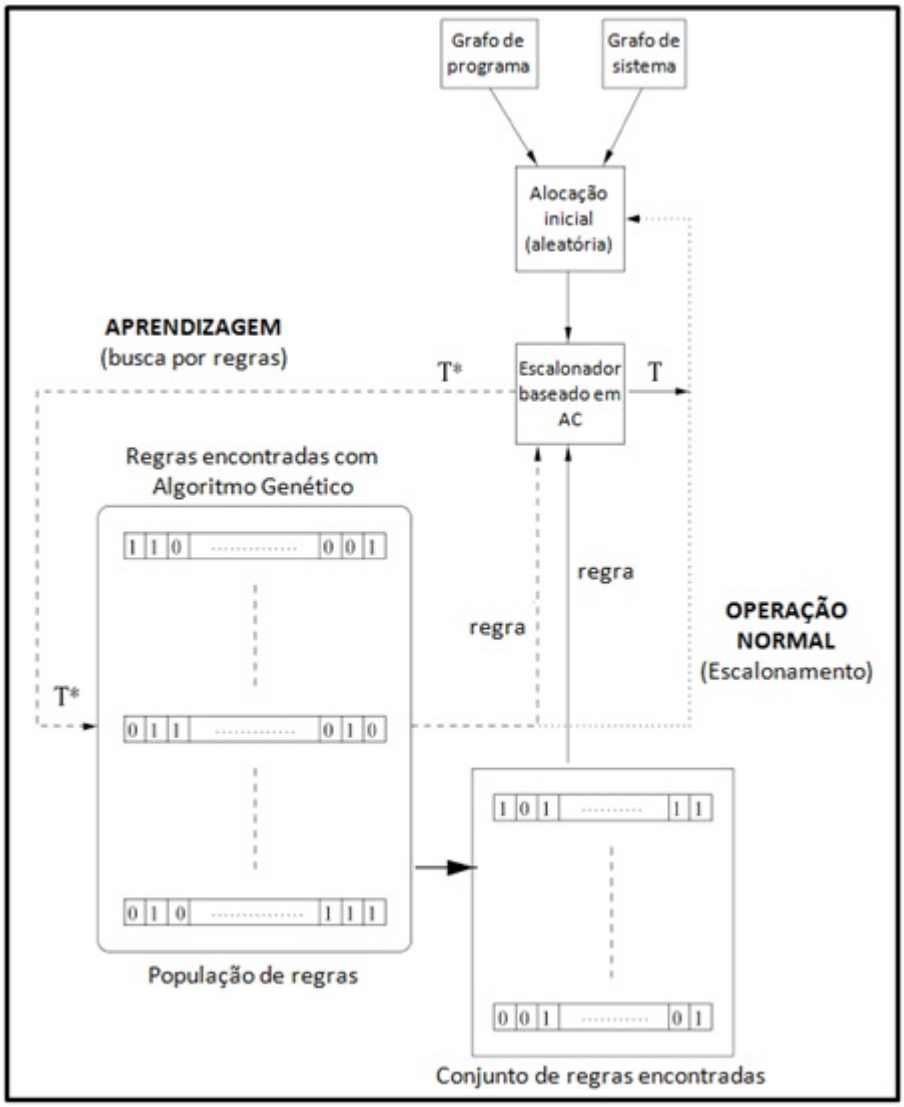


Figura 4.3: Representação do modelo de escalonador proposto em [49].

instâncias do problema de escalonamento são tratadas como antígenos. Dessa forma, um dado anticorpo reconhece um antígeno específico se ele pode encontrar um escalonamento ótimo para ele.

O modelo discutido em [57] é baseado nos conceitos de escalonador propostos em [46, 47, 49, 51, 52] e tem como principal contribuição o uso de uma estratégia denominada evolução conjunta para o AG utilizado no modo de aprendizagem. Esta estratégia consiste em utilizar, paralelamente, mais de um grafo de programa durante a evolução das regras. Apenas o modelo sequencial foi avaliado neste trabalho. O trabalho [56] ainda apresenta uma abordagem de algoritmos genéticos coevolutivos, onde os modos de atualização sequencial e síncrono foram analisados.

Uma característica para discernir os trabalhos de [52] e [53] dos demais trabalhos é o modo de vizinhança empregado. Em [46, 48, 49, 56, 57] apesar de utilizarem modelos de vizinhança linear, o foco é no emprego de um modelo de vizinhança não linear no qual os

vizinhos de uma célula não são definidos simplesmente pelo raio. Essas vizinhanças são mais complexas e definem um espaço de regras de alta cardinalidade, tornando o processo de aprendizagem mais lento.

4.3 Modelos de Vizinhança

Para que o escalonador baseado em AC obtenha bons resultados é importante definir um modelo de vizinhança capaz de capturar as relações entre as tarefas expressas no grafo de programa. Os modelos de vizinhança investigados na literatura podem ser divididos em lineares e não lineares. No primeiro, assume-se que a estrutura não linear de um grafo de programa é aproximada para a estrutura linear de um AC tradicional como apresentado na Figura 3.8. Já no modelo de vizinhança não linear, a estrutura não linear de um grafo de programa é modelada através de uma estrutura também não linear de vizinhança de AC elaborada especificamente para o problema de escalonamento.

As Seções 4.3.1 e 4.3.2 apontam exemplos e características dos modelos de vizinhança lineares e não lineares, respectivamente, utilizadas em trabalhos anteriores.

4.3.1 Vizinhança linear

A vizinhança linear no modelo de escalonador baseado em AC [52, 53], consiste em aproximar a estrutura não linear de um grafo de programa composto por N tarefas em uma estrutura linear de um AC unidimensional com N células e utilização de uma vizinhança padrão baseada em raio [44]. Na Figura 4.4 tem-se a representação da vizinhança linear para o grafo de programa *gauss18* que já foi exibido na Figura 2.5, com destaque para a vizinhança 8, em diferentes valores de raio. Para efeito de exemplo, considerou-se para as Figuras 4.4(a), 4.4(b) e 4.4(c), tamanho do raio $R = 1, 2$ e 3, respectivamente. É interessante notar que nesse modelo, cada uma das tarefas do grafo de programa está relacionada a uma célula do reticulado, assim a tarefa 0 está representada na célula 0, a tarefa 1 na célula 1, e assim por diante.

É interessante perceber que esse tipo de vizinhança, utilizada em alguns trabalhos como em [53], não é apta a capturar as relações entre as tarefas, pois utiliza-se das posições no reticulado para definir as células vizinhas de uma determinada tarefa, sendo que a posição de uma tarefa no reticulado é definida simplesmente pelo número de ordem da mesma. Por exemplo, independente da relação entre duas tarefas identificadas como 7 e 8 em um grafo, elas sempre serão vizinhas no reticulado. Contudo, é importante destacar a simplicidade desse modelo de vizinhança.

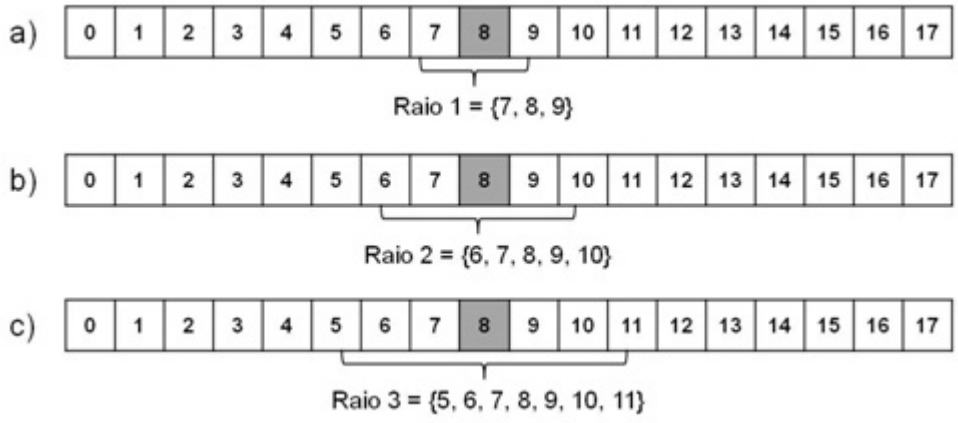


Figura 4.4: Modelo de vizinhança linear para o *gauss18*: (a) Raio = 1; (b) Raio = 2; Raio = 3.

4.3.2 Vizinhança não linear

Uma outra forma de definir uma vizinhança sobre a estrutura não linear de um grafo de programa é utilizar um modelo também não linear [46]. Dessa forma, a principal diferença entre os modelos de vizinhança linear e não linear se refere à maneira como a vizinhança de uma célula é especificada, embora o posicionamento das tarefas no reticulado seja feito da mesma forma anterior, pelo número de ordem da tarefa. De fato, nos modelos não lineares a proximidade entre duas tarefas no reticulado não as tornam vizinhas, uma vez que a vizinhança é determinada pelas relações entre as tarefas no próprio grafo de programa. Assim, esse tipo de vizinhança é baseada no conjunto de tarefas predecessoras, irmãs e sucessoras da tarefa analisada. A Figura 4.5 mostra uma representação da vizinhança não linear para o grafo de programa *gauss18*. Na figura, a vizinhança da tarefa 8 é constituída por ela mesma e também pelo seu conjunto de predecessoras (3 e 6), irmãs (7, 9, 10 e 11) e sucessoras (11 e 12), além da própria tarefa 8. Nas sub-seções a seguir são apresentados dois tipos de vizinhança não lineares apresentados originalmente em [46]: selecionada e totalística.

Apesar das vizinhanças não lineares considerarem as relações entre as tarefas, uma de suas características é o custo computacional bastante elevado quando comparado à vizinhança linear. Além disso, os modelos existentes na literatura têm apresentado limitações quanto ao número de processadores e ainda possuem estruturas complexas.

Vizinhança Selecionada

Originalmente proposta em [46], a vizinhança selecionada de uma tarefa i consiste da própria tarefa e de três subvizinhanças, onde cada subvizinhança é constituída por duas tarefas representativas de um conjunto de tarefas predecessoras, irmãs (tarefas que possuem ao menos uma tarefa predecessora em comum) e sucessoras da tarefa i. Estas tarefas representativas são, respectivamente, a tarefa com o valor máximo e a tarefa com

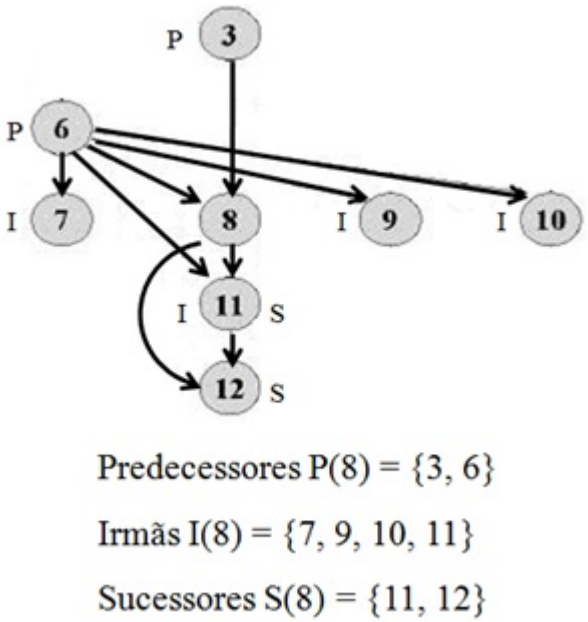


Figura 4.5: Vizinhança não linear para o *gauss18*.

o valor mínimo, de acordo com algum atributo das tarefas escolhido a priori. Dentre os atributos apresentados na Seção 2.4, que podem ser utilizados, pode-se citar custo computacional, custo de comunicação, b-*level*, t-*level*, entre outros.

Uma vez que a estrutura de um grafo de programa é irregular, o número de predecessoras, irmãs ou sucessoras de uma dada tarefa pode ser menor que dois. Além disso, pode ser que os atributos entre duas ou mais tarefas sejam iguais. Para esses casos especiais, o seguinte tratamento foi proposto em [46]:

- Se predecessoras (irmãs ou sucessoras) não existem para uma dada tarefa, uma subvizinhança correspondente para cada situação é criada adicionando-se um par de tarefas "postiças" e associando-se estas tarefas com um par de células; os estados destas células (representando processadores onde as tarefas estão alocadas) são considerados indefinidos e essa subvizinhança recebe um tratamento diferenciado;
- Se apenas uma predecessora (irmã ou sucessora) existe para uma dada tarefa, adiciona-se apenas uma tarefa "postiça" para a subvizinhança correspondente e o estado da nova célula receberá o mesmo estado da tarefa existente. Ou seja, a tarefa postiça e a tarefa real existente para uma dada subvizinhança estarão alocadas no mesmo processador;
- Se o número de predecessoras (irmãs ou sucessoras) for maior que dois e todas elas tiverem o mesmo valor para um dado atributo, serão selecionadas as duas tarefas com o menor e maior número de ordem.

Após identificar as tarefas pertencentes à vizinhança (conforme apresentado na Figura

4.5), é necessário definir os estados das subvizinhanças q_i^P (predecessoras de i), q_i^I (irmãs de i), q_i^S (sucessoras de i) e o estado $q_i^{'}$, que representa o novo estado da tarefa i. O estado da célula i (q_i) pode assumir os valores 0 ou 1, enquanto que os valores de cada par de células (máximo, mínimo), correspondentes às subvizinhanças, são mapeados em um de cinco valores descrevendo o estado do par da seguinte maneira:

- **estado 0**: os valores das células do par são iguais a 0 (ambas as tarefas estão alocadas em P_0);
- **estado 1**: a primeira célula tem o valor 0 (está em P_0) e a segunda célula tem o valor 1 (está em P_1);
- **estado 2**: a primeira célula tem o valor 1 (está em P_1) e a segunda célula tem o valor 0 (está em P_0);
- **estado 3**: os valores das duas células são iguais a 1 (ambas as tarefas estão alocadas em P_1);
- **estado 4**: os valores das duas células são indefinidos (par de tarefas "postiças").

Um exemplo de formação da vizinhança selecionada é ilustrado na Figura 4.6, onde foram considerados os atributos t-*level* (t) para o conjunto de predecessoras, custo computacional (w) para o conjunto de irmãs e custo de comunicação (c) para o conjunto de sucessoras. Os valores dos atributos selecionados para cada conjunto estão identificados na própria figura, para cada tarefa pertencente ao respectivo conjunto. Além disso, a figura também apresenta a composição da vizinhança em relação à célula/tarefa 8, onde suas subvizinhanças são $q_8 = 1$, $q_8^P = 1$ ($menor^t - 20$ em P_1 e $maior^t - 32$ em P_0), $q_8^I = 0$ ($menor^w - 3$ em P_0 e $maior^t - 4$ em P_0) e $q_8^S = 0$ ($menor^c - 8$ em P_0 e $maior^c - 8$ em P_0).

Ao analisar o número total de vizinhanças possíveis na vizinhança selecionada, tem-se que o estado q_i pode assumir dois valores distintos (0 ou 1) e os demais estados, q_i^P, q_i^I e q_i^S podem assumir cinco valores diferentes (0, 1, 2, 3 ou 4). Dessa forma, o comprimento de uma regra de transição é de 250 bits (dado por 2 x 5 x 5 x 5).

Vizinhança Totalística

A vizinhança totalística, também originalmente proposta em [46], difere da vizinhança selecionada por considerar todas as tarefas pertencentes aos conjuntos de predecessoras, irmãs e sucessoras de uma dada tarefa central ao invés de apenas as mais representativas (maior e menor) de cada conjunto. A idéia desse modelo é reduzir o tamanho da regra de transição do AC por reduzir o tamanho da vizinhança. Assim, ao invés das 7 células utilizadas na vizinhança selecionada, a totalística utiliza quatro, pois a vizinhança de uma tarefa i é representada pelo estado da célula correspondente à tarefa q_i, além de outras

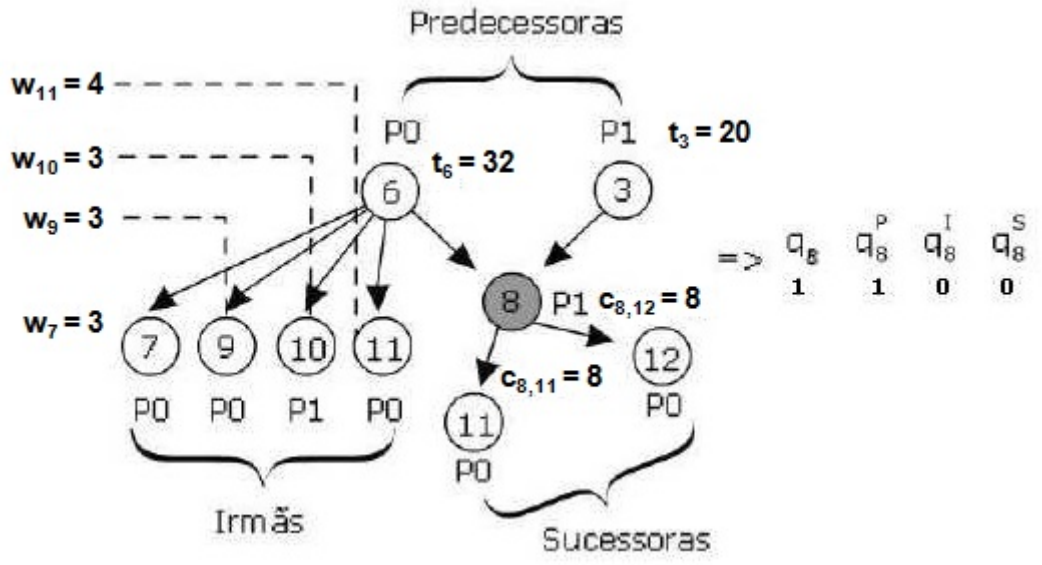


Figura 4.6: Vizinhança selecionada para o *gauss18* (adaptada de [56]).

três células que representam, respectivamente, o conjunto de predecessoras de i (q_i^P), o conjunto de irmãs de i (q_i^I) e o conjunto de sucessoras de i (q_i^S).

Vale destacar também que os mesmos atributos do grafo de programa que podem ser utilizados na formação da vizinhança selecionada também podem ser utilizados na vizinhança totalística. Um exemplo de obtenção da vizinhança totalística pode ser realizado considerando o conjunto de tarefas predecessoras de uma tarefa i alocadas em P_0 ($P_i^{P_0}$) e em P1 ($P_i^{P_1}$), onde o atributo selecionado foi o t-*level* (t). Logo, o estado da célula q_i^P é definido conforme os estados abaixo, sendo $x \in P_i^{P_0}$ e $y \in P_i^{P_1}$:

- $q_i^P = 0$, se $\sum_x t_x > \sum_y t_y$
- $q_i^P = 1$, se $\sum_x t_x < \sum_y t_y$
- $q_i^P = 2$, se $\sum_x t_x = \sum_y t_y$
- $q_i^P = 3$, se não há predecessores.

Para os conjuntos de tarefas irmãs e sucessoras, a obtenção dos estados q_i^I e q_i^S é feita de forma similar, utilizando-se os atributos selecionados para esses conjuntos.

A Figura 4.7 mostra um exemplo de formação da vizinhança totalística. Na figura, foram considerados os atributos t-*level*, custo computacional e custo de comunicação para os conjuntos de predecessoras, irmãs e sucessoras, respectivamente. Novamente, destaca-se que a vizinhança não linear da tarefa 8 foi exibida na Figura 4.5 e os valores dos atributos selecionados para cada conjunto estão identificados na própria Figura 4.7, para cada tarefa pertencente ao respectivo conjunto. Considerando-se a figura, temos:

- $q_8 = 1$, pois a tarefa 8 está em P_1;

- $q_8^P = 0$, pois $\sum_x t_x = 32$ e $\sum_y t_y = 20$;
- $q_8^I = 0$, pois $\sum_x t_x = 3 + 3 + 4 = 10$ e $\sum_y t_y = 3$;
- $q_8^S = 0$, pois $\sum_x t_x = 8 + 8 = 16$ e $\sum_y t_y = 0$.

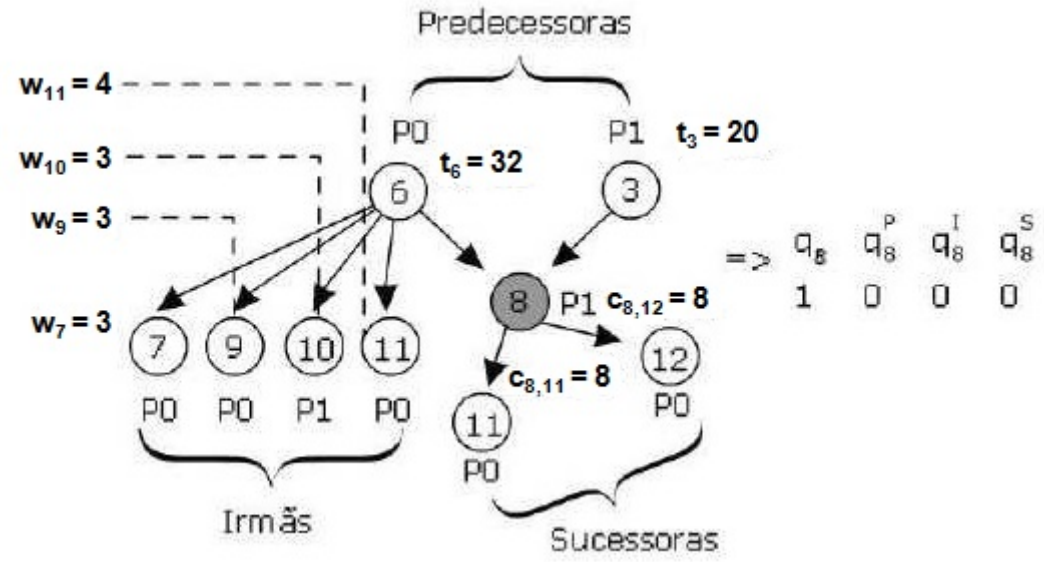


Figura 4.7: Vizinhança totalística para o *gauss18* (adaptada de [56]).

Ao analisar o número total de vizinhanças possíveis na vizinhança totalística, tem-se que o estado q_i pode assumir dois valores distintos (0 ou 1) e os demais estados, q_i^P, q_i^I e q_i^S podem assumir quatro valores diferentes (0, 1, 2 ou 3). Dessa forma, o comprimento de uma regra de transição é de 128 *bits* (dado por 2x4x4x4). Portanto, o modelo totalístico utiliza uma regra de menor comprimento que o modelo de vizinhança selecionada (250 *bits*).

4.4 Tipos de Aprendizagem

A maior parte dos trabalhos encontrados na literatura utilizam apenas um grafo de programa e uma população de regras durante a fase de aprendizagem do escalonador. Tal característica é denominada aprendizagem simples ou, segundo [57], evolução simples. Contudo, mais recentemente outros tipos possíveis de aprendizagem no escalonador foram investigados, tais como a aprendizagem coevolutiva [49, 56] e a evolução conjunta [57].

De fato, a estrutura de busca do AG e a forma como se dá a evolução do AC definem o tipo de aprendizagem. É certo que a aprendizagem simples parte de um algoritmo genético padrão, enquanto que a aprendizagem coevolutiva baseia-se no uso de duas populações simultâneas durante o processo evolutivo e a aprendizagem conjunta consiste em utilizar dois ou mais grafos de programa durante a evolução das regras. Neste trabalho, a

aprendizagem coevolutiva não é investigada, por apresentar uma complexidade computacional elevada sem uma melhora significativa comparada às demais. Já a aprendizagem conjunta, apesar de não ser a principal forma de aprendizado abordada nesta dissertação, tem como principal objetivo a melhoria da capacidade de generalização e apresenta custo computacional menor que a coevolutivo, embora tenhamos iniciado algumas investigações com este tipo de aprendizagem, não foi possível concluí-las até a elaboração desta dissertação e assim não serão apresentadas neste texto. Por outro lado, a aprendizagem simples oferece algumas vantagens relacionadas especialmente ao baixo custo computacional e à sua simplicidade, razões pelas quais foi empregada neste trabalho.

Capítulo 5

Novas Abordagens para o Escalonamento baseado em Autômatos Celulares

5.1 Metodologia para Avaliação dos Novos Modelos

Como meio de realizar uma avaliação adequada do desempenho dos modelos propostos neste trabalho, foram reproduzidos os principais trabalhos relacionados ao escalonamento baseado em AC e também implementados métodos heurísticos, meta-heurísticos e exatos. O intuito geral é comparar os resultados obtidos por estes métodos, especialmente na reprodução dos modelos anteriores do escalonador, com as novas abordagens.

É importante considerar os algoritmos clássicos para o PEET nesta avaliação porque eles são capazes de gerar bons escalonamentos a partir de informações do programa paralelo (atributos). As meta-heurísticas por outro lado, apesar de na maioria das vezes não considerarem qualquer informação do grafo de programa, possuem uma estrutura de busca robusta, normalmente baseada em algum modelo científico ou natural, o que lhes garantem, em muitos casos, bons resultados frente a problemas complexos como o PEET. Os resultados obtidos por algoritmos exatos são extremamente úteis, pois permitem avaliar precisamente a qualidade de uma solução, sendo assim esses algoritmos também foram incorporados à metodologia de avaliação deste trabalho.

Contudo, mais importante que utilizar todos estes métodos computacionais, é avaliar o desempenho dos novos modelos frente aos modelos já existentes na literatura, uma vez que o conceito de escalonador baseado em AC não está relacionado apenas a encontrar um escalonamento para uma dada instância tal como acontece nestes métodos, mas extrair conhecimento do processo de escalonamento a fim de reusá-lo em outras instâncias. De fato, o uso dos modelos anteriores para a análise das abordagens propostas permitiram apontar e dirigir grande parte do esforço na busca por um escalonamento bem sucedido,

tanto na aprendizagem quanto na fase de execução. Afinal, mais que um *benchmark*, esses modelos representam um ambiente de teste, onde novas ferramentas e novos modelos podem ser incorporados e testados com o intuito de obter um melhor desempenho.

