Escalonamento Estático de Tarefas Bi-objetivo e Tolerante a Falhas para Sistemas Distribuídos

Idalmis Milián Sardiña

Tese de Doutorado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito para a obtenção do título de Doutor. Área de concentração: Redes e Sistemas Distribuídos e Paralelos.

Orientadores
Professora PhD. Cristina Boeres
Professora Dsc. Lúcia Drummond

Niterói, 2010

Escalonamento Estático de Tarefas
Bi-objetivo e Tolerante a Falhas para Sistemas Distribuídos

Idalmis Milián Sardiña

Tese de Doutorado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito para a obtenção do título de Doutor.

Aprovada por:

Profª. Maria Cristina Silva Boeres / UFF (Presidente)

Profª. Lúcia Maria de Assumpção Drummond / UFF

Prof. Eugene Francis Vinod Rebello / UFF

Prof. Luiz Satoru Ochi / UFF

Prof. Ricardo Cordeiro Corrêa / UFC

Prof. Cesar Augusto Fonticielha De Rose / PUCRS

Prof. Felipe Maia Galvão França / UFRJ

Niterói, junho de 2010.

A Dios,
a mi familia, a mis amigos,
que mismo con la distancia
son mi constante fuerza e inspiración.

Agradecimentos

Em primeiro lugar, gostaria de agradecer a Deus por me iluminar e me dar forças para me levantar em cada momento difícil.

A minha família, em especial a minha mãe Hortensia e a minha irmã Beatriz por serem meu amor total e incondicional, a meu pai por sua enorme bondade, a meu esposo e família pelo amor e a companhia paciente, a minhas primas por serem irmãs e a todos meus parentes queridos por me apoiar.

A minhas orientadoras Cristina e Lúcia pela paciência, empenho e competência.

A banca por aceitar e avaliar meu trabalho com total dedicação.

A minha família brasileira, minha mãe Nira, minha irmã Valeria por me adotar e me cuidar como parte de seu sangue.

A todos meus amigos. Aos de longe pela linda lembrança dos melhores momentos, em especial a Deborah, Maydita, Liset, Idalmis, Jemar, Massiel e German. Aos mais próximos do dia a dia pela amizade acalmando os momentos de estresse. No IC-UFF por essa família bonita de alunos, professores e funcionários: Geiza, Jacques, Aline, Alexandre, Cristiano, Renathinha, Cris, Stenio, Lu, Copeti, Janine, Luciene, Ju, Diego, Dani, Rodrigo, Dri, Roma, Simone, Flavia, Glauco, Alex e outros. A Teresa, Angela, Viviane, Carlinhos, Dulcinea, Mauricio, Henrique etc. Aos que fora da UFF não deixaram por nada sentir-me sozinha, Luciana, Pri, Roxana, Renata e Ilza. A Dorinha, Paulo, Rita, Leo e mãe da Lu por me incluir na família. A meus amigos e colegas da Plínio que amenizam a cansativa jornada de trabalho.

Em geral, a muitos e a este país maravilhoso que fazem parte da minha estrada e me fizeram uma pessoa melhor.

A bolsa FAPERJ que permitiu meus estudos para a realização deste trabalho.

Resumo

Em sistemas distribuídos de larga escala a propensão à ocorrência de falhas em recursos é um problema que deve ser priorizado para a execução confiável da aplicação. Diversas heurísticas de escalonamento de aplicações não consideram a confiabilidade dos recursos ao escalonar as tarefas da aplicação, ou especificam esquemas de tratamento de falhas centralizados e pouco flexíveis que se tornam inapropriados para sistemas distribuídos heterogêneos. Esta tese propõe uma estratégia de escalonamento estático bi-objetivo e tolerante a falhas, onde na primeira etapa, a estratégia tem como finalidade estabelecer não somente um tempo de execução desejável, mas também maior confiabilidade. Para resolver este problema, uma função de custo ponderada integra tanto a minimização do tempo de execução quanto a maximização da confiabilidade nos recursos selecionados. O trabalho inclui também propostas relativas ao ajuste de pesos dos objetivos do problema, para que um bom escalonamento seja definido de acordo com as necessidades do usuário. Para tal, uma metodologia é desenvolvida, que auxilia na escolha de soluções de compromisso. Na segunda parte do trabalho, explora-se uma técnica de replicação passiva para escalonar aplicações considerando tolerância a falhas. Para garantir flexibilidade e consistência, os algoritmos propostos introduzem critérios para o escalonamento de tarefas *backups*, adicionando também a função ponderada proposta na primeira etapa. A estratégia de escalonamento foi também avaliada com o desenvolvimento de mecanismos de recuperação com replicação passiva no *middleware* EasyGrid. Estes mecanismos consideram informações produzidas pelo escalonamento estático para a recuperação de falhas com execução eficiente e confiável da aplicação paralela no ambiente real. A tese destaca a importância de usar abordagens de escalonamento flexíveis com tolerância a falha em ambientes distribuídos com recursos heterogêneos.

Palavras-chave: Escalonamento Estático, Tolerância a Falhas, Escalonamento Bi-objetivo, Sistemas Distribuídos

Abstract

In large-scale distributed systems, the occurrence of faults in resources is a problem that should be prioritized. Several scheduling heuristics do not consider the resources reliability to schedule the tasks of a given application, or specify inappropriate fault treatments, which are centralized and not flexible for heterogeneous distributed systems. This thesis proposes a fault-tolerant bi-objective static scheduling strategy, where during the first stage the strategy addresses a problem of bi-objective scheduling in order to establish not only a desirable execution time, but also a high reliability. To resolve this problem, a weighted cost function integrates both, the minimization of execution time as the maximization of execution reliability. The work also includes a proposal for the adjustment of weights of the bi-objetive problem, so that a good schedule is set according to user needs. To this end, a methodology is developed, to aid on the choice of a tradeoff solution. In the second part, this work explores a passive replication technique for the applications scheduling considering fault tolerance. To ensure flexibility and consistency, the proposed algorithms introduce rules for the scheduling of backups tasks, based also on weighted function proposed in first part of this thesis. The scheduling strategy is evaluated with the development of recovery mechanisms with passive replication in the middleware EasyGrid. The mechanisms consider information produced by weighted bi-objetive static scheduling, helping on the recovery from faults, resulting in an efficient and reliable execution of parallel application in the real environment. This thesis emphasize the importance of using flexible scheduling approaches with fault tolerance in distributed environments with heterogeneous resources.

Keywords: Static Scheduling, Fault Tolerance, Bi-objetive Scheduling, Distributed Systems

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Motivações e Problema

O crescimento maior do uso de sistemas de recursos heterogêneos e distribuídos para executar aplicações de alto desempenho tem levado a um estudo aprofundado das características destes sistemas e das aplicações executadas nestes ambientes, com objetivo de aproveitar ao máximo o poder computacional oferecido pelos recursos disponíveis. A literatura mostra que os ganhos obtidos em desempenho são elevados [3, 6, 43]. Atualmente, pode-se notar como diversas aplicações das ciências exatas e da engenharia, aquelas com alto consumo computacional e que manipulam enorme volume de dados [7, 42, 58, 73, 77], são executadas de forma razoavelmente eficiente em ambientes distribuídos de larga escala, como por exemplo, as grades computacionais. Para executar eficientemente, estas aplicações são paralelizadas e as distintas tarefas são distribuídas nos diferentes processadores do ambiente computacional.

Como a arquitetura de sistemas distribuídos atuais pode ser vista, como um conjunto de *clusters* de computadores heterogêneos conectados por redes metropolitanas, o ambiente alvo deste trabalho pode ser representado por um grande *cluster* de *cluster*s de computadores ou uma grade computacional. Esses ambientes tipicamente constituídos por recursos heterogêneos, também se caracterizam por apresentarem conexões mais rápidas intra-*clusters* e mais lentas entre máquinas de diferentes *clusters*. Isto faz com que tais ambientes possam ser modelados por meio de topologias hierárquicas e descentralizadas, onde a maneira de distribuir os processos de tarefas da aplicação e a forma de realizar a comunicação nos diferentes níveis da hierarquia representam importantes questões a ser abordadas. Neste trabalho, características destes sistemas como a velocidade dos processadores, a confiabilidade dos recursos, assim como, a latência de comunicação são levados em consideração.

O escalonamento de tarefas de uma aplicação paralela nos recursos disponíveis de um ambiente distribuído é crucial para atingir um bom desempenho. Este problema amplamente estudado é NP-completo [32], e pode ser dividido em escalonamento estático [46] e dinâmico [10]. O escalonamento estático de tarefas da aplicação, diferente do escalonamento dinâmico garante melhor planejamento e previsão dos custos antes de executar

a aplicação, mesmo com a ocorrência de falhas. Decisões podem ser tomadas com um tempo maior, empregando técnicas de escalonamento que garantem maior desempenho e consistência da execução. O escalonamento dinâmico exige um tempo menor, mas pode contemplar as variações no sistema com a tomada de novas decisões durante a execução da aplicação. Entretanto, o escalonamento dinâmico não têm uma visão global da execução, de maneira que uma decisão em determinado momento pode prejudicar decisões futuras.

As heurísticas de construção propostas para o problema de escalonamento estático [19, 46, 70], na maioria das vezes, especificam um escalonamento para aplicações representadas por *Grafos Acíclicos Direcionados* (GADs), o que é foco deste trabalho de tese. Os nós do grafo representam as tarefas da aplicação e os arcos a comunicação entre as tarefas. As tarefas são processos que recebem inicialmente dados, computam e enviam os resultados no final. Kwok em [46] propôs uma classificação para esse conjunto de heurísticas de acordo com os seguintes aspectos: pesos associados às tarefas; pesos associados aos arcos unitários ou não; custos associados à comunicação ou não; número de processadores limitados ou não; e processadores completamente conectados ou de acordo com uma certa topologia de rede. Dentre as heurísticas utilizadas, três estratégias de escalonamento se destacam: *list scheduling*, aglomeração de tarefas sem duplicação destas; e aglomeração com duplicação de tarefas.

As heurísticas do tipo *list scheduling*, em sua forma geral, apresentam baixa complexidade e manipulam mais facilmente um conjunto limitado e heterogêneo de processadores, sendo assim bastante utilizadas para o escalonamento de tarefas em redes de processadores heterogêneos, como *cluster* de processadores e até grades computacionais. Em um algoritmo do tipo *list scheduling* [19, 70], uma lista de tarefas ordenadas é construída atribuindo-se prioridade a cada tarefa. Tarefas selecionadas na ordem das suas prioridades são escalonadas nos processadores de forma que se minimize uma função de custo especificada.

Devido à crescente necessidade de alcançar alto desempenho, diferentes tipos de heurísticas de escalonamento de aplicações [10, 11, 38, 75] têm sido propostas com o objetivo de alcançar uma distribuição eficiente das tarefas da aplicação. Muitos destes algoritmos de escalonamento só minimizam o tempo de execução, embora pela natureza heterogênea das aplicações e da arquitetura, outros fatores afetem também a execução da aplicação, como por exemplo, a confiabilidade.

Além da heterogeneidade, a propensão à ocorrência de falhas em recursos deve ser atacada em sistemas distribuídos. Este problema têm sido extensivamente estudado [4, 5, 21, 25, 28, 33, 34, 35, 37, 39, 41, 48, 53, 60] e deve ser priorizado para a execução efetiva de aplicações. Exemplos e dados experimentais [63] através de estudos estatísticos sobre a confiabilidade dos recursos em sistemas heterogêneos atuais, destacam a importância deste problema. Pesquisas realizadas mostram a necessidade de novas abordagens para o tratamento das falhas que permitam uma execução com sucesso da aplicação.

Embora alguns algoritmos de escalonamento na literatura busquem formas de melhorar o tempo de execução das aplicações, considerando aspectos de confiabilidade e tolerância a falhas, muitos deles [4, 26, 48, 53, 59, 62, 74, 79] utilizam esquemas centralizados

e pouco flexíveis ou consideram ambientes homogêneos. Portanto, para sistemas heterogêneos e distribuídos, estes algoritmos propostos se tornam inapropriados, sendo necessário novas pesquisas com o objetivo de melhorar o desempenho e a confiabilidade das aplicações. Por exemplo, os mesmos podem sobrecarregar o tempo de execução ou até provocar a inconsistência da execução, quando os modelos não reproduzem as condições do ambiente.

1.2 Objetivos e Abordagem de Solução

Este trabalho de tese tem como objetivo realizar um estudo que relaciona duas problemáticas acima citadas: desenvolver uma heurística de escalonamento estático em conjunto com uma técnica de tolerância a falhas, para conseguir além de um bom desempenho, maior confiabilidade em sistemas heterogêneos e distribuídos.

Este trabalho aborda um problema de escalonamento bi-objetivo [26, 28, 60, 62], onde o tempo de execução da aplicação deve ser minimizado e ainda, a confiabilidade de sua execução deve ser maximizada. Por considerar um sistema distribuído com recursos heterogêneos, a função de custo além de integrada, deve ser flexível, com pesos associados aos objetivos, que possa ajustar o escalonamento das tarefas à heterogeneidade do sistema antes de executar a aplicação dada. Esta abordagem tem como objetivo encontrar entre múltiplas soluções de escalonamento, uma solução de compromisso adequada que leve a uma execução da aplicação perto do esperado.

Em geral, em um problema de otimização multi-objetivo em que pode haver objetivos conflitantes, uma solução pode ser a melhor do ponto de vista de um objetivo, mas não em relação aos demais objetivos [24]. Quando vários objetivos, provalvemente conflitantes são otimizados simultaneamente, não existirá uma solução ótima, mas sim um conjunto de possíveis soluções de qualidade ou não dominadas. Para se adotar uma boa solução, será necessário recorrer a informações adicionais e subjetivas que irão contribuir na escolha. No problema tratado neste trabalho, os dois objetivos podem ser conflitantes, ou seja, enquanto um processador pode finalizar a execução de uma tarefa da aplicação rapidamente, uma alta taxa de falha pode estar atribuída a este processador dependendo do modelo de confiabilidade.

Com o propósito de encontrar uma solução apropriada, neste trabalho de tese é proposta uma metodologia para a classificação e ajuste dos pesos na função de custo. A mesma permite avaliar o compromisso entre os objetivos do escalonamento com informações adicionais sobre um conjunto total de soluções.

A estratégia de escalonamento proposta deve funcionar para um modelo de uma aplicação paralela com restrições de precedência, representada por um Grafo Acíclico Direcionado (GAD) e para uma arquitetura distribuída com recursos heterogêneos propensa a falhas. Para uma melhor escolha dos recursos no sistema considerando tolerância a falhas, devem ser definidos novos critérios de escalonamento baseados em técnicas de replicação de tarefas da aplicação para múltiplas falhas permanentes de processador.

A estratégia de escalonamento de tarefas da aplicação proposta neste trabalho de tese é mais direcionada para ser utilizada por *midlewares*. Uma proposta inicial foi dada

em [64] como parte de uma ferramenta MPI. Ambientes distribuídos como as grades computacionais podem ser representados por um esquema de três níveis principais. A primeiro nível formado por componentes de *hardware* e *software* integrados em uma rede física de diferentes recursos. No segundo nível o *middleware* é composto de ferramentas e serviços responsáveis por disponibilizar os recursos do primeiro nível às aplicações do terceiro nível. As aplicações devem explorar os recursos disponíveis do sistema distribuído durante a execução através do *midleware*. Portanto, o objetivo principal do *middleware* é facilitar o trabalho do desenvolvedor ao fornecer um único ambiente de programação distribuída, integrado e consistente, que permite uma comunicação entre os outros dois níveis extremos. O *middleware* inclui um subnível de serviços que provê alto nível de abstração com ambientes de desenvolvimento de aplicações, ferramentas de programação, escalonamento de aplicações, tolerância a falhas, entre outros.

Em particular, o *middleware* SGA EasyGrid [23] permite que aplicação se adapte ao comportamento de uma grade computacional, considerando a disponibilidade dos recursos e as características específicas de cada aplicação. Para viabilizar a estratégia de escalonamento proposta, este trabalho tem como objetivo também a implementação de um mecanismo tolerante a falhas em MPI (*Message Passing Interface*) [72] no *middleware* SGA EasyGrid [23]. O mecanismo considera as informações do escalonamento estático, produzidas pela estratégia bi-objetivo, e permite a recuperação da aplicação na presença de falhas.