5.1.1 Algoritmos Clássicos

Além de permitir incorporar alguns conceitos e técnicas aos novos modelos de escalonador com AC, o estudo aprofundado do PEET também permitiu empregar heurísticas de construção bastante utilizadas para o problema como parte da metodologia para avaliar as abordagens propostas. Assim, foram implementados três algoritmos clássicos: DHLFET, MCP e ETF, todos apresentados na Seção 2.4.

5.1.2 Meta-heurísticas

Devido à complexidade do PEET, é muito frequente a aplicação de meta-heurísticas sobre o problema. O intuito é quase sempre o mesmo, encontrar boas soluções para uma dada aplicação paralela, uma vez que algoritmos exatos não conseguem resolver muitas instâncias do problema em tempo considerável e heurísticas nem sempre apresentam o desempenho esperado.

As meta-heurísticas implementadas neste trabalho foram: algoritmo genético (AG) e *simulated annealing* (SA). De fato, essas técnicas são bastante investigadas no problema de escalonamento e se destacam pelas boas soluções que costumam encontrar.

Três ambientes baseados nessas meta-heurísticas foram implementados: AG-1, AG-2 e SA.

Algoritmo Genético AG-1

O primeiro algoritmo genético desenvolvido para o PEET é responsável por alocar as tarefas entre os processadores e definir a ordem de execução delas. Esse algoritmo foi denominado AG-1 e teve seus principais passos descritos no Capítulo 3.

Após alguns estudos empíricos, os parâmetros utilizados em AG-1, salvo alguns casos especificados, foram: tamanho da população $T_{pop} = 200$, torneio simples $Tour = 2$, taxa de cruzamento $P_{cross} = 100\%$, taxa de mutação $P_{mut} = 3\%$ e número de gerações $G = 200$. O tipo de cruzamento escolhido foi o cíclico (Seção 3.1.4), a mutação foi do tipo permutação (Seção 3.1.5) e a re-inserção foi realizada baseada na aptidão (Seção 3.1.6). As principais etapas de AG-1 são descritas com maiores detalhes entre as Seções 3.1.1 e 3.1.7. Em todos os testes foram realizadas 20 execuções.

Algoritmo Genético AG-2

O segundo AG implementado para o PEET tem por objetivo apenas a distribuição das tarefas entre os processadores, de modo que a ordem de execução das tarefas é definida por uma política de escalonamento. Em todos os testes com esse AG, a política de escalonamento foi a mesma utilizada nos modelos de escalonador baseado em AC propostos neste trabalho: "a tarefa com maior nível dinâmico primeiro". Esse algoritmo genético foi denominado AG-2.

Os parâmetros utilizados em AG-2 foram idênticos aos utilizados em AG-1. Contudo a representação dos indivíduos e alguns tipos de operadores genéticos foram alterados a fim de lidar melhor com as características peculiares do AG-2. A nova representação, que também foi utilizada no SA, consiste em uma matriz unidimensional onde os índices estão relacionados às tarefas e o elemento representado em cada posição denomina o processador em que aquela tarefa está alocada. Dessa forma, o tipo de cruzamento escolhido foi o ponto-simples (Seção 3.1.4) e a mutação foi do tipo troca ou complemento do *bit* (Seção 3.1.5). Os demais métodos são os mesmos de AG-1.

Simulated Annealing SA

Da mesma forma como AG-2, também foi implementado o SA para o PEET com o objetivo apenas de alocar as tarefas entre os processadores, sendo a ordem de execução das tarefas definida através da política de escalonamento utilizada nas novas abordagens para o escalonamento baseado em AC: "a tarefa com maior nível dinâmico primeiro".

Após alguns estudos empíricos, o mapeamento da temperatura *temp* em função do tempo t_{SA} foi definido pela fórmula $temp(t_{SA}) = 100 * 0,9995 t_{SA}$. Além disso, considerou-se $\epsilon = 1.10^{-9}$ e o número de movimentos laterais $mov_lat = N/6$. Em todos os testes para SA também foram realizadas 20 execuções. Maiores detalhes sobre o funcionamento do *Simulated Annealing* podem ser vistos em [42].

5.1.3 Reprodução de Trabalhos Correlatos

Como primeiro passo para desenvolver nossas abordagens, foi realizada a reprodução dos modelos originais de escalonamento baseado em AC existentes na literatura. O intuito, mais do que definir um esquema comparativo posteriormente, foi entender os principais conceitos, vantagens e desvantagens desses modelos.

Para uma análise coerente, foram considerados todos os grafos de programa utilizados em trabalhos anteriores [46–49, 51–53, 56, 57] com possibilidade de reprodução. Além disso, por avaliar as abordagens propostas de uma forma sistematicamente robusta, testes inéditos com a reprodução desses trabalhos foram realizados.

Nas Seções 5.2 e 5.3 apresentam os resultados e conclusões sobre o desempenho dessas reproduções em relação às abordagens propostas.

5.1.4 Análise Estatística

Uma vez que os modelos baseados em autômatos celulares fazem uso de um algoritmo genético para a busca de regras e também utilizam configurações iniciais geradas aleatoriamente, é importante considerar o caráter probabilístico destas ferramentas. Desse modo, ao contrário, dos trabalhos apresentados na literatura [46–49, 51–53, 56, 57], não considerou-se nas análises realizadas apenas a melhor execução obtida, uma vez que nem sempre essa metodologia reflete o desempenho geral do modelo e certamente dificulta a sua reprodução. Pelo contrário, além do desempenho da melhor execução obtida, o resultado obtido em todas as execuções também foi considerado.

Diante disso, a população em cada teste é composta por 20 amostras, cada uma delas relacionada a uma execução do escalonador. Os testes estatísticos apresentados nesta dissertação visam comparar dois modelos (M_1 e M_2). Porém, antes disso utilizamos alguns testes, como Shapiro Wilk, para descobrir se as populações destes modelos seguem uma distribuição normal ou anormal. Se as distribuições são normais, utiliza-se o teste t de Student, caso contrário, realiza-se o teste de Mann-Whitney, também conhecido como o teste de Wilcoxon para amostras independentes. O nível de significância adotado nos testes de hipótese foi de 5%, o que equivale a um nível de confiança de 95%. O teste foi formulado da seguinte maneira:

- $H_0 : M_1 = M_2$;
- $H_1 : M_1 \neq M_2$, onde $M_1 < M_2$ ou $M_1 > M_2$.

É importante destacar que a melhor solução para o PEET é aquela que minimiza o tempo de escalonamento, sendo assim, o modelo com melhor desempenho é aquele que apresenta uma população menor que o outro. Por exemplo, se $M_1 < M_2$, sabe-se que o modelo M_1 superou M_2, caso contrário ($M_1 > M_2$), tem-se que M_1 foi superado por M_2.

5.2 Escalonador Baseado em AC Síncrono (EACS)

5.2.1 Modelos Anteriores

Localidade de interações celulares, simplicidade de componentes básicos (células) e possibilidade de implementação em *hardware* paralelo estão entre as mais notáveis caracteristicas dos autômatos celulares [50]. Contudo, os modelos de escalonamento baseado em AC da literatura que obtiveram melhores resultados não são capazes de explorar a capacidade intrínseca de paralelismo dos ACs, uma vez que utilizam o modo de atualização sequencial (apenas uma célula pode atualizar seu estado por vez) [49].

Assim, o foco inicial desse trabalho teve por objetivo a construção de um modelo de escalonador baseado em AC que, através do modo de atualização síncrono, obtivesse resultados satisfatórios de escalonamento, similares aos obtidos nos trabalhos anteriores

com o modo sequencial. A abordagem proposta para atingir tal objetivo foi denominada EACS.

5.2.2 Arquitetura EACS

O Escalonador baseado em AC com atualização Síncrona (EACS) utiliza vizinhança linear por três razões: simplicidade, baixo custo computacional e número arbitrário de processadores. É certo que em trabalhos anteriores [49] as vizinhanças não lineares apresentaram resultados melhores do que as lineares. Entretanto, elas estão limitadas a uma arquitetura multiprocessada com dois nós e possuem uma estrutura bastante complexa. Como exemplo, a vizinhança não linear investigada em [49] emprega regras com 250 *bits*, enquanto na vizinhança linear, considerando o tamanho do raio igual a 3, o tamanho das regras é igual a 128 *bits*. No caso da vizinhança não-linear investigada em [49], os modelos de vizinhança só prevêem o uso de 2 processadores no grafo de sistema e uma generalização desses modelos para um número maior de processadores seria bastante complexa e levaria a uma regra de dimensão ainda maior. No caso da vizinhança linear, para um número maior de processadores no grafo de sistema, basta utilizar um número maior de estados por célula.

Outra característica importante do EACS diz respeito à condição de contorno utilizada, que assim como nos modelos anteriores, também é nula. Contudo, no EACS, as células de contorno à esquerda do reticulado possuem valor 0 e as da direita 1, diferente dos demais modelos que utilizam 0 dos dois lados. Para decidir por esses valores foi feito um estudo acerca da influência da condição de contorno nos resultados encontrados pelas vizinhanças onde concluiu-se que tal condição (0 e 1) oferece maior equilíbrio (uso mais distribuído dos estados de transição das regras) na evolução temporal do reticulado.

Além das mudanças na estrutura do AC, o AG empregado no EACS também utiliza uma abordagem diferente dos modelos anteriores por não utilizar estratégia elitista (métodos de seleção e re-inserção utilizam elitismo). Dessa forma, no novo modelo, a seleção é realizada por meio de torneio simples e a re-inserção acontece baseada na aptidão, isto é, a população total (pais e filhos) é ordenada e selecionam-se as melhores regras. O intuito dessas modificações é estimular a competição entre os indivíduos da população a fim de permitir uma busca mais ampla no espaço de possíveis soluções para o problema e, consequentemente, formar um conjunto de regras mais eficiente em sua totalidade, o que é bastante difícil utilizando uma estratégia elitista.

Na Figura 5.1 é apresentado um *framework* do modelo de escalonador proposto. O EACS recebe como entrada um grafo de programa e um grafo de sistema. EACS verifica se já existem regras em RDB para o grafo de programa recebido. No modo de aprendizagem utiliza-se um AG para buscar regras de AC capazes de encontrar o escalonamento ótimo para o grafo de programa. A população de indivíduos desse AG é avaliada a cada passo

do processo evolutivo através de um conjunto de configurações iniciais (CIs) geradas aleatoriamente. Ainda no processo de avaliação, as regras de AC realizam a evolução temporal destas configurações e obtêm o escalonamento das tarefas que é tomado como valor de aptidão pelo AG. A Figura 5.2 apresenta o pseudo-código do AG utilizado no modelo. Ao final da execução do AG a população é carregada na Base de Regras (RDB). No modo de execução, o grafo de programa é inicializado com uma CI aleatória e o AC é equipado com uma regra da RDB. O AC então atualiza sincronamente o reticulado por S passos de tempo obtendo a alocação final das tarefas que é então submetida à política de escalonamento. O resultado final é dado pelo tempo de execução (*makespan*) associado ao escalonamento do grafo de programa definido pela configuração final do reticulado.

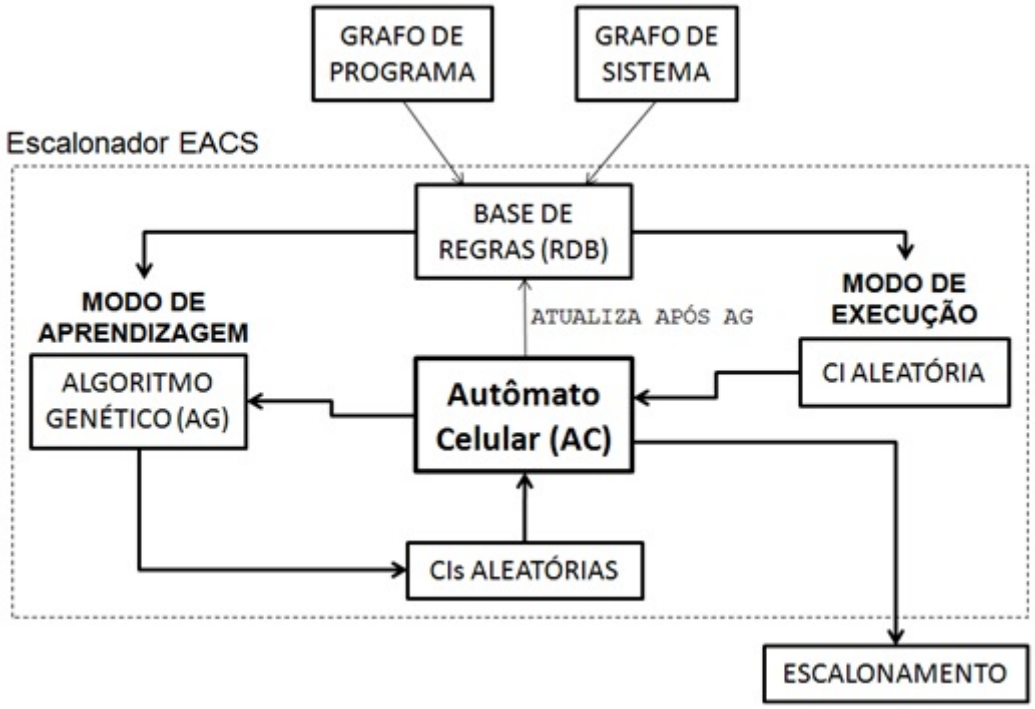


Figura 5.1: Modelo de escalonador baseado em AC (EACS).

5.2.3 Resultados Experimentais

Nesta seção são descritos experimentos realizados a fim de avaliar o desempenho do modelo EACS. O objetivo é comparar os resultados obtidos pelo EACS com a reprodução do modelo de escalonador proposto em [53], em que o modo de atualização síncrono e assíncrono foram investigados. Dessa forma, a meta principal é avaliar se o modo de atualização síncrono em EACS é apto a obter um desempenho satisfatório em relação aos modelos anteriores com atualização síncrona e com resultados mais próximos aos de modelos com atualização sequencial. Os grafos considerados nos testes iniciais foram os mesmos utilizados em [53]: *g18*, *g40* e *gauss18*. Também foram gerados três grafos aleatórios: *random30*, *random40* e *random50*, com 30, 40 e 50 tarefas, respectivamente.

Modo de Aprendizagem

Passo 0 Gere aleatoriamente uma população de T_{pop} regras;

Passo 1 Enquanto (!condição_término) // faça Passos 2-6

Passo 2 Calcule o *fitness* para todas as regras // faça Passos 2.1-2.3

Passo 2.1 Gere aleatoriamente um conjunto de configurações iniciais (CI);

Passo 2.2 Utilizando evolução paralela, aplique as T_{pop} regras sobre cada CI por S passos de tempo e calcule o tempo total de execução T para cada alocação final;

Passo 2.3 Obtenha o *fitness* para cada regra através da soma dos valores de T encontrados para cada CI avaliada;

Passo 3 Selecione pares de regras em T_{pop} utilizando torneio simples;

Passo 4 Aplique o *crossover* de ponto simples para os pares selecionados gerando P_c regras;

Passo 5 Submeta P_{cross} regras à mutação com probabilidade P_{mut};

Passo 6 Ordene T_{pop}+P_{cross} e escolha as T_{pop} melhores regras para a nova geração;

Figura 5.2: Pseudo-código do AG utilizado no modo de aprendizagem em EACS.

As vizinhanças utilizadas para o AC nestes experimentos foram raio $R \in \{1, 2, 3\}$. O número de passos do AC (S) foi definido de acordo com a complexidade do grafo de programa considerado, sendo igual ao número de tarefas para grafos simples como os *out-trees* e igual a 50 para os demais grafos. Outros parâmetros utilizados foram: torneio simples $Tour \in \{2, 3\}$, taxa de cruzamento $P_{cross} = 100\%$, taxa de mutação $P_{mut} = 3\%$ e conjunto de configurações iniciais aleatórias $CI = 50$. Os parâmetros que apresentaram maior variação de um experimento para outro foram tamanho da população T_{pop} e número de gerações G. Foram realizadas 20 execuções para cada experimento e a política de escalonamento adotada em todos eles foi "a tarefa com maior nível dinâmico primeiro". Em todos os experimentos também foram reproduzidos os modelos síncrono e sequencial apresentados em [53] por considerá-los modelos de referência para a vizinhança linear. Assim, nas figuras e tabelas exibidas nesta seção, "Reprod. Seq" está relacionado aos resultados obtidos na reprodução do modelo apresentado em [53] para o modo de atualização sequencial e "Reprod. Sinc" está relacionado aos resultados encontrados para a reprodução do modelo apresentado em [53], utilizando o modo de atualização síncrono.

EACS e modelos relacionados

A Tabela 5.1 mostra os resultados de *makespan* encontrados na fase de aprendizagem pelo EACS, pela reprodução com modo síncrono e pela reprodução com modo sequencial. Na tabela, "BEST" está relacionado à aptidão da melhor regra para o modo de aprendizagem em 20 execuções. É importante lembrar que as regras são avaliadas a partir de um conjunto de configurações iniciais aleatórias e que a média do tempo de escalonamento

encontrado em todas estas CIs é considerada a aptidão da regra de transição, o que justifica os valores decimais nas colunas de "BEST". "AVG" aponta a média de escalonamento considerando a aptidão da melhor regra em cada execução Os resultados dessa tabela são discutidos em maiores detalhes a seguir.

Tabela 5.1: Comparativo entre EACS e os modelos reproduzidos: fase de aprendizagem.

G_P	**EACS**		**Reprod. Sinc**		**Reprod. Seq**	
	BEST	AVG	BEST	AVG	BEST	AVG
g18	46,00	46,00	46,00	46,00	46,00	46,00
g40	80,00	80,81	80,00	80,89	80,00	80,76
gauss18	44,00	47,86	47,00	49,31	44,00	47,60
random30	1225,84	1267,50	1250,76	1275,81	1239,00	1247,71
random40	996,52	1024,19	1008,32	1027,58	1006,00	1020,47
random50	661,04	673,98	669,68	676,80	659,04	667,78

O primeiro experimento foi realizado com o grafo de programa apresentado na Figura 2.6(a), *g18*. A vizinhança mínima na literatura para resolver este grafo utiliza raio 2 [53] e o *makespan* ótimo (T_{OT}) em um sistema de dois processadores é igual a 46. Utilizando $R = 2$, $T_{pop} = 100$ e $G = 100$, o AG rapidamente encontra regras de AC capazes de encontrar T_{OT} para qualquer CI. Na reprodução de [53], tanto para o modo de atualização sequencial como para o síncrono, o AG também foi capaz de encontrar regras aptas a realizar o escalonamento ótimo, utilizando-se os mesmos parâmetros empregados no EACS. Assim, de acordo com a Tabela 5.1, os resultados entre os métodos foram considerados equivalentes na aprendizagem do grafo *g18*.

As Figuras 5.3(a) e 5.3(d) mostram o número de soluções ótimas alcançadas para o grafo *g18* para cada uma das 100 regras da população final considerando-se o modelo EACS e as reproduções de [53], no modo de execução. Ao contrário dos resultados apresentadas na Tabela 5.1, que se referem ao modo de aprendizagem, nestas figuras não foi apresentado os resultados médios das regras, mas sim o número de CIs (de 1000 CIs avaliadas) que alcançaram o *makespan* ótimo. De acordo com trabalhos anteriores [46, 49, 53], o intuito em utilizar um número maior de CIs na fase de operação/execução é avaliar se realmente a regra é capaz de levar qualquer configuração aleatória para uma configuração que represente a alocação ótima das tarefas. Em termos de resultados, é importante notar que o modelo sequencial reproduzido a partir de [53] apresenta melhor desempenho que o modelo síncrono também baseado em [53], contudo o melhor desempenho foi obtido pela nova abordagem (EACS) por apresentar um número maior de regras capazes de convergir as CIs para alocações ótimas. Os gráficos na parte inferior da figura apresentam um detalhamento dos gráficos da parte superior apresentando apenas a escala de 950 a 1000 CIs, com o objetivo de tornar mais claro o desempenho dos modelos reproduzidos.

No segundo experimento, o grafo de programa considerado foi o *g40* que pode ser visto na Figura 2.6. A menor vizinhança na literatura para resolver este grafo emprega

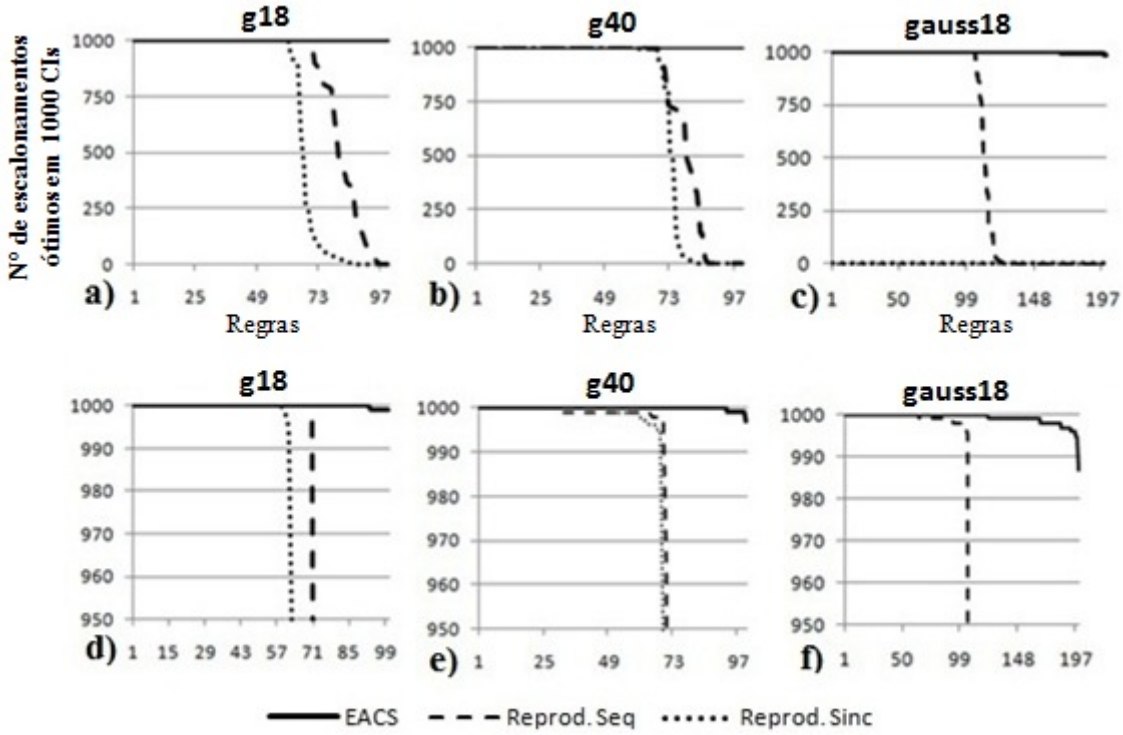


Figura 5.3: Modo de execução dos escalonadores: (a) *g18*; (b) *g40*; (c) *gauss18*; (d) *g18* (escala de 950 a 1000); (e) *g40* (escala de 950 a 1000); (f) *gauss18* (escala de 950 a 1000).

raio 2 e o tempo ótimo T_{OT} em um sistema com dois processadores é igual a 80 [53]. Os parâmetros utilizados foram $R = 2$, $T_{pop} = 100$ e $G = 100$. Apesar de utilizarem os mesmos parâmetros, a convergência do modo de aprendizagem para o *g40* é mais demorada que para o *g18*. Entretanto, todos os modelos encontraram regras capazes de realizar o escalonamento ótimo conforme apresentado na Tabela 5.1. Assim, avaliou-se o conjunto de regras obtidos através do escalonamento de 1000 CIs no modo de execução. O resultado dessa avaliação é apresentado nas Figuras 5.3(b) e 5.3(e) que destacam o número de soluções ótimas alcançadas em cada uma das regras de cada modelo. Novamente, o EACS obteve os melhores resultados. Comparando-se os modelos reproduzidos, o modelo sequencial retornou um resultado médio melhor de acordo com as figuras, por outro lado considerando a convergência do ótimo, o modelo síncrono apresentou um melhor desempenho de acordo com a Tabela 5.2.

Além do número de soluções ótimas alcançadas, a qualidade das regras obtidas também foi examinada. A Figura 5.4 mostra a média do tempo de escalonamento obtida pelas regras armazenadas em RDB, no modo de execução (em 1000 CIs), para a melhor execução (de 20), considerando os grafos de programa *g18* (a) e *g40* (b). Os resultados apresentados na figura mostram que "Reprod. Seq" e "Reprod. Sinc" criaram um conjunto de regras com grande variação em termos de desempenho do escalonamento. Por outro lado, o EACS foi capaz de criar um conjunto onde quase todas as regras apresentaram desempenho ótimo.

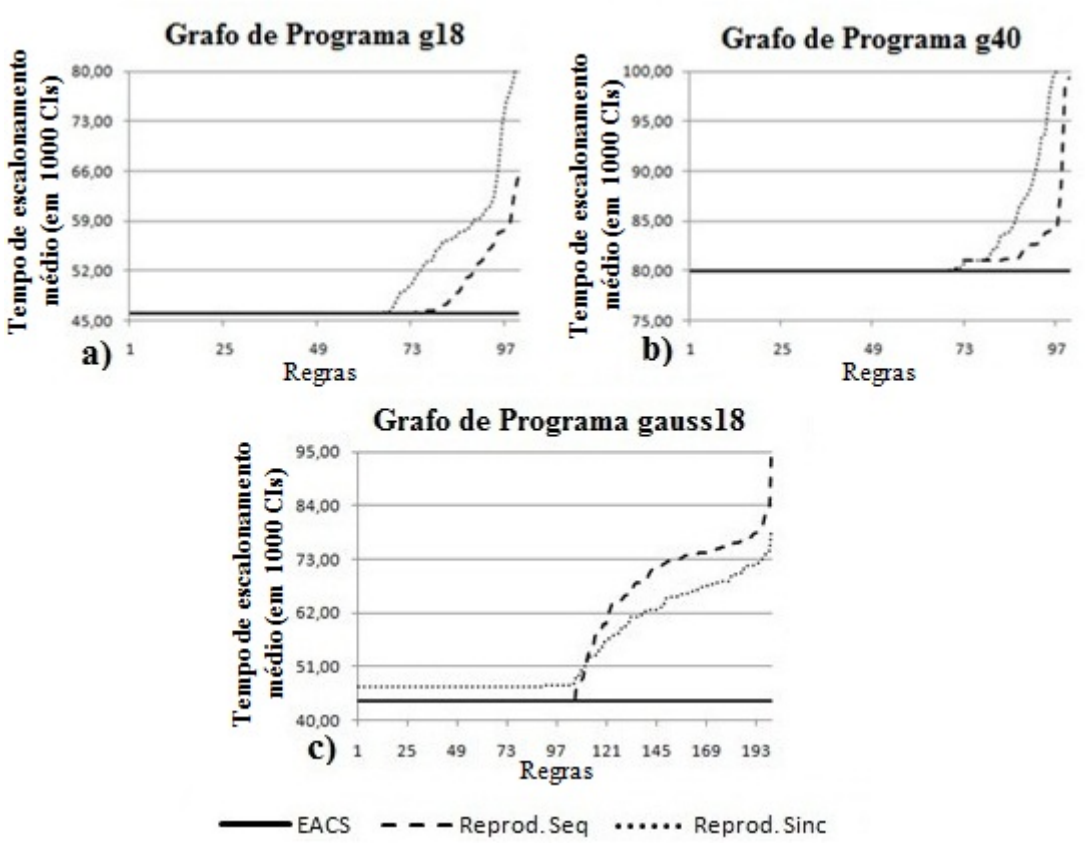


Figura 5.4: Avaliação do conjunto de regras para (a)*g18*, (b)*g40* e (c)*gauss18* em EACS.

O grafo de programa usado no terceiro experimento foi o *gauss18* que é apresentado na Figura 2.5. Ele é considerado mais difícil que os demais para o modo de aprendizagem devido à sua estrutura irregular [53]. O seu tempo ótimo T_{OT} em um sistema com dois processadores é igual a 44 [49]. Os parâmetros utilizados no modelo EACS foram os mesmos dos utilizados em [53] ($R = 3$, $T_{pop} = 200$ e $G = 1000$). Vale destacar também que nenhum dos trabalhos relacionados [49, 53, 57] foi capaz de encontrar T_{OT} para este grafo utilizando vizinhança linear e modo de atualização síncrono. Contudo, utilizando modo de atualização sequencial, a vizinhança mínima necessária para alcançar T_{OT} foi raio 3 [53].

A reprodução de [53] confirma as afirmações do parágrafo anterior, uma vez que na melhor execução, a melhor regra obtida na "Reprod. Sinc" retornou 47 para qualquer configuração inicial gerada enquanto que a melhor regra da "Reprod. Seq" conseguiu obter 44. Contudo, o modelo EACS foi apto a levar qualquer CI gerada para um escalonamento ótimo utilizando a atualização síncrona. As Figuras 5.3(c) e 5.3(f) mostram que no modo de execução, o EACS obteve melhor desempenho que "Reprod. Seq".

Na Figura 5.4(c) é possível perceber uma grande diferença entre os modelos "Reprod. Seq" e EACS: o conjunto de regras encontrado. Enquanto em "Reprod. Seq" mais de 50% das regras não foram capazes de encontrar escalonamentos ótimos (ou próximos do ótimo)

para todas as configurações iniciais, no EACS quase todas as regras encontraram escalonamentos ótimos para qualquer CI. As piores regras obtidas com EACS, "Reprod. Sinc" e "Reprod. Seq", retornaram escalonamentos médios de 44.1, 79.2 e 94.0, respectivamente (considerando 1000 configurações iniciais).

A Tabela 5.2 apresenta o percentual das regras que, na fase de execução do escalonador, foram aptas a escalonar 1000 CIs para o tempo ótimo. É interessante perceber que para os três grafos de programa, o modelo EACS obteve o melhor desempenho. Para o *gauss18*, por exemplo, 61% do das regras do EACS foram capazes de encontrar T_{OT} para as 1000 CIs, contra $31,5\%$ da "Reprod. Seq" e 0% da "Reprod. Sinc".

Tabela 5.2: Percentual de regras aptas a escalonar 1000 CIs para T_{OT} no modo de execução dos grafos *g18*, *g40* e *gauss18*.

Abordagens	*g18*	*g40*	*gauss18*
Reprod. Seq	70%	31%	31,5%
Reprod. Sinc	58%	49%	0%
EACS	92%	92%	61%

O quarto experimento foi conduzido com grafos de programa gerados aleatoriamente: *random30*, *random40* e *random50*. É possível observar na Tabela 5.1 que o modelo EACS retornou os melhores resultados para a fase de aprendizagem utilizando *random30* e *random40*. Para o grafo de programa *random50*, os melhores resultados foram encontrados por "Reprod. Seq", embora a melhor regra de EACS obteve um desempenho próximo. A Figura 5.5 mostra o desempenho do conjunto de regras obtidos considerando cada modelo no modo de execução: Figuras 5.5(a), 5.5(b) e 5.5(c) referem-se aos grafos de programa *random30*, *random40* e *random50*, respectivamente. É possível perceber na figura uma grande variação no desempenho das regras em todos os grafos de programa aleatório. Contudo para as regras encontradas pelo EACS essa variação é muito menos significativa, como se pode perceber ao considerar a pior regra encontrada em cada conjunto de regras. Por exemplo, observando o grafo de programa *random30*, a pior regra obtidas por EACS, "Reprod. Sinc" e "Reprod. Seq" retornou uma média de 1265.42, 1875.97 e 2186.54, respectivamente. Considerando o grafo de programa *random40*, a pior regra obtidas por EACS, "Reprod. Sinc" e "Reprod. Seq" retornou uma média de 1012.54, 1286.56 e 1414.88, respectivamente. E finalmente, a pior regra, considerando o grafo de programa *random50*, obtida com EACS, "Reprod. Sinc" e "Reprod. Seq" retornou uma média de 668.24, 959.70 e 966.15, respectivamente.

Uma análise do desempenho geral de cada um dos modelos também foi feita. Para cada um dos modelos considerou-se uma população de 20 amostras referente a cada uma das execuções. Testes de hipótese conforme descrito na Seção 5.1.4 foram realizados. A Tabela 5.3 apresenta o resultado obtido nestes testes. Inicialmente, a população de Reprod. Sinc foi analisada em relação à população de Reprod. Seq. Os resultados obtidos

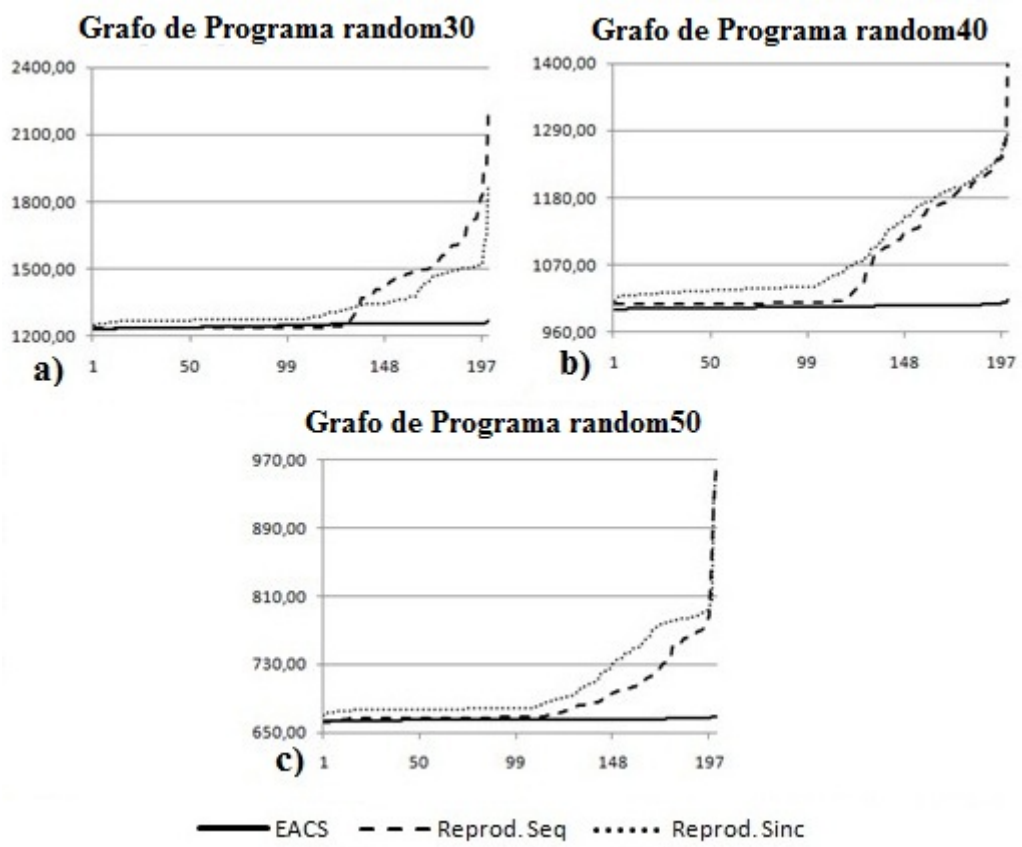


Figura 5.5: Avaliação do conjunto de regras para *random30*, *random40* e *random50* em EACS.

nesta análise estão dispostos na coluna "I" e permitem concluir que o modelo da literatura com atualização síncrona realmente apresenta uma grande desvantagem em relação ao de atualização sequencial, uma vez que há evidências estatísticas significativas de que o Reprod. Seq foi melhor que Reprod. Sinc em 4 grafos de programa. Posteriormente, na coluna "II" são exibidos os resultados encontrados para a comparação entre as amostras do modelo EACS e de Reprod. Seq, a partir dos quais conclui-se que há evidências estatísticas significativas de que Reprod. Seq foi melhor em apenas um grafo de programa e que a hipótese nula (H_0: $M_1 = M_2$) não pode ser rejeitada nos demais. Os resultados apresentados através destes testes permitem apontar uma melhora promissora em relação ao modo de atualização síncrono das células alcançada por meio do modelo EACS.

A Tabela 5.4 destaca os resultados obtidos no modo de execução utilizando os modelos EACS, "Reprod. Sinc" e "Reprod. Seq". É esperado que, para qualquer alocação inicial de tarefas, as regras descobertas sejam capazes de minimizar o *makespam*. Cada valor na tabela representa o desempenho da melhor regra obtida no modo de execução, quando cada regra é utilizada para evoluir 1000 configurações iniciais geradas aleatoriamente. Todos os resultados para o modelo EACS foram iguais (*g18*, *g40*) ou melhores (*gauss18*, *random30*, *random40* e *random50*) que "Reprod. Sinc". Além disso, EACS apresentou melhores resultados que "Reprod. Seq" para *random30* e *random40*, enquanto que os

Tabela 5.3: Análise estatística: I (M_1: Reprod. Sinc e M_2: Reprod. Seq); II (M_1: EACS e M_2: Reprod. Seq).

G_P	I	II
g18	=	=
g40	=	=
gauss18	>	=
random30	>	>
random40	>	=
random50	>	=

resultados para *random50* ficaram próximos.

Tabela 5.4: Valores obtidos para o modo de execução em EACS.

G_P	**EACS**	**Reprod. Sinc**	**Reprod. Seq**
g18	46,00	46,00	46,00
g40	80,00	80,00	80,00
gauss18	44,00	47,00	44,00
random30	1226,95	1251,36	1239,00
random40	997,27	1010,40	1006,00
random50	662,90	669,48	661,18

Reuso do conhecimento

Também foi avaliada a capacidade de reuso de regras em outras instâncias de grafos de programa. Um último experimento foi conduzido com o grafo de programa *outtree15*. A vizinhança foi definida por $R = 1$ e os parâmetros do AG utilizados foram $T_{pop} = 30$ e $G = 30$. Este grafo foi considerado fácil uma vez que o AG rapidamente encontrou regras de AC capazes de encontrar o *makespan* para qualquer CI apresentada. Contudo, o propósito foi escolher aleatoriamente uma regra do conjunto de regras obtido para realizar o escalonamento com os grafos de programa *outtree31*, *outtree63*, *outtree127*, *outtree255*, *outtree511* e *outtree1023*. Além disso, foi utilizado um algoritmo genético (AG-1, descrito na Seção 5.1.2), com os mesmos parâmetros do AG utilizado no modo de aprendizagem do escalonador e seu resultado e tempo computacional foram comparados aos alcançados por "EACS" no modo de execução. O intuito desta comparação é mostrar que, apesar do modelo proposto também fazer uso de um AG, o seu uso ocorre apenas no modo de aprendizagem e seu foco é a busca no espaço de regras do AC, enquanto que o AG padrão tem como objetivo encontrar uma solução para determinado grafo de programa. A Tabela 5.5 apresenta o comparativo para 10 execuções. Assim, "BEST" e "tempo (ms)" referem-se, respectivamente, ao menor custo de escalonamento encontrado e ao tempo médio, em milissegundos, consumido em cada execução.

Tabela 5.5: Resultados obtidos por EACS e um AG simples para os grafos de programa *outtree*.

G_P	EACS				AG	
	Aprendizagem		Execução			
	BEST	tempo (ms)	BEST	tempo (ms)	BEST	tempo (ms)
outtree15	9	17.874	9	0	9	11
outtree31	-	-	17	1	17	43
outtree63	-	-	33	3	33	196
outtree127	-	-	65	11	66	1.084
outtree255	-	-	129	45	140	6.587
outtree511	-	-	257	191	287	53.267
outtree1023	-	-	513	725	602	282.592

Considerando a Tabela 5.5, inicialmente utilizou-se o modo de aprendizagem do EACS para extrair informações sobre o grafo de programa *outtree15*. O *outtree15* foi considerado fácil, pois o modelo EACS precisou de poucos passos para encontrar uma regra cuja alocação das tarefas é ótima para o grafo. Terminada a aprendizagem, o modo de execução de EACS foi utilizado sobre o mesmo grafo, onde também foi capaz de encontrar a alocação ótima. Contudo, as regras extraídas para o *outtree15* foram também utilizadas no reuso sobre os outros grafos exibidos na tabela e em todos eles, o modelo EACS foi capaz de encontrar a solução ótima. O tempo computacional gasto em cada aplicação do EACS também é apresentado na tabela. Posteriormente, AG-1 foi aplicado sobre cada um dos grafos da tabela e o seu tempo computacional também foi calculado. Diante das informações da tabela, é possível entender a principal finalidade do escalonamento baseado em AC, o reuso do conhecimento extraído de um programa paralelo em outros grafos de programa. De fato, a fase de aprendizagem em EACS apresenta um custo computacional maior que o do AG, contudo no modo de execução, o custo computacional de EACS é menor e o seu desempenho melhor. É interessante observar que aumentando o número de tarefas, o espaço de busca de possíveis soluções também aumenta e o AG, por não utilizar qualquer conhecimento, precisa realizar novamente, para cada aplicação, todos os passos evolutivos. Características como a extração e o reuso do conhecimento enfatizam a importância em se investigar escalonadores baseados em AC para o PEET.

5.2.4 Considerações Finais

O EACS (Escalonador baseado em Autômato Celular com atualização Síncrona) é o modelo inicial proposto neste trabalho com o intuito de possibilitar o uso do modo de atualização paralelo das células no escalonamento baseado em AC com desempenho satisfatório. De fato, trabalhos anteriores [49,53,57], apesar de destacarem bons resultados no uso dos ACs com atualização sequencial, desvalorizaram o modo síncrono por apresentar resultados inferiores. Na verdade, uma das principais características dos ACs é a sua

adequação à implementação massivamente paralela e um modelo de escalonador capaz de usufruir dessa capacidade deve ser almejado.

Estabeleceu-se um esquema comparativo entre o EACS e os modelos anteriores a fim de avaliar a qualidade da nova abordagem. Como resultado, há evidências estatísticas significativas de que o desempenho do modelo EACS no modo de aprendizagem foi superior aos modelos da literatura com atualização síncrona e, ainda que não tenha superado, esteve próximo dos modelos com atualização sequencial. Porém, no modo de execução, o modelo EACS apresentou o maior número de regras capazes de convergir qualquer CI para o escalonamento ótimo (grafos de programa da literatura), e também obtiveram um conjunto de regras com resultados superiores àqueles conjuntos encontrados pelos modelos relacionados, tanto síncronos como assíncronos. Além disso, o modo de execução também alcançou bom desempenho no reuso das regras e destacou a maior vantagem do escalonamento baseado em ACs sobre a maioria das outras meta-heurísticas: a capacidade de extração e reuso do conhecimento.

Os resultados obtidos com o modelo EACS permitiram a submissão e publicação de dois artigos [9, 10].

5.3 Escalonador com Inicialização Baseada em Heurística de Construção (EACS-H)

5.3.1 Introdução

Mesmo com as melhorias e os bons resultados alcançados pelo EACS frente aos trabalhos apresentados na literatura [9, 10], um problema notado nessa abordagem e também nas anteriores aponta a grande dificuldade em se encontrar bons resultados diante do aumento do número de processadores e também considerando a fase de reuso das regras.

Em [53] é apresentada uma tentativa de aumentar o número de processadores (V_s), mas os resultados encontrados não foram satisfatórios e mostraram que este modelo de escalonador não tem escalabilidade. A Tabela 5.6 mostra os resultados publicados em [53] (aptidão da melhor regra na fase de aprendizagem) utilizando modo de atualização sequencial e, como referência, os valores alcançados por um AG padrão também implementado em [53] (tempo de escalonamento).

Nesta dissertação, um AG padrão também foi utilizado para se obter valores de referência para análise dos resultados obtidos pelo escalonador baseado em AC (AG-1, descrito na Seção 5.1.2). A Tabela 5.7 destaca o reuso do conhecimento extraído durante o escalonamento do grafo de programa *gauss18* (para $V_s = 2$) sobre os demais grafos (considerando a média do tempo de escalonamento obtida pela melhor regra em 1000 CIs). De fato, os valores encontrados estão distantes daqueles tomados por referência obtidos pelo AG-1.

Tabela 5.6: Resultados publicados em [53].

Abordagens	V_s	**g18**	**g40**	**gauss18**
AC em [53]	2	46,00	80,00	44,00
	3	38,00	57,00	48,00
	4	27,00	46,00	52,00
	8	26,00	39,00	52,00
AG em [53]	2	46,00	80,00	44,00
	3	36,00	57,00	44,00
	4	26,00	45,00	44,00
	8	24,00	33,00	44,00

Tabela 5.7: Modo de reuso (*gauss18*) sobre grafos de programa distintos para a reprodução do modelo sequencial apresentado em [53] .

G_P	V_s	**AG-1**	**Reprod. Seq**
g18	2	46	49,49
g40	2	80	86,25
random30	2	1222	1336,61
random40	2	983	1136,55
random50	2	624	730,70

Entendemos que a principal razão para a dificuldade desses modelos em lidarem com o reuso em instâncias diferentes de grafos de programa e com o aumento do número de processadores deve-se à complexidade da busca pelas regras no modo de aprendizagem, oriunda da necessidade de que uma regra precisa convergir qualquer configuração inicial das tarefas para um alocação ótima. De fato, considerar um conjunto de configurações diferentes a cada geração torna a busca ainda mais díficil e computacionalmente mais intensiva. Na verdade, o que se deseja é um conjunto de regras aptas a encontrarem uma alocação ótima das tarefas para uma determinada instância do problema e que esse mesmo conjunto possa ser utilizado com sucesso no escalonamento de outros grafos de programa. Diante disso, a metodologia empregada no modo de aprendizagem para avaliação do conhecimento foi investigada e um novo modelo de escalonador baseado em AC foi proposto: EACS-H.

5.3.2 Arquitetura do EACS-H

O modelo de escalonador baseado em AC com inicialização por heurística de construção (EACS-H) proposto nesta seção tem por objetivo um melhor desempenho no que se refere ao aumento do número de processadores e à capacidade de reuso das regras e pode ser diferenciado dos modelos anteriores [9, 49, 53, 57], por duas características principais:

- heurística de construção: no novo modelo EACS-H não há um conjunto de configurações iniciais (CI) de reticulados aleatórios a cada geração do AG. Na verdade,

existe apenas uma CI obtida através do escalonamento realizado por uma heurística de construção determinística escolhida a priori. É a partir dessa CI que a avaliação de todas as regras é realizada;

- capacidade de reuso: em modelos anteriores, o foco da avaliação da qualidade final das regras obtidas pelo AG é a capacidade de uma regra de transição levar qualquer CI para uma configuração ótima, que representa a alocação ideal das tarefas nos processadores. Contudo a busca por essa independência em relação ao reticulado inicial torna a busca pelas regras mais complexa, dificultando a convergência por parte do AG. Além disso, nós não encontramos nenhuma razão prática para a necessidade desta capacidade de generalização ser exigida de uma regra a fim de que ela seja apta a realizar um escalonamento eficiente. Assim, nós acreditamos que a generalização mais importante não está relacionada às CIs do reticulado, mas à capacidade de uma regra escalonar outras instâncias no modo de reuso [11].

Na Figura 5.6 é apresentado um esboço da arquitetura do novo modelo. Assim como acontece no EACS, o modelo EACS-H recebe como entrada um grafo de sistema com V_s processadores e um grafo de programa. A "Heurística de Construção" está relacionada a um método determinístico utilizado para criar um bom ponto de partida para o escalonador: o reticulado inicial a ser utilizado por todas as regras da população na fase de aprendizagem. Posteriormente, também será utilizada a mesma heurística para gerar os reticulados iniciais associados aos grafos que venham a ser escalonados pelas regras da RDB.

Assim como no modelo EACS, o EACS-H também utiliza modo de atualização paralelo das células e o uso de uma política de escalonamento. Assim, o modo de aprendizagem pode ser descrito da seguinte maneira: (i) obtenção da configuração inicial (CI) do reticulado através do escalonamento realizado por uma heurística de construção determinística escolhida a priori; (ii) atualização paralela desta CI pela população de regras por S passos de tempo; (iii) cálculo do tempo de escalonamento para cada regra através da aplicação da política de escalonamento sobre o reticulado.

No reuso, o AC é equipado com um conjunto de regras no repositório e as etapas (i) até (iii) são realizadas para cada regra. Na etapa de reuso, a regra que minimiza o escalonamento do novo grafo de programa considerado é retornada. A mesma heurística da fase de aprendizagem é utilizada também no reuso. Neste trabalho, uma heurística de construção comum e muito utilizada na literatura foi adaptada: HLFET (Figura 2.8), explicada na Seção 2.4.1. Por ela apresentar alguns passos aleatórios, foi elaborada uma versão determinística chamada DHLFET, que é o HLFET sem as escolhas aleatórias no caso de empate na lista de tarefas prontas (Seção 2.4.1). Assim, no caso de duas tarefas com mesmo sl, a tarefa com menor número de ordem é escolhida.

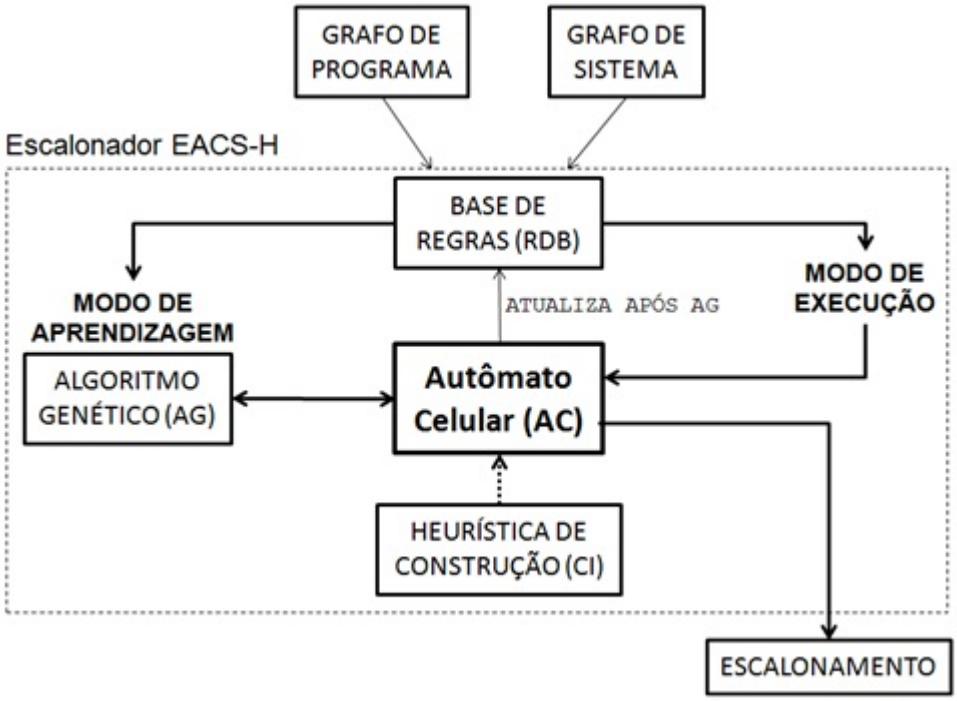


Figura 5.6: Arquitetura EACS-H (escalonador com AC baseado em heurística).