Em geral, os mecanismos de tolerância a falhas seguem distintos estágios para realizar o tratamento da falha: detecção da falha, localização da falha, contenção da falha e recuperação da falha. Logo que se detecta uma falha de processador, os processos gerenciadores no *middleware* SGA devem iniciar a recuperação, a partir do mecanismo proposto, baseado na estratégia de escalonamento estático com replicação passiva. Para executar a aplicação, os gerenciadores leêm os dados produzidos pelo escalonamento estático, como por exemplo, a ordem de execução das primárias e *backups*, os processadores onde estão escalonadas as tarefas e a lista de *backups* que devem executar quando determinada tarefa primária falhar. Este mecanismo deve tolerar falhas permanentes de processador, que como resultado provocam também falhas nos processos da aplicação: *falha por hardware* e *falha induzida*.

Como o SGA EasyGrid utiliza o modelo de execução 1PTask [67], onde os programas consistem de um largo número de processos de curta duração, determinados pelo paralelismo da aplicação e não pelo número de recursos, não é necessário utilizar técnicas de *checkpointing* para fazer a recuperação. Os beneficios de evitar a implementação de sofisticados esquemas de *checkpointing* e de manter longos *log*s de mensagens, podem compensar o custo de gerenciar um maior número de processos. Desta forma para recuperar a aplicação se justifica o uso e implementação de técnicas de replicação de processos no *middleware* SGA.

1.3 Contribuições

Em relação à estratégia de escalonamento de tarefas, foi proposta uma função de custo ponderada que integra dois objetivos: minimização do tempo de execução e maximização da confiabilidade, e que se diferencia das propostas anteriores nos seguintes pontos. A função é integrada e não hierárquica como em [60]. As funções integradas de outros trabalhos relacionados diferem na maneira como os objetivos são combinados e a maioria não apresenta ponderação de cada objetivo. Neste trabalho, a função proposta é ponderada como em [39], mas utiliza um operador e um metodo de normalização diferente ao associar os termos. Os resultados gerados a partir da função proposta neste trabalho mostram grandes vantagens em relação às funções de trabalhos correlatos da literatura.

No algoritmo é definida uma métrica D denominada *Diferença Média* dos objetivos, cujo própósito é auxiliar o ajuste dos pesos. A partir desta informação adicional gerada pelo escalonamento estático, o usuário pode tomar decisões para chegar a uma solução de escalonamento conveniente. Da mesma forma, é proposta também uma metodologia para avaliar os pesos na função de custo que utiliza a métrica proposta e conceitos de dominância. O método sugere como selecionar possíveis soluções de compromisso para o problema de escalonamento bi-objetivo, mais adequadas a determinados interesses do usuário.

O trabalho apresenta um estudo de casos com variações dos pesos que mostra a importância de utilizar uma abordagem flexível (múltiplas soluções) para escalonar aplicações em sistemas distribuídos heterogêneos. Na maioria dos casos, o algoritmo bi-objetivo proposto pode encontrar soluções eficientes e dominantes quando comparadas com outras heurísticas. Os resultados desta primeira parte podem ser vistos em [65, 66].

Em trabalhos correlatos, propostas tais como empregar uma metodologia para ajustar a função de custo e informações adicionais no algoritmo de escalonamento, não são nem consideradas. Diferentemente da literatura, este trabalho responde a questão de *como* achar determinada solução de escalonamento que represente um compromisso conveniente entre o tempo de execução e confiabilidade. A abordagem pode ser aplicada aos trabalhos já existentes, desde que se introduzam, nestes algoritmos de escalonamento, informações adicionais sobre os objetivos e um método de normalização adequado.

A segunda parte deste trabalho, adiciona tolerância a falhas no algoritmo de escalonamento. O escalonamento baseia-se no esquema primária-*backup* da técnica de replicação passiva, e inclui também a abordagem bi-objetivo ponderada proposta inicialmente, com o propósito de minimizar os custos da tolerância a falhas.

Para efetuar o escalonamento de tarefas *backups* foram definidos critérios de escalonamento e distintos métodos que fornecem flexibilidade, assim como dados importantes para garantir o funcionamento consistente e a recuperação eficiente da aplicação. Diferente da literatura com replicação passiva [60, 61], múltiplas falhas de processador podem ser toleradas. Além disso, a adição da ponderação nos objetivos oferece também flexibilidade ao algoritmo tolerante a falhas, sendo que distintas soluções de escalonamento com tolerância a falha podem ser geradas para uma mesma aplicação e cenário da arquitetura, assim como

compromissos com o número de falhas.

Finalmente, diferentemente de muitos trabalhos da literatura, avaliá-se a estratégia de escalonamento tolerante a falhas proposta sobre um ambiente real. Foram propostos e desenvolvidos mecanismos tolerantes a falhas em MPI que utilizam o escalonamento estático proposto baseado na técnica de replicação passiva. Os arquivos gerados pelos algoritmos propostos são lidos inicialmente e assim, a recuperação da aplicação é ativada em caso de falhas.

Desta forma, para recuperar grandes aplicações sobre sistemas computacionais de maior escala a estratégia foi integrada ao SGA EasyGrid da UFF [23], com a implementação de novas funcionalidades para explorar o gerenciamento hierárquico do *middleware*. A integração adicionou uma nova abordagem com replicação passiva para tolerar falhas ao SGA EasyGrid, e a estratégia proposta ganhou um funcionamento descentralizado e de maior escala. Assim, demostra-se a viabilidade da estratégia de escalonamento tolerante a falhas proposta e a importância de novas abordagens flexíveis e mais eficientes, para executar aplicações maiores sobre arquiteturas heterogêneas e distribuídas.

1.4 Organização

A tese está organizada como segue. O Capítulo 2 descreve a definição do problema e os modelos heterogêneos utilizados a longo da explicação do trabalho. Em seguida, o Capítulo 3 resume alguns trabalhos relacionados na literatura sobre escalonamento de tarefas e tolerância a falhas. Os Capítulos 4 e 5 apresentam as duas abordagens da estratégia de escalonamento proposta. Os mecanismos de recuperação desenvolvidos para a execução das aplicações sobre uma ambiente computacional real são descritos no Capítulo 6 com os resultados experimentais. Para finalizar, conclusões finais e perspectivas futuras são relacionadas no último capítulo.

Capítulo 2

Modelos e Definições

Neste capítulo são descritos os distintos modelos adotados ao longo deste trabalho, assim como é apresentado o problema que é abordado na tese.

2.1 Modelos Adotados

Heurísticas de escalonamento guiam suas decisões baseadas em caracteríticas tanto da aplicação quanto do sistema alvo, que efetivamente influenciam no desempenho. Com o intuito de representar tais características, os modelos da aplicação, arquitetural e de confiabilidade são especificados a seguir, considerando o objetivo almejado neste trabalho: especificar estratégias de escalonamento de aplicações paralelas, tolerantes as falhas, em ambientes com recursos heterogêneos distribuídos.

2.1.1 Modelo da Aplicação

Este trabalho considera aplicações paralelas modeladas por *Grafos Acíclicos Direcionados* (GAD), onde os nós do grafo representam as tarefas da aplicação e os arcos, a precedência entre estas. Um GAD $G = (V, E, e, c)$ modela uma aplicação paralela sendo que V é o conjunto de vértices que representam as tarefas, E a relação de precedência, $e(v_i)$ com $v_i \in V$, o peso de execução associado à tarefa v_i e, $c(v_j, v_i)$ com $(v_j, v_i) \in E$, o peso de comunicação associado ao arco (v_j, v_i).

A relação de precedência que define o GAD, indica que se $(v_j, v_i) \in E$ então, a execução de v_i não pode ser iniciada enquanto não seja completada a execução de v_j e os dados de v_j para v_i sejam recebidos por este. O conjunto de predecessores imediatos de v_i é denotado por $Pred(v_i)$, enquanto $Succ(v_j)$ representa o conjunto os sucessores imediatos da tarefa v_j.

A seguir são relacionados alguns exemplos de aplicações GADs utilizadas na tese. No primeiro caso, o GAD G_n é uma paralelização do método matemático Eliminação de Gauss que representa uma aplicação real, utilizada para solucionar sistemas de equações lineares, conforme especificado em [19, 70]. Uma característica das instâncias desta classe é que a estrutura destes grafos possuem pesos de computação variável, uma vez que esses pesos são maiores inicialmente na parte superior do GAD e diminuem a cada nível. A

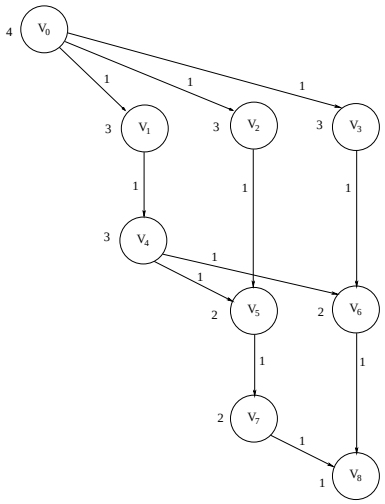


Figura 2.1: GAD da Eliminação de Gauss com 9 tarefas

Figura 2.1 mostra um exemplo, considerando uma matriz do sistema de equações lineares 4×4. Este tipo de GAD serve como modelo para estudar aplicações heterogêneas em relação as tarefas. A segunda classe R_n, são grafos gerados aleatóriamente que permitem estudar diferentes topologias irregulares. Já o GAD diamante Di_n, é utilizado para paralelizar a multiplicação de matrizes. Di_n caracteriza-se por apresentar uma estrutura regular, mas como em R_n, o GAD é homogêneo em relação ao tamanho das tarefas.

2.1.2 Modelo da Arquitetura

No modelo deste trabalho, $P = \{p_0, p_1, ..., p_{m-1}\}$ é o conjunto de m processadores heterogêneos, sendo que a cada p_j é associado o índice de retardo (*computational slowdown index*), denotado por $csi(p_j)$, conforme identificado em [51], sendo esta métrica inversamente proporcional ao poder computacional de p_j. Desta forma, o tempo de execução da tarefa v no processador p_j é dado por $eh(v, p_j) = e(v) \times csi(p_j)$. Para duas tarefas adjacentes v_i e v_j alocadas em processadores distintos p_l e p_k, respectivamente, supõe que o custo associado à comunicação de $c(v_i, v_j)$ dados é definido como $ch(v_i, v_j) = c(v_i, v_j) \times L(p_l, p_k)$, onde a latência $L(p_l, p_k)$ é o tempo de transmissão por *byte* sobre o *link* (p_l, p_k).

Um programa modelador [52] foi usado para capturar as características da arquitetura disponível, como a capacidade de processamento dos processadores *csi* e a latência associada aos canais de comunicação L. Estes valores são coletados antes da execução da aplicação do usuário.

2.1.3 Modelo de Confiabilidade

A confiabilidade de um sistema pode ser definida como a probabilidade que o sistema tem de realizar com sucesso sua função durante um certo período de tempo e sob condições definidas. Uma alta confiabilidade pode ser entendida como uma alta probabilidade de o sistema não falhar. Neste trabalho são consideradas somente falhas permanentes de processador. Supõe-se que estas falhas são eventos independentes entre si, e acontecem

de acordo com uma distribuição de *Poisson* com taxa ou probabilidade de falha $FP(p_j)$ $\forall\ p_j \in P$, com valor constante, conforme definido em [34, 61]. A taxa de falha de um processador p_j representa a quantidade de falhas por unidade de tempo que podem ocorrer em p_j.

O custo de confiabilidade $RC(v, p_j)$ de execução da tarefa v em um processador p_j é definido com a seguinte equação:

$$RC(v, p_j) = FP(p_j) \times eh(v, p_j) \tag{2.1}$$

A confiabilidade do escalonamento da tarefa em um processador é medida pelo custo de confiabilidade. Portanto, $RC(v, p_j)$ deve ser minimizado para que a confiabilidade se aproxime a 1. Desta maneira, o custo de confiabilidade de um processador baseado em um dado escalonamento de tarefas pode ser calculado como a soma dos custos de confiabilidade de suas tarefas. Assim, sendo $ltask(p_j)$ a lista de tarefas atribuídas a p_j, o custo de confiabilidade associado à p_j é $RC_p(p_j) = \sum_{\forall v \in ltask(p_j)} RC(v, p_j)$. Para um sistema heterogêneo com um total de m processadores, o custo de confiabilidade de escalonar uma aplicação nos processadores de P, pode ser definido como:

$$RC(G, P) = \sum_{\forall p_j \in P} RC_p(p_j) \tag{2.2}$$

Assim, a confiabilidade da aplicação G escalonada em P é dada pela equação:

$$R = e^{-RC(G,P)} \tag{2.3}$$

Com base neste modelo, escalonar as tarefas mais críticas, ou seja com maiores pesos de computação, em processadores com menores custos de confiabilidade (mais confiáveis) é uma forma de maximizar a confiabilidade R da aplicação no sistema alvo.

2.2 Definição do Problema e Estratégia Proposta

Em plataformas heterogêneas de grande escala, falhas de recursos (processadores ou *links*) podem ocorrer com certa frequência [63] com efeitos fatais sobre a execução das aplicações. Consequentemente, existe uma grande necessidade de desenvolver técnicas para tolerar falhas nestes sistemas.

Diversas heurísticas de escalonamento de tarefas vem sendo desenvolvidas na busca de melhores estratégias para executar a aplicação, embora muitas não incluam técnicas que considerem a ocorrência de falhas no ambiente. O escalonamento estático de tarefas é um problema amplamente estudado e uma vasta literatura mostra diversas formas de abordá-lo [19, 40, 46, 68, 70, 71, 75]. Heurísticas deste tipo podem ser agrupadas em diferentes grupos: *list scheduling*, clusterização, replicação e métodos de buscas aleatórias [46].

A classificação é feita de acordo com a maneira que as heurísticas distribuem, organizam e dão prioridades às tarefas ao associá-las com um determinado recurso, geralmente

com o objetivo de obter um melhor desempenho quando a aplicação é executada. Em particular, no grupo *list sheduling*, o escalonamento mantêm uma lista com todas as tarefas de um GAD de entrada, seguindo certas prioridades. Este tipo de heurística apresenta duas etapas principais: a seleção da tarefa, etapa onde é selecionada a tarefa pronta que apresenta maior prioridade para ser escalonada, e a seleção do processador, etapa para selecionar um processador disponível que minimiza uma função objetivo predefinida.

Muitas dessas heurísticas do tipo *list scheduling* existentes na literatura empregam simples modelos onde aspectos sobre confiabilidade e tolerância a falhas não são considerados. Geralmente, estes modelos assumem que os processadores e os *links* de comunicação são seguros. Portanto, com o objetivo de garantir a execução confiável e eficiente das aplicações, estudos realizados mostram a importância de revisar as heurísticas *list scheduling* tradicionais, quando utilizadas em ambientes heterogêneos de larga escala.

É importante ressaltar que um custo maior está associado ao uso de técnicas de tolerância a falha nos algoritmos de escalonamento. Se ambos problemas, escalonamento de aplicações e tolerância a falha, respectivamente, são problemas difíceis por si só em ambientes heterogêneos, abordá-los em conjunto [25, 28, 33, 34, 35, 37, 39, 41, 48, 53, 60] pode apresentar uma dificuldade ainda maior.

A estratégia de escalonamento proposta neste trabalho, de forma geral tem como objetivos principais:

1. Encontrar uma alocação estática de tarefas do GAD G sobre a arquitetura P que satisfaça o modelo descrito na Seção 2.1 (para ambientes distribuídos de larga escala) a partir de uma heurística simples como a *list scheduling*;

2. minimizar o tempo total de execução (*makespan*) da aplicaçã;.

3. maximizar a confiabilidade do sistema, ou seja, reduzir o custo de confiabilidade durante o escalonamento;

4. permitir que falhas permanentes de processador possam ser toleradas.