5.3.3 Resultados Experimentais

Nesta seção são relatados alguns experimentos para avaliar o desempenho do novo modelo EACS-H especialmente em relação ao aumento do número de processadores. Os resultados são comparados com algumas metaheurísticas, heurísticas de construção e trabalhos relacionados.

O número de processadores V_s utilizado nos experimentos foram 2, 3, 4 e 8. O tamanho da vizinhança foi $R \in \{2, 3\}$ para 2 processadores e $R \in 1$ para 3 ou mais processadores. O número de passos de atualização do AC $S = 50$. Os parâmetros relacionados ao AG foram: tamanho da população $T_{pop} \in \{200, 300\}$, torneio simples $Tour = 2$, taxa de cruzamento $P_{cross} = 100\%$, taxa de mutação $P_{mut} = 3\%$ e número de gerações $G = 200$. Foram realizadas 20 execuções para cada experimento e a política de escalonamento adotada em todos elas foi "a tarefa com maior nível dinâmico primeiro".

EACS-H e modelos anteriores

Um esquema comparativo do desempenho de EACS-H em relação aos modelos anteriores, considerando o aumento de processadores, é exibido na Tabela 5.8. A tabela apresenta uma comparação entre resultados encontrados na reprodução de [53] para o modo de atualização sequencial ("Reprod. Seq") e para o modo de atualização paralelo ("Reprod. Sin"), além do próprio EACS-H que utiliza atualização síncrona das células.

Na Tabela 5.8, "BEST" representa a aptidão da melhor regra no modo de operação normal para as reproduções de [53] e para o modelo EACS-H, e "AVG" representa a média

do melhor escalonamento considerando todas as execuções. Nos modelos anteriores, no modo de operação normal cada regra precisa evoluir 1000 CIs para configurações ótimas. Aliás, um problema com os modelos anteriores de escalonador baseado em AC é que eles não asseguram o mesmo escalonamento da aprendizagem no reuso para o mesmo grafo de programa, pois as configurações iniciais são geradas aleatoriamente e as regras podem ou não levá-las a configurações com resultado igual ao encontrado na aprendizagem. Por outro lado, o modo de operação normal não é necessário na arquitetura do EACS-H, pois concluído o modo de aprendizagem para um grafo de programa, é certo afirmar que o escalonamento do mesmo grafo de programa no modo de reuso é igual ao obtido pela melhor regra na aprendizagem, pois sempre se parte da mesma configuração inicial: aquela obtida pela heurística de construção. Por esse motivo, os valores apresentados em BEST para as reproduções dos modelos de [53] são valores médios (em 1000 CIs aleatórias), enquanto os valores na coluna BEST para o EACS-H são valores obtidos a partir de uma única heurística de inicialização.

Tabela 5.8: Comparativo do desempenho de EACS-H e modelos sequenciais e síncronos reproduzidos.

G_P	V_s	**Reprod. Seq**		**Reprod. Sinc**		**EACS-H**	
		BEST	AVG	BEST	AVG	BEST	AVG
g18	2	46,00	46,00	46,00	46,00	46,00	46,00
	3	38,00	38,36	38,00	38,50	36,00	36,55
	4	26,00	26,90	26,51	26,94	26,00	26,05
	8	26,68	29,26	34,99	33,60	24,00	24,00
g40	2	80,00	80,76	80,00	80,89	80,00	80,00
	3	57,00	57,15	57,00	57,37	57,00	57,00
	4	45,28	45,82	45,67	46,05	45,00	45,55
	8	36,00	41,64	46,76	46,05	32,00	32,00
gauss18	2	44,00	47,60	47,00	49,31	44,00	44,00
	3	52,00	52,65	52,00	53,28	44,00	44,05
	4	51,86	52,68	52,00	53,35	44,00	44,05
	8	52,02	56,49	52,36	56,76	46,00	51,40
random30	2	1239,00	1247,71	1251,36	1275,81	1222,00	1224,45
	3	963,00	1021,89	1020,95	1024,45	853,00	892,90
	4	911,19	1011,58	1019,98	1024,77	828,00	849,90
	8	1085,11	1074,98	1100,87	1080,90	804,00	828,75
random40	2	1006,00	1020,47	1010,40	1027,58	983,00	984,80
	3	810,94	833,40	820,98	854,95	694,00	708,45
	4	681,00	719,65	716,13	754,67	607,00	631,85
	8	814,01	797,67	816,23	798,04	551,00	568,85
random50	2	661,18	667,78	669,48	676,80	628,00	642,40
	3	643,09	659,60	660,00	664,30	532,00	539,40
	4	620,00	643,98	633,38	656,60	524,00	583,80
	8	652,05	667,78	683,54	764,56	636,00	665,00

Um ponto significante na Tabela 5.8 diz respeito aos resultados obtidos em "Reprod. Seq" que geralmente são melhores que aqueles encontrados em "Reprod. Sinc". Considerando, por exemplo, os grafos de programa para o caso de $V_s = 8$ é fácil perceber que o desempenho do modelo com modo de atualização sequencial é melhor. Tal informação confirma a escolha de outros trabalhos por esse tipo de atualização das células [49,53,57]. Todavia, os resultados na Tabela 5.8 destacam uma boa vantagem da abordagem de EACS-H sobre os trabalhos relacionados [53], tanto sobre o modo de atualização sequencial quanto paralelo. Analisando os grafos de programa considerados, é possível notar que *g18* (para V_s igual a 2 e 4) e *g40* (para V_s igual a 2, 3 e 4) apresentaram resultados similares para todos os modelos, contudo em todos os outros testes realizados EACS-H encontrou soluções melhores que "Reprod. Seq" e "Reprod. Sinc". Em alguns testes, a diferença entre estas soluções foi considerada bastante significativa como, por exemplo, no *random30* para 8 processadores, onde os tempos de escalonamento (*makespan*) obtidos por "Reprod. Seq" e "Reprod. Sinc" ultrapassaram em mais de 34% e 36%, respectivamente, a solução encontrada por EACS-H.

De fato, os melhores resultados da Tabela 5.8 estão associados ao novo modelo EACS-H. Esta superioridade pode ser vista considerando o aumento do número de processadores ($V_s > 2$) nos experimentos com o grafo de programa *gauss18*, onde os melhores resultados em "Reprod. Seq" e "Reprod. Sinc" para $V_s = 3$ foram iguais a 52, enquanto a média e o melhor resultado encontrado pelo EACS-H é 44 (solução ótima). Outro exemplo é o grafo de programa *random50*, onde os melhores valores encontrados em "Reprod. Seq" e "Reprod. Sinc" para $V_s = 3$ são, respectivamente, 643,09 e 660, enquanto a média das execuções de EACS-H é 539,40 e o melhor resultado é 532.

Pelos resultados apresentados por "Reprod. Seq" e "Reprod. Sinc" na Tabela 5.8 é possível notar que o aumento no número de processadores gera uma grande complexidade para o escalonador nesses modelos, pois os resultados não estão próximos dos valores de referência (AG) apresentados na Tabela 5.6. Isso fica claro nas reproduções de [53], onde o custo das soluções obtidas para 8 processadores é geralmente maior que os resultados obtidos para $V_s = 4$. Na Tabela 5.8, somente para "Reprod. Seq" no grafo de programa *g40*, a solução para $V_s = 8$ foi melhor. Nos demais grafos de programa (tanto em "Reprod. Seq" como em "Reprod. Sinc"), o aumento no número de processadores de 4 para 8 sempre degradou as soluções em termos de resultado obtido. Contudo, os resultados obtidos pelo modelo EACS-H mostram que essa abordagem lida melhor com o aumento da complexidade causada pelo incremento de processadores. Por exemplo, os resultados encontrados pelo EACS-H para os grafos de programa *g18*, *g40*, *random30* e *random40* apresentaram melhor desempenho de 4 para 8 processadores. Apenas no *gauss18* e no *random50*, o modelo EACS-H retornou resultados inferiores com 8 processadores em relação aos resultados com 4 processadores.

O desempenho computacional do modelo EACS-H em relação aos trabalhos reproduzi-

dos também foi avaliado. Como era esperado, os resultados revelaram que o tempo gasto para a fase de aprendizagem foi muito reduzido, uma vez que o modelo EACS-H realiza a evolução do AC apenas uma vez, partindo de uma única inicialização, enquanto que as reproduções fazem 100 evoluções a partir de 100 reticulados iniciais diferentes. Por exemplo, o tempo computacional para concluir o modo de aprendizagem em "Reprod. Seq" para o grafo de programa *random30* considerando $V_s = 3$ foi aproximadamente 62 vezes maior que o tempo computacional gasto pelo EACS-H para terminar o mesmo modo. Outro exemplo, é o tempo computacional gasto em "Reprod. Sinc" para a fase de extração do conhecimento no grafo de programa *gauss18* considerando $V_s = 8$ que foi quase 53 vezes maior que o tempo gasto pelo EACS-H. Assim, uma outra vantagem do novo modelo EACS-H é a redução do tempo de avaliação das soluções, que consequentemente diminui o custo computacional do modelo.

Como meio de realizar uma análise mais quantitativa em relação aos resultados obtidos por EACS-H frente aos modelos reproduzidos, testes de hipótese foram realizados considerando a fase de operação dos modelos e seguindo as descrições apresentadas na Seção 5.1.4. A Tabela 5.9 apresenta os resultados dos testes. Na coluna "Reprod. Sinc" analisa-se o desempenho de EACS-H(M_1) em relação a Reprod. Sinc (M_2) e na coluna "Reprod. Seq" em relação a Reprod. Seq (M_2). De acordo com os resultados, é possível concluir que há evidências estatísticas significativas de que EACS-H obteve melhor desempenho que Reprod. Sinc em 23 de 24 testes e melhor que Reprod. Seq em 22 de 24 testes. Tamanha vantagem comprova a grande melhoria da nova abordagem sobre a anterior.

Tabela 5.9: Análise estatística do desempenho de EACS-H (M_1) em relação ao dos modelos reproduzidos (M_2): Reprod. Sinc e Reprod. Seq.

G_P	**Reprod. Sinc**				**Reprod. Seq**			
	2	3	4	8	2	3	4	8
g18	=	<	<	<	=	<	<	<
g40	<	<	<	<	<	<	<	<
gauss18	<	<	<	<	<	<	<	<
random30	<	<	<	<	<	<	<	<
random40	<	<	<	<	<	<	<	<
random50	<	<	<	<	<	<	<	=

EACS-H e outras técnicas computacionais

Também foram conduzidos experimentos para avaliar o desempenho do modelo EACS-H em relação à outros métodos comumente empregados no PEET. Inicialmente, três grafos de programa encontrados em trabalhos relacionados foram utilizados no modo de aprendizagem, de modo que o conhecimento extraído foi reusado em outros grafos de

programa. Em paralelo, algumas heurísticas e metaheurísticas foram implementadas e testadas.

A Tabela 5.10 exibe os resultados de escalonamento para a heurística DHLFET (ponto de partida do EACS-H, pois define o reticulado inicial), EACS-H na fase de aprendizagem, algoritmo genético AG-1 (AG que aloca e determina a ordem das tarefas em cada processador, descrito na Seção 5.1.2), algoritmo genético AG-2 (AG que apenas distribui as tarefas entre os processadores e utiliza uma política de escalonamento para definir a ordem das tarefas, também descrito na Seção 5.1.2) e *Simulated Annealing* SA (Seção 5.1.2), que também utiliza uma política de escalonamento. A coluna "BEST" apresenta o melhor escalonamento obtido em todas as execuções e "AVG" mostra a média do melhor escalonamento.

Tabela 5.10: Análise dos resultados obtidos no modo de aprendizagem do EACS-H e por heurísticas e meta-heurísticas.

G_P	V_s	**DHL-FET**	**EACS-H**		**AG-1**		**AG-2**		**SA**	
			BEST	AVG	BEST	AVG	BEST	AVG	BEST	AVG
g18	2	46	46	46,00	46	46,00	46	46,00	46	46,00
	3	39	36	36,55	36	36,00	36	36,00	36	36,10
	4	27	26	26,05	26	26,00	26	26,00	26	26,00
	8	24	24	24,00	24	24,00	24	24,00	24	24,00
g40	2	81	80	80,00	80	80,00	80	80,00	82	86,90
	3	57	57	57,00	57	57,15	57	57,00	66	67,85
	4	46	45	45,55	46	46,45	46	46,60	55	57,75
	8	32	32	32,00	33	33,75	32	33,00	40	41,80
gauss18	2	54	44	44,00	44	44,95	44	44,00	44	46,30
	3	52	44	44,05	44	45,65	44	44,30	44	46,15
	4	52	44	44,05	44	45,65	44	45,45	46	47,35
	8	52	46	51,40	44	48,30	44	49,10	46	50,55
r.30	2	1311	1222	1224,45	1222	1222,05	1222	1222,20	1222	1244,60
	3	946	853	892,90	821	860,90	823	844,35	970	1027,05
	4	825	828	849,90	753	785,35	751	771,25	853	914,70
	8	778	804	828,75	753	776,75	751	768,60	753	813,95
r.40	2	1001	983	984,80	983	983,15	983	983,00	997	1046,65
	3	733	694	708,45	685	699,05	680	686,75	794	861,35
	4	620	607	631,85	561	585,55	555	568,50	684	759,90
	8	478	551	568,85	471	517,25	443	475,05	578	626,70
r.50	2	724	628	642,40	624	626,80	624	624,00	664	709,00
	3	636	532	539,40	504	532,60	496	520,20	624	680,20
	4	572	524	583,80	508	528,80	496	517,00	600	671,80
	8	556	636	665,00	532	553,80	512	534,60	600	638,00

Na Tabela 5.10, todos os algoritmos apresentaram escalonamentos idênticos aos valores de referência para o *g18*, com exceção de DHLFET para $V_s = 3$. Para o grafo de programa *g40*, SA exibiu o pior desempenho enquanto EACS-H apresentou os melhores escalonamentos considerando-se $V_s = \{4, 8\}$. Para o *gauss18*, a heurística DHLFET retornou o pior desempenho, AG-1 e AG-2 apresentaram os melhores escalonamentos, superando EACS-H apenas para $V_s = 8$. Entretanto, para $V_s = \{2, 3, 4\}$, o modelo EACS-H retornou

as melhores médias. Para os grafos de programa gerados aleatoriamente (*r.30*, *r.40* e *r.50* são abreviações para designar os grafos de programa *random30*, *random40* e *random50*, respectivamente), percebeu-se que o modelo EACS-H teve maior dificuldade em acompanhar os resultados dos algoritmos genéticos, especialmente para $V_s = 8$, onde os resultados são até inferiores aos da heurística de inicialização (DHLFET). Entretanto, mesmo com essa degradação de desempenho com o aumento do número de processadores, esse resultado é bastante superior ao obtido pelos modelos anteriores baseados em AC, conforme apresentado na Tabela 5.8. Comparando-se ao SA, vê-se que o modelo EACS-H se saiu melhor, exceto para o caso $V_s = 8$ nos grafos *random30* e *random50*. De forma geral, os resultados encontrados pelo EACS-H na fase de aprendizagem até 4 processadores podem ser comparados aos dos algoritmos genéticos nestes experimentos e são superiores ao SA. Apenas nos casos de 8 processadores, o EACS-H degrada o desempenho e se assemelha ao SA.

Para analisar o reuso no modelo EACS-H, foram utilizados os grafos de programa da Tabela 5.10 e um grafo de sistema com 2 processadores ($V_s = 2$). No modo de reuso, o AC foi equipado com regras extraídas para o *g18*, *g40* e *gauss18* na fase de aprendizagem que foram reaplicadas no escalonamento dos demais grafos (*r.30*, *r.40* e *r.50* são abreviações para designar os grafos de programa *random30*, *random40* e *random50*, respectivamente). A Tabela 5.11 mostra o melhor escalonamento obtido pelo EACS-H nos modos de aprendizagem e reuso, e também os resultados obtidos por DHLFET, AG-1, AG-2 e SA. Os melhores valores, que foram tomados como referência, são apresentados pelos AGs. De fato, os resultados encontrados por EACS-H no modo de aprendizagem confirmam a proximidade com os resultados dos AGs, conforme já apresentado na Tabela 5.10. Sobre os resultados no modo de execução, eles podem ser considerados adequados de modo geral, pois eles encontram tempos de escalonamento próximos aos AGs para muitos grafos de programa e em quase todos os casos conseguiram melhorar o tempo obtido pela heurística DHLFET, que equivale aos tempos obtidos nos reticulados iniciais, ou seja, o ponto de partida para o escalonamento do AC. A única exceção é o *random40*, onde o DHLFET consegue um resultado de partida mais próximo ao melhor obtido, e o EACS-H no modo reuso (g40) não consegue melhorar esse valor. Além disso, é possível notar uma melhoria significativa nessa abordagem em relação às utilizadas anteriormente na literatura, conforme o resultado de reuso na reprodução do modelo em [53] apresentado na Tabela 5.7 e na última linha da Tabela 5.11. Contudo, para um melhor entendimento sobre o reuso no escalonamento baseado em AC, experimentos adicionais foram realizados.

O esforço computacional necessário na descoberta das regras do AC é justificado apenas se estas regras puderem ser reutilizadas em novos problemas. Em [57] foi apresentado um experimento relacionado ao reuso de conhecimento no escalonamento baseado em AC. O conhecimento extraído no grafo de programa *gauss18* e de 5 variações do mesmo durante o modo de aprendizagem foi reusado em 10 outras variações distintas deste grafo

Tabela 5.11: Análise dos resultados obtidos no modo de aprendizagem (Ap.) e reuso (Re.) do EACS-H com 2 processadores ($V_s = 2$).

Abordagens	g18	g40	gauss18	r.30	r.40	r.50
DHLFET	46	81	54	1311	1001	724
AG-1	46	80	44	1222	983	624
AG-2	46	80	44	1222	983	624
SA	46	82	45	1222	996	668
EACS-H Ap.	46	80	44	1222	983	628
EACS-H Re. (g18)	-	80	47	1225	983	652
EACS-H Re. (g40)	46	-	46	1236	1001	648
EACS-H Re. (gauss18)	46	80	-	1233	1000	656
Reprod. Seq Re. (gauss18)	49,49	86,25	-	1336,61	1136,55	730,70

de programa. A Tabela 5.12 mostra 15 variações do *gauss18*. A notação utilizada na Seção 2.2 foi utilizada para descrever estas variações. Por exemplo, na linha *gauss18-1*, tem-se a composição desse grafo de programa (V1, E1) gerado a partir de modificações no *gauss18* (V,E) tais como: a troca do custo computacional da tarefa 6 de 6 unidades de tempo para 10, a exclusão da restrição de precedência entre as tarefas 11 e 13, a troca do custo de comunicação entre as tarefas 11 e 14 de 12 unidades de tempo para 4 e, finalmente, a adição de uma nova restrição de precedência entre as tarefas 5 e 13 com custo de comunicação igual a 12.

O objetivo do experimento descrito em [57] foi avaliar a capacidade de generalização das regras encontradas por um escalonador baseado em AC que utiliza uma abordagem de evolução conjunta (uso simultâneo de dois ou mais grafos de programa para o modo de aprendizagem) para o problema. Além disso, a vizinhança usada pelos autores foi não linear (vizinhança selecionada) e o modo de atualização das células foi o sequencial. Nos experimentos apresentados em [57], foram utilizados 6 grafos de programa para o modo de aprendizagem (*gauss18*, *gauss18-1*, *gauss18-2*, *gauss18-3*, *gauss18-4* e *gauss18-5*). Assim, o processo de aprendizagem é similar ao modelo original proposto em [49], exceto pela forma de avaliação, pois uma dada regra é avaliada a partir da evolução temporal de 25 configurações iniciais e a consequente aplicação da política de escalonamento sobre a alocação final obtida em cada CI sobre cada um dos 6 grafos de programa considerados, sendo a aptidão da regra igual a média de todos os escalonamentos encontrados ($T/25$).

A Tabela 5.13 apresenta um esquema comparativo entre o novo modelo EACS-H, a heurística DHLFET, as metaheurísticas AG e SA e trabalhos relacionados [53, 57]. A coluna "Joint" representa o resultado publicado em [57]. Os valores de *gauss18_1* a *gauss18_5*, marcados com *, se referem a valores obtidos na fase de aprendizagem, enquanto os valores de *gauss18_6* a *gauss18_15* foram usados na fase de reuso. "Reprod. Seq" representa os resultados obtidos com a reprodução do modelo descrito em [53] com modo sequencial de atualização das células e vizinhança linear. AG-1, SA, DHLFET são os mesmos algoritmos aplicados nos testes anteriores. Note que na fase de aprendizagem,

Tabela 5.12: Variações do grafo de programa *gauss18*.

G_P	**Descrições**
gauss18-1	V1 = V - {w(6)} + {(w(6) = 10)}; E1 = E - {e(11, 13), e(11, 14)} + {e(11, 14), (c(11, 14) = 4), e(5, 13), (c(5, 13) = 12)}
gauss18-2	V2 = V - {w(6)} + {(w(6) = 10)}; E2 = E - {e(3,8), e(11,13)} + {e(3,6), (c(3,6) = 8), e(5, 13), (c(5, 13) = 12)}
gauss18-3	V3 = V; E3 = E - {e(3,8), e(11,15)} + {e(3,6), (c(3,6) = 8), e(10,13), (c(10,13) = 12)}
gauss18-4	V4 = V; E4 = E - {e(3,8), e(11,15), e(5,10)} + {e(3,6), (c(3,6) = 8), e(10,13), (c(10,13) = 12), e(5,10), (c(5,10) = 15)}
gauss18-5	V5 = V - {w(9) } + {(w(9) = 6)}; E5 = E - {e(3,8), e(11,15), e(5,10)} + {e(3,6), (c(3,6) = 8), e(10,13), (c(10,13) = 12), e(5,10), (c(5,10) = 15)}
gauss18-6	V6 = V - {w(6) } + {(w(6) = 10)}; E6 = E
gauss18-7	V7 = V; E7 = E - {e(11,14)} + {e(11,14), (c(11,14) = 4)}
gauss18-8	V8 = V; E8 = E - {e(11,13)} + {e(5, 13), (c(5, 13) = 12)$
gauss18-9	V9 = V; E9 = E - {e(3,8)} + {e(3,6), (c(3,6) = 8)}
gauss18-10	V10 = V; E10 = E - {e(11,15)}
gauss18-11	V11 = V; E11 = E + {e(10,13), (c(10,13) = 12)}
gauss18-12	V12 = V - {w(6)} + {(w(6) = 10)}; E12 = E - {c(11, 14)} + {(c(11, 14) = 4)}
gauss18-13	V13 = V; E13 = E - {e(11, 13), e(11, 14)} + {e(11, 14), (c(11, 14) = 4), e(5, 13), (c(5, 13) = 12)}
gauss18-14	V14 = V; E14 = E - {e(3,8), e(11,13)} + {e(3,6), (c(3,6) = 8), e(5, 13), (c(5, 13) = 12)}
gauss18-15	V15 = V; E15 = E - {e(3,8), e(11,15)} + {e(3,6), (c(3,6) = 8)}

os modelos EACS-H e "Reprod. Seq" escalonam apenas o grafo de programa *gauss18*, enquanto que os resultados apresentados em [57] (coluna "Joint") foram obtidos usando 6 grafos de programa. No modo de execução é realizado o escalonamento das variações do *gauss18*, sem que haja necessidade de um novo processo evolucionário de aprendizagem, ao contrário das metaheurísticas.

Os resultados na Tabela 5.13 destacam o bom desempenho do EACS-H no modo de execução. Os dois modelos de AG e o SA apresentaram as melhores médias, mas a principal vantagem do modelo EACS-H sobre eles é que o seu processo evolucionário ocorreu somente uma vez para o *gauss18* e o conhecimento extraído foi utilizado para escalonar todos os outros grafos de programa apresentados na tabela. Assim, o custo computacional foi bastante reduzido. Por exemplo, enquanto o AG-1 gasta em média $7,04$ seg. para o escalonamento de cada grafo da Tabela 5.13, o conjunto de regras em EACS-H gasta em média apenas $0,42$ seg. para escalonar os 15 grafos de programa. A heurística DHLFET apresentou o pior desempenho. Tal fato comprova que as regras obtidas por EACS-H são capazes de melhorar o escalonamento a partir do reticulado inicial gerado por DHLFET. "Reprod. Seq" apresentou pior média entre os modelos baseados em AC e também apresentou resultados inferiores à EACS-H na maioria dos grafos de programa no qual a regra evoluída para o *gauss18* foi reusada. Na análise da média geral obtida por

Tabela 5.13: Análise do reuso do EACS-H para variações do *gauss18*.