2.2.1 Escalonamento Bi-objetivo de Aplicações Paralelas

Com o objetivo de melhorar o desempenho e a confiabilidade da execução de aplicações em sistemas distribuídos, estudos realizados mostram a importância de estratégias flexíveis que se adaptem à heterogeneidade destes ambientes e das aplicações. No entanto, em ambientes heterogêneos propensos a falha, a maior parte das heurísticas de escalonamento não consideram a confiabilidade dos recursos ao escalonar as tarefas, o que pode aumentar a probabilidade de falha durante a execução da aplicação.

Neste trabalho para abordar o problema de escalonamento, o algoritmo proposto emprega uma heurística do tipo *list scheduling* com uma função de custo integrada que incorpora ambos objetivos, tanto a minimização do tempo de execução quanto a maximização da confiabilidade. A função de custo proposta é ponderada, tal que o escalonamento da aplicação considere os modelos especificados e ainda ofereça ao usuário, a possibilidade

de especificar suas preferências em relação à execução de sua aplicação em um ambiente heterogêneo propenso a falhas. O algoritmo também oferece flexibilidade na determinação da importância de cada objetivo ao variar os pesos na função de custos. Isto permite buscar resultados esperados de tempo de execução e confiabilidade da aplicação.

Usando a flexibilidade do algoritmo de escalonamento, é proposta também uma metodologia para ajudar ao usuário a encontrar um compromisso conveniente entre *makespan* e confiabilidade. Para isto, utiliza-se uma medida para estabelecer uma relação subjetiva do desequilíbrio entre os objetivos do problema. A abordagem proposta auxilia na decisão de determinado escalonamento, onde tipos de soluções distintas são definidas com o intuito de indicar escalonamentos que sejam de maior interesse para o usuário.

2.2.2 Escalonamento Tolerante a Falhas usando Replicação Passiva

Para escalonar grafos de precedência em uma arquitetura real considerando os aspectos anteriores, é proposto neste trabalho um algoritmo de escalonamento estático tolerante a falhas para recuperar falhas permanentes de processador. O algoritmo de escalonamento inclui também a abordagem bi-critério proposta antes, que além de minimizar o tempo total de execução, maximiza a confiabilidade no sistema. Para oferecer tolerância a falhas, o escalonamento é baseado em um esquema primária-*backup* da técnica de replicação passiva que, diferentemente da replicação ativa, utiliza mecanismos de detecção e manipulação de falhas.

A estratégia tolerante a falhas proposta inicialmente foi baseada no algoritmo de [61] por utilizar um *list scheduling* para escalonar as tarefas com uma técnica de replicação passiva para tolerar uma falha permanente de processador. No entanto, o algoritmo em [61] foi projetado para sistemas de tempo real onde as aplicações apresentam restrições de tempo (*deadlines*). Neste caso, a confiabilidade tem maior prioridade do que o tempo de execução, o que prejudica o *makespan* da aplicação. No trabalho proposto algumas definições e técnicas como a sobreposição de *backups*, utilizadas em outros trabalhos anteriores [33, 53, 61], são estendidas e reformuladas para funcionar no ambiente computacional considerado.

Na estratégia proposta, cada tarefa apresenta duas cópias denominadas primária (v^P) e *backup* (v^B). Este esquema primária-*backup* [61] é um dos mais importantes esquemas empregados em técnicas de escalonamento tolerante a falhas, onde cada versão da tarefa é escalonada sobre diferentes processadores. Assim, na técnica de replicação passiva, a *backup* pode ser executada se o processador, no qual a respectiva tarefa primária é atribuída, falhar. Só serão executadas aquelas *backups* das primárias que não terminaram de executar no processador que apresentou falha. Para melhorar a qualidade do escalonamento e reduzir mais seu tamanho (*makespan*), *backups* podem ser escalonadas sobrepostas em um mesmo processador, desde que não precisem executar ao mesmo tempo.

Em particular, na técnica de replicação ativa [8, 9], ambas versões da tarefa do esquema primária-*backup* são executadas ao mesmo tempo e não utilizam mecanismos de detecção e manipulação de falhas. Note que na replicação passiva [25, 33, 35, 37, 48, 53, 61],

a versão *backup* da tarefa é ativada somente se for detectada falha na versão primária.

Outras abordagens com novas classificações e critérios de escalonamento são introduzidas no algoritmo proposto e definem funções para oferecer tolerância a falhas, além de informações de saída necessárias para garantir a recuperação consistente e eficiente da aplicação em um ambiente real na presença de falha. Também diferente de outros trabalhos [60, 61], múltiplas falhas de processador podem ser toleradas.

Capítulo 3

Trabalhos Relacionados

Neste capítulo são abordados aspectos importantes relacionados com a proposta deste trabalho. Em geral são discutidas abordagens distintas de trabalhos existentes na literatura sobre escalonamento de tarefas de aplicações paralelas, ressaltando seus principais destaques e limitações que motivaram o desenvolvimento desta tese. Primeiro são relacionados trabalhos sobre escalonamento estático baseados em heurísticas construtivas que inicialmente só priorizavam o tempo de execução da aplicação. Na próxima seção, contribuições mais recentes sobre escalonamentos considerando múltiplos objetivos são apresentados. Para finalizar o capítulo, é feita uma análise sobre diferentes algoritmos de escalonamento tolerante a falha que utilizam replicação de tarefas.

3.1 Escalonamento Estático de Aplicações

As heurísticas de construção propostas para o problema de escalonamento estático, na maioria das vezes, especificam um escalonamento para aplicações representadas por *Grafos Acíclicos Direcionados* (GADs). Kwok em [46] propôs uma classificação para esse conjunto de heurísticas. Dentre os mecanismos utilizados nas heurísticas, três estratégias de escalonamento se destacam: *list scheduling*, aglomeração de tarefas sem duplicação destas; e aglomeração com duplicação de tarefas.

As heurísticas do tipo *list scheduling*, em sua forma geral, apresentam baixa complexidade e manipulam mais facilmente um conjunto limitado e heterogêneo de processadores, sendo assim bastante utilizadas para o escalonamento de tarefas em redes de processadores heterogêneos, como grades computacionais.

Dois algoritmos baseados na metodologia *list scheduling* têm sido bastante utilizados: o *Earliest Task First* (ETF) [40] e o *Heterogeneous Earliest First Task* (HEFT) [75]. O ETF escalona um GAD com pesos arbitrários em um conjunto limitado de processadores homogêneos. A análise apresentada em [40] mostra que ETF pode gerar escalonamentos ótimos quando custos de comunicação são negligíveis. Até o final da década de 90, muitos trabalhos baseados nesta classe de algoritmos foram propostos, considerando ambientes de processadores homogêneos.

Já nos últimos anos, observa-se um crescimento maior de heurísticas desenvolvidas

para sistemas heterogêneos. Em [75] foram apresentados algoritmos de escalonamento de aplicações representados por GAD em ambientes heterogêneos, sendo que o algoritmo HEFT supera os outros algoritmos, tanto em relação ao desempenho, quanto em relação à qualidade da solução gerada. HEFT primeiramente ordena todas as tarefas de acordo com uma certa prioridade. Posteriormente, seguindo a ordem determinada nesta lista, HEFT associa cada tarefa a um processador de forma que seja minimizado o tempo de fim da tarefa escolhida. HEFT, no entanto, realiza o procedimento de inserção da tarefa em espaços de tempo ociosos nos processadores que podem ser definidos ao longo do próprio processo de escalonamento.

Em [71], foi proposta uma extensão do *list scheduling* para o escalonamento de GADs em arquiteturas arbitrárias heterogêneas, incluindo mecanismos para escalonamento dos arcos do GAD (que representam as comunicações entre tarefas) nos canais do sistema. Algoritmos de escalonamento estáticos tais como apresentados em [12, 13, 45, 46, 47] consideram comunicação, mas não especificam a topologia de rede e contenção das redes de comunicação atuais. Macey em [49] mostra experimentalmente a importância de incluir estes aspectos para produzir escalonamentos mais eficientes. O trabalho em [71] escalona as mensagens da aplicação nos caminhos especificados pelos canais de comunicação da rede em questão, procurando minimizar o problema de contenção.

Em [68] foi desenvolvida uma ferramenta para escalonamento estático de processos em programas MPI, baseada em diferentes versões de heurísticas tipo *list scheduling*. A ferramenta disponibiliza heurísticas do tipo *list scheduling* como ETF, e de replicação como *Critical Path Fast Duplication* (CPFD) [1], para o escalonamento de tarefas em programas paralelos com o objetivo de reduzir os tempos de execução em ambientes MPI. As aplicações de entrada são representadas por GADs. A ferramenta proposta [68] permite o estudo de heurísticas de escalonamento baseadas em replicação de tarefas, mas não considera aspectos de confiabilidade.

Em geral, as heurísticas de escalonamento estático apresentadas não consideram a confiabilidade do sistema nem a possível ocorrência de falhas ao escalonar as tarefas, estes aspectos são analisados nas próximas seções.

3.2 Escalonamento Multi-objetivo

Quando lida-se com ambientes heterogêneos, além do tempo de execução a ser minimizado alguns trabalhos da literatura, como [4, 5, 28, 34, 39, 41, 60], também atacam o problema de confiabilidade no sistema, especificando em suas abordagens um problema de escalonamento bi-objetivo. Em ambientes distribuídos de grande escala como as grades computacionais, o uso de múltiplos objetivos para alocar os recursos torna-se uma abordagem ainda mais interessante. Note que nestes ambientes múltiplos fatores importantes intercedem durante a execução das aplicações paralelas.

Na literatura, existe uma variedade de trabalhos que abordam problemas de escalonamento bi-objetivo. Importantes contribuições e abordagens com relação a heurísticas construtivas de escalonamento bi-objetivo são resumidas na Tabela 3.1 ordenadas por ano.

Pode ser observado que muitas delas consideram sistemas de tempo real.

Tabela 3.1: Trabalhos relacionados de escalonamento multi-objetivo

Trabalho	**Ano**	**Abordagens**	**Contribuições**
Assayad *et al.* [5]	2004	Escalonamento estático, sistemas de tempo real, sistemas heterogêneos	Confiabilidade e escalonamento fixando objetivos, adaptação iterativa
Amin *et al.* [4]	2005	Escalonamento estático, sistemas de tempo real, sistemas homogêneos	Grafos *tandem* e *fork-join*, função de custo com critérios distintos
Qin and Jiang [59]	2005	Escalonamento dinâmico, sistemas de tempo real, sistemas heterogêneos	Minimizar custo de confiabilidade, rejeição de tarefas, ordenação de tarefas por *deadline*
Qin and Jiang [60]	2006	Escalonamento estático, sistemas de tempo real, sistemas heterogêneos	Função de custo hierárquica
Dongarra *et al.* [28]	2007	Escalonamento estático, sistemas heterogêneos	Estensão de *List scheduling*, taxa de falha $\times$ tempo de fim da tarefa
Hakem*et al.* [39]	2007	Escalonamento estático, sistemas heterogêneos	função de custo normalizada com pesos
Jeannot *et al.* [41]	2008	Escalonamento estático, tarefas independentes, sistemas heterogêneos	aproximação $(2+\delta, 1)$ do conjunto de Pareto
Girault *et al.* [34]	2009	Escalonamento estático sistemas heterogêneos	Algoritmo de dois passos: Aleatório seguido por uma fase *List scheduling*

Os autores em [5] apresentam uma heurística de escalonamento bi-objetivo para GADs, de acordo com os critérios: primeiro minimização do comprimento do escalonamento, e segundo a maximização da confiabilidade do sistema. Também utiliza replicação ativa de tarefas para melhorar a confiabilidade. Se os objetivos não são atingidos, então um parâmetro da função de compromisso pode mudar, e o algoritmo é re-executado até achar ambos requerimentos.

Em [4], uma função objetivo combina vários fatores, como *deadlines* para tempo real, confiabilidade, medidas quantitativas da comunicação e nível de paralelismo, numa função integrada. Neste trabalho, a taxa de aceitação total das aplicações e o efeito da confiabilidade sobre esta taxa são analisadas.

O trabalho em [59] apresenta uma heurística de escalonamento dinâmico de *jobs* paralelos em tempo real sobre *clusters* heterogêneos. O mesmo assume que os *jobs* são modelados por GADs e que chegam ao sistema seguindo um processo de *Poisson*. O algoritmo considera uma medida de confiabilidade e existe um controle de admissão: se não é garantido o *deadline* do *job*, então ele é rejeitado. Cada *job* é subdividido em tarefas que são escalonadas por um escalonador centralizado. O escalonamento é considerado dinâmico em relação aos *jobs*, embora seja na verdade estático em relação as tarefas. Para escalonar as tarefas, presentes antes do ínicio de cada etapa de execução, é utilizado *list scheduling*. Aspecto interessante considerado neste trabalho é a confiabilidade, embora use

um escalonador completamente centralizado.

Os algoritmos relacionados [4, 5, 34, 59, 60] não são projetados especificamente para sistemas distribuídos como as grades computacionais, os mesmos foram desenvolvidos para aplicações com restrições de tempo para sistemas de tempo real. Em particular, em [60] o algoritmo de escalonamento proposto ordena as tarefas da aplicação por *deadline*. Desta forma, não explora critérios de prioridades para escolha da tarefa e do processador. Os objetivos são manipulados de forma hierárquica, ou seja, verifica primeiro a condição de confiabilidade e somente, se é satisfeita, analisa então a condição de tempo. Como consequencia, este escalonamento favorece mais a confiabilidade e prejudica o tempo de execução da aplicação.

Em [28] um algoritmo de aproximação para tarefas independentes e unitárias é proposto, onde para obter um melhor compromisso entre os dois objetivos, o produto *taxa de falha* $\times$ *tempo de execução da tarefa* é especificado como critério para alocar tarefas não unitárias sobre processadores uniformes. Assim, a função de custo é baseada neste produto, e pode ser usada segundo [28], para incluir o conhecimento de confiabilidade em heurísticas de tipo *list scheduling* que só consideram a minimização do *makespan*. Os autores derivam a importância deste produto para o escalonamento de tarefas independentes e propõem uma extensão da heurística de escalonamento HEFT [75] para selecionar o processador que minimiza o produto do tempo de fim mais cedo de uma tarefa e o custo de confiabilidade de um processador. A função de custo não é ponderada produzindo uma única solução de escalonamento.

O trabalho [39] é outra abordagem bi-objetivo que associa pesos diretamente a seus objetivos em uma função de custo integrada. Os objetivos *makespan* e confiabilidade são normalizados por seus valores máximos e combinados na função. Seu algoritmo é somente comparado em relação a complexidade, com um algoritmo que não utiliza ponderação. Este trabalho não apresenta maneiras de escolher determinado compromisso que leve a um bom escalonamento.

Jeannot *et al.* [41] apresenta uma abordagem para a otimização simultânea do *makespan* e a confiabilidade para um conjunto de tarefas independentes ser escalonadas sobre um conjunto de processadores heterogêneos. A heurística proposta busca, para um valor fixo dado para o *makespan* $\mathcal{M}$, a solução com *makespan* no máximo $2\mathcal{M}$, e também soluções com confiabilidade ótima entre os escalonamentos com menores $\mathcal{M}$. Esta solução é utilizada para derivar uma aproximação $(2+\delta, 1)$ do conjunto de Pareto do problema, para qualquer $\delta > 0$.

Em Girault *et al.* [34], embora otimize ambos os objetivos, i.e., tempo de execução e confiabilidade, um algoritmo de dois passos é apresentado. Durante o primeiro passo, a confiabilidade é maximizada utilizando um algoritmo aleatório, que considera somente a alocação espacial. No segundo passo, o *makespan* é minimizado aplicando um algoritmo *list scheduling*. O algoritmo utiliza replicação ativa de tarefas para aumentar o desempenho, replicando também informação da confiabilidade.

É importante também destacar que existem trabalhos como [15] e [27] que empregam metaheurísticas para resolver o problema de escalonamento multi-objetivo. Como

esta abordagem usualmente requer mais tempo de computação para encontrar soluções, a mesma está fora do escopo deste trabalho e portanto não é considerada na Tabela 3.1.