G_P	AG-1	SA	DHLFET	Reprod. Seq	Joint [57]	EACS-H
gauss18-1	47	47	56	48	47*	48
gauss18-2	47	47	55	52	47*	49
gauss18-3	46	46	56	55	47*	47
gauss18-4	47	47	49	55	47*	49
gauss18-5	50	50	60	55	50*	50
gauss18-6	47	47	54	48	47	48
gauss18-7	44	44	54	44	47	44
gauss18-8	44	46	54	47	47	46
gauss18-9	46	46	52	48	47	47
gauss18-10	44	45	54	44	47	44
gauss18-11	46	46	49	51	47	47
gauss18-12	47	47	58	48	47	47
gauss18-13	44	45	54	47	47	46
gauss18-14	46	46	53	48	47	47
gauss18-15	46	46	52	48	47	47
Média	46,07	46,33	54,00	49,20	47,20	47,07

Joint [57] e os resultados obtidos no reuso do EACS-H, o EACS-H possui um desempenho melhor (47,20 e 47,07), embora a magnitude dessa diferença seja pequena. Entretanto, os resultados dos grafos *gauss18-1* a *gauss8-5* na coluna "Joint" não se referem a reuso, mas aos resultados obtidos na fase de aprendizagem. Se for calculada a média apenas para os casos *gauss18-6* a *gauss18-15*, onde foi feito reuso das regras nos dois métodos, a vantagem do EACS-H é mais significativa: 46,30 contra 47,00 obtidos em Joint [57]. Além disso, uma das vantagens do modelo EACS-H em relação a Joint [57] está no fato de que o custo computacional de "Joint" na fase de aprendizagem é muito mais elevado, uma vez que um número maior de grafos de programa é utilizado. Além disso, o atributo utilizado para definição da vizinhança no modelo em [57] é dinâmico e precisa ser recalculado a cada célula atualizada, o que aumenta ainda mais o custo computacional. Outras vantagens do modelo EACS-H estão ainda relacionadas à vizinhança adotada que é linear, mais simples que a vizinhança selecionada utilizada em [57], e o modo de atualização das células que é síncrono, ao contrário de "Joint" que utiliza modo de atualização sequencial [57]. Portanto, considerando uma execução em *hardware* paralelo, um ambiente implementando baseado no modelo EACS-H, seria potencialmente muito mais rápido que um modelo baseado em [57].

5.3.4 Considerações Finais

O grande objetivo da abordagem proposta nesta seção foi desenvolver um escalonador que além de explorar o paralelismo intrínseco nos ACs fosse capaz de diminuir a complexidade computacional, considerando o aumento do número de processadores, e obtivesse

desempenho satisfatório no reuso do conhecimento extraído. A abordagem construída foi denominada EACS-H, por ser baseada no modelo anterior EACS e por empregar uma heurística de construção para obter o reticulado inicial.

De fato, trabalhos anteriores avaliaram a capacidade de uma regra de transição levar qualquer configuração inicial para uma configuração que represente o escalonamento ótimo. Contudo, a busca por essa "independência" em relação ao reticulado inicial torna a busca pelas regras mais complexa, dificultando a convergência do AG. Entendemos que a generalização mais relevante não está relacionada à configuração inicial do reticulado, mas à capacidade de uma regra escalonar outras instâncias no modo de execução. Diante disso, o novo modelo EACS-H apresenta um novo conceito no escalonamento baseado em AC, em relação aos modelos anteriores [9, 10, 46–49, 51–53, 56, 57]: a avaliação da habilidade de uma regra em realizar um escalonamento eficiente a partir de uma única configuração inicial. Essa nova abordagem se mostrou mais eficiente tanto em termos de tempo de processamento quanto nos resultados obtidos no reuso.

Vários experimentos foram realizados visando avaliar o desempenho do modelo EACS-H. Foram utilizados grafos de programa da literatura e também gerados aleatoriamente. Os experimentos foram separados em três grupos. No primeiro, EACS-H foi analisado em um esquema comparativo com trabalhos anteriores, que também empregaram escalonadores baseados em ACs, e apresentou vantagens importantes em termos de resultados, desempenho computacional e escalabilidade em relação ao aumento do número de processadores. No segundo grupo de experimentos, os modos de aprendizagem e execução do EACS-H foram avaliados em relação a outras técnicas computacionais e os resultados também foram bons, especialmente no modo de aprendizagem que apresentou valores muito próximos aos do AG, tomado por referência. No terceiro grupo, o desempenho do EACS-H no reuso para várias instâncias distintas obteve resultados muito bons em relação à trabalhos relacionados e outras técnicas computacionais, se mostrando competitivo diante de técnicas com maior custo computacional.

Os resultados e análises relacionados ao modelo EACS-H, levou à submissão de um artigo em janeiro de 2012 [11].

5.4 Novas Estruturas para o Escalonamento Baseado em AC: Vizinhanças Pseudo-lineares e Avaliação com Penalização de Dinâmica

5.4.1 Introdução

Como meio de aperfeiçoar a estrutura do escalonamento baseado em ACs foram investigados novos tipos de vizinhança. O objetivo destas vizinhanças é mesclar boas ca-

racterísticas das vizinhanças linear e não linear. Assim, desenvolveu-se as vizinhanças que denominamos pseudo-lineares, que levam em consideração o tamanho do raio e o número de estados possíveis (processadores) do AC, como no caso da linear. Entretanto, diferentemente da vizinhança linear, as células da vizinhança são definidas, como nas vizinhanças não lineares, considerando as relações entre as tarefas no grafo de programa e não a posição no reticulado. Uma grande preocupação que houve na construção desses novos modelos de vizinhança é a possibilidade de mudança da arquitetura multiprocessada, não havendo limitações sobre o número de processadores.

Inicialmente, as investigações realizadas apontaram que o modelo de vizinhança linear, apresentado originalmente em [51], não possui uma capacidade propriamente adequada ao escalonamento. Uma importante conclusão desse estudo inicial é que nos modelos anteriores uma regra deve ser capaz de levar qualquer configuração aleatória das tarefas de um grafo de programa para uma configuração que represente a alocação ótima para esse grafo, uma característica imposta pela forma de avaliação. Assim, é desejado que em uma regra "perfeita" todas essas configurações possíveis possam ser evoluídas para a mesma configuração final desde que tal configuração garanta o escalonamento ótimo. A princípio isso não é um problema. Contudo, a grande falha dessa abordagem envolve a capacidade de reuso da regra. As regras evoluídas dessa forma possuem uma tendência natural a levar qualquer reticulado com o mesmo número de tarefas para a mesma configuração, independentemente do grafo de programa utilizado na execução ser ou não o mesmo. Isso ocorre porque o modelo linear não considera a relação entre as tarefas na definição da vizinhança. Isso pode ser melhor explicado através de um experimento que foi realizado utilizando quatro grafos de programa distintos cuja única semelhança está relacionada ao número de tarefas (todos apresentam 18 tarefas), e o modelo de escalonador baseado em vizinhança linear apresentado em [53].

O modelo do escalonador foi reproduzido a partir de [53], utilizando vizinhança linear e modo sequencial de atualização. Inicialmente, realizou-se a fase de aprendizagem para o grafo de programa *gauss18*. Posteriormente, na fase de operação normal realizou-se o reuso das regras obtidas nesse mesmo grafo de programa e também no *g18*, *rand18.1* e *rand18.2*, sendo estes dois últimos grafos gerados aleatoriamente com 18 tarefas. A Figura 5.7 mostra o reticulado final encontrado para a aplicação da melhor regra encontrada na fase de aprendizagem do *gauss18* sobre cada um dos grafos de programa. De fato, a alocação final encontrada para os quatro grafos de programa foi a mesma. Essa alocação representa a alocação ótima para o *gauss18*, contudo está muito distante de ser adequada para os demais grafos de programa.

A Figura 5.7 também apresenta o valor do tempo de escalonamento associado ao reticulado obtido (T_{AC}) e o valor de referência para o grafo: T_{otimo}, para os casos em que se conhece o valor ótimo e T_{AG}, para os casos em que o valor de referência foi obtido utilizando o algoritmo AG-1, descrito na Seção 5.1.2. Assim, é possível perceber que o

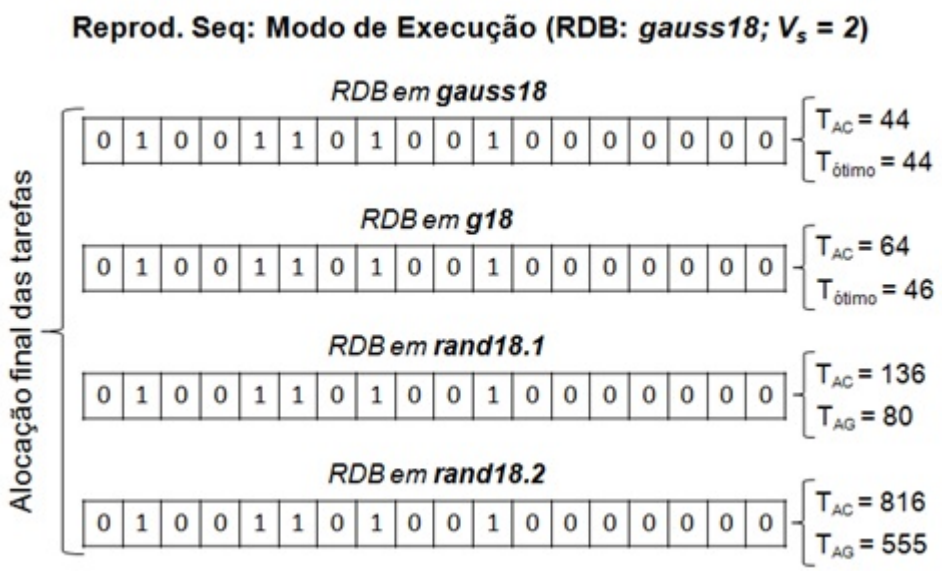


Figura 5.7: Reuso da regra evoluída no *gauss18* para os grafos de programa *gauss18*, *g18*, *rand18.1* e *rand18.2*, usando a reprodução do modelo sequencial de [53].

modo de operação onde é feito o reuso da regra é extremamente prejudicado pela avaliação baseada em um conjunto de configurações iniciais aleatórias, especialmente com o uso da vizinhança linear. Antes de prosseguir, cabe ressaltar que no modelo EACS-H descrito na seção anterior esse problema é atenuado, mesmo com o uso da vizinhança linear. O modelo EACS-H também emprega uma vizinhança baseada apenas na proximidade das células no reticulado, o que não necessariamente reflete uma relação de proximidade entre as tarefas no grafo. Entretanto, o processo de busca do AG não força a obtenção de regras insensíveis à condição inicial, uma vez que a regra é avaliada apenas no escalonamento de um único reticulado inicial, obtido pela aplicação da heurística de construção. Além disso, essa condição inicial varia de uma instância para outra, uma vez que o resultado da heurística de construção reflete as relações de cada grafo de programa. Para exemplificar, a Figura 5.8 apresenta os reticulados finais obtidos utilizando-se o modelo EACS-H, para os mesmos casos de reuso apresentados na Figura 5.7. É possível notar que a melhor regra encontrada na aprendizagem do *gauss18* é apta a levar os diferentes grafos de programa utilizados no reus, para diferentes alocações finais. Os resultados obtidos por essa regra também foram melhores que aqueles exibidos na figura anterior.

No caso das vizinhanças não lineares utilizadas nos modelos anteriores em [51–53], observou-se que este problema é contornado pelas características da vizinhança, que consideram as relações existentes entre as tarefas no grafo de programa.

Uma vez identificada a limitação do modelo linear de vizinhança, baseado no simples posicionamento das tarefas no reticulado (dado pela ordem da tarefa), partiu-se para a proposição de novos modelos de vizinhança que mantivessem a simplicidade e a escalabilidade do modelo linear. Esta seção apresenta os principais modelos de vizinhança e

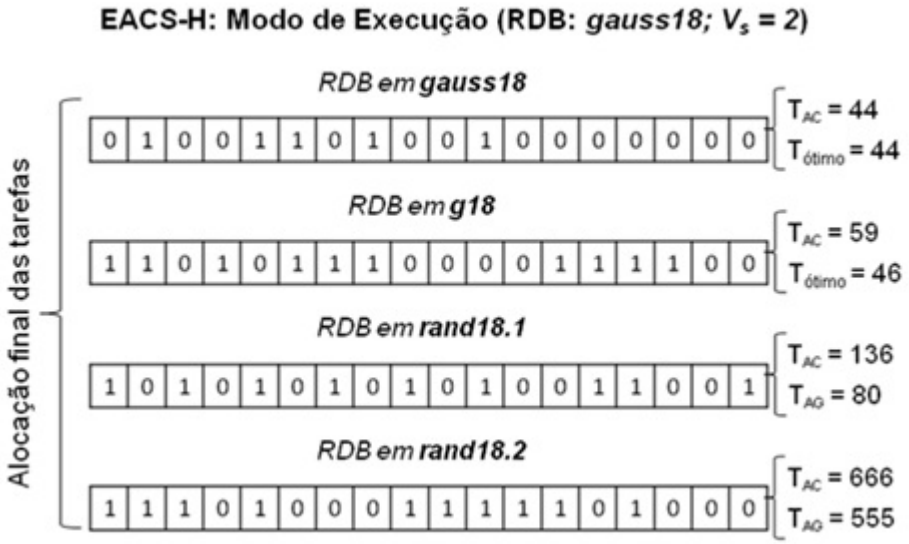


Figura 5.8: Reuso da regra evoluída no *gauss18* para os grafos de programa *gauss18*, *g18*, *rand18.1* e *rand18.2*, usando o modelo EACS-H.

outras estruturas desenvolvidas durante a pesquisa.

5.4.2 Novos Modelos de Vizinhança

Antes de apresentar os principais modelos de vizinhança desenvolvidos durante a pesquisa, são mostrados alguns conceitos utilizados em suas definições:

- conjuntos de vizinhança primário e secundário: essa denominação designa o agrupamento de tarefas que possuem relação direta (predecessoras ou sucessoras) ou indireta (irmãs) com uma determinada tarefa i, a qual se deseja determinar a vizinhança. Como conjunto primário, destaca-se o agrupamento de relação direta sendo por isso designado por $c1$, enquanto que o agrupamento de ligação indireta é considerado um conjunto secundário determinado por $c2$.
- atributos da vizinhança: as tarefas dentro de cada conjunto são ordenadas (de forma crescente ou decrescente) de acordo com atributos definidos a priori. É importante observar que como mais de um atributo pode ser utilizado, as tarefas dentro de um mesmo conjunto podem estar organizadas de diferentes maneiras. A referência à ordenação realizada pelo primeiro atributo é designada por $Atrib1[i]$, com $i = \{1, ..., z\}$, onde z é o número total de tarefas pertencentes àquele agrupamento. É importante observar também que a tarefa relacionada à $Atrib1[1]$, em caso de maximização, é a que apresenta maior valor do atributo, enquanto em $Atrib1[z]$ está a tarefa com menor valor do mesmo. A ordenação também pode ser realizada por um segundo atributo sendo representada por $Atrib2[i]$, para um terceiro $Atrib3[i]$, e assim por diante.

Diversos modelos de vizinhança foram investigados, sendo que dois deles retornaram um melhor desempenho e serão detalhados a seguir.

O primeiro modelo de vizinhança pseudo-linear desenvolvido foi denominado V_{pl}-$c1$. Ele se baseia em apenas um conjunto (primário) e no uso de dois atributos. Uma generalização de V_{pl}-$c1$ é apresentada a seguir: considere um AC com raio R e reticulado A composto por N células (tarefas), onde o estado A_i representa o processador em que a tarefa i está alocada. A vizinhança da célula i, utilizando V_{pl}-$c1$, será dada conforme apresentado na Figura 5.9. O esquema apresentado na figura mostra que da célula central (A_i) para a esquerda, têm-se os estados das tarefas do conjunto $c1$ ordenadas por um atributo $Atrib1$ e da célula central para a direita, têm-se as tarefas do mesmo conjunto, ordenadas por um $Atrib2$. Conforme a figura, o número de tarefas consideradas em cada conjunto é definido por R, ou seja, se $R = 2$ serão consideradas, de acordo com a ordenação dos conjuntos, duas tarefas à direita e à esquerda.

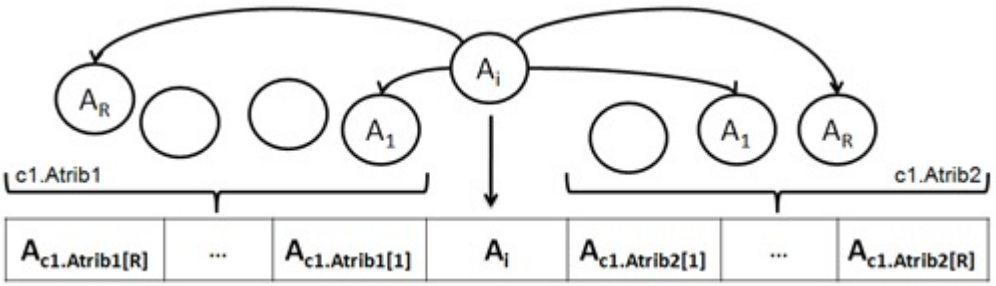


Figura 5.9: Modelo de vizinhança V_{pl}-$c1$.

Em V_{pl}-$c1$ há apenas um caso especial, quando $z < R$. Nessa condição, a formação da vizinhança continua a partir da primeira tarefa ($Atrib1[1]$). Tal procedimento é realizado até que se tenham R tarefas vizinhas.

O segundo modelo de vizinhança pseudo-linear foi denominado V_{pl}-$c2$. Ele pode ser caracterizado por utilizar dois conjuntos (primário e secundário) e apenas um atributo. Uma generalização de V_{pl}-$c2$ também é apresentada. Considerando as mesmas condições descritas para V_{pl}-$c1$, a vizinhança da célula i, utilizando V_{pl}-$c2$, será dada conforme apresentado na Figura 5.10. De acordo com a figura, da célula central (A_i) para a esquerda, têm-se os estados das tarefas do conjunto $c1$ ordenadas por um atributo $Atrib1$ e da célula central para a direita, têm-se as tarefas do outro conjunto ($c2$) ordenadas pelo mesmo $Atrib1$. Assim como em V_{pl}-$c1$, o número de tarefas consideradas em cada conjunto é definido por R.

Em V_{pl}-$c2$ há dois casos especiais, quando $z < R$ e quando $c2$ é um conjunto vazio, ou seja, a tarefa associada à vizinhança não possui irmãs. O primeiro caso é resolvido de forma idêntica à V_{pl}-$c1$. Já no outro caso, quando o conjunto secundário de uma tarefa não possui qualquer elemento, a vizinhança é dada pelo conjunto primário $c1$ da tarefa.

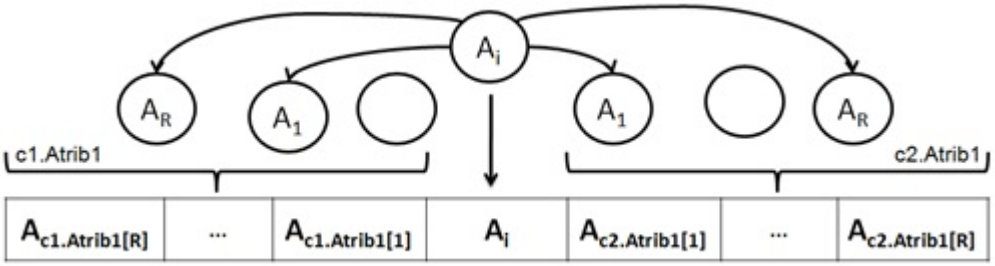


Figura 5.10: Modelo de vizinhança V_{pl}-$c2$.

Neste trabalho, os atributos considerados em V_{pl}-$c1$ foram: nível inferior (b-*level*) e nível superior (t-*level*). Já em V_{pl}-$c2$, que utiliza apenas um atributo, considerou-se o nível inferior. Para os dois métodos, as tarefas foram ordenadas em função da maximização dos atributos. Contudo, para uma condição especial de empate no valor dos atributos, nos conjuntos ordenados pelo valor de b-*level* escolhe-se a tarefa que apresenta maior número de ordem enquanto que em conjuntos ordenados pelo valor de t-*level* faz-se o oposto. Os Algoritmos 1 e 2, exibidos na Seção 2.4, apresentam, respectivamente, os principais passos para cálculo do b-*level* e do t-*level* de uma tarefa.

5.4.3 Avaliação Composta: Desempenho e Dinâmica

No estudo das novas vizinhanças, foi possível observar que o comportamento dinâmico exibido pelas regras durante a sua evolução temporal exerce um papel fundamental na avaliação das mesmas. Este fato foi constatado a partir de uma única condição inicial. O escalonador possui duas fases: o modo de aprendizagem, caracterizado pela busca de regras aptas a encontrarem bons escalonamentos para o grafo de programa considerado e o modo de execução, caracterizado pelo reuso de regras em outros grafos de programa. Do ponto de vista do comportamento dinâmico, regras com dinâmicas mais estáveis, como ponto-fixo heterogêneas ou periódicas de ciclo pequeno, são desejáveis na etapa de aprendizagem, uma vez que regras caóticas estariam realizando um escalonamento aleatório, enquanto regras nulas (ponto-fixo homogêneas) não estariam distribuindo tarefas mas concentrando-as em um único processador. Foi possível observar nos modelos anteriores, que adotar um conjunto de CIs no modo de aprendizagem, ao invés de um único ponto de partida, estimula o escalonador na busca por regras com tais dinâmicas, uma vez que dificilmente uma regra caótica ou nula retornaria valores baixos de avaliação, partindo-se de diferentes condições iniciais. Contudo, conforme já foi apresentado anteriormente, encontrar um conjunto de regras capazes de levar um conjunto de CI para uma configuração ótima é uma busca complexa e que não gerou o devido retorno em termos de reuso, como pôde ser visto em experimentos da Seção 5.3. Por outro lado, observou-se que, ao utilizar uma única condição inicial na avaliação, o risco de convergência para regras de dinâmica caótica é considerável.

Diante disso, foi proposta uma nova forma de avaliação no modelo de escalonador com o intuito de auxiliar o AG na busca por regras com dinâmicas ponto-fixo ou periódicas (com ciclo pequeno) [62]. Basicamente, essa avaliação composta consiste em realizar um novo cálculo (denominado avaliação penalizada) sobre o tempo de escalonamento encontrado pela regra, onde um acréscimo a esse tempo é dado ou não de acordo com a dinâmica apresentada pela regra, e o resultado final desse cálculo é o valor de aptidão da regra.

O Algoritmo 4 apresenta um esboço sobre como é realizada a penalização da regra. A taxa de penalização Tp_{rule} determina o grau de incremento na aptidão final da regra, baseado no comportamento dinâmico da mesma. Uma taxa muito alta pode limitar demasiadamente a busca do AG fazendo-o convergir para regras com dinâmicas desejadas, porém com desempenhos distantes dos satisfatórios. Por outro lado, um valor extremamente pequeno pode não ter qualquer efeito sobre a busca e permitir a convergência do AG para regras com ciclos grandes ou caóticas.

Ainda sobre o Algoritmo 4, foram criadas seis classes para diferentes dinâmicas de regras sendo que para cada uma delas é associado um peso distinto: classe 1 com peso nulo, para regras ponto-fixo; classe 2 com peso 1, para regras periódicas com ciclo igual a 2; classe 3 com peso 2, para regras periódicas com ciclo igual 3; classe 4 com peso 3, para regras periódicas com ciclo igual a 4; classe 5 com peso 4, para regras com ciclo igual a 5; e classe 6 com peso 10, para todas as regras com ciclo maior que 5 ou caóticas. Desse modo, a fórmula para cálculo da taxa de penalização é dada por $Tp_{rule} = P_{inc} * pesoClasse * valorEscalonamento$, onde P_{inc} é o percentual de incremento, $pesoClasse$ é o peso associado à classe da regra e $valorEscalonamento$ é o tempo de escalonamento obtido pela própria regra. Foi determinado experimentalmente valores adequados entre $0,1$ e $0,8\%$ para P_{inc}.

Algoritmo 4 Esboço do cálculo de penalização da regra

1: $P_{inc} \Leftarrow X\%$
2: $maxCiclo \Leftarrow 6$
3: $pesoClasse[maxCiclo] = \{0, 1, 2, 3, 4, 10\}$
4: **if** $(tamanhoCicloRegra >= maxCiclo)$ **then**
5: $\quad valorPonderado = P_{inc} * pesoCiclo[maxCiclo] * valorEscalonamento$
6: **else**
7: $\quad valorPonderado = P_{inc} * pesoCiclo[tamanhoCicloRegra] * valorEscalonamento$
8: **end if**
9: $aptidaoRegra = valorEscalonamento + valorPonderado$

Ao incorporar a penalização da regra no modelo EACS-H, acrescenta-se uma nova etapa ao modo de avaliação na fase de aprendizagem (Seção 5.3.2): (iv) verificação e penalização sobre a dinâmica da regra, a partir do reticulado, por no máximo $nCiclo$ passos de tempo. Nos experimentos realizados foi considerado $nCiclo = 5$ como referência às classes criadas. Assim incentiva-se a busca do AG por regras com dinâmicas estáveis, tais como ponto-fixo ou periódicas com ciclos pequenos ($<= 5$).

Nas Tabelas 5.14, 5.15 e 5.16 são exibidos alguns resultados experimentais referentes à penalização da regra. Nos experimentos utilizou-se V_{pl}-$c2$ como modelo de vizinhança para EACS-H, sendo que V_{pl}-$c2$-**P** representa a coluna onde a penalização da regra foi aplicada. Os parâmetros utilizados foram número de processadores $V_s \in \{3, 4, 8\}$. O tamanho da vizinhança usada nos experimentos foi $R \in \{1, 2\}$ e o número de passos de evolução do AC foi $S = 3 * N$, onde N representa o número de tarefas do grafo de programa. Os grafos de programa utilizados foram: *gauss18*, *random30*, *random40* e *random50*. Devido aos resultados pouco conclusivos em experimentos anteriores, não foram considerados neste experimento os grafos *g18* e *g40* (foram considerados fáceis de serem resolvidos). Novamente, foram realizadas 20 execuções.

Tabela 5.14: Análise do modo de aprendizagem no EACS-H utilizando a penalização da regra.