3.3 Escalonamento e Replicação para Tolerar Falhas

Para aplicações de alto desempenho executadas em ambientes heterogêneos e distribuídos, é essencial que mecanismos de tolerância a falhas sejam aplicados. Falhas em processadores, canais e até mesmo de processos da própria aplicação podem ocorrer com uma certa freqüência em tais ambientes. Em particular, algoritmos de escalonamento atuais vêm considerando a heterogeneidade da arquitetura, mas muitas vezes não incorporam tolerância a falhas nem aspectos sobre a confiabilidade do sistema. Da mesma forma, existem trabalhos da literatura que não consideram informações geradas pelo escalonamento da aplicação dentro dos mecanismos de tolerância a falhas propostos, que possa auxiliar no tratamento das falhas. Assim, os mecanismos abordam o tratamento de falhas sem considerar como a alocação das tarefas da aplicação aos processadores poderia maximizar a confiabilidade e o desempenho da execução da aplicação.

A replicação é uma técnica amplamente usada é importante para atingir alta disponibilidade e tolerância a falhas em sistemas distribuídos. Já trabalhos como [17, 37, 50] usam replicação para tolerar falhas. Em [37], por exemplo, são introduzidos conceitos importantes em relação a tolerância a falha por replicação sobre sistemas distribuídos. O artigo descreve dois tipos de técnicas de replicação, passiva e ativa, mostrando as vantagens da replicação passiva. No modelo passivo existe, em um determinado momento uma réplica primária e um ou mais *backups* ou escravos. A primária executa as operações e envia cópias dos dados atualizados para ser utilizados pelas *backups*. Se a primária falhar, uma das *backups* sera promovida para atuar como primária. A replicação passiva é um ponto de interesse em diversas pesquisas e está fortemente relacionada a este trabalho de tese, tendo como objetivo incorporar tolerância à falhas a escalonadores em ambientes distribuídos.

Com o objetivo de atingir alto desempenho através de estratégias de escalonamento considerando a ocorrência de falhas em ambientes distribuídos, trabalhos como [25, 50, 33, 44, 48, 53] integram heurísticas de escalonamento com mecanismos que abordam tolerância a falhas. Muitas destas propostas usam mecanismos de replicação passiva, fazendo uso de esquemas primária-*backup* das tarefas da aplicação. Como o problema de escalonamento de tarefas é NP-completo [57], boas heurísticas são necessárias para escalonar as tarefas primárias e suas *backups*.

[25, 33, 35, 37, 48, 53] utilizam um esquema primária-*backup* da técnica de replicação para abordar problemas de escalonamento dinâmico. Já trabalhos da literatura que propõem algoritmos de escalonamento estático baseados no mesmo esquema, em muitos casos são projetados para sistemas de tempo real como em [60, 61], ou utilizam técnicas de replicação ativa como em [2, 8]. Na Tabela 3.2 são relacionadas algumas contribuições de algoritmos de escalonamento que utilizam técnicas de replicação de tarefas.

Em [33], qualquer heurística dinâmica para escalonar as tarefas sobre sistemas de tempo real pode ser utilizada, tendo como objetivo principal estudar técnicas de replicação

Tabela 3.2: Trabalhos relacionados de escalonamento tolerante a falhas

Trabalho	**Ano**	**Abordagens**	**Contribuições**
Ghosh *et al.* [33]	1997	Escalonamento dinâmico, sistemas de tempo real não distribuído, tarefas independentes	Múltiplas falhas, *desallocation* e *overloading* de *backups*
Naedele *et al.* [53]	1999	Escalonamento dinâmico, sistemas de tempo real, sistemas homogêneos, tarefas independentes	Análise de desempenho das heurísticas, escalonamento adaptativo
Liberato *et al.* [48]	2000	Restrições de precedência, sistemas de tempo real, sistemas homogêneos	Múltiplas falhas de tarefas transientes
Qin and Jiang [61]	2002	Escalonamento estático, sistemas de tempo real, sistemas heterogêneos, restrições de precedência	Uma falha permanente de processador, função de custo hierárquica (confiabilidade) sobreposição de *backups*, classificação de primárias
Girault *et al.* [2]	2003	Escalonamento estático, sistemas de tempo real, sistemas heterogêneos, restrições de precedência, replicação ativa	Múltiplas falhas permanentes de processador
Qin and Jiang [60]	2006	Escalonamento estático, sistemas de tempo real, sistemas heterogêneos, restrições de precedência	Sobreposição de primárias e *backups*
Benoit *et al.* [8]	2008	Escalonamento estático, sistemas heterogêneos, restrições de precedência, replicação ativa	Múltiplas falhas permanentes de processador, redução da comunicação

passiva para tolerar falhas. Neste artigo múltiplas cópias de uma tarefa são escalonadas para tolerar múltiplas falhas separadas por intervalos de tempo, utilizando as técnicas *desallocation* e *overloading*. *Desallocation* de *backups* libera os recursos reservados pelas *backups* uma vez que as primárias correspondentes tenham completado a execução. *Overloading* é o escalonamento de mais de uma cópia no mesmo intervalo de tempo sobre o mesmo processador. As técnicas foram avaliadas com simulações, mostrando um uso eficiente dos recursos durante o escalonamento de primárias e *backups*, embora sejam necessários algoritmos de escalonamento rápidos e simples para as *backups*. A proposta não considera restrições de precedência das tarefas e o ambiente não é distribuído.

Já em [53], investiga-se o desempenho de diferentes heurísticas de escalonamento que usam o esquema primária-*backup*. Três heurísticas de seleção do processador foram comparadas: busca seqüencial, seleção baseada na carga e seleção aleatória do candidato. O algoritmo baseado na carga obteve melhor desempenho em relação ao número de tarefas rejeitadas, mas ambos, busca sequencial e seleção baseada em carga, se comportaram

melhor que seleção aleatória. Foi concluído que, embora muitos trabalhos deste tipo não concedam importância a heurística de escalonamento, a escolha da heurística terá um papel fundamental no desempenho. Também foi recomendado o uso de um escalonador adaptativo que monitore a variação dos parâmetros, ajustando em cada caso a estratégia conveniente. Só aplicações com tarefas independentes foram consideradas e escalonadas *dinamicamente* sobre um conjunto de processadores homogêneos.

Em [48] é apresentado um esquema de escalonamento tolerante a falhas onde as tarefas são consideradas com restrições de precedência, e múltiplas falhas transientes podem ser toleradas, mas o sistema é homogêneo. O interessante deste trabalho foi conseguir a recuperação de múltiplas falhas de tarefas, e o desenvolvimento de uma solução ótima usando a política de escalonamento tolerante a falhas *Earliest-Deadline-First* (EDF). O trabalho aborda o problema: dado um conjunto de n tarefas aperiódicas, é possível determinar se cada tarefa no conjunto pode completar a sua execução antes do *deadline* usando escalonamento EDF, mesmo quando o sistema precisa recuperar-se para k falhas.

[59, 61] mostram um algoritmo onde tarefas de uma aplicação com restrições de precedência são escalonadas em um ambiente heterogêneo. A abordagem considera não somente a heterogeneidade de recursos de processamento e comunicação, como também características de confiabilidade do sistema. Para tolerar uma falha permanente de processador, este algoritmo utiliza também um esquema primária/*backup* das tarefas da aplicação. Para melhorar a qualidade do escalonamento da aplicação, a heurística considera a possibilidade de sobreposição da execução de *backups* de diferentes tarefas no mesmo processador. O algoritmo tem como objetivo minimizar o tempo de execução da aplicação e o custo de confiabilidade associado a esse escalonamento. A função de custo prioriza a confiabilidade e só em caso de empate considera o tempo. Em [59], diferentemente de [61], adiciona-se a sobreposição de *backups* com primárias e melhora-se a sobreposição entre *backups* de [61]. Os escalonamentos em ambos os trabalhos apresentam inconsistências nas suas formulações, como será explicado no Capítulo 5.

Diferentemente dos trabalhos anteriores, [2, 8] utilizam replicação ativa. Nesta técnica, múltiplas cópias de uma tarefa primária executam em paralelo para alcançar a tolerância a falhas. Mesmo que esta abordagem não seja considerada no escopo deste trabalho, pois acrescenta uma sobrecarga de computação e comunicação durante a execução da aplicação, seus esquemas são também analisados. Ambos algoritmos [2, 8] são estáticos e toleram múltiplas falhas permanentes de processador para aplicações com restrições de precedência das tarefas em ambientes heterogêneos. No entanto, durante o escalonamento são alocadas múltiplas cópias de uma tarefa (como número de falhas) em diferentes processadores aumentando a sobrecarga da execução. Em particular, em [2] o algoritmo é orientado a sistemas de tempo real, e em [8] para reduzir o problema de contenção o algoritmo aborda a minimização de comunicações induzidas por usar a técnica de replicação ativa.

Capítulo 4

Escalonamento Bi-objetivo Proposto

Estudos realizados mostram a importância de estratégias de escalonamento flexíveis que se adaptem à heterogeneidade das aplicações e das arquiteturas como as de grades computacionais, visando melhorar o desempenho e a confiabilidade da execução nesses ambientes. Portanto, neste capítulo é proposto um algoritmo de escalonamento bi-objetivo que emprega uma heurística do tipo *list scheduling* com uma função de custo integrada diferente de [28, 39, 60, 61]. A função incorpora ambos os objetivos, tanto a minimização do tempo de execução quanto a maximização da confiabilidade.

O escalonamento da aplicação é também ponderado e tem como premisa os modelos especificados no Capítulo 2. A função de custo é proposta ponderada para permitir a especificação de distintas preferências em relação aos objetivos do escalonamento, antes de executar a aplicação. Com a possibilidade de variar os pesos na função, é possível obter resultados diferentes de tempo final de execução e confiabilidade da aplicação para um mesmo ambiente heterogêneo propenso a falhas. Um estudo das variações de pesos é realizado e mostra a importância de considerar uma abordagem ponderada de escalonamento flexível em ambientes de larga escala como as grades computacionais.

Aproveitando a flexibilidade do algoritmo de escalonamento pela função ponderada, é definida uma métrica *Diferença Média D*, introduzida no escalonamento para estabelecer uma relação entre o *makespan* e a confiabilidade. Esta informação adicional é calculada com o objetivo principal de auxiliar no ajuste dos pesos da função de custo. O usuário pode tomar diferentes decisões até escolher uma solução de escalonamento conveniente. Em particular, para ajudar ao usuário a achar um compromisso de maior qualidade, é proposta uma classificação, onde tipos distintos de soluções podem ser obtidos com o intuito de indicar escalonamentos de maior interesse. A metodología utiliza a métrica D e conceitos de dominância, que sugerem possíveis soluções mais adequadas para o problema bi-objetivo. Em trabalhos correlatos [4, 5, 28, 34, 39, 41, 60], aspectos como estes não são considerados.

4.1 Estratégia de Escalonamento Proposta

Nesta seção é apresentada a abordagem de escalonamento bi-objetivo proposta neste trabalho. A seguir, são descritos com detalhes as principais etapas, a função de custo e os passos que conformam o algoritmo de escalonamento.

Em heurísticas do tipo *list scheduling* sobre ambientes heterogêneos, a escolha da tarefa usando o critério de prioridade *blevel* tem mostrado melhor desempenho para um número maior de GADs de aplicações [75]. O uso deste critério com GADs é possível já que a ordem decrescente dos *blevels* provê uma ordem topológica linear que preserva as restrições de precedência do grafo. Ao priorizar tarefas mais críticas, em relação a computação e comunicação, para escalonar em processadores mais confiáveis e de forma que terminem mais cedo, aumenta-se o desempenho, e ainda a probabilidade de a aplicação não falhar.

Algoritmo 1 : *Biobj-framework(G,P,w)*

1 $V_{ordG} = \langle v_0, \ldots, v_{n-1} \rangle / blevel(v_i) \leq blevel(v_{i+1}), i = 0, \ldots, n-2;$
2 $\langle\ Sch,\ S(Sch),\ D(Sch)\ \rangle$=MRCD($V_{ordG}$,P,w);

O algoritmo de escalonamento estático bi-objetivo ponderado proposto é chamado de *Makespan Reliability Cost Driven* (MRCD) e faz parte do *Biobj-framework* no Algoritmo 1. Primeiramente, as tarefas são ordenadas por seus *blevels* (*static bottom level*), conforme [75], denotado por $blevel(v), \forall v \in V$. O critério de prioridade *blevel* para selecionar uma tarefa v de um GAD, pode ser definido como o maior caminho que existe entre v e outra tarefa fim (sem sucessores), considerando a média dos tempos de execução $\overline{eh(v)}$ em P, e a média de comunicação $\overline{ch(v,v')}$ em qualquer canal. Assim, este critério é computado como:

$$blevel(v) = \begin{cases} \overline{eh(v)} & \text{se } Succ(v) = \emptyset,\ v \in V \\ max_{v' \in Succ(v)} \left\{ \overline{eh(v)} + \overline{ch(v,v')} + blevel(v') \right\} & \text{se } Succ(v) \neq \emptyset,\ v \in V \end{cases}, \tag{4.1}$$

onde

$$\overline{eh(v)} = e(v) \times \frac{\sum_{\forall p_j \in P} csi(p_j)}{|P|} \tag{4.2}$$

$$\overline{ch(v,v')} = c(v,v') \times \frac{\sum_{\forall (p_l,p_k)} L(p_l,p_k)}{|E|} \tag{4.3}$$

Com as prioridades calculadas na lista V_{ordG}, o algoritmo MRCD executa o escalonamento Sch das tarefas da aplicação G. Primeiramente escalona as tarefas usando uma função de custo ponderada que integra os dois objetivos tempo de fim e custo de confiabilidade. Depois, calcula uma informação adicional definida como *Diferença Média* $D(Sch)$ que representa uma relação entre os objetivos do escalonamento Sch. Esta informação tem o propósito de auxiliar o usuário durante o ajuste dos pesos na função de custo, caso deseje buscar uma nova solução de escalonamento $S(Sch)$.

4.1.1 Função de Custo Ponderada

Na primeira etapa do algoritmo MRCD, a lista V_{ordG} de G ordenada por $blevel(v)$ é percorrida e, para cada tarefa $v \in V$, é efetuada uma procura pelo melhor processador, de acordo com os seguintes critérios. Seja $v \in V$ a próxima tarefa a ser escalonada, o melhor processador $p_j \in P$ a ser escolhido para execução de v é aquele que minimiza a função de custo

$$F(v,p) = min_{\forall p_j \in P} \left\{(1-w)EFT(v,p_j) + wRC(v,p_j)\right\}, \tag{4.4}$$

ou seja, é escolhido o processador p que satisfaz $F(v,p)$, com $0 \leq w \leq 1$.

O tempo disponível mais cedo para a tarefa v ser escalonada no processador p_j é denotado por $EAT(v,p_j(v))$ (tempo em que v recebe todas mensagens das tarefas predecessoras), e serve para calcular o tempo de ínicio $EST(v,p_j(v))$ de v no processador $p_j(v)$. $EST(v,p_j(v))$ é o menor tempo que v pode ser escalonada em p_j. Para calcular $EST(v_i,p_j(v_i))$, utiliza-se uma política de inserção de tarefas em espaços ociosos do processador, conforme [75]. A partir de $EST(v,p_j(v_i))$ calcula-se o tempo de fim da tarefa v escalonada em $p_j(v)$, da forma $EFT(v,p_j(v)) = EST(v,p_j(v)) + et(v,p_j(v))$. Note que para computar $EFT(v,p_j(v))$, todas as tarefas predecessoras imediatas de v devem ter sido escalonadas.

A função de custo proposta tem pesos associados a cada objetivo agregado, tempo de fim $EFT(v,p_j)$ e custo de confiabilidade $RC(v,p_j)$, pesos esses que representam o nível de importância dado a cada um deles, no problema de escalonamento. O custo de confiabilidade $RC(v,p_j)$ é calculado conforme definido no modelo de confiabilidade. Depois de todas as tarefas v_i da aplicação G serem escalonadas, o *makespan* do escalonamento da aplicação denotado por Sch, pode ser calculado como $\mathcal{M}(Sch) = max_{\forall v_i \in V} \{EFT(v_i,p(v_i))\}$.