G_P	V_s	**EACS-H com V_{pl}-$c2$**		**EACS-H com V_{pl}-$c2$-P**	
		BEST	CLASSE	BEST	CLASSE
gauss18	3	44	ponto-fixo	44	ponto-fixo
	4	44	ponto-fixo	44	ponto-fixo
	8	44	ponto-fixo	44	ponto-fixo
random30	3	876	Ciclo 4	840	ponto-fixo
	4	778	Caótica	778	Ciclo 3
	8	778	Caótica	778	ponto-fixo
random40	3	681	Caótica	679	Ciclo 3
	4	565	Ciclo 4	567	Ciclo 3
	8	443	Caótica	443	Ciclo 3
random50	3	516	Ciclo 4	512	Ciclo 3
	4	508	Ciclo Duplo	516	Ciclo Duplo
	8	524	Caótica	512	Ciclo 3

A Tabela 5.14 faz referência aos resultados encontrados no modo de aprendizagem para cada um dos grafos de programa. É possível perceber uma distinção entre a classificação dinâmica das regras ("CLASSE") entre V_{pl}-$c2$ e V_{pl}-$c2$-**P**, especialmente porque alguns dos resultados de V_{pl}-$c2$ foram obtidos por regras de dinâmica periódica com ciclos longos ou até mesmo com comportamento caótico (classe 6). Contudo, os desempenhos dos modelos em geral ficaram próximos, com uma pequena vantagem para V_{pl}-$c2$-**P**. A importância do comportamento dinâmico das regras pode ser vista nas Tabelas 5.15 e 5.16, onde são exibidos os resultados para o modo de execução, considerando as regras obtidas, respectivamente, para o *gauss18* e para o *random30* no modo de aprendizagem. Nesse modo, V_{pl}-$c2$-**P** é superior a V_{pl}-$c2$ em todos os casos das tabelas, enfatizando assim a principal vantagem em incorporar a penalização da regra na arquitetura de EACS-H: o reuso em grafos de programa distintos durante o modo de execução do escalonador.

Tabela 5.15: Análise do modo de execução (*gauss18*) em EACS-H utilizando a penalização da regra.

G_P	V_s	**EACS-H com** V_{pl}-$c2$	**EACS-H com** V_{pl}-$c2$**-P**
random30	3	963	927
	4	862	830
	8	954	879
random40	3	767	743
	4	711	698
	8	636	579
random50	3	620	588
	4	624	616
	8	696	656

Tabela 5.16: Análise do modo de execução (*random30*) em EACS-H utilizando a penalização da regra.

G_P	V_s	**EACS-H com** V_{pl}-$c2$	**EACS-H com** V_{pl}-$c2$**-P**
gauss18	3	45	44
	4	44	44
	8	44	44
random40	3	767	744
	4	657	638
	8	623	540
random50	3	612	600
	4	636	548
	8	724	656

5.4.4 Escalonador Baseado em AC Síncrono com Inicialização por Heurística de Construção e Modelo de Vizinhança Pseudo-linear

Além dos dois novos modelos de vizinhança V_{pl}-$c1$ e V_{pl}-$c2$, diversos outros modelos foram avaliados. Todavia, foi identificado que esses modelos foram aqueles com melhor desempenho neste trabalho, especialmente quando utilizados com a estratégia de penalização. O Apêndice B apresenta os resultados de experimentos e análises efetuados com o objetivo de escolher a melhor opção, dentre as duas vizinhanças, a serem incorporadas no modelo EACS-H apresentado na seção anterior. De forma geral, a vizinhança, V_{pl}-$c2$ foi escolhida como melhor modelo. Assim, foi elaborado um novo modelo de escalonador, denominado EACS-HV. Além de utilizar a estratégia de inicialização dos reticulados e avaliação das regras baseada em uma heurística de construção, o modelo EACS-HV também incorporou o novo modelo de vizinhança pseudo-linear V_{pl}-$c2$ e a estratégia de penalização da regra com dinâmica indesejável, durante a fase de aprendizagem.

EACS-HV e EACS-H

Uma comparação entre EACS-HV e o modelo anterior (EACS-H) é apresentada a seguir. A Tabela 5.17 apresenta os resultados de escalonamento obtidos na fase de aprendizagem com o modelo anterior (EACS-H) e o modelo EACS-HV. Na tabela, é exibido também o ponto de partida de ambos os modelos, gerado a partir do DHLFET. Ao analisar os resultados em relação à heurística, é possível perceber que EACS-HV sempre consegue superá-la, ou seja, ele sempre consegue melhorar o escalonamento inicial obtido pelo DHLFET, enquanto que EACS-H não consegue fazê-lo nos grafos de programa *random30*, *random40* e *random50* para $V_s = 8$. Comparando os dois modelos, é possível perceber uma proximidade entre os resultados obtidos para a aprendizagem do *gauss18*, embora o desempenho de EACS-HV tenha sido superior para $V_s = 8$. Os resultados também estão próximos considerando os experimentos para $V_s = 2$, entretanto, a diferença se torna maior quando analisa-se o desempenho apresentado pelos modelos diante do aumento do número de processadores. A vantagem de EACS-HV sobre EACS-H chega a ser muito significativa em alguns casos, tais como nos grafos de programa *random30* e *random40*, para $V_s = 8$. Nos demais experimentos, EACS-HV também obteve melhores resultados que EACS-H.

Tabela 5.17: Comparativo entre EACS-HV e EACS-H para o modo de aprendizagem dos grafos de programa *gauss18*, *random30*, *random40* e *random50*.

G_P	V_s	**DHLFET**	**EACS-H**		**EACS-HV**	
			BEST	AVG	BEST	AVG
gauss18	2	54	44	44,00	44	44,00
	3	52	44	44,05	44	44,00
	4	52	44	44,05	44	44,00
	8	52	46	51,40	44	44,00
random30	2	1311	1222	1224,45	1222	1227,13
	3	946	853	892,90	840	909,86
	4	825	828	849,90	778	818,71
	8	778	804	828,75	778	780,53
random40	2	1001	983	984,80	983	984,45
	3	733	694	708,45	679	692,40
	4	620	607	631,85	567	587,96
	8	478	551	568,85	443	474,44
random50	2	724	628	642,40	624	629,90
	3	636	532	539,40	512	540,86
	4	572	524	583,80	520	538,17
	8	556	636	665,00	512	549,19

De forma geral, EACS-HV obteve desempenho superior à EACS-H no modo de aprendizagem. Isso pode ser visto na Tabela 5.18 que apresenta os resultados dos testes de hipótese realizados entre EACS-HV (M_1) e EACS-H (M_2) considerando o modo de apren-

dizagem. Em apenas 1 dos 16 casos de teste houve evidências estatísticas significativas de que EACS-H foi melhor do que EACS-HV, enquanto que em outros 9 casos EACS-HV teve desempenho superior à EACS-H.

Tabela 5.18: Análise estatística sobre o desempenho dos modelos EACS-HV e EACS-H durante a fase de aprendizagem.

V_s	**gauss18**	**random30**	**random40**	**random50**
2	=	>	=	<
3	=	=	<	=
4	=	<	<	<
8	<	<	<	<

O modo de execução nos modelos EACS-HV e EACS-H também foi examinado. A Tabela 5.19 apresenta um comparativo entre as duas abordagens considerando as regras extraídas durante a fase de aprendizagem do grafo de programa *gauss18*. Na Tabela 5.17 é possível notar que os resultados encontrados pelos modelos para este grafo foram bastante próximos. De fato, os melhores resultados obtidos no reuso sobre os grafos de programa *random30* e *random40*, ora são do EACS-HV, ora do EACS-H. Contudo, ao analisar o reuso sobre o grafo *random50*, o desempenho do novo modelo é superior em todos os casos.

Tabela 5.19: Comparativo entre EACS-HV e EACS-H no modo de execução (*gauss18*) para os grafos de programa *random30*, *random40* e *random50*.

G_P *gauss18*	V_s	**EACS-H**		**EACS-HV**	
		BEST	AVG	BEST	AVG
random30	2	1231	1253,40	1229	1249,65
	3	973	1062,75	927	1006,35
	4	896	979,75	861	992,15
	8	872	916,40	879	959,20
random40	2	986	1005,25	990	994,55
	3	750	823,65	746	793,00
	4	681	724,65	751	794,70
	8	590	632,25	579	649,75
random50	2	656	679,80	640	666,00
	3	612	665,20	600	640,00
	4	652	671,60	616	651,00
	8	692	728,60	656	691,40

A fim de avaliar o desempenho geral dos modelos EACS-H e EACS-HV no reuso das regras extraídas para o *gauss18*, testes de hipótese foram realizados. A Tabela 5.20 apresenta os resultados obtidos por estes modelos. Nos testes, considerou-se EACS-HV como M_1 e EACS-H como M_2. De acordo com a tabela, há evidências estatísticas significativas de que o desempenho de EACS-HV foi melhor em 6 dos 12 casos de teste, enquanto que

EACS-H foi melhor em 3. Dessa forma, é possível concluir um desempenho superior por parte de EACS-HV.

Tabela 5.20: Análise estatística do desempenho de EACS-HV e EACS-H durante o modo de execução (*gauss18*).

V_s	**random30**	**random40**	**random50**
2	=	=	<
3	<	<	<
4	=	>	<
8	>	>	<

Na Tabela 5.19 é exibido outro comparativo entre os modelos EACS-HV e EACS-H, considerando, dessa vez, as regras extraídas durante a fase de aprendizagem do grafo de programa *random30*. EACS-HV encontrou melhor resultado que EACS-H em todos os experimentos da tabela, com exceção do reuso no grafo de programa *random40* para $V_s = 3$, onde EACS-H encontrou 744 e EACS-HV obteve 743. Em alguns casos a diferença entre os resultados obtidos pelos modelos foi considerada alta, como por exemplo no reuso para o grafo de programa *random50*, considerando $V_s = 4$, onde EACS-HV realizou o escalonamento com *makespan* igual a 548 e EACS-H obteve 660.

Tabela 5.21: Comparativo entre EACS-HV e EACS-H no modo de execução (*random30*) para os grafos de programa *gauss18*, *random40* e *random50*.

G_P *random30*	V_s	**EACS-H**		**EACS-HV**	
		BEST	AVG	BEST	AVG
gauss18	2	47	51,95	44	47,05
	3	52	60,30	44	47,10
	4	53	61,00	44	47,10
	8	60	71,20	44	45,90
random40	2	991	1006,70	987	996,85
	3	743	804,70	744	781,75
	4	668	707,25	638	748,25
	8	591	628,35	540	632,55
random50	2	648	673,60	636	661,00
	3	620	661,00	616	644,80
	4	660	680,60	548	641,80
	8	696	719,00	656	702,20

Análises estatísticas também foram realizadas tomando os valores obtidos por EACS-HV e EACS-H no reuso das regras extraídas durante o modo de aprendizagem do *random30*. A Tabela 5.22 apresenta os resultados obtidos com testes de hipótese desenvolvidos conforme a Seção 5.1.4. Ao avaliar os resultados da tabela, é possível concluir que o desempenho de EACS-HV foi superior ao de EACS-H, uma vez que em 9 de 12 casos, houveram evidências estatísticas significativas de que a população de EACS-HV é menor que

a de EACS-H, enquanto que em apenas um caso, pode-se afirmar o contrário (população de EACS-HV é maior que a de EACS-H).

Tabela 5.22: Análise estatística do desempenho de EACS-HV e EACS-H durante o modo de execução (*random30*).

V_s	**gauss18**	**random40**	**random50**
2	$<$	$<$	$<$
3	$<$	$=$	$<$
4	$<$	$>$	$<$
8	$<$	$=$	$<$

EACS-HV e outros escalonadores baseados em AC

Como meio de se realizar um comparativo geral entre os modelos de escalonadores baseados em AC desenvolvidos neste trabalho e também os reproduzidos, a Tabela 5.23 mostra o desempenho de todos eles considerando o modo de execução sobre os grafos de programa *gauss18*, *random30*, *random40* e *random50*. Em relação ao *gauss18*, é importante lembrar que ele é tido como um dos grafos mais difíceis da literatura e que trabalhos relacionados obtiveram grande dificuldade para lidar com ele, tal como exibido na Figura 5.6. Entretanto, observando na tabela o desempenho dos novos modelos, é possível perceber uma grande evolução em relação a esse grafo e mais que isso, o excelente desempenho de EACS-HV sobre ele, alcançando tempos de escalonamento ótimos para 2, 3, 4 e 8 processadores. Aliás, a complexidade gerada pelo acréscimo de processadores é outro problema encontrado em abordagens da literatura, contudo há uma grande evolução das novas abordagens nesse sentido, uma vez que EACS-H e, principalmente, EACS-HV têm trabalhado muito melhor diante dessa complexidade, tal como pode ser observado na tabela.

EACS-HV e outras técnicas computacionais

De modo geral, o desempenho de EACS-HV também foi analisado em relação à outras técnicas computacionais, conforme é exibido na Tabela 5.24. Na tabela são apresentados os resultados de algoritmos clássicos (ETF, MCP e DHLFET), metaheurísticas (SA, AG-1 e AG-2) e EACS-HV para a fase de aprendizagem (Ap) e execução (Re). Ao analisar o desempenho de EACS-HV (Ap) nota-se um modelo robusto em termos dos resultados obtidos, quando comparado com outras técnicas computacionais, uma vez que ele consegue superar até mesmo os AGs (valores de referência) em alguns casos. De fato, o modo de aprendizagem em EACS-HV apresenta um desempenho bastante satisfatório. Em EACS-HV (Re) considerou-se os resultados obtidos através do conhecimento extraído em grafos de programas distintos daquele considerado no modo de aprendizagem. Por

Tabela 5.23: Análise dos modelos de escalonadores baseados em AC apresentados neste trabalho.

G_P	V_s	Rep. Sinc	Rep. Seq	EACS	EACS-H	EACS-HV
gauss18	2	47,00	44,00	44,00	44	44
	3	52,00	52,00	-	44	44
	4	51,86	52,00	-	44	44
	8	52,36	52,36	-	46	44
random30	2	1251,36	1239,00	1226,95	1222	1222
	3	1020,95	963,00	-	853	854
	4	1019,98	911,19	-	828	778
	8	1100,87	1085,11	-	804	778
random40	2	1010,40	1006,00	997,27	983	983
	3	820,98	810,94	-	694	679
	4	716,13	681,00	-	607	567
	8	816,23	814,01	-	551	443
random50	2	669,48	661,18	662,90	628	624
	3	660,00	643,09	-	532	512
	4	633,76	620,00	-	524	520
	8	683,54	652,05	-	636	512

exemplo, considerando os resultados para o *random50*, eles foram obtidos através do reuso das regras extraídas para o *gauss18*, *random30* e *random40*. De forma geral, apesar da dificuldade desse modo em lidar com o aumento do número de processadores, conclui-se que o desempenho de EACS-HV (Re) tem se mostrado eficiente para grafos de programa com menor número de tarefas, tal como o *gauss18* e também diante de arquiteturas com poucos processadores.

Tabela 5.24: Análise geral de EACS-HV em relação à outras técnicas computacionais.

G_P	V_s	ETF	MCP	DHLFET	EACS-HV (Ap)	EACS-HV (Re)	SA	AG-1	AG-2
gauss18	2	52	52	54	44	44	44	44	44
	3	52	52	52	44	44	44	44	44
	4	52	52	52	44	44	46	44	44
	8	52	52	52	44	44	46	44	44
random30	2	1229	1271	1311	1222	1225	1222	1222	1222
	3	921	850	946	854	927	970	821	823
	4	826	776	825	778	861	853	753	751
	8	806	776	778	778	835	753	753	751
random40	2	983	996	1001	983	985	997	983	983
	3	699	689	733	679	737	794	685	680
	4	571	558	620	567	601	684	561	555
	8	471	443	478	443	527	578	471	443
random50	2	624	636	724	624	632	664	624	624
	3	568	572	636	512	600	624	504	496
	4	568	572	572	520	548	600	508	496
	8	568	572	556	512	628	600	532	512

5.4.5 Considerações Finais

De acordo com os resultados obtidos nesta seção, é importante destacar o bom desempenho das vizinhanças pseudo-lineares sobre o modelo linear comumente utilizado na literatura [53, 56]. Diante dos experimentos, V_{pl}-$c2$ se mostrou um modelo de vizinhança promissor para o escalonamento baseado em AC. Além disso, nesta seção foi apresentado outro modelo de vizinhança pseudo-linear denominado V_{pl}-$c1$. Uma característica interessante das vizinhanças pseudo-lineares é que elas mesclam qualidades dos modelos lineares e não lineares de modo a tentar contornar as falhas de cada um deles. Assim, elas são capazes de considerar as relações entre as tarefas ao invés de apenas a ordem das mesmas no reticulado, tal como acontece no modelo linear [53]. Por outro lado, elas não apresentam uma estrutura complexa e limitada a 2 processadores como os modelos de vizinhança não lineares utilizados na literatura [49, 57].

Outro ponto importante apresentado nesta seção foi uma falha no modelo de escalonador original relacionada ao modelo de avaliação das regras. Observou-se que apesar desse modelo funcionar como uma espécie de filtro para a escolha de regras com dinâmicas estáveis pelo AG, ele também é responsável por exibir uma tendência natural nas regras de sempre levar uma CI qualquer para uma mesma configuração. Por outro lado, o fato do modelo EACS-H considerar apenas uma configuração dificulta o seu processo de busca por regras estáveis, uma vez que não há um filtro para auxiliar o AG na busca por regras de dinâmica estável. Dessa forma, foi incorporada uma forma de avaliação à arquitetura de EACS-H, denominada penalização da regra, que foi apta a melhorar o desempenho de EACS-H no reuso.

Também foi mostrado nesta seção uma visão geral sobre as abordagens de escalonamento baseadas em AC desenvolvidas nesta dissertação, bem como a evolução que elas representam diante dos trabalhos anteriores relacionados ao assunto e o desempenho competitivo, e muitas vezes superior, que elas alcançaram quando comparadas às heurísticas construtivas e metaheurísticas, técnicas estas frequentemente utilizadas no problema.

Capítulo 6

Conclusões

6.1 Considerações Finais

Nesta dissertação foram investigadas novas abordagens do escalonamento baseado em AC para o problema de escalonamento estático de tarefas (PEET). Um conjunto de mudanças importantes em relação às abordagens anteriores foi definido. Assim, os principais objetivos alcançados nesta pesquisa foram: (i) garantir o aproveitamento do paralelismo nos ACs através do uso do modo de atualização síncrono das células do reticulado, (ii) utilizar modelos de vizinhança de estrutura simples e não limitada, (iii) considerar e avaliar o escalonamento sobre arquiteturas com mais de dois processadores, (iv) investigar novos métodos para direcionar, avaliar e utilizar o conhecimento adquirido e (v) comparar os resultados alcançados pelos modelos com outras formas de resolver o problema, sejam elas exatas, heurísticas ou meta-heurísticas.

A contribuição inicial deste estudo foi a investigação aprofundada do PEET, que envolveu conceitos, taxonomias, formulações técnicas, cálculo de atributos, além de instâncias e heurísticas utilizadas no problema. Desse modo, foi possível apresentar o problema de maneira detalhada e incorporar alguns de seus conceitos e técnicas já investigadas ao escalonamento baseado em AC.

O Escalonador baseado em Autômato Celular com atualização Síncrona (EACS) foi o primeiro modelo de escalonador desenvolvido. O objetivo do desenvolvimento do EACS foi, utilizando uma vizinhança simples e padrão, possibilitar o uso bem sucedido do modo de atualização síncrono das células no escalonamento com AC, o que ainda não havia sido alcançado por nenhum dos trabalhos pesquisados na literatura. Ao contrário, os melhores resultados publicados nos trabalhos anteriores utilizam o modo de atualização sequencial. Dessa forma, o modelo EACS é apto a usufruir de uma das principais características dos ACs, a sua adequação à implementação massivamente paralela. As principais mudanças do escalonador proposto em relação ao modelo original [46], estiveram relacionadas à estrutura do AG e à condição de contorno no AC. O desempenho de EACS foi avaliado

através de experimentos onde outros modelos da literatura, com modo de atualização tanto paralelo como sequencial, foram reproduzidos. Os resultados na fase de aprendizagem mostraram a superioridade de EACS frente ao modelo reproduzido para o modo paralelo e a similaridade com os resultados do modo de atualização sequencial. Contudo, analisando a convergência para o ótimo e a qualidade do conjunto de regras, o EACS apresentou o melhor desempenho dentre os demais modelos (considerando o modo de atualização paralelo e sequencial). Também foi realizado um teste simples sobre o modo de execução do EACS que teve seus resultados comparados ao de um algoritmo genético e destacou a maior vantagem do escalonamento com AC sobre a maioria das outras meta-heurísticas: a capacidade de extração e reuso do conhecimento.

O segundo modelo de escalonador foi desenvolvido com o intuito de diminuir a complexidade computacional no aumento do número de processadores Ele foi denominado EACS-H por ser um escalonador baseado em AC, mas que usa uma heurística de construção para definição do reticulado inicial. Esta é a principal diferença do modelo EACS-H em relação ao EACS e à qualquer outro modelo já proposto na literatura para o escalonamento baseado em AC. De fato, trabalhos anteriores avaliaram a capacidade de uma regra de transição levar qualquer configuração inicial do reticulado para uma configuração que represente o escalonamento ótimo. Contudo, nós percebemos que a busca por essa independência em relação ao reticulado inicial torna a busca pelas regras mais complexa e computacionalmente intensiva. Por outro lado, acreditamos que a mais importante habilidade de generalização não está relacionada à configuração inicial do reticulado, mas à capacidade da regra escalonar outras instâncias no modo de execução. Diante disso, o modelo EACS-H apresenta um novo conceito no escalonamento baseado em AC: a capacidade de avaliar a habilidade de uma regra ao executar a tarefa a partir de uma única configuração inicial. Além disso, o EACS-H herda boas características advindas do EACS, tais como: estrutura do AG, vizinhança linear e modo síncrono de atualização das células.

Vários experimentos foram realizados visando avaliar o desempenho do EACS-H, inclusive em relação ao reuso do conhecimento extraído de um dado grafo de programa em outros distintos. Em alguns experimentos os resultados do EACS-H foram comparados com as reproduções de abordagens anteriores e o novo modelo apresentou vantagens importantes em termos de resultados, desempenho computacional e escalabilidade com o número de processadores. Em outros experimentos, as fases de aprendizagem e execução do EACS-H foram avaliadas em relação a outras técnicas computacionais - heurísticas de construção e meta-heurísticas - e os resultados também foram bons, especialmente no modo de aprendizagem que apresentou valores muito próximos aos melhores valores obtidos por uma meta-heurística (AG-1). Finalmente, outros experimentos avaliaram o desempenho do EACS-H no reuso para vários grafos de programa distintos comparado a modelos de trabalhos relacionados e outras técnicas computacionais. Os resultados obtidos no modo de execução foram promissores e o escalonador inclusive foi superior a outras

técnicas computacionalmente mais intensivas.

Duas razões nos levaram à investigação de novas estruturas de vizinhança. A primeira relacionada aos aspectos positivos e negativos apresentados pelos modelos de vizinhança lineares e não lineares. Os modelos lineares apresentam uma estrutura bastante simples e de custo computacional baixo, contudo eles não são capazes de capturar a relação entre as tarefas no grafo de programa, pois utilizam-se das posições no reticulado para definir as células vizinhas de uma determinada tarefa. Por outro lado, os modelos não lineares conseguem traduzir adequadamente na vizinhança de uma tarefa as suas relações no grafo de programa. Todavia, eles apresentam um custo computacional elevado quando comparado aos modelos lineares, além de apresentarem limitações em relação ao aumento do número de processadores e uma estrutura um tanto quanto complexa.

A segunda razão foi decorrente das próprias investigações realizadas que identificaram uma falha no modelo de avaliação utilizado nos trabalhos anteriores que empregaram a vizinhança linear [51–53, 56]. Na avaliação das regras nessas abordagens foi definido que uma regra deve ser capaz de levar qualquer configuração aleatória das tarefas de um grafo de programa para uma configuração que representa a alocação ótima. Assim, uma regra apta a levar todas as configurações iniciais para a mesma configuração final é desejável, desde que o resultado do escalonamento obtido por essa alocação seja ótimo (ou sub-ótimo). Contudo, a grande falha nesse conceito está relacionada à capacidade de reuso da regra, uma vez que no modo de operação normal essa mesma regra terá uma tendência natural a levar qualquer reticulado que apresente o mesmo número de tarefas para a mesma configuração (devido ao modelo linear não considerar a relação entre as tarefas na definição da vizinhança), independentemente do grafo de programa utilizado ser ou não o mesmo.

Diante das razões apresentadas, dois métodos foram propostos como modelos de vizinhança para a nossa abordagem. Eles foram chamados de modelos pseudo-lineares, uma vez que se assemelham ao modelo linear, mas não definem suas relações baseadas simplesmente na proximidade das células no reticulado. De fato, os novos modelos V_{pl}-$c1$ e V_{pl}-$c2$ apresentam estrutura simples, pois levam em consideração o tamanho do raio e o número de estados possíveis do AC. Entretanto, diferentemente dos modelos lineares, as células da vizinhança são definidas como nos modelos não lineares, considerando as relações entre as tarefas no grafo de programa e não a posição no reticulado. Por outro lado, diferentemente do que ocorre nos modelos não lineares dos trabalhos pesquisados, não há limitações em relação ao número de nós da arquitetura multiprocessada. Um novo modelo de escalonador foi elaborado incorporando a vizinhança V_{pl}-$c2$ ao EACS-H. Esse novo modelo foi denominado EACS-HV. Além disso, o EACS-HV utiliza também uma estratégia de avaliação diferente do EACS-H, que foi chamada de penalização da regra, que consiste, basicamente, em realizar um cálculo sobre o tempo de escalonamento encontrado pela regra onde um acréscimo a esse tempo é dado ou não de acordo com a dinâmica

apresentada pela regra. Dessa forma, temos uma espécie de filtro para a escolha de regras com dinâmicas estáveis pelo AG.