O objetivo da função ponderada é encontrar um peso estático conveniente w para essa aplicação executar naquele ambiente. Analisando os casos extremos de w, se $w = 0$ obtemos o algoritmo HEFT [75] e significa que o termo custo de confiabilidade não é considerado, sendo somente o *makespan* minimizado. Por outro lado, se $w = 1$ o escalonamento é especificado somente de acordo com a maximização da confiabilidade. No entanto, o interesse é achar um compromisso entre os dois objetivos.

Um problema inerente em uma função de custo é que, valores de diferentes objetivos podem não ser comparáveis entre si, o que inviabiliza sua agregação imediata. Para resolver este problema, os valores de $EFT(v,p_j)$ e $RC(v,p_j)$ para cada tarefa v são normalizados numa mesma escala sobre o conjunto total de processadores P. Para garantir a mesma magnitude dos objetivos, foi utilizada a normalização da amplitude que consiste em transformar todas as variáveis de modo a que partilhem do mesmo valor mínimo e máximo. Uma forma simples consiste em aplicar um operador a todas as variáveis objetivos. Seja O^i, o i-ésimo objetivo, correspondente ao conjunto de objetivos sendo tratados, e sejam O^i_{min} and O^i_{max} os valores mínimo e máximo de O^i, respectivamente. O valor normalizado do i-ésimo objetivo é então

$$O_n^i = norm(0, 100, O_{min}^i, O_{max}^i) = 100 \times \frac{O_i - O_{min}^i}{O_{max}^i - O_{min}^i}. \quad (4.5)$$

Neste trabalho, $EFT(v, p_j)$ e $RC(v, p_j)$ foram normalizados na função de custo (Equação 4.4) aplicando a Equação 4.5. Detalhes do algoritmo são apresentados a seguir.

4.1.2 Algoritmo de Escalonamento Bi-objetivo

O Algoritmo 2 mostra os passos do algoritmo MRCD para um dado $G = (V, E, e, c)$, e um conjunto de processadores P, conforme os modelos da aplicação e arquitetura adotados. A partir da lista de tarefas V_{ordG} ordenada em ordem crescente de $blevel(v)$. A etapa de escalonamento é executada para cada $v_i \in V_{ord}$ da linha 2 até a 19, onde o tempo de fim mais cedo $EFT(v_i, p_j)$ e o custo de confiabilidade da tarefa $RC(v_i, p_j)$ são calculados para cada $p_j \in P$ (linhas 3 a 6). Entre as linhas 8 e 10, os valores mínimo e máximo para $EFT(v_i, p_j)$ e $RC(v_i, p_j)$ são calculados para todo $p_j \in P$. Logo depois, é aplicada a normalização da Equação 4.5 para calcular os objetivos normalizados $EFT_n(v_i, p_j)$ e $RC_n(v_i, p_j)$ na função de custo $f_n(v_i, p_j)$ da linha 16. Assim, o processador $p_j = p_{v_i}$ que minimiza f_n para escalonar v_i, é identificado.

Dado um valor w para o parâmetro de ponderação da função de custo, o algoritmo MRCD gera um escalonamento Sch, conforme visto na linha 19 e ainda com um conjunto solução $S(Sch) = (\mathcal{M}, R_T)$. O *makespan* $\mathcal{M}(Sch)$ e a confiabilidade total $R_T(Sch)$ do escalonamento Sch são calculados nas linhas 17 e 20, respectivamente. Para ajudar na busca de uma solução de compromisso entre os diferentes valores de w, uma informação adicional, denotada *Diferença Média* $D(Sch)$, é calculada na linha 21. O sinal de $D(Sch)$ mostra qual objetivo foi mais priorizado para a especificação ou saída produzida de escalonamento. Se o sinal é negativo, o custo de confiabilidade foi privilegiado no escalonamento construído para v_i; caso contrário, a minimização do *makespan* foi mais favorecida. Porém, se o valor de $D(Sch)$ é próximo a zero significa que o escalonamento, em média, priorizou igualmente ambos objetivos, ou seja, não houve muita diferença entre eles. Mais detalhes sobre a métrica $D(Sch)$ são apresentados a seguir.

Algorithm 2 : *MRCD(V_{ordG},P,w)*

1 for $i = 0, \ldots, n-1$
2 $\quad F = \infty$;
3 $\quad \forall p_j \in P$
4 $\quad\quad$ Calculate $EST(v_i, p_j)$ using $taskInsertion(v_i, p_j)$;
5 $\quad\quad EFT(v_i, p_j) = EST(v_i, p_j) + eh(v_i, p_j)$;
6 $\quad\quad RC(v_i, p_j) = FP(p_j) \times eh(v_i, p_j)$;
7 $\quad EFT_{min} = \min_{p_j \in P}\{EFT(v_i, p_j)\}$;
8 $\quad EFT_{max} = \max_{p_j \in P}\{EFT(v_i, p_j)\}$;
9 $\quad RC_{min} = \min_{p_j \in P}\{RC(v_i, p_j)\}$;
10 $\quad RC_{max} = \max_{p_j \in P}\{RC(v_i, p_j)\}$;
11 $\quad \forall p_j \in P$
12 $\quad\quad EFT_n(v_i, p_j) = norm(0, 100, EFT_{min}, EFT_{max}, EFT(v_i, p_j))$;
13 $\quad\quad RC_n(v_i, p_j) = norm(0, 100, EFT_{min}, RC_{max}, RC(v_i, p_j))$;
14 $\quad\quad f_n(v_i, p_j) = (1 - w) \times EFT_n(v_i, p_j) + w \times RC_n(v_i, p_j)$;
15 $\quad\quad$ if $(f_n(v_i, p_j) < F_n)$
16 $\quad\quad\quad F_n = f_n(v_i, p_j)$; $p_{v_i} = p_j$;
17 $\quad$ if $(\mathcal{M}(Sch) < EFT(v_i, p_{v_i}))$ $\mathcal{M}(Sch) = EFT(v_i, p_{v_i})$;
18 $\quad RC_s(Sch) = RC_s(Sch) + RC(v_i, p_{v_i})$;
19 $\quad Sch = Sch \cup \langle v_i, p_{v_i}, EST(v_i, p_{v_i})\rangle$;
20 $R_T(Sch) = e^{-RC_s}$;
21 $D(Sch) = \frac{\sum_{v_i \in V} RC_n(v_i, p_{v_i}) - EFT_n(v_i, p_{v_i})}{n}$;

4.2 Classificação dos Pesos Aplicada à Função de Custo

No algoritmo de escalonamento estático MRCD proposto, cada tarefa v da aplicação é escalonada no processador que minimiza a função de custo ponderada da Equação 4.4, mas normalizada, para um valor de w dado. Executando MRCD para diferentes valores de w, são gerados escalonamentos distintos para uma mesma aplicação G e arquitetura P. Para buscar uma solução de escalonamento apropriada, é possível analisar o resultado obtido pelo escalonamento e prever novas soluções para novos valores de *makespan* e confiabilidade. Porém, uma questão importante [28] e pouco explorada na literatura é como determinar uma solução de escalonamento que represente um compromisso esperado entre o *makespan* e a confiabilidade.

Analisar os escalonamentos gerados pelo algoritmo bi-objetivo ponderado para uma variedade significativa de valores dos pesos (w) pode ser impraticável devido à enorme quantidade de soluções possíveis, o que torna o problema de difícil solução. Desta forma, é necessário otimizar esta busca, e utilizar critérios para escolher qual é a solução adequada em um dado momento, entre soluções de mais qualidade, como será visto na seção. Para

isto, podem ser consideradas informações adicionais que permitam analisar por exemplo, características do escalonamento produzido, interesses do usuário, informações sobre a aplicação e a arquitetura, entre outros aspectos.

Em geral, em um problema de otimização multi-objetivo em que pode haver objetivos conflitantes, uma solução pode ser a melhor do ponto de vista de um objetivo, mas não em relação aos demais objetivos [24]. Quando vários objetivos, provalvemente conflitantes são otimizados simultaneamente, não existirá uma solução ótima, mas sim um conjunto de possíveis soluções de qualidade ou não dominadas. Para se adotar uma boa solução, será necessário recorrer a informações adicionais e subjetivas que irão contribuir na escolha. No problema tratado neste trabalho, os dois objetivos podem ser conflitantes, ou seja, enquanto um processador pode finalizar a execução de uma tarefa da aplicação rapidamente, uma alta taxa de falha pode estar atribuída a este processador dependendo do modelo de confiabilidade.

Com o propósito de encontrar uma solução apropriada, neste trabalho de tese é proposta uma metodologia para a classificação e ajuste dos pesos na função de custo. A mesma permite avaliar o compromisso entre os objetivos do escalonamento com informações adicionais sobre um conjunto total de soluções.

4.2.1 Diferença Média dos Objetivos no Escalonamento

Com o objetivo de ajudar a encontrar uma solução de compromisso entre o *makespan* $\mathcal{M}$ e a confiabilidade R_T, a métrica $D(Sch)$ é uma informação adicional, gerada pelo algoritmo que permite escolher determinado w. Esta informação permitirá usar critérios para atribuir pesos (w) diferentes aos objetivos do problema e reajustar a função de custo na busca de uma nova solução de compromisso. A seguir, são formalizados alguns conceitos que conduzem à classificação proposta, definindo melhor esta métrica.

Seja $v_i \in V$ uma tarefa qualquer do escalonamento Sch em P obtido por MRCD. O processador $p_{v_i} \in P$ que tem v_i alocada nele é aquele que minimiza a função de custo da Equação (4.4) na forma normalizada, ou seja, que satisfaz $F(v_i, p_{v_i})$ para os objetivos normalizados $EFT_n(v_i, p_{v_i})$ e $RC_n(v_i, p_{v_i})$ calculados para v_i com a Equação 4.5.

Definição 1. Sejam $EFT_n(v_i, p_{v_i})$ e $RC_n(v_i, p_{v_i})$ de $F(v_i, p_{v_i})$ para cada v_i em Sch e seja $RC_n(v_i, p_{v_i}) - EFT_n(v_i, p_{v_i})$ a *diferença parcial* em Sch para v_i escalonada em p_{v_i}. A ***Diferença Média*** $D(Sch)$ entre os objetivos EFT_n e RC_n em Sch é então definida como a média de todas as *diferenças parciais* para todo $v_i \in V$ em Sch e calculada como

$$D(Sch) = \frac{\sum_{v_i \in V} RC_n(v_i, p_{v_i}) - EFT_n(v_i, p_{v_i})}{n}, \tag{4.6}$$

onde p_{vi} é o processador em que cada v_i está alocada de acordo com Sch.

Esta definição é baseada na normalização aplicada com a Equação 4.5 na função de custo, que permite distribuir os valores de um objetivo de forma similar na nova escala entre

0 e 100. Como a distância entre os objetivos normalizados na nova escala é proporcional à distância dos objetivos na escala real, esta abordagem permite medir a distância de cada objetivo aos seus extremos, mínimo e máximo. Desta forma, analisando o valor normalizado escolhido pelo escalonamento da tarefa, é possível saber quanto foi priorizado um objetivo dentre seus possíveis valores nos distintos recursos de P. Como o escalonamento MRCD considera a minimização dos objetivos EFT e RC, se o valor normalizado ficou próximo do mínimo 0, então o objetivo correspondente foi altamente priorizado, quando próximo a 100, o contrário ocorre.

A diferença parcial entre os objetivos normalizados para v_i, $RC_n(v_i, p_{v_i}) - EFT_n(v_i, p_{v_i})$ indica quanto $EFT_n(v_i, p)$ e $RC_n(v_i, p)$ foram desigualmente priorizados no escalonamento de v_i. Se o valor absoluto desta diferenca é alta um dos objetivos foi muito mais privilegeado que o outro, e o sinal indica qual deles: no caso negativo, RC foi priorizado; se positivo, EFT foi priorizado.

Desta forma, calculando a média de todas as diferencas parciais entre os objetivos normalizados das tarefas v_i, este valor equivale a $D(Sch)$ na Equação 4.6, e indica pela análise anterior quanto em média foram desigualmente priorizados os objetivos normalizados para o conjunto total de escalonamentos de tarefas v_i em Sch.

Uma vantagem desta métrica $D(Sch)$ é que pode ser utilizada para obter informação sobre determinado escalonamento, sem necessariamente conhecer as outras soluções do problema. Cada escalonamento Sch_w, para cada w, calcula seu proprio $D(Sch_w)$ que informa como tomar decisões, mesmo sem conhecer a saida dos outros escalonamentos. Note que com o uso da função ponderada, inúmeras soluções são possíveis, o que dificulta a decisão de escolha de uma determinada solução. Com a idéia de se chegar a um compromisso conveniente, é possivel, com esta métrica avaliar, a solução e saber qual dos objetivos no escalonamento está sendo mais priorizado e em que medida. Um exemplo com um estudo de caso é mostrado depois na Seção 4.3.

4.2.2 Conceitos de Dominância

Nesta subseção são apresentadas a seguir algumas definições que serão utilizadas durante a classificação e ajuste dos pesos.

Definição 2. Seja S o conjunto de todas as soluções viáveis/factíveis para o problema bi-objetivo abordado. Sejam $(Sch_k, \mathcal{M}(Sch_k), R_T(Sch_k), D(Sch_k))$ e $(Sch_q, \mathcal{M}(Sch_q), R_T(Sch_q), D(Sch_q))$ duas soluções de S com suas correspondentes informações adicionais $D(Sch)$ associadas. A solução $S_k \in S$ **domina** a solução $S_q \in S$ se as seguintes condições são satisfeitas:

1. S_q não é melhor que S_k nos dois objetivos, ou seja, $\mathcal{M}(Sch_k) \leq \mathcal{M}(Sch_q)$ e $R_T(Sch_k) \geq \mathcal{M}(Sch_q)$
2. S_k é estritamente melhor que S_q em ao menos um dos objetivos, ou seja, $\mathcal{M}(Sch_k) < \mathcal{M}(Sch_q)$ ou $R_T(Sch_k) > R_T(Sch_q)$

Desta forma, S_k é uma **solução dominante** e S_q é uma **solução dominada** por S_k. Se S_k não domina S_q e vice-versa, então as soluções são **incomparáveis** [31].

Uma variedade de soluções possíveis podem ser produzidas por MRCD para a mesma entrada G e P, se diferentes valores de w são dados ao algoritmo. Seja W o conjunto destes valores. De acordo com as condições de dominância descritas na Definição 2, o conjunto de todas as soluções de S, que são dominantes e incomparáveis (não repetidas), é denotado por S'. Note que as soluções em S' não são dominadas por nenhuma outra solução em S. O conceito de dominância pode ser as vezes fraco para aplicações, onde no caso de soluções incomparáveis um objetivo pode ser melhor significativamente, ao custo de uma pequena deteriorização do outro objetivo. Assim, os conceitos de dominância não indicam necessariamente quais soluções selecionar, mas mostram quais soluções deve-se evitar.

A partir de um conjunto S de soluções do problema de escalonamento, o subconjunto S' de soluções não-dominadas de S é aquele cujos elementos são não dominados por qualquer elemento do conjunto S. Então, quaisquer duas soluções de S' são não-dominadas entre si, e as soluções em $S - S'$ são dominadas por pelo menos um elemento de S'. O conjunto S' forma uma aproximação denotada na literatura como *near-optimal pareto-front* [18, 24, 31, 80].

Para selecionar determinadas soluções dentro do espaço de soluções S, podem ser considerados somente os elementos do conjunto S', já que para qualquer solução fora dele existe uma solução melhor nele. No entanto, em S' não há preferência, em princípio, por nenhuma das soluções. Portanto, é necessário acrescentar informações adicionais, ou seja, expressões que contemplem relações entre as funções objetivas para escolher determinada solução. Estas informações são subjetivas, podendo ser, inclusive, informações puramente qualitativas ou baseadas na experiência. Desta forma, utilizando informações de mais alto nível e conceitos de dominância, será possível selecionar soluções em S' com mais qualidade ou não dominadas que satisfaçam determinados interesses.

4.2.3 Classificação e Ajuste dos Pesos

Com o conhecimento das soluções que formam S' a partir dos conceitos anteriores e utilizando a métrica de Diferença Média $D(Sch)$ definida, é então proposta uma classificação para ajustar os pesos na função de custo. Para apresentar a metodologia, primeiro são formuladas algumas definições necessárias.

Definição 3. Para um conjunto de soluções S obtido com MRCD, uma aplicação G e um ambiente P, um **ponto de equilíbrio total** em S é definido como uma solução $(Sch, \mathcal{M}(Sch),\ R_T(Sch), D(Sch))$ em que o escalonamento Sch de G prioriza igualmente ambos objetivos do problema, $EFT(v,p)$ e $RC(v,p)$, ou seja, $D(Sch) = 0$.

Dada uma solução do conjunto S gerada por MRCD, denotada por $(Sch_k, \mathcal{M}(Sch_k),$ $R_T(Sch_k), D(Sch_k))$, se $|D(Sch_k)| > 0$, então houve certa diferença entre os objetivos ao serem priorizados em Sch_k. Portanto, pela Definição 3, é possível obter outra nova solução

em que esta diferença seja menor (com maior equilíbrio), de forma que se aproxime mais a um ponto de equilíbrio total ($D(Sch) = 0$). Para diminuir $|D(Sch_k)|$ e equilibrar a solução de escalonamento de Sch_k obtida, é necessário variar os pesos dos objetivos, w, na função de custo. Se $D(Sch_k) < 0$ é necessário diminuir o valor de w; se $D(Sch) > 0$ deve-se aumentar o valor de w. Assim, a Definição 3 junto com a métrica $D(Sch)$ oferece critérios para atribuir pesos diferentes aos objetivos do problema em busca de uma solução de compromisso conveniente, que tenha maior ou menor equilíbrio entre a prioridades de seus objetivos.

A partir destes critérios e os conceitos de dominância anteriores, uma classificação é então proposta para soluções de escalonamento obtidas com MRCD.

Tendo o conhecimento das soluções de escalonamento que formam o conjunto S' para G e P, e definido o ponto de equilíbrio total em S' pela Definição 3. Soluções em S' podem ser classificadas em:

1. **soluções de equilíbrio** S_e, soluções que mais se aproximam ao ponto de equilíbrio total, aquelas com menor valor de $|D(Sch_k)|$,

2. **soluções de confiabilidade** S_{R_T}, soluções com os valores mais altos de confiabilidade $R_T(Sch_k)$, aquelas mais distantes do ponto de equilíbrio total e com menor valor de $D(Sch_k)$,

3. **soluções de desempenho** $S_{\mathcal{M}}$, soluções com o os menores valores de *makespan* $\mathcal{M}(Sch_k)$, aquelas mais distantes do ponto de equilíbrio total e com o maior valor de $D(Sch_k)$.

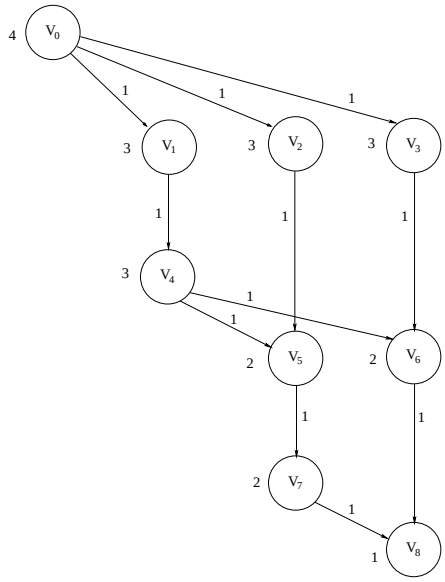
Baseando-se na classificação proposta, pode ser aplicada a seguinte metodologia para o ajuste dos pesos em MRCD. Dados, V_{ordG} e P, pode ser atribuído um passo h em $(0, 1)$ para calcular o peso da forma $w_i \in W \ / \ w_{i+1} = w_i + h$. Assim, $W = \{w_0, w_1, ..., w_l\}$, tal que $w_i \in (0, 1)$ é um conjunto finito de valores de pesos equidistantes a serem utilizados por MRCD. As soluções das distintas execuções MRCD(V_{ordG},P,w_i) formam o conjunto S de soluções iniciais a serem analisadas. MRCD(V_{ordG},P,w_i) além de informar $\mathcal{M}(Sch_i)$ e $R(Sch_i)$ para cada w_i, gera também a informação adicional $D(Sch_i)$ sobre Sch_i. O próximo passo é construir S' com soluções não dominadas a partir de S pela Definição 2. Em S', é aplicada a classificação anterior, de forma que são geradas as soluções, $S_{\mathcal{M}}$, S_{R_T} e S_e. As soluções $S_{\mathcal{M}}$ e S_{R_T} priorizam, respectivamente, *makespan* e confiabilidade. Já S_e representa um maior equilíbrio entre ambos objetivos do problema. O conjunto S' fornece uma informação mais completa para um usuário que prefere realizar um estudo de casos com escalonamentos de mais qualidade da aplicação G no cenário P.

Note que refinando o intervalo $(0, 1)$ ou seja usando um passo h menor, será possível aplicar a metodologia de forma a melhorar a solução de escalonamento. Assim, aumenta-se o conjunto S e portanto um número maior de valores w ou possíveis soluções de escalonamento podem ser considerados.

A abordagem para classificação e ajuste dos pesos proposta neste trabalho pode ser aplicada a algoritmos de escalonamentos bi-objetivos ponderados existentes, que empregam

também uma função de custo integrada. Para isto é necessário que os objetivos sejam normalizados na forma da Equação 4.5. Embora seja empregada uma função de custo diferente, pode ser adicionado no algoritmo o cálculo da métrica $D(Sch)$ para estimar a diferença média entre as prioridades alcançadas pelos objetivos no escalonamento final Sch. Da mesma forma, a classificação e o ajuste dos pesos propostos nesta seção podem ser utilizados para avaliar soluções de compromisso.

4.3 Estudo de Caso: Gauss com 9 Tarefas



Processador	$FP()$	$csi()$
p_1	10^{-6}	73
p_2	10^{-5}	53
p_3	10^{-4}	33

Figura 4.1: Eliminação de Gauss com 9 tarefas e características do ambiente com 3 processadores.

Com o objetivo de entender a aplicação dos pesos na função de custo e a classificação das soluções que podem ser geradas por MRCD para diferentes valores de w, da forma $W = \{0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1\}$, a aplicação Eliminação de Gauss com 9 tarefas foi escalonada sobre um conjunto de três processadores p_i, com as características mostradas na Figura 4.1.

Tabela 4.1: Soluções geradas por MRCD para G_9

w	0.1	0.2	**0.3**	**0.4**	0.5	0.6	**0.7**	**0.8**	**0.9**
$\mathcal{M}$	49.5	49.5	**49.5**	**79.5**	84.9	84.9	**84.9**	**109.5**	**167.8**
R_T	0.94897	0.94897	**0.94897**	**0.98195**	0.99150	0.99150	**0.99150**	**0.99467**	**0.99832**
D	69.8	69.8	**69.8**	**-21.2**	-41.8	-41.8	**-41.8**	**-70.2**	**-100**

Os resultados produzidos por MRCD são mostrados na Tabela 4.1, onde w é o peso aplicado à função de custo $f(v, p_j)$; $\mathcal{M}$ é o *makespan* do escalonamento; R_T é a confiabilidade total; e D, a diferença média associada. As colunas em negrito mostram o conjunto S', soluções não dominadas. Em S', três soluções são particularmente interessantes pela classificação proposta: $S_{\mathcal{M}}$, S_{R_T} e S_e, geradas com $w = 0.3$, $w = 0.4$ e $w = 0.9$, respectivamente. Note que as soluções idênticas não se encontram repetidas em S'.

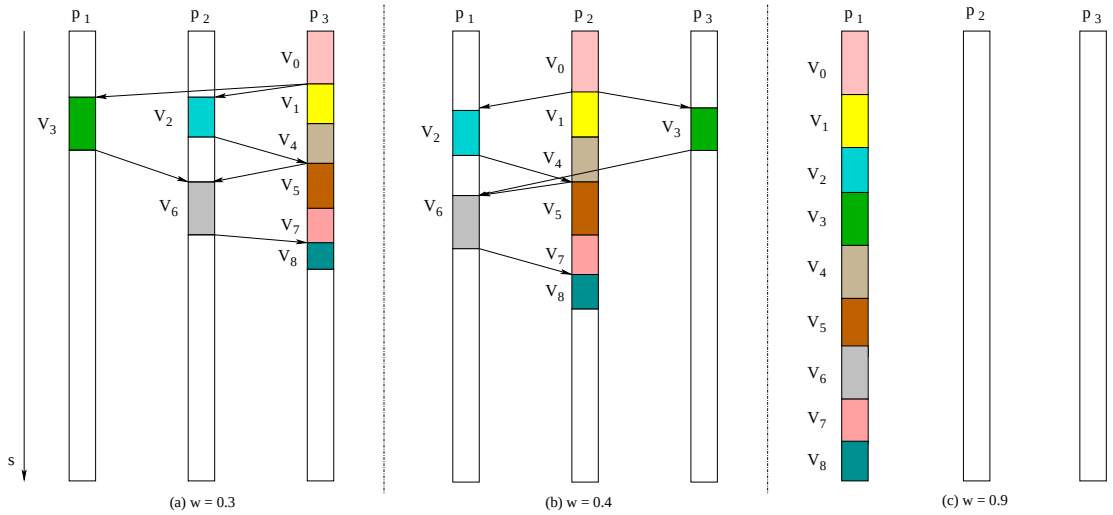


Figura 4.2: MRCD escalona o GAD da Eliminação de Gauss com 9 tarefas para $w = 0.3, 0.4$ e 0.9, respectivamente, sobre 3 processadores

Em geral, pode ser observado na Tabela 4.1 que para altos valores de w e consequentemente, menores valores de D, MRCD tende a maximizar a confiabilidade, caso contrário, o *makespan* é minimizado. Para ilustrar melhor como foram obtidas as soluções variando o peso w em G_9, a Figura 4.2 mostra os escalonamentos produzidos de $S_{\mathcal{M}}$, S_{R_T} e S_e por MRCD sobre 3 processadores.

Inicialmente, em $w = 0.3$ ($S_{\mathcal{M}}$) na Tabela 4.1 o *makespan* da aplicação é altamente priorizado, sacrificando o valor final da confiabilidade. Observe na Figura 4.2 que as tarefas do caminho crítico foram escalonadas em p_3, processador mais rápido e menos confiável. Desta forma se garante o melhor tempo final $\mathcal{M} = 49.5$ para executar a aplicação, embora seja alcançado com menor confiabilidade. Em $w = 0.4$ (S_e) o escalonamento mostra um maior equilíbrio entre os objetivos ($D = -21.2$), uma diferença menor comparado com as outras soluções. Ainda neste escalonamento pela Figura 4.2, todos os processadores são utilizados, porém as tarefas do caminho crítico são alocadas em p_2, processador nem muito rápido nem muito confiável. Finalmente, com $w = 0.9$ (S_{R_t}) todas as tarefas foram escalonadas em p_1, o único processador mais confiável. Neste exemplo, a confiabilidade da aplicação é absolutamente privilegiada (veja Tabela 4.1).

4.4 Análise de Desempenho

Com o objetivo de avaliar o desempenho da estratégia de escalonamento proposta, o algoritmo MRCD é comparado com outras heurísticas de escalonamento da literatura, para diferentes topologias diferentes de aplicações GADs e considerando cenários arquiteturais distintos. Antes de analisar os resultados, são definidas nesta seção algumas métricas empregadas durante o estudo realizado.

4.4.1 Heurísticas de Escalonamento

O algoritmo de escalonamento proposto MRCD foi comparado com estratégias de escalonamento da literatura (por Seção 3.2), as quais foram divididas em duas classes: a primeira produz uma única solução, denotada como Classe 1, que engloba as heurísticas HEFT [75], RCDMod (baseada em [60]) e RHEFT (baseada em [28]); e a Classe 2, contendo BSAMod baseada em [39], que de forma similar a MRCD, pode gerar múltiplas soluções. BSAMod emprega uma função de custo integrada como MRCD, porém combina ambos objetivos com um operador e um método de normalização distintos. Desta forma, resultados diferentes são obtidos. Com o propósito de ser compararadas com MRCD, estas heurísticas foram implementadas com pequenas mudanças que serão descritas a seguir.

Na Classe 1, HEFT minimiza somente o *makespan* do escalonamento. Para comparar com MRCD, este trabalho adicionalmente calcula R_T de acordo com o escalonamento produzido por HEFT. O trabalho [60] também fornece uma solução para o problema de escalonamento bi-objetivo, mas usa uma função de custo hierárquica. Diferentemente da proposta original, RCDMod não foi implementada com restrições de tempo (*deadlines*) e a função de custo usa tempo de fim no lugar de tempo de ínicio para escalonar as tarefas. Finalmente, RHEFT implementa a abordagem *list scheduling* de HEFT, mas considera ambos os objetivos na sua função de custo com a multiplicação do tempo de execução e a taxa de falha como em [28]. Na Classe 2, BSAMod também considera o problema bi-objetivo, neste caso usando a função $f(v,p) = \sqrt{w\left(\frac{EFT(v,p_j)}{\max_{p\in P}\{EFT(v,p)\}}\right)^2 + (1-w)\left(\frac{RC(v,p_j)}{\max_{p\in P}\{RC(v,p)\}}\right)^2}$ como proposto em [39], mas ordena as tarefas pelos $blevels(v)$ e executa o procedimento de inserção de tarefas como em HEFT.

4.4.2 Aplicações GADs

Todas as heurísticas descritas foram executadas para aplicações sintéticas representadas por três tipos de GADs, conforme o modelo da aplicação da Seção 2.1.1: um que representa a Eliminação Gausseana, denotada por G_n; o Diamante Di_n, que representa, por exemplo, a multiplicação de matrizes; e o último R_n formado de grafos gerados aletóriamente com topologia irregular. Nos três casos, n denota o número de tarefas do GAD. Em todos os grafos, um custo unitário foi associado com as comunicações entre as tarefas. O mesmo não acontece com os pesos das tarefas. Em G_n, cada tarefa $v \in V$ apresenta distintos pesos de computação que depende do nível da tarefa no GAD, de acordo com [70], enquanto em ambos R_n e Di_n, um mesmo peso é atribuído a todas as tarefas, $\epsilon(v) = 50$.

4.4.3 Cenários da Arquitetura

O sistema de recursos computacionais distribuído P para esta análise foi dividido em três grupos diferentes em relação ao índice de retardo, denotados como P_0, P_1 e P_2, cada um com $m/3$ processadores. A taxa de falha dos processadores, $FP(p_i)$, em todos os cenários foi uniformemente gerada no intervalo $[10^{-5}, 10^{-4}]$. Em particular, FP foi obtida em $[10^{-5}, 3.3\times10^{-5}]$, para $p_i \in P_0$; $[3.4\times10^{-5}, 6.6\times10^{-5}]$, para $p_i \in P_1$; $[6.7\times10^{-5}, 10^{-4}]$,

para $p_i \in P_2$. Observe que P_0 é composto por processadores com a menor taxa de falha e P_2 com a maior.

Três cenários arquiteturais foram definidos. No cenário do pior caso (CPC), os valores adotados foram, $csi(p_i) = 73$ para $p_i \in P_0$; $csi(p_i) = 53$, para $p_i \in P_1$; $csi(p_i) = 33$ para $p_i \in P_2$. CPC é considerado um cenário de pior caso ou conflitante, porque os processadores com menor velocidade são os que apresentam as menores taxas de falha e vice-versa. Esta configuração foi escolhida para estudar casos de risco, onde é difícil chegar a um compromisso entre os objetivos do escalonamento (conflitantes).

Por outro lado, o cenário com o melhor caso (CMC) é o cenário onde os processadores mais rápidos apresentam as menores taxas de falha, ou seja, $csi(p_i) = 33$ para $p_i \in P_0$; $csi(p_i) = 53$, para $p_i \in P_1$; $csi(p_i) = 73$, para $p_i \in P_2$.