Alguns experimentos foram conduzidos com o modelo EACS-HV, analisando o desempenho do mesmo em relação ao EACS-H, que utiliza um modelo de vizinhança linear, tanto no modo de aprendizagem quanto na fase de execução. Os resultados e análises estatísticas mostraram que o novo modelo de vizinhança pseudo-linear é bastante promissor, uma vez que foram superiores ao modelo de vizinhança linear nos dois modos.

Uma característica interessante e inédita deste trabalho é a análise sistemática do reuso das regras a partir das características do AC, uma vez que trabalhos anteriores sempre estiveram mais voltados ou para análise da fase de aprendizagem ou para mudanças na estrutura do AG que pudessem auxiliar o reuso. De fato, a principal ferramenta do escalonador é o AC e mudanças em sua estrutura também são necessárias e podem contribuir tanto para a extração quanto para o reuso das regras. Conforme foi mostrado nesta dissertação, há evidências estatísticas significativas que atestam o melhor desempenho das novas abordagens em relação àquelas apresentadas em trabalhos correlatos.

Em suma, as novas abordagens apresentadas nesta dissertação, EACS-H e EACS-HV, abrem novas perspectivas para o escalonamento baseado em AC. Os conceitos introduzidos, bem como as estruturas desenvolvidas, acrescentaram bastante ao desempenho do modelo, como pôde ser visto nos experimentos realizados. Mais do que uma abordagem promissora, este trabalho mostra o desempenho eficiente dos ACs como estruturas realmente propícias à extração e reuso do conhecimento no problema de escalonamento, sem mencionar ainda a enorme capacidade de paralelismo a ser explorada nestas estruturas.

6.2 Trabalhos Futuros

As contribuições deste trabalho abrem uma série de novas possibilidades para as abordagens de escalonamento baseado em AC. O bom desempenho alcançado por EACS-HV, especialmente na fase de aprendizagem, frente à modelos anteriores e outras técnicas computacionais, é um diferencial do modelo. Além disso, o modo de reuso se mostrou promissor, pois apresentou resultados muito bons para arquiteturas com menor número de processadores, especialmente até 4 processadores. Todavia, a complexidade do aumento de processadores ainda é uma questão que precisa ser explorada, uma vez que o espaço de busca aumenta sua cardinalidade drasticamente. Nesse sentido, propõe-se como trabalhos futuros a investigação de novas estruturas relacionadas tanto à aprendizagem quanto à execução, que permitam reduzir essa complexidade. Por exemplo, sugere-se investigar o emprego de parâmetros de previsão do comportamento dinâmico das regras de AC durante a fase de aprendizagem. Foi realizada a investigação e a implementação de alguns desses parâmetros para construir o procedimento de penalização da regra no EACS-HV e, apesar de não incorporá-los às abordagens apresentadas neste trabalho, acredita-se que

eles tenham potencial para melhorar o desempenho do modelo.

Uma das grandes diferenças entre EACS-H e os modelos de escalonamento baseado em AC apresentados na literatura [9, 10, 46, 49, 53, 57] é que estes utilizam um conjunto de reticulados aleatórios na avaliação das regras enquanto que aquele utiliza apenas um reticulado "determinístico". Diante disso, é normal que uma regra seja eliminada da população porque não conseguiu levar esse determinado reticulado à uma boa alocação final das tarefas. Contudo, é possível também pensar que o problema possa estar na configuração do reticulado e não apenas na regra. De fato, existem algumas regras que apesar de apresentarem dinâmica estável, podem ter isso alterado diante de alguns reticulados. Dessa forma, propõe-se um modelo semelhante à EACS-H, todavia baseado em mais de uma heurística. Acredita-se que utilizar dois ou mais reticulados como ponto de partida para a regra possa também auxiliar na busca de regras com dinâmicas as quais se deseja, especialmente na fase de reuso.

Além disso, dois modelos de vizinhaça foram apresentados neste trabalho, $V_{pl} - c1$ e $V_{pl} - c2$. Os experimentos mostraram que as vizinhanças auxiliaram diretamente na melhoria de desempenho por parte do EACS-H (Apêndice B) e que ambos são modelos de vizinhança promissores. Contudo, foi avaliado apenas o emprego de $V_{pl} - c2$ em EACS-HV, devido ao seu bom desempenho, especialmente na fase de aprendizagem. Dessa forma, propõe-se também a realização de experimentos utilizando $V_{pl} - c1$ como modelo de vizinhança para uma estrutura similar à EACS-HV, a fim de avaliar as vantagens e desvantagens oferecidas por este modelo em relação à $V_{pl} - c2$ e outros métodos computacionais.

Em trabalhos relacionados, outras estratégias evolutivas foram investigadas na fase de aprendizagem ou como suporte ao reuso nos escalonadores baseados em AC, tais como, sistemas imunológicos artificiais em [53], evolução conjunta em [57] e co-evolução em [49, 56]. Dessa forma, sugere-se a investigação de algumas dessas estratégias como meio de aperfeiçoar o modelo EACS-HV, especialmente em relação ao desempenho das regras evoluídas na fase de reuso.

De fato, o paralelismo é uma das características principais dos ACs. A aplicação bem sucedida do modo de atualização síncrono nos novos modelos de escalonadores baseados em AC permitem usufruir dessa característica. Logo, sugere-se também a construção do projeto e implementação de EACS-HV em *hardwares* paralelos, tais como FPGA (*Field-Programmable Gate Array*) e GPU (*Graphics Processing Unit*).

Outra sugestão de trabalho futuro é a construção de um modelo de escalonador baseado em EACS-HV capaz de lidar com o escalonamento dinâmico. De fato, o intuito em se estudar o escalonamento baseado em AC no PEET é justamente a possibilidade de desenvolver e avaliar o sistema para que posteriormente, ele possa ser utilizado em ambientes reais de escalonamento.

Referências Bibliográficas

[1] Adam, T. L., Chandy, K. M., and Dickson, J. R. A comparison of list schedules for parallel processing systems. *Commun. ACM 17* (December 1974), 685–690.

[2] Ahmad, I., and Kwok, Y.-K. A parallel approach for multiprocessor scheduling. In *Proceedings of the 9th International Symposium on Parallel Processing* (Washington, DC, USA, 1995), IPPS '95, IEEE Computer Society.

[3] Berlekamp, E. R., Conway, J. H., and Guy, R. K. *Winning Ways for Your Mathematical Plays*, vol. 2. Academic Press, 1982.

[4] Binder, P. A phase diagram for elementary cellular automata. *Complex Systems 7* (1993), 241–247.

[5] Binder, P. Parametric ordering of complex systems. *Physical Review E 49* (1994), 2023–2025.

[6] Cardoso, D. F. Escalonamento estático de tarefas em ambientes computacionais heterogêneos sob o modelo logp. Master's thesis, Universidade Federal Fluminense - Programa de Pós-Graduação em Computação, 2004.

[7] Carneiro, M. G. Artificial intelligence in games evolution. In *Business, Technological, and Social Dimensions of Computer Games: Multidisciplinary Developments*, M. M. Cruz-Cunha, V. H. Carvalho, and P. Tavares, Eds. IGI Global, 2011, ch. 7, pp. 98–114.

[8] Carneiro, M. G., and Fernandes, M. A. An algorithm for the dial-a-ride problem using optimization techniques. *International Conference on Information Systems and Technology Management* (2010).

[9] Carneiro, M. G., and Oliveira, G. M. B. Cellular automata-based model with synchronous updating for task static scheduling. In *Proceedings of 17th International Workshop on Cellular Automata and Discrete Complex System* (2011).

[10] Carneiro, M. G., and Oliveira, G. M. B. Um modelo baseado em autômatos celulares síncronos para o escalonamento de tarefas em multiprocessadores. In *Anais do X Congresso Brasileiro de Inteligência Computacional* (2011).

[11] Carneiro, M. G., and Oliveira, G. M. B. Synchronous cellular automata-based scheduler with construction heuristic to static task scheduling in multiprocessors. *(submitted to GECCO'12)* (2012).

[12] Carvalho, M. H., Cerioli, M., Dahab, R., Feofiloff, P., Fernandes, C. G., Ferreira, C. E., Guimaraes, K. S., Miyazawa, F. K., Jr, J. C. P., and Soares, J. A. *Uma Introdução Sucinta a Algoritmos de Aproximação*. IMPA - Instituto de Matemática Pura e Aplicada, 2001.

[13] Casavant, T. L., and Kuhl, J. G. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Enginnering 14*, 2 (1988), 141–154.

[14] Cormen, T. H., Stein, C., Rivest, R. L., and Leiserson, C. E. *Introduction to Algorithms*, 2nd ed. McGraw-Hill Higher Education, 2001.

[15] Cosnard, M., Marrakchi, M., Robert, Y., and Trystram, D. Parallel gaussian elimination on an mimd computer. *Parallel Computing 6*, 3 (1988), 275 – 296.

[16] Culik, K., Hurd, L. P., and Yu, S. Computation theoretic aspects of cellular automata. *Physica D 45* (1990), 357–378.

[17] DAG generation program. http://www.loria.fr/∼suter/dags.html, 2011.

[18] Das, R., Mitchell, M., and Crutchfield, J. P. A genetic algorithm discovers particle-based computation in cellular automata. In *International Conference on Evolutionary Computation* (1994), pp. 344–353.

[19] El-Rewini, H., and Lewis, T. G. Scheduling parallel program tasks onto arbitrary target machines. *J. Parallel Distrib. Comput. 9* (June 1990), 138–153.

[20] Fidelis, M. V., Lopes, H. S., and Freitas, A. A. Discovering comprehensible classification rules with a genetic algorithm. In *Proceedings of the Congress on Evolutionary Computation* (2000), vol. 1, pp. 805–810.

[21] Garey, M. R., and Johnson, D. S. *Computers and Interactability. A Guide to the Theory of NPCompleteness.* Freemann And Company, 1979.

[22] Goldbarg, M., and Luna, H. *Otimização Combinatória e Programação Linear.* Campus, 2005.

[23] Goldberg, D. E. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, 1989.

[24] Holland, J. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* University of Michigan Press, 1975.

[25] Hou, E. S. H., Ansari, N., and Ren, H. A genetic algorithm for multiprocessor scheduling. *IEEE Transactions Parallel Distributed Systems 5*, 2 (1994), 113–120.

[26] Jin, S., Schiavone, G., and Turgut, D. A performance study of multiprocessor task scheduling algorithms. *The Journal of Supercomputing 43* (2008), 77–97.

[27] Khan, A. A., McCreary, C. L., and Jones, M. S. A comparison of multiprocessor scheduling heuristics. *International Conference on Parallel Processing* (1994), 243–250.

[28] Kwok, Y. K., and Ahmad, I. Benchmarking and comparison of the task graph scheduling algorithms. *Journal of Parallel and Distributed Computing 59*, 3 (1999), 381–422.

[29] Kwok, Y. K., and Ahmad, I. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys 31*, 4 (1999), 406–471.

[30] Lee, Y.-C., and Zomaya, A. A novel state transition method for metaheuristic-based scheduling in heterogeneous computing systems. *IEEE Trans. Parallel Distrib. Syst. 19* (September 2008), 1215–1223.

[31] Li, W. Parameterizations of cellular automata rule space. Tech. rep., Santa Fe Institute, 1991.

[32] Li, W. Phenomenology of non-local cellular automata. *Journal of Statistical Physics* (1992).

[33] Li, W., and Packard, N. The structure of the elementary cellular automata rule space. *Complex Systems 4* (1990), 281–297.

[34] Li, W., Packard, N. H., and Langton, C. Transition phenomena in cellular automata rule space. *Physica D 45* (1990), 77–94.

[35] Mitchell, M., Crutchfield, J. P., and Das, R. Evolving cellular automata with genetic algorithms: A review of recent work. *In Proceedings of the First International Conference on Evolutionary Computation and Its Applications (EvCA'96)* (1996).

[36] Mitchell, M., Crutchfield, J. P., and Hraber, P. T. Evolving cellular automata to perform computations: mechanisms and impediments. In *International Seminar on Complex Systems: from complex dynamical systems to sciences of artificial reality* (1994), pp. 361–391.

[37] Oliveira, G. M. B. Autômatos celulares: aspectos dinâmicos e computacionais. *III Jornada de Mini-cursos em Inteligência Artificial (MCIA) 8* (2003), 297–345.

[38] Oliveira, G. M. B., Oliveira, P. P. B., and Omar, N. Guidelines for dynamics-based parameterization of one-dimensional cellular automata rule spaces. *Complexity 6*, 2 (2000), 63–71.

[39] Oliveira, G. M. B., Oliveira, P. P. B., and Omar, N. Improving genetic search for onedimensional cellular automata, using heuristics related to their dynamic behavior forecast. In *Congress on Evolutionary Computation* (2001).

[40] Pacheco, M. A. C. Algoritmos genéticos: Princípios e aplicações. *Apostila* (1999).

[41] Pinedo, M. L. *Scheduling: Theory, Algorithms, and Systems*, third ed. Springer Science, 2008.

[42] Russell, S., and Norvig, P. *Artificial intelligence: a modern approach*, third ed. Prentice Hall series in artificial intelligence. Prentice Hall, 2010.

[43] Salleh, S., and Zomaya, A. Y. Multiprocessor scheduling using mean-field annealing. In *IPPS/SPDP Workshops* (1998), pp. 288–296.

[44] Sarkar, P. A brief history of cellular automata. *ACM Comp. Surveys 32*, 1 (2000), 80–107.

[45] Sarkar, V. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* MIT Press, Cambridge, MA, USA, 1989.

[46] Seredynski, F. Scheduling tasks of a parallel program in two-processor systems with use of cellular automata. *IPPS/SPDP Workshops* (1998).

[47] Seredynski, F. *Evolving cellular automata-based algorithms for multiprocessor scheduling.* Wiley, 2001, pp. 179–207.

[48] Seredynski, F., Swiecicka, A., and Zomaya, A. Y. Discovery of parallel scheduling algorithms in cellular automata-based systems. *IPDPS* (2001).

[49] Seredynski, F., and Zomaya, A. Y. Sequential and parallel cellular automata-based scheduling algorithms. *IEEE Transactions on Parallel and Distributed Systems 13*, 10 (2002), 1009–1022.

[50] Sipper, M. *Evolution of Parallel Cellular Machines, The Cellular Programming Approach.* Springer, 1997.

[51] Swiecicka, A., and Seredynski, F. Cellular automata approach to scheduling problem. In *Proceedings of the International Conference on Parallel Computing in Electrical Engineering* (2000).

[52] Swiecicka, A., and Seredynski, F. Applying cellular automata in multiprocessor scheduling. In *Proceedings of the International Conference on Parallel Computing in Electrical Engineering* (2002), pp. 177–182.

[53] Swiecicka, A., Seredynski, F., and Zomaya, A. Y. Multiprocessor scheduling and rescheduling with use of cellular automata and artificial immune system support. *IEEE Transactions on Parallel and Distributed Systems 17*, 3 (2006), 253–262.

[54] Tanomaru, J. Motivação, fundamentos e aplicações de algoritmos genéticos. *II Congresso Brasileiro de Redes Neurais* (1995).

[55] Ullman, J. D. NP-complete scheduling problems. *Journal of Computer and System Science 10*, 3 (1975), 384–393.

[56] Vidica, P. M. Autômatos celulares e algoritmos genéticos aplicados ao problema do escalonamento de tarefas em sistemas multiprocessadores. Master's thesis, Universidade Federal de Uberlândia - Programa de Pós-Graduação em Ciência da Computação, 2006.

[57] Vidica, P. M., and Oliveira, G. M. B. Cellular automata-based scheduling: A new approach to improve generalization ability of evolved rules. *Brazilian Symposium on Artificial Neural Networks (SBRN'06)* (2006).

[58] von Neumann, J., and Burks, A. W. *Theory of Self-Reproducing Automata.* University of Illinois Press, Champaign, IL, USA, 1966.

[59] Wolfram, S. Cellular automata. *Los Alamos Science* (1983).

[60] WOLFRAM, S. Statistical mechanics of cellular automata. *Rev. Modern Physics 55* (1983), 601–644.

[61] WOLFRAM, S. Computation theory of cellular automata. *Communications in Mathematical Physics* (1984).

[62] WOLFRAM, S. Universality and complexity in cellular automata. *Physica D: Nonlinear Phenomena 10* (1984), 1 – 35.

[63] WOLFRAM, S. Complex systems theory. In *Emerging Syntheses in Science: Proceedings of the Founding Workshops of the Santa Fe Institute* (1988), Addison-Wesley, pp. 183–189.

[64] WOLFRAM, S. *Cellular Automata and Complexity.* Addison-Wesley, 1994.

[65] WOLFRAN, S. Cryptography with cellular automata. *Advances in Cryptology: Crypto '85 Proceedings 218* (1986), 429–432.

[66] WUENSCHE, A. *Complexity in one-D cellular automata: gliders, basins of attraction and the Z parameter.* Cognitive science research papers. University of Sussex, School of Cognitive and Computing Sciences, 1994.

[67] WUENSCHE, A. Classifying cellular automata automatically; finding gliders, filtering, and relating space-time patterns, attractor basins, and the z parameter. *Complexity 4* (1999), 47–66.

[68] WUENSCHE, A., AND LESSER, M. *The global dynamics of cellular automata: an atlas of basin of attraction fields of one-dimensional cellular automata.* Addison-Wesley, 1992.

[69] YOU WU, M., AND GAJSKI, D. D. Hypertool: A programming aid for message-passing systems. *IEEE Transactions on Parallel and Distributed Systems 1* (1990), 330–343.

Apêndice A

Outros tipos de heurísticas de construção para o PEET

A.1 UNC - Número Ilimitado de Agrupamentos

Os algoritmos classificados como UNC estudados neste trabalho foram: EZ, LC, DSC e DCP. Na Tabela A.1 é feito um comparativo como meio de oferecer uma visão geral acerca das características das heurísticas.

Tabela A.1: Comparativo entre heurísticas UNC.

Heurística	**Prioridade**	**Atributos**
EZ	Estática	sl
LC	Dinâmica	CP
DSC	Dinâmica	b-level, t-level, SD
DCP	Dinâmica	b-level, t-level, CP

A.1.1 EZ

O EZ (*Edge Zeroing*) caracteriza-se por selecionar agrupamentos para mesclagem baseado nos pesos de aresta. Assim, a cada passo o algoritmo encontra a aresta com o maior peso. Os dois agrupamentos incidentes pela aresta serão mesclados se tal procedimento não aumenta o tempo de completude. Após mesclados a ordem das tarefas no agrupamento resultante é baseada no atributo estático sl. Na Figura A.1 tem-se um esboço do algoritmo.

Na Figura A.2 tem-se o resultado do escalonamento do grafo de programa *gp9* utilizando a heurística EZ. Na figura, o campo "Aresta" está ligado às relações de precedência entre duas tarefas e "Valor" se refere aos custos de comunicação dessas relações. O tempo de conclusão do escalonamento obtido pelo algoritmo foi de 18 unidades de tempo.

EZ (Edge Zeroing)	
Passo 0	Coloque as arestas do GAD em uma ordem descendente de acordo com seus pesos.
Passo 1	Inicialmente, tome todas as arestas como não examinadas.
Passo 2	Enquanto todas as arestas não são examinadas, faça (Passo 3).
Passo 3	Pegue uma aresta não examinada que tenha o maior peso. Tome-a como examinada. Zere o peso da aresta maior se o tempo de conclusão não aumenta. Neste processo de zeragem, dois *clusters* são mesclados de forma que outras arestas entre esses dois *clusters* também precisam ser zeradas e tomadas como examinadas. A ordenação dos nós no *cluster* resultante é baseada no sl (decrescente).

Figura A.1: Algoritmo EZ.

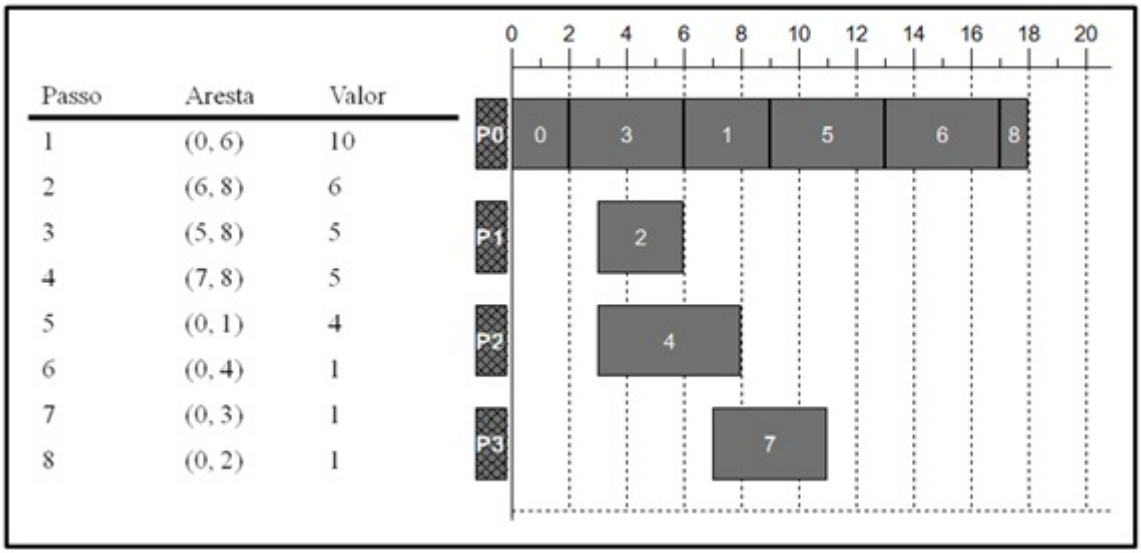


Figura A.2: Escalonamento obtido pelo EZ para o grafo de programa *gp9*.

A.1.2 LC

A heurística LC (*Linear Clustering*) trabalha com o conceito de CP para realizar o escalonamento das tarefas. Ela primeiramente determina o conjunto de nós que constituem o CP, então os escalona em um único processador por vez. Depois disso, estes nós e todas arestas incidentes neles são então removidas do GAD. Na Figura A.3 tem-se uma breve descrição do algoritmo.

O resultado do escalonamento do grafo de programa *gp9* pelo algoritmo LC é apresentado na Figura A.4 e, como pode ser visto, obteve tempo de conclusão equivalente a 19 unidades de tempo. Na figura, o campo "CP: Custo" está relacionado ao caminho crítico de tarefas ainda não examinadas e seus respectivos valores.

LC (Linear Clustering)	
Passo 0	Inicialmente, tome todas as arestas como não examinadas.
Passo 1	Enquanto todas as arestas não são examinadas, faça (Passo 2-4).
Passo 2	Determine o CP (*Critical Path*) composto somente de arestas não examinadas.
Passo 3	Crie um *cluster* pela zeragem de todas as arestas no CP.
Passo 4	Tome todas as arestas incidentes no CP e todas as arestas incidentes para os nós no *cluster* como examinadas.

Figura A.3: Algoritmo LC.

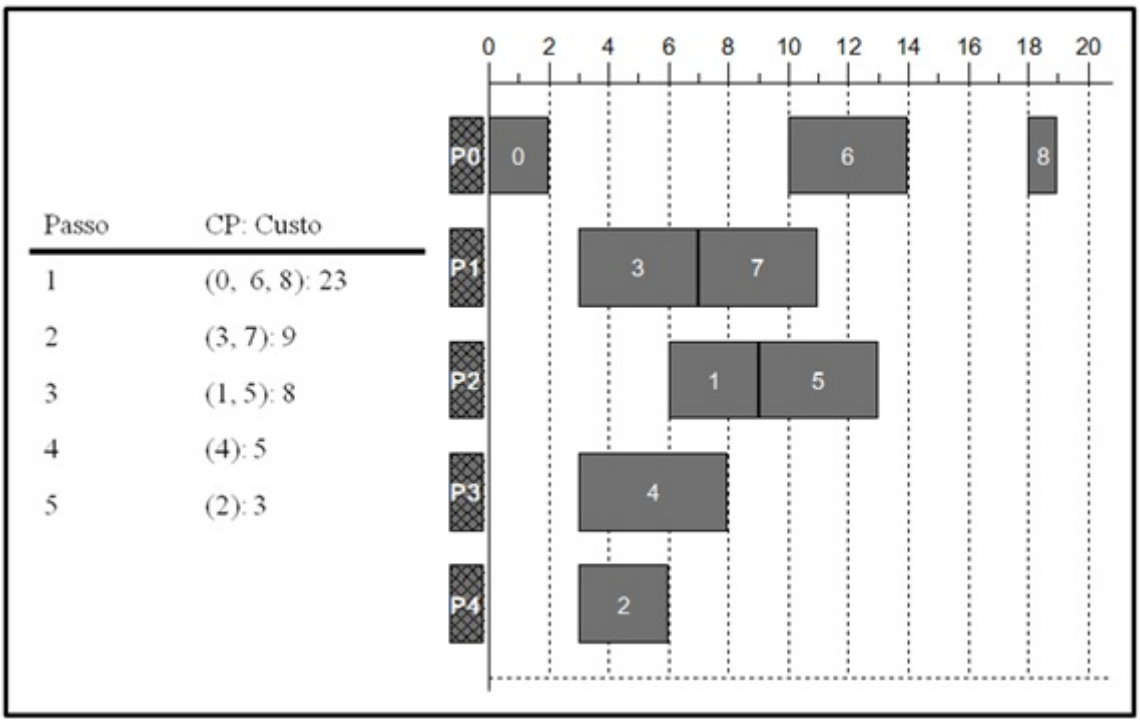


Figura A.4: Escalonamento obtido pelo LC para o grafo de programa *gp9*.