O terceiro caso, denominado cenário de processadores homogêneos (CPH) foi escolhido com velocidade igual $csi(p_i) = 53$ para todo $p_i \in P_0 \cup P_1 \cup P_2$.

Os parâmetros da arquitetura, tais como índice de retardo csi e latência L, foram obtidos de acordo com as métricas coletadas por [52] e [55], calculadas nesses trabalhos com o objetivo de modelar o ambiente computacional real das máquinas de nossa instituição. As taxas de falha dos processadores FP, foram calculadas a partir de [60].

4.4.4 Métricas Utilizadas

Cada heurística de escalonamento, dentro das classes relacionadas, foi comparada com MRCD utilizando métricas distintas. Em todos os casos foram utilizados os conceitos de dominância da Definição 2 para comparar soluções de duas heurísticas. Um par solução $(\mathcal{M}_1, R_{T_1})$ em Sch_1 de uma heurística é considerada dominada por outra $(\mathcal{M}_2, R_{T_2})$ em Sch_2 de outra heurística, se e somente se, a solução de Sch_2 apresenta pelo menos uma de suas soluções objetivas ($\mathcal{M}_2$ ou R_{T_2}) melhor que a de Sch_1, e nenhuma pior que a de Sch_1. Neste caso a solução de Sch_2 domina a de Sch_1, assim a solução de Sch_2 é **dominante** e a de Sch_1 **dominada**. As soluções são **iguais** quando ambos objetivos são idênticos. Em qualquer outro caso, as soluções são consideradas **incomparáveis** entre si.

O número total de soluções dominantes e dominadas foi uma métrica considerada para comparar as heurísticas da Classe 2. Nestas heurísticas de múltiplas soluções, várias soluções possíveis podem ser produzidas para a mesma entrada G e P, se valores diferentes de w são fornecidos ao algoritmo. Seja $W = \{w_0, w_1, ..., w_l\}$ o conjunto com os valores considerados para formar o conjunto de soluções S. Para comparar o algoritmo BSAMod com MRCD foram somente consideradas as soluções contidas no conjunto S' para cada heurística. Cada solução em S' para MRCD foi comparada com as soluções em S' da outra heurística, baseado nos conceitos de dominância descritos. Estes critérios podem não ser suficientes para casos de soluções incomparáveis, onde um objetivo da solução pode ser significativamente melhor, a um custo de uma pequena deteriorização do outro objetivo. Portanto, foram consideradas também outras métricas de comparação entre duas heurísticas, relacionadas a seguir.

O cálculo da percentagem,

$$Perc^{i}_{MRCD}(\%) = \frac{(O^{i}_{Heur} - O^{i}_{MRCD}) \times 100}{O^{i}_{MRCD}}, \tag{4.7}$$

é uma métrica utilizada neste trabalho para comparar um objetivo O^{i}_{MRCD} ($\mathcal{M}$ ou R_T) de uma solução MRCD, com o objetivo correspondente O^{i}_{Heur} na solução de outra heurística. Esta métrica permite medir a **percentagem de deteriorização ou de melhoria (%)** de um objetivo em relação ao outro, na solução MRCD quando comparada com a solução de outra heurística.

Outra métrica,

$$Diss = (\sqrt{(\frac{\mathcal{M}Heur - \mathcal{M}}{\max \mathcal{M}})^2 + (\frac{R_T Heur - R_T}{\max R_T})^2} \tag{4.8}$$

foi utilizada também neste trabalho para calcular a **dissemelhança** (ou proximidade) de uma solução MRCD com soluções das heurísticas comparadas. $Diss$ calcula a distância euclidiana entre a solução de MRCD ($\mathcal{M}$ e R_T) e a solução de outra heurística ($\mathcal{M}Heur$ e $R_T Heur$).

4.4.5 Comparação de MRCD com Heurísticas da Classe 1

O primeiro grupo de experimentos compara os resultados de MRCD com as heurísticas na Classe 1, relacionadas na Seção 3.2, nos três cenários arquiteturais descritos anteriormente. As tabelas mostram os resultados obtidos nas diferentes heurísticas na solução do escalonamento $(\mathcal{M}, R_T)$, considerando as três comparações seguintes. A primeira comparação C1 fornece os resultados de HEFT e MRCD, C2 mostra os resultados de RCDMod e MRCD, e C3 compara RHEFT e MRCD. Em cada caso, as soluções MRCD selecionadas foram: $S_{\mathcal{M}}$ de S' em C1 para uma comparação com HEFT; $S_{R_T} \in S'$ no caso de C2 (RCDMod prioriza confiabilidade), e em C3 como RHEFT, devido a sua função de custo, acaba priorizando um pouco mais processadores que maximizam confiabilidade, então sua solução foi comparada com a solução MRCD mais próxima. Uma solução é mais proxima ou semelhante a outra solução de outra heurística, se apresenta o menor valor de dissemelhança calculada com a Equação 4.8. Uma outra coluna com o cálculo da percentagem (%) é adicionada nas comparações C2 e C3, para analisar onde a dominância não é suficiente para comparar as soluções. A diferença média D, para cada GAD, é mostrada separadamente em outras tabelas.

Cenário de Pior Caso (CPC)

A Tabela 4.2 apresenta os resultados do pior cenário CPC para $m = 24$ processadores. Em geral, como esperado pode ser observado que com o incremento do número de tarefas n, o *makespan* também aumenta e a confiabilidade diminui. Como o número de tarefas executadas cresce para um número fixo de processadores, incrementa-se a carga dos processadores e consequentemente seus custos de confiabilidade. A tabela mostra também que as soluções em C1 apresentaram pior confiabilidade que em C2 e C3, mas o *makespan* foi consideralvemente menor. O algoritmo MRCD apresentou soluções melhores (todas

Tabela 4.2: Resultados para Heurísticas da Classe 1 com G_n, R_n e Di_n sobre $m = 24$ processadores no *Cenário de Pior Caso* (CPC)

	C1		C2			C3		
DAG	HEFT	MRCD	RCDMod	MRCD	(%)	RHEFT	MRCD	(%)
G_{152}	190.0, 0.9207	**190.0, 0.9329**	2406.0, 0.9576	1283.3, 0.9554	87.4, 0.22	694.9, 0.9457	449.4, 0.9428	54.6, 0.30
G_{252}	318.7, 0.8351	**318.7, 0.8587**	5201.9, 0.9106	2715.6, 0.9060	91.5, 0.49	1309.6, 0.8813	856.3, 0.8784	52.9, 0.32
G_{377}	480.4, 0.7184	**480.4, 0.7514**	9603.8, 0.8412	4981.5, 0.8335	92.7, 0.92	2204.6, 0.7869	1494.8, 0.7856	47.4, 0.16
G_{527}	675.1, 0.5853	**675.1, 0.6122**	15976.7, 0.7500	8211.0, 0.7385	94.5, 1.56	3560.9, 0.6679	2273.0, 0.6657	56.6, 0.33
G_{702}	902.8, 0.4406	**902.8, 0.4553**	24685.6, 0.6412	12687.4, 0.6260	94.5, 2.42	5258.9, 0.5286	**3463.3, 0.5318**	51.8, -0.59
R_{80}	330.0, 0.8264	**330.0, 0.8415**	5840.0, 0.9002	3066.0, 0.8952	90.4, 0.55	1022.0, 0.8527	**949.0, 0.8647**	7.69, -1.39
R_{98}	330.0, 0.7905	**330.0, 0.8076**	7154.0, 0.8791	3650.0, 0.8730	96.0, 0.70	1460.0, 0.8306	**1314.0, 0.8403**	11.1, -1.16
R_{152}	396.0, 0.6959	**396.0, 0.6990**	11096.0, 0.8189	5694.0, 0.8101	94.8, 1.08	1679.0, 0.7334	**1533.0, 0.7533**	9.52, -2.64
R_{256}	528.0, 0.5421	**528.0, 0.5421**	18688.0, 0.7143	10512.0, 0.7027	77.7, 1.64	1606.01, 0.5665	**1314.0, 0.5830**	22.2, -2.82
R_{364}	759.0, 0.4159	**759.0, 0.4180**	26572.0, 0.6198	13505.0, 0.6038	96.7, 2.64	2993.0, 0.4568	**1825.0, 0.4674**	64.0, -2.26
Di_{81}	580.9, 0.8300	**580.9, 0.8321**	5913.0, 0.8990	3066.0, 0.8939	92.8, 0.57	3285.0, 0.8924	**3066.0, 0.8939**	7.17, -0.16
Di_{100}	660.0, 0.7949	**660.0, 0.7966**	7300.0, 0.8768	3723.0, 0.8706	96.0, 0.71	4599.0, 0.8721	3723.0, 0.8706	23.5, 0.17
Di_{144}	825.0, 0.7153	**825.0, 0.7190**	10512.0, 0.8276	5329.0, 0.8190	97.2, 1.04	6935.0, 0.8217	5329.0 0.8190	30.1, 0.32
Di_{256}	1155.0, 0.5407	**1155.0, 0.5494**	18688.0, 0.7143	9417.0, 0.7012	98.4, 1.87	12994.0, 0.7062	9417.0, 0.7012	37.9, 0.71
Di_{361}	1452.0, 0.4158	**1452.0, 0.4209**	26353.0, 0.6222	13286.0, 0.6062	98.3, 2.64	19126.0, 0.6133	13286.0, 0.6062	43.9, 1.17

dominantes) do que HEFT em todos os grafos. É importante ressaltar que, selecionando soluções MRCD com valores D maiores (soluções $S_{\mathcal{M}}$), como mostra Tabela 4.3, o algoritmo mostrou ser capaz de produzir soluções com *makespans* tão bons quanto os de HEFT, mas com melhor confiabilidade.

Em C2, RCDMod apresentou a confiabilidade um pouco melhor do que os resultados de S_{R_T} para MRCD, com soluções incomparáveis para cada GAD em todas as tabelas. Note que esta heurística emprega uma função de custo hierárquica que prioriza somente a confiabilidade e em caso de empate é que considera o *makespan*. Mesmo que as soluções MRCD também consideram a confiabilidade, como sua função de custo integra ambos os objetivos, o *makespan* produzido não se prejudica igualmente. Contudo, é importante observar que o *makespan* de RCDMod é quase o dobro do valor apresentado por MRCD.

Finalmente em C3, as soluções MRCD dominam as soluções RHEFT em muitos grafos, produzindo nos outros casos soluções incomparáveis em relação a dominância. RHEFT emprega uma função de custo que também integra ambos os objetivos mas, de forma diferente a MRCD, priorizando mais a confiabilidade por causa de sua função de custo (*taxa de*

falha × *tempo de execução da tarefa*). Em C3, as soluções MRCD apresentaram *makespan*s que são quase a metade dos valores obtidos por RHEFT, enquanto os valores da confiabilidade ficaram muito próximos. Como em MRCD a confiabilidade é menos priorizada que em RHEFT, observa-se que G_n e Di_n obtiveram mais soluções incomparáveis, por ser os objetivos conflitantes neste cenário.

Tabela 4.3: Valores de D e w para os GADs G_n, R_n e Di_n sobre $m = 24$ processadores no Cenário de Pior Caso (CPC) e as comparações C1, C2 e C3.

		G_n					R_n				
		152	252	377	527	702	80	98	152	256	364
C1	D	30.3	31.0	32.7	35.6	41.0	27.7	29.9	42.0	38.2	44.1
	w	0.5	0.5	0.5	0.5	0.4	0.6	0.6	0.4	0.4	0.3
C2	D	-58.7	-58.1	-57.1	-57.1	-57.3	-58.7	-57.9	-56.9	-60.0	-54.5
	w	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9
C3	D	-13.2	-7.6	-6.3	-4.4	-4.8	-10.6	-18.1	-5.95	12.8	15.3
	w	0.7	0.7	0.7	0.7	0.7	0.7	0.7	0.7	0.6	0.6

		Di_n				
n		81	100	144	256	361
C1	D	43.6	43.9	44.7	46.6	49.5
	w	0.6	0.6	0.4	0.3	0.2
C2	D	-59.7	-60.5	-61.0	-62.5	-62.7
	w	0.9	0.9	0.9	0.9	0.9
C3	D	-59.7	-60.5	-61.0	-62.5	-62.7
	w	0.9	0.9	0.9	0.9	0.9

Neste cenário observa-se uma diferença significativa entre os valores (em C1, C2 e C3) das soluções de MRCD para um mesmo GAD. Na maioria das soluções foi necessário sacrificar bastante o valor de um objetivo para melhorar o outro, gerando um desequilíbrio ou diferença média maior entre os objetivos (valores de $|D|$ mais altos), como mostra a Tabela 4.3. Em todos os cenários os valores de D são colocados em tabelas distintas para analisar seu comportamento em relação aos objetivos.

Cenário de Processadores Homogêneos (CPH)

Na Tabela 4.4 para o cenário de processadores homogêneos (CPH), em relação a velocidade dos processadores, são mostrados os resultados produzidos pelas heurísticas da Classe 1, considerando as mesmas aplicações GADs anteriores. Conclusões similares as derivadas com CPC foram também obtidas neste cenário, porém pode ser observada uma menor diferença entre os *makespans* das soluções produzidas por MRCD para um mesmo GAD. Como a velocidade dos processadores é a mesma, os valores do *makespan* só dependem do escalonamento das tarefas e da topologia do GAD. Neste cenário, houve menor conflito entre os objetivos durante o escalonamento, ou seja, para melhorar a confiabilidade não sacrificou-se muito o valor do *makespan*.

Além disso, é interessante notar que na maioria dos GADs os valores menores de $|D|$, como visto na Tabela 4.5, para as soluções $S_{\mathcal{M}}$ e S_{R_T}, mostram que se alcança maior equilíbrio entre os objetivos nos escalonamentos obtidos que no cenário anterior. Na maioria dos GADs, os valores de D ficam muito próximos do ponto de equilíbrio (D próximo a zero

Tabela 4.4: Resultados para Heurísticas da Classe 1 com G_n, R_n e Di_n sobre $m = 24$ processadores no *Cenário de Processadores Homogêneos* (CPH)

	C1		C2			C3		
DAG	HEFT	MRCD	RCDMod	MRCD	(%)	RHEFT	MRCD	(%)
G_{152}	305.2, 0.9402	**305.2, 0.9569**,	1746.8, 0.9690	666.7, 0.9652	162.0, 0.39	504.5, 0.9595	343.4, 0.9591	46.9, 0.04
G_{252}	511.9, 0.8594	**511.9, 0.8982**	3776.7, 0.9342	1409.8, 0.9261	167.8, 0.87	983.6, 0.9086	**722.9, 0.9134**	36.0, -0.52
G_{377}	771.6, 0.7346	**771.6, 0.7958**	6972.6, 0.8820	2546.1, 0.8675	173.8, 1.66	1657.8, 0.8246	**1263.5**, **0.8447**	31.4, -2.38
G_{527}	1084.3, 0.5803	**1084.3, 0.6584**	11599.5, 0.8115	4249.5, 0.7897	172.9, 2.76	2558.8, 0.7143	**1958.8**, **0.7517**	30.5, -4.97
G_{702}	1450.0, 0.4139	**1450.0, 0.4988**	17922.4, 0.7242	6434.2, 0.6935	178.5, 4.43	3713.1, 0.5867	**2988.1, 0.5872**	24.2, -8.73
R_{80}	530.0, 0.8486	**530.0, 0.8869**	4240.0, 0.9265	1643.0, 0.9178	158.0, 0.94	848.0, 0.8743	**795.0, 0.9033**	6.66, -3.21
R_{98}	530.0, 0.8240	**530.0, 0.8543**	5194.0, 0.9107	2014.0, 0.9005	157.8, 1.13	1113.0, 0.8576	**901.0, 0.8816**,	23.5, -2.7
R_{152}	636.0, 0.7052	**636.0, 0.7411**	8056.0, 0.8650	3127.0, 0.8500	157.6, 1.76	1325.0, 0.7667	**1060.0, 0.8118**	25.0, -5.54
R_{256}	583.0, 0.5065	**583.0, 0.5089**	13568.0, 0.7833	5724.0, 0.7631	137.0, 2.64	1484.0, 0.5834	901.0, 0.5656	64.7, -3.14
R_{377}	848.0, 0.3832	**848.0, 0.3840**	19292.0, 0.7066	7155.0, 0.6757	169.6, 4.56	2438.0, 0.4810	**2014.0, 0.5582**	21.0, -13.8
Di_{81}	900.9, 0.8888	**900.9, 0.9006**	4293.0, 0.9256	1536.9, 0.9155	179.3, 1.09	2385.0, 0.9207	1536.9, 0.9155	55.1, 0.56
Di_{100}	1007.1, 0.8444	**1007.1, 0.8749**	5300.0, 0.9090	1908.0, 0.8969	177.7 1.34	3339.0, 0.9054	1908.0, 0.8969	75.0, 0.94
Di_{144}	1219.1, 0.7525	**1219.1, 0.8111**	7632.0, 0.8716	2703.0, 0.8550	182.3, 1.93	5035.0, 0.8671	2703.0 0.8550	86.2, 1.40
Di_{256}	1643.1, 0.5752	**1643.1, 0.6486**	13568.0, 0.7833	4717.0, 0.7566	187.6, 3.52	9434.0, 0.7768	4717.0 0.7566	100, 2.67
Di_{361}	1961.1, 0.4223	**1961.2, 0.5106**	19133.0, 0.7086	6625.0, 0.6747	188.8, 5.01	13886.0, 0.7012	6625.0, 0.6747	109.6, 3.92

0). Observe que ao selecionar um processador que maximiza confiabilidade, a velocidade dos processadores não prejudica a minimização do *makespan*. Entretanto, a topologia do GAD influencia na minimização do *makespan*, mas não na minimização do custo de confiabilidade pela função de custo de MRCD.