A.1.3 DSC

O DSC (*Dominant Sequence Cluster*) utiliza o conceito de sequência dominante (SD) de um grafo para realizar o escalonamento de tarefas. SD é o CP do GAD parcialmente escalonado. O algoritmo DSC é brevemente descrito na Figura A.5 abaixo.

Na Figura A.6 tem-se o resultado do escalonamento do grafo de programa *gp9* utilizando a heurística DSC. O algoritmo encontrou um escalonamento com tempo de conclusão igual a 17 unidades de tempo. Na figura, os campos "n_i", "n_y" e "Pai" referem-se, respectivamente, a tarefa escalonada naquele passo, uma tarefa não pronta com maior prioridade na SD e uma tarefa predecessora de n_i.

DSC (Dominant Sequence Clustering)	
Passo 0	Inicialmente, tome todos os nós como não examinados e inicialize uma lista de nós prontos L para conter todos os nós de entrada. Calcule o b-level de cada nó e ajuste o t-level para cada nó apto a ser executado.
Passo 1	Enquanto todos os nós não são examinados, faça (Passo 2-4).
Passo 2	Se a cabeça de L, n_i, é um nó na DS (*Dominant Sequence*), zere a aresta entre n_i e um de seus pais para que o t-level de n_i seja minimizado. Se nenhuma zeragem é permitida, o nó fica em um *cluster* de nó único.
Passo 3	Se a cabeça de L, n_i, não é um nó na DS (*Dominant Sequence*), zere a aresta entre n_i e um de seus pais para que o t-level de n_i seja minimizado sob a restrição chamada DSRW (*Dominant Sequence Reduction Warranty*). Se algum de seus pais são nós de entrada que não possuem qualquer outro filho além de n_i, mescle parte deles para que o t-level de n_i seja minimizado. Se nenhum zeramento é permitido, o nó fica em um *cluster* de nó único.
Passo 4	Atualize o t-level e o b-level dos sucessores de n_i e tome n_i como examinado.

Figura A.5: Algoritmo DSC.

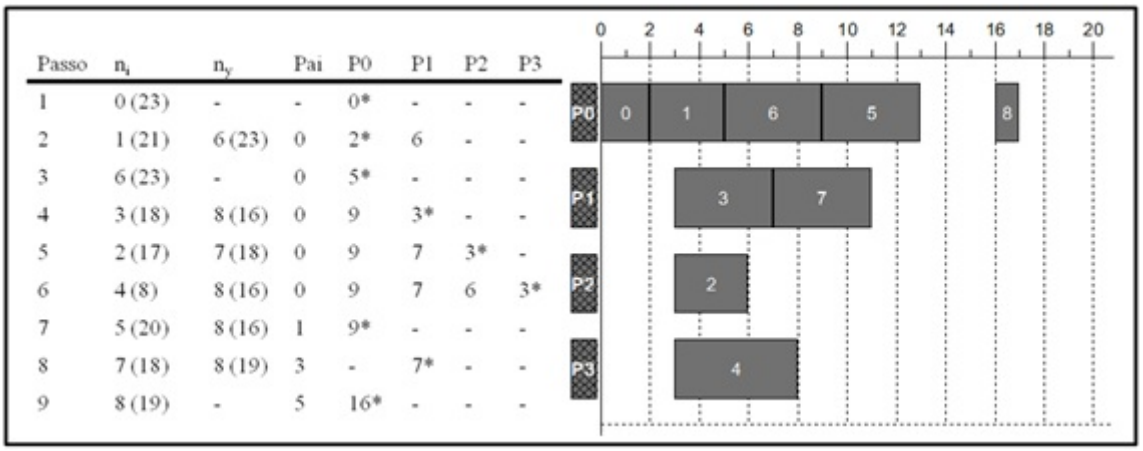


Figura A.6: Escalonamento obtido pelo DSC para o grafo de programa *gp9*.

A.1.4 DCP

A heurística DCP (*Dynamic Critical Path*) calcula "(Cur_ CP_ Lenght - (b-level + t-level))" para todas as tarefas e escalona aquela que possui menor valor. "Cur_CP_Lenght" pode ser entendido como o tamanho do caminho crítico do passo atual do escalonamento. Na Figura A.7, tem-se o algoritmo DCP, que utiliza prioridade dinâmica e uma estratégia do tipo "lookahead" a fim de encontrar um melhor agrupamento para uma dada tarefa.

O resultado do escalonamento do grafo de programa *gp9* através da heurística DCP pode ser visto na Figura A.8. O tempo de conclusão do escalonamento obtido é de 16 unidades de tempo. Na figura, o campo "n_c" faz referência a tarefa sucessora de "n_i"

DCP (Dynamic Critical Path)	
Passo 0	Enquanto todos os nós não são escalonados, faça (Passo 1-4).
Passo 1	Calcule "(Cur_CP_Lenght - (b-level(n_i) + t-level(n_i)))" como a prioridade de cada nó. Cur_CP_Lenght é o tamanho do caminho crítico atual.
Passo 2	Considere n_x o nó com a maior prioridade. Permita n_c ser o *critical child* (nó filho de n_x que possui maior prioridade).
Passo 3	Selecione um *cluster P* tal que a soma "$T_s(n_x)$ + ($T_x(n_c)$)" é a menor entre todos os *clusters* explorados pais ou filhos de n_x. Na análise de um cluster, primeiro tente não utilizar qualquer nó que crie ou aumente um espaço de tempo vazio. Se não é possível encontrar um espaço para n_x, varra o *cluster* por espaços de tempo vazio adequados novamente, possivelmente colocando alguns nós já escalonados para baixo.
Passo 4	Escalone n_x para *P*.

Figura A.7: Algoritmo DCP.

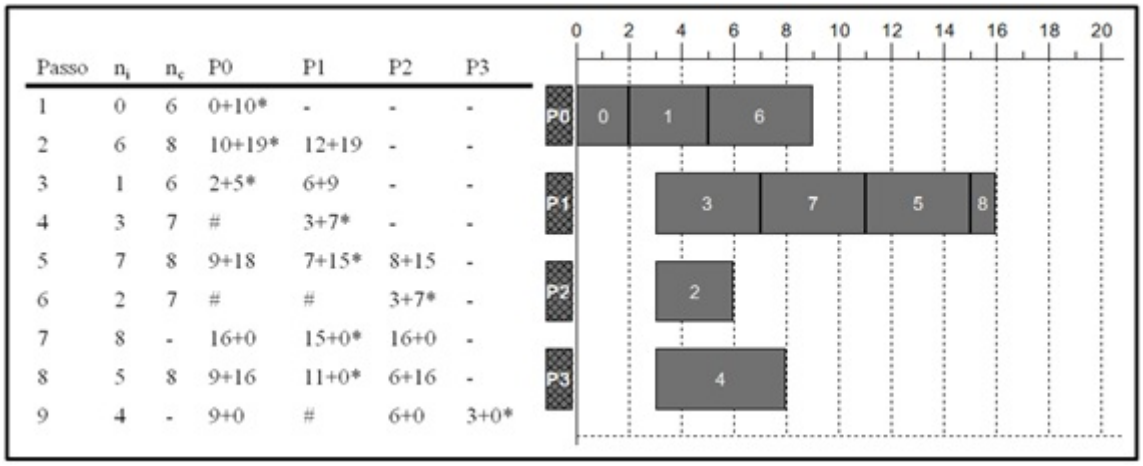


Figura A.8: Escalonamento obtido pelo DCP para o grafo de programa *gp9*.

(tarefa escalonada naquele passo) com maior prioridade.

A.2 TDB - Duplicação de Tarefas

Os algoritmos classificados como TDB apresentados a seguir são: DSH e CPFD. Na Tabela A.2 é feito um comparativo geral entre as duas heurísticas.

Tabela A.2: Comparativo entre heurísticas TDB.

Heurística	**Prioridade**	**Atributos**
DSH	Dinâmica	sl, TIC
CPFD	Dinâmica	CPN, IBN, OBN

A.2.1 DSH

A heurística DSH (*Duplication Scheduling Heuristic*) realiza a duplicação de tarefas quando há um espaço de tempo vazio entre o tempo de término da última tarefa no processador e o tempo de início da tarefa atual. O algoritmo utiliza prioridade estática (sl) e é apresentado em síntese na Figura A.9.

DSH (Duplication Scheduling Heuristic)	
Passo 0	Calcule o nível estático (sl) para cada nó.
Passo 1	Enquanto todos os nós não são escalonados, faça (Passo 2-4).
Passo 2	Permita n_i ser um nó não escalonado com maior sl.
Passo 3	Para cada processador *P* faça (Passo 3.1 - 3.3).
Passo 3.1	Considere o tempo pronto de *P*, denotado por *RT*, ser o tempo de término do último nó em *P*. Calcule o tempo de início de n_i em *P* e denote-o por *ST*. Então o espaço de tempo de duplicação em *P* tem tamanho "(*ST* - *RT*)". Considere o *candidate* ser n_i.
Passo 3.2	Considere o conjunto de pais do *candidate*. Deixe n_x ser o pai de n_i que não é escalonado em *P* e cuja mensagem para *candidate* tem o maior tempo de chegada. Tente duplicar n_x dentro do espaço de tempo de duplicação.
Passo 3.3	Se a duplicação não é bem sucedida, então grave *ST* para este processador e tente outro processador; caso contrário, considere *ST* ser o *candidate* de novo tempo de início e *candidate* ser n_x. Então volte para (Passo 3.2).
Passo 4	Considere *P'* ser o processador que dá o tempo de início mais cedo de n_i. Escalone n_i para *P'* e realize toda duplicação necessária em *P'*.

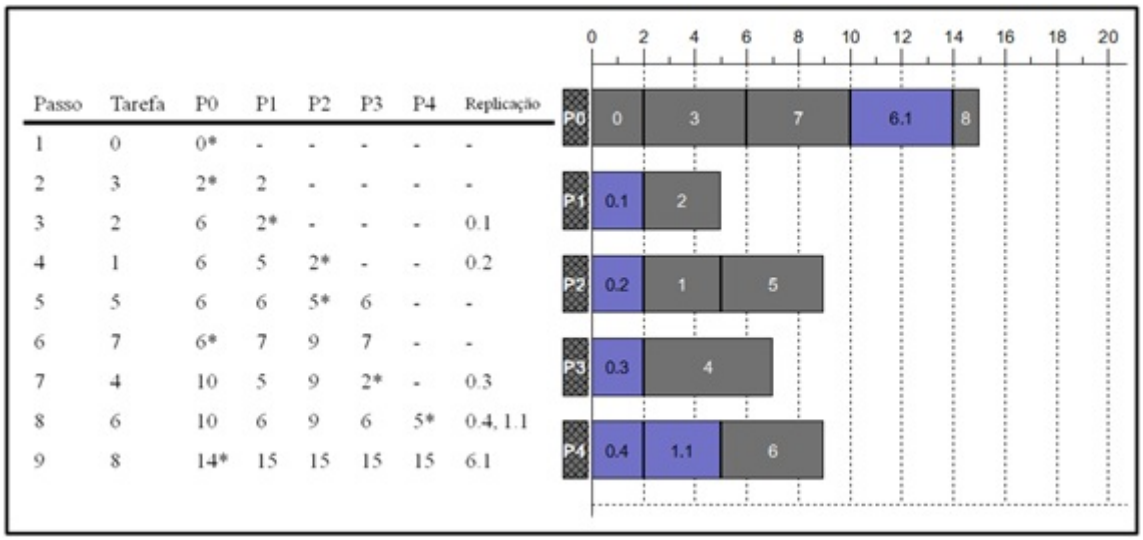
Figura A.9: Algoritmo DSH.

Na Figura A.10 é exibido o resultado do escalonamento do grafo de programa *gp9* pela heurística DSH. O tempo de conclusão encontrado pelo algoritmo foi de 15 unidades de tempo. Note que, na figura, o campo "Replicação" se refere às tarefas replicadas naquele passo.

A.2.2 CPFD

O algoritmo CPFD (*Critical Path Fast Duplication*) é baseado no particionamento do GAD em três categorias: CPN, IBN e OBN. Usando tal procedimento tem-se uma lista denominada Sequência Dominante de Nós do Caminho Crítico (*Critical Path Node - Dominant Sequence*, CPN-DS). O algoritmo CPN-DS é apresentado na Figura A.11 abaixo. Posterior a execução desse algoritmo é executada a heurística CPFD, que é exibida na Figura A.12.

O resultado do escalonamento obtido pela heurística CPFD sobre o grafo de programa *gp9* é apresentado na Figura A.13. O algoritmo encontrou um escalonamento cujo tempo



Passo	Tarefa	P0	P1	P2	P3	P4	Replicação
1	0	0*	-	-	-	-	-
2	3	2*	2	-	-	-	-
3	2	6	2*	-	-	-	0.1
4	1	6	5	2*	-	-	0.2
5	5	6	6	5*	6	-	-
6	7	6*	7	9	7	-	-
7	4	10	5	9	2*	-	0.3
8	6	10	6	9	6	5*	0.4, 1.1
9	8	14*	15	15	15	15	6.1

Figura A.10: Escalonamento obtido pelo DSH para o grafo de programa *gp9*.

CPN – Dominant Sequence	
Passo 0	Tome o CPN de entrada para ser o primeiro nó na lista de sequência *S*. Ajuste *Position* para 2. Considere n_x ser o próximo CPN.
Passo 1	Enquanto todos os CPNs não estão em *S*, faça (Passo 2-3).
Passo 2	Se n_x tem todos os seus nós pai *S*: então (Passo 2.1); senão (Passo 2.2)
Passo 2.1	Coloque n_x na *Position* em *S* e incremente *Position*.
Passo 2.2	Enquanto todos os nós pai de n_x não estão em *S*, faça (Passo 2.2.1). Coloque n_x em *S* na *Position*.
Passo 2.2.1	Suponha n_y ser o nó pai de n_x que não está em *S* e tem o maior b-level. Em caso de empate, escolhe-se o pai com menor t-level. Se n_y tem todos os seus nós pai em *S*, coloque n_y na *Position* em *S* e incremente *Position*. Caso contrário, recursivamente inclua todos os nós ancestrais de n_y em *S* de forma que os nós com maior comunicação sejam considerados primeiro.
Passo 3	Considere n_x ser o próximo CPN.
Passo 4	Acrescente todos os OBNs para *S* em ordem decrescente de b-level.

Figura A.11: Algoritmo CPN-DS.

de conclusão é igual a 15 unidades de tempo. É importante ressaltar que a heurística CPFD se baseia na lista gerada por CPN-SD para realizar o escalonamento (0, 1, 6, 3, 2, 7, 5, 8, 4) e o campo "Replicação" se refere às tarefas replicadas naquele passo.

CPFD (Critical Path Fast Duplication)	
Passo 0	Determine o caminho crítico. Particione o grafo de tarefa em CPNs, IBNs e OBNs. Considere candidate ser o CPN (Critical Path Node) de entrada.
Passo 1	Enquanto todos os CPNs não são escalonados, faça (Passo 2-5).
Passo 2	Considere P_SET ser o conjunto de processadores contendo as acomodações dos pais de candidate mais um processador vazio.
Passo 3	Para cada processador P em P_SET, faça (Passo 3.1-3.3).
Passo 3.1	Determine o tempo de início de candidate em P e denote-o por ST.
Passo 3.2	Considere o conjunto de pais de candidate. Considere m ser o pai que não é escalonado em P e cuja mensagem para candidate tem o maior tempo de chegada.
Passo 3.3	Tente duplicar m no espaço de tempo vazio mais cedo em P. Se a duplicação é bem sucedida e o novo tempo de início de candidate é menor que ST, então considere ST ser o novo tempo de início de candidate. Altere candidate para m e volte para (Passo 3.1). Se a duplicação não é bem sucedida, então retorne o controle para examinar outro pai do candidate anterior.
Passo 4	Escalone candidate para o processador P′ que dá o tempo de início mais cedo e realize toda a duplicação necessária.
Passo 5	Considere candidate ser o próximo CPN.
Passo 6	Repita o algoritmo do Passo 2-5 para cada OBN com P_SET contendo todos os processadores em uso junto com um processador vazio. Os OBNs são considerados um por um topologicamente.

Figura A.12: Algoritmo CPFD.

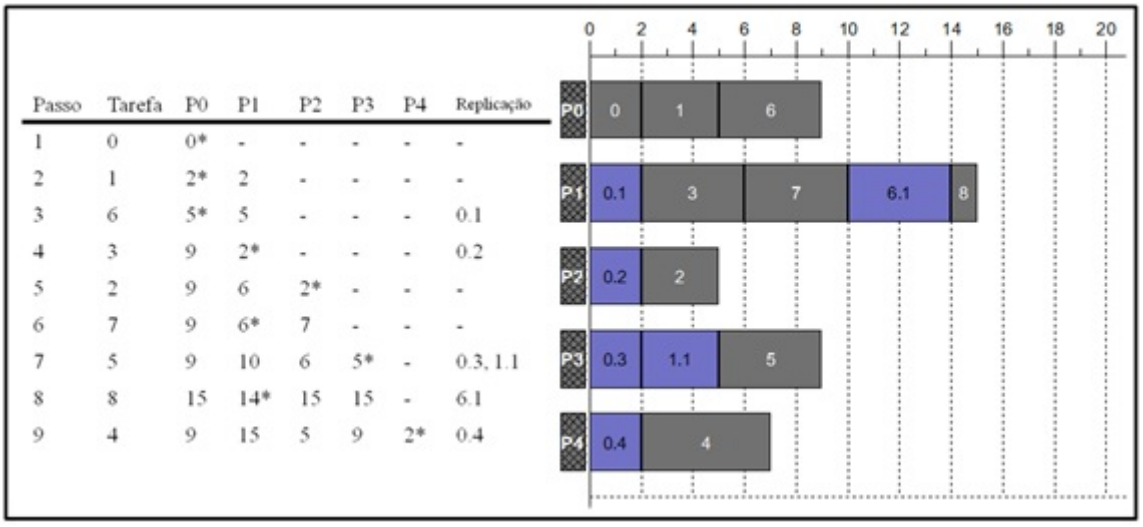


Figura A.13: Escalonamento obtido pelo CPFD para o grafo de programa *gp9*.

Apêndice B

Análise das novas vizinhanças: $V_{pl} - c1$ e $V_{pl} - c2$

No capítulo 5, foram apresentadas as novas vizinhanças e os novos conceitos desenvolvidos na arquitetura do escalonador. Neste apêndice, relatamos experimentos que foram realizados a fim de avaliar a melhor vizinhança para o modelo EACS-H. Assim, foram considerados o modelo de vizinhança linear ("linear") utilizado em [51–53, 56], que também foi utilizado na definição dos modelos EACS e EACS-H, além das vizinhanças propostas e apresentadas na Seção 5.4.2: $V_{pl}-c1$ e $V_{pl}-c2$. O intuito deste estudo é definir qual destes modelos de vizinhança apresenta melhor desempenho a fim de incorporá-lo às próximas abordagens. Cabe ressaltar que a penalização da regra foi aplicada nos três modelos de vizinhança, pois foi identificado que mesmo no caso da vizinhança linear, a penalização melhora o desempenho das regras.

Nestes experimentos foram considerados os grafos de programa *gauss18*, *random30*, *random40* e *random50*. Os grafos de programa *g18* e *g40* não foram considerados devido aos resultados pouco conclusivos apresentados em experimentos anteriores. Diferentes valores para o número de processadores $V_s \in \{2, 3, 4, 8\}$ foram avaliados. O tamanho da vizinhança usada nos experimentos foi $R \in \{1, 2, 3\}$. O número de passos de atualização do AC novamente foi $S = 3 * N$. Os parâmetros usados no AG foram: tamanho da população $T_{pop} \in \{200, 400, 600\}$, torneio simples $Tour = 2$, taxa de cruzamento $P_{cross} = 100\%$, taxa de mutação $P_{mut} = 3\%$ e número de gerações $G \in \{100, 200, 300\}$. O percentual de incremento adotado no cálculo da taxa de penalidade da regra foi $P_{inc} = 0,5\%$ para as três vizinhanças. Foram realizadas 20 execuções para cada experimento e a política de escalonamento adotada em todas elas foi "a tarefa com maior nível dinâmico primeiro".

A Tabela B.1 exibe uma comparação entre o desempenho de três modelos de vizinhança para o modo de aprendizagem do modelo EACS-H. Na tabela, "AVG" representa a média do melhor indivíduo obtido considerando todas as execuções enquanto que "BEST" refere-se ao valor encontrado na melhor execução. Analisando os resultados da tabela, tem-se que para o *gauss18* há uma leve desvantagem da vizinhança **linear** sobre as de-

mais. Entretanto, considerando os demais grafos de programa, essa desvantagem se torna bastante significativa. A única excessão é para o *random30* com $V_s = 8$, onde o melhor resultado obtido ("BEST") por **linear** foi um pouco melhor que as outras vizinhanças, contudo basta apenas uma análise da média obtida para se verificar que em geral o ambiente com vizinhança linear retornou regras distantes desse desempenho, ao contrário dos ambientes que empregaram $V_{pl} - c1$ e $V_{pl} - c2$.

Tabela B.1: Análise dos novos modelos de vizinhança no modo de aprendizagem do modelo EACS-H.

G_P	V_s	**DHLFET**	**linear-P**		$V_{pl} - c1$**-P**		$V_{pl} - c2$**-P**	
			AVG	BEST	AVG	BEST	AVG	BEST
gauss18	2	54	44,04	44	44,39	44	44,00	44
	3	52	44,42	44	44,00	44	44,00	44
	4	52	44,20	44	44,00	44	44,00	44
	8	52	46,61	44	44,01	44	44,00	44
random30	2	1311	1227,71	1222	1232,08	1222	1227,13	1222
	3	946	931,00	859	927,10	854	909,86	840
	4	825	880,50	823	798,77	778	818,71	778
	8	778	830,20	776	782,43	778	780,53	778
random40	2	1001	985,90	983	985,65	983	984,45	983
	3	733	741,60	695	741,15	695	692,40	679
	4	620	654,55	602	623,03	579	587,96	567
	8	478	564,65	519	542,85	509	474,44	443
random50	2	724	637,12	628	632,66	624	629,90	624
	3	636	618,80	556	531,10	512	540,86	512
	4	572	604,52	548	529,52	512	538,17	520
	8	556	670,20	616	622,87	524	549,19	512

Ainda analisando a Tabela B.1, é possível perceber uma certa proximidade entre os resultados alcançados em $V_{pl} - c1$ e $V_{pl} - c2$, na maioria dos grafos. Todavia, no caso do grafo de programa *random40*, onde as diferenças foram muito significativas, considerou-se o desempenho em $V_{pl} - c2$ melhor.

A fase de execução do EACS-H também foi avaliada para cada vizinhança. As Tabelas B.2 e B.3 apresentam respectivamente o reuso das regras obtidas na fase de aprendizagem para *gauss18* e *random30*. Ao analisar a Tabela B.2 é possível afirmar que $V_{pl} - c1$ apresentou o melhor desempenho. Por outro lado, o desempenho de **linear** foi considerado inferior ao de $V_{pl} - c2$, especialmente pelo fraco desempenho que apresentou no reuso para o *random50* e pelos altos valores de média encontrados para alguns casos.

Na análise da Tabela B.3, $V_{pl} - c2$ apresentou melhor desempenho no reuso para o *gauss18* e *random50*, enquanto **linear** apresentou pior desempenho para esses grafos de programa. Por outro lado, **linear** apresentou desempenho um pouco melhor que $V_{pl} - c1$ no reuso para o *random40*, e $V_{pl} - c2$ obteve os piores resultados para este grafo.

Tabela B.2: Análise dos novos modelos de vizinhança no modo de execução (*gauss18*) de EACS-H.

G_P *gauss18*	V_s	**linear-P**		$V_{pl} - c1$**-P**		$V_{pl} - c2$**-P**	
		AVG	BEST	AVG	BEST	AVG	BEST
random30	2	1250,40	1225	1239,00	1225	1249,65	1229
	3	1019,50	929	961,85	913	1006,35	927
	4	1228,60	964	937,30	833	992,15	861
	8	948,45	850	815,90	778	959,20	879
random40	2	1008,80	987	994,55	983	994,55	990
	3	784,65	736	752,90	705	793,00	746
	4	1023,10	762	706,40	651	794,70	751
	8	663,50	591	565,20	548	649,75	579
random50	2	677,80	652	676,40	640	666,00	640
	3	660,20	608	626,60	588	640,00	600
	4	743,40	656	629,20	584	651,00	616
	8	701,40	672	722,20	656	691,40	656

Tabela B.3: Análise dos novos modelos de vizinhança no modo de execução (*random30*) de EACS-H.

G_P *random30*	V_s	**linear-P**		$V_{pl} - c1$**-P**		$V_{pl} - c2$**-P**	
		AVG	BEST	AVG	BEST	AVG	BEST
gauss18	2	51,55	46	48,30	44	47,05	44
	3	52,75	49	50,30	47	47,10	44
	4	57,15	47	51,00	44	47,10	44
	8	64,35	60	52,30	47	45,90	44
random40	2	997,20	984	995,30	989	996,85	987
	3	739,00	698	750,15	711	781,75	744
	4	678,50	645	685,70	622	748,25	638
	8	589,50	538	558,80	540	632,55	540
random50	2	663,20	648	661,20	636	661,00	636
	3	631,80	612	656,60	600	644,80	616
	4	660,80	648	642,40	592	641,80	548
	8	688,00	656	720,40	704	702,20	656