Em C2, os valores de D na Tabela 4.5 são um pouco mais altos do que aqueles da Tabela 4.3 para o pior cenário, ou seja, os *makespan* são um pouco menores. As soluções S_{R_T} priorizam a minimização dos custos de confiabilidade, assim D é negativo. Já em C3 praticamente todos os valores de D são relativamente mais altos que os de CPC. Mesmo ainda negativos, o valor absoluto $|D|$ é pequeno (menor diferença média entre os objetivos). Na Tabela 4.4, como o ambiente é homogêneo, é obvio que quando a função de custo de RHEFT é minimizada, mais tarefas são escalonadas sobre processadores que minimizam o custo de confiabilidade, devido ao produto especificado na função de custo de RHEFT. Entretanto, como RHEFT considera o tempo de fim na função de custo, este cenário de velocidades iguais, apresenta soluções com *makespan*s melhores do que os de CPC, mas com confiabilidade um pouco mais alta na última coluna da Tabela 4.4.

Tabela 4.5: Valores de D para os GADs G_n, R_n e Di_n sobre $m = 24$ processadores no Cenário de Processadores Homogêneos (CPH) e as comparações C1, C2 e C3.

		G_n					R_n				
		152	252	377	527	702	80	98	152	256	364
C1	D	6.18	10.6	18.8	24.2	29.2	1.39	3.61	25.6	54.4	55.5
	w	0.7	0.6	0.5	0.5	0.5	0.6	0.6	0.5	0.4	0.4
C2	D	-38.5	-43.3	-43.6	-46.2	-46.6	-50.0	-50.8	-51.1	-58.1	-51.4
	w	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9
C3	D	-0.37	-8.63	-9.03	-8.82	-11.8	-15.8	-17.7	-17.3	12.8	-21.1
	w	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.7	0.5	0.6

		Di_n				
n		81	100	144	256	361
C1	D	11.6	13.1	17.1	25.2	31.1
	w	0.6	0.6	0.5	0.5	0.5
C2	D	-35.7	-39.0	-42.6	-45.1	-46.1
	w	0.9	0.9	0.9	0.9	0.9
C3	D	-35.7	-39.0	-42.6	-45.1	-46.1
	w	0.9	0.9	0.9	0.9	0.9

Cenário de Melhor Caso (CMC)

Os resultados deste cenário CMC são mostrados na Tabela 4.6, onde há menos conflito ou diferença entre os objetivos (csi e FP são melhores nos mesmos processadores) que nos cenários CPH e CPC. Os objetivos do escalonamento podem ser alcançados selecionando um processador que minimize ambos *makespan* e custo de confiabilidade, para as tarefas da aplicação. Note que, para um mesmo GAD as soluções de MRCD, em C1, C2 e C3, estão mais próximas. Na Tabela 4.7, os valores de $|D|$ menores (próximos a zero) apresentam uma menor diferença média entre os objetivos (maior equilíbrio), comparado com os outros cenários. Portanto, em CMC, a minimização do *makespan* não se compromete tanto como em CPC e CPH, assim neste cenário (Tabela 4.6), os *makespan*s de MRCD em C2 e C3 são muito menores. Note também, que em CMC assim como nos outros cenários, para MRCD não há nenhuma solução dominada.

Variação do Número de Processadores no CPC

Outro conjunto de experimentos foi realizado variando o número de processadores, sobre GADs de maior tamanho: G_{1034}, R_{546} e Di_{529}, como mostra a Tabela 4.8 para o pior cenário CPC. Com o incremento da quantidade de processadores, ambos objetivos tendem a melhorar, o *makespan* diminui enquanto a confiabilidade aumenta. Observe que desta forma, as estratégias de escalonamento podem alocar as tarefas mais adequadamente nos processadores, com mais opções em relação as taxas de falha e os índices de retardo. Para as comparações C1, C2 e C3, o mesmo comportamento descrito anteriormente é detectado também.

Tabela 4.6: Resultados para Heurísticas da Classe 1 com G_n, R_n e Di_n sobre $m = 24$ processadores no *Cenário de Melhor Caso* (CMC)

	C1		C2			C3		
DAG	HEFT	MRCD	RCDMod	MRCD	(%)	RHEFT	MRCD	(%)
G_{152}	190.0, 0.9516	**190.0, 0.9733**	1087.6, 0.9806	190.0, 0.9733	472.4, 0.74	314.1, 0.9747	190.0, 0.9733	65.3, 0.18
G_{252}	318.7, 0.8820	**318.7, 0.9341**	2351.5, 0.9585	368.2, 0.9423	538.6 1.72	627.0, 0.9405	**368.2**, **0.9423**	91.9, -0.05
G_{377}	480.4, 0.8036	**480.4, 0.8474**	4341.4, 0.9248	652.7, 0.8953	565.1, 3.28	1030.9, 0.8803	**652.7, 0.8953**	82.2, -1.03
G_{527}	675.1, 0.6403	**675.1, 0.7190**	7222.3, 0.8780	1062.6, 0.8316	579.6, 5.58	1652.6, 0.7922	**1062.6, 0.8316**	81.7, -3.11
G_{702}	902.8, 0.4949	**902.8, 0.5267**	11159.2, 0.8180	1620.9, 0.7523	588.4, 8.73	2508.0, 0.6937	**1620.9, 0.7523**	54.7, -7.78
R_{80}	330.0, 0.8669	**330.0, 0.9293**	2640.0, 0.9535	462.0, 0.9375	471,4 1.70	528.0, 0.9141	**462.0, 0.9375**	14.2, -2.50
R_{98}	330.0, 0.8434	**330.0, 0.8841**	3234.0, 0.9434	495.0, 0.9219	553.3 2.33	726.0, 0.9006	**495.0, 0.9219**	46.6, -2.31
R_{152}	396.0, 0.7296	**396.0, 0.7528**	5016.0, 0.9136	726.0, 0.8798	590.9, 3.83	924.0, 0.8287	**726.0**, **0.8798**	27.2, -5.8
R_{256}	528.0, 0.5453	**528.0, 0.5453**	8448.0, 0.8589	1551.0, 0.8123	444.6, 5.73	1122.0, 0.6653	**759.0, 0.6848**	47.8, -2.84
R_{364}	759.0, 0.4257	**759.0, 0.4298**	12012.0, 0.8055	1947.0, 0.7412	516.9, 8.67	1914.0, 0.5751	**1023.0, 0.5759**	87.1, -0.13
Di_{81}	580.8, 0.9298	**580.8, 0.9364**	2673.0, 0.9530	594.0, 0.9390	350.0 1.48	1485.0, 0.9498	594.0, 0.9390	150.0, 1.15
Di_{100}	660.0, 0.9120	**660.0, 0.9232**	3300.0, 0.9423	693.0, 0.9242	376.1, 1.95	2079.0, 0.9400	693.0, 0.9242	200.0, 0.94,
Di_{144}	825.0, 0.8704	**825.0, 0.8883**	4752.0, 0.9180	891.0, 0.8905	433.3, 3.08	3135.0, 0.9150	891.0, 0.8190	251.8, 2.75
Di_{256}	1155.0, 0.7404	**1155.0, 0.7528**	8448.0, 0.8589	1353.0, 0.8085	524.3, 6.23	5874.0, 0.8545	1353.0, 0.8085	334.1, 5.68
Di_{361}	1452.0, 0.6287	**1452.0, 0.6454**	11913.0, 0.8069	1782.0, 0.7384	568.5, 9.28	8679.0, 0.8017	1782.0, 0.7384	387.0, 8.58

Outras Conclusões da Comparação com Heurísticas da Classe 1

R_n se comporta melhor em todos os cenários em relação a dominância, ou seja, apresenta mais soluções dominantes quando comparado com os outros grafos, seguido por G_n e depois Di_n. Para R_n com MRCD foi possível em C3 achar soluções próximas a RHEFT, dominantes na maioria dos casos, embora os percentais (%) de melhora do *makespan* são menores do que nos outros tipos de GADs. G_n tem todas suas soluções dominantes nos dois últimos cenários CPH e CMC, e só no cenario conflitante CPC que a confiabilidade fica um pouco menor que a de RHEFT e portanto MRCD nao consegue dominar. O GAD G_n se caracteriza por ser uma aplicação heterogênea com menor computação (diminui a cada nível) que os outros tipos de GADs, mas com bastante dependência entre as tarefas o que aumenta um pouco o custo de confiabilidade no cenário mais desequilibrado. O percentual de melhora do *makespan* é 50% em média superior em CPC. Note que diferentemente de RHEFT, MRCD tende a minimizar mais o tempo que o custo de confiabilidade, buscando maior equilíbrio.

Tabela 4.7: Valores de D para os GADs G_n, R_n e Di_n sobre $m = 24$ processadores no Cenário de Melhor Caso (CMC) e as comparações C1, C2 e C3.

		G_n					R_n				
		152	252	377	527	702	80	98	152	256	364
C1	D	-1.91	4.0	9.75	14.6	21.5	-16.2	4.17	20.6	37.2	26.6
	w	0.9	0.6	0.6	0.5	0.4	0.7	0.5	0.4	0.1	0.3
C2	D	-1.91	-3.70	-7.59	-11.0	-13.1	-18.2	-20.7	-20.1	-35.4	-28.2
	w	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9
C3	D	-1.91	-3.79	-7.59	-11.0	-13.1	-18.2	-20.7	-20.1	-31.5	-26.0
	w	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.6	0.6

		Di_n				
n		81	100	144	256	361
C1	D	4.56	2.37	-0.35	10.8	11.7
	w	0.6	0.8	0.8	0.3	0.3
C2	D	1.19	-1.84	-7.27	-13.0	-16.8
	w	0.9	0.9	0.9	0.9	0.9
C3	D	1.19	-1.84	-7.27	-13.0	-16.8
	w	0.9	0.9	0.9	0.9	0.9

Di_n teve poucas soluções dominantes para MRCD. Observe que para Di_n o valor absoluto da diferença média $|D|$ é relativamente alto nos dos primeiros cenários, mostrando mais desequilibrio das soluções objetivas. Enquanto o *makespan* de MRCD obtém melhores resultados que nas outras heurísticas, o custo de confiabilidade é maior. Mesmo que os valores negativos de D para MRCD mostram, em C2 e C3, que a confibilidade é mais priorizada que o tempo, este GAD apresenta maior dificuldade ao paralelizar a aplicação por ter muitas dependências. Além disso RHEFT e RCDMod se caracterizam por priorizar mais a confiabilidade, por causa de suas funções de custo, dominando em relação a este objetivo. Contudo comparado com os outros grafos, Di_n para MRCD obteve o melhor resultado nos percentais (%) do *makespan* do que as outras heurísticas.

Note que, a maioria das soluções MRCD nos experimentos da Classe 1 foram dominantes ou incomparáveis, com muitas dominantes e nenhuma dominada por outra heurística. Como pode ser observado nas tabelas desta classe, para soluções incomparáveis, as melhorias no *makespan* (em percentuais %) fornecidas por MRCD, foram significativas, a um custo de uma pequena deterioração da confiabilidade (em percentuais %).

4.4.6 Análise Comparativa entre MRCD e uma Heurística da Classe 2

A heurística proposta em [39] pertence a mesma classe de MRCD (múltiplas soluções). O algoritmo BSAMod segue a estrutura *list scheduling* de HEFT, mas implementa uma função de custo e um método de normalização diferentes. Comparando MRCD com BSAMod, são analisadas as vantagens da função de custo e dos valores normalizados para MRCD. Primeiramente, os resultados gerados por MRCD usando a normalização de BSA (MRCD$_{BSA}$) são comparados com a implementação original de MRCD, para um melhor entendimento da normalização empregada.

Comparando todas as soluções geradas em W, por MRCD com as soluções de BSA (em qualquer das versões implementadas), foi calculado o número total de dominantes

Tabela 4.8: Comparação na Classe 1 para G_{1034}, R_{546} and Di_{529} com a variação de m processadores no Cenário de Pior Caso (CPC)

		C1			C2			C3		
DAG	m	HEFT	MRCD	*D w*	RCDMod	MRCD	*D w*	RHEFT	MRCD	*D w*
G_{1034}	24	1504.1,	**1499.8,**	41.3	44389.8,	22758.4,	-55.8	8743.9,	**5309.0,**	2.1
		0.8625	**0.8662**	0.4	0.9232	0.9192	0.9	0.8897	**0.8901**	0.7
	45	1335.8,	**1335.8,**	35.2	44389.8,	23805.3,	-51.5	9291.4,	**4623.8,**	-11.0
		0.8466	**0.8803**	0.4	0.9355	0.9298	0.9	0.9173	**0.9181**	0.8
	66	1335.8,	**1335.8,**	31.5	22229.9,	9640.3,	-49.8	5673.5,	**4940.6,**	-29.3
		0.8516	**0.8939**	0.4	0.9523	0.9469	0.9	0.9305	**0.9392**	0.8
	87	1335.8,	**1335.8,**	28.4	22229.9,	7797.8,	-54.9	4885.1,	**4137.6,**	-25.5
		0.8461	**0.9004**	0.4	0.9523	0.9484	0.9	0.9340	**0.9387**	0.8
	108	1335.8,	**1335.8,**	28.4	44389.8,	11617.2,	-54.4	6556.8,	4108.4,	-15.6
		0.8450	**0.9025**	0.4	0.9523	0.9473	0.9	0.9344	0.9334	0.8
R_{546}	24	1122.0,	**1122.0,**	36.4	39858.0,	20513.0,	54.8	4453.0,	**2628.0,**	16.5
		0.8770	**0.8773**	0.4	0.9307	0.9271	0.9	0.8886	**0.8919**	0.6
	45	629.0,	**627.0,**	57.7	39858.0,	20805.0,	-49.8	2920.0,	**2263.0,**	1.44
		0.8724	**0.8735**	0.2	0.9419	0.9365	0.9	0.8970	**0.9097**	0.6
	66	627.0,	**627.0,**	46.5	19929.0,	8541.0,	-51.3	2847.0,	**1533.0,**	5.8
		0.8756	**0.8819**	0.4	0.9571	0.9521	0.9	0.9095	**0.9142**	0.6
	87	627.0,	**627.0,**	49.3	19929.0,	6935.0,	-56.0	2409.0,	**1347.0,**	2.91
		0.8703	**0.8832**	0.3	0.9571	0.9535	0.9	0.9139	**0.9183**	0.6
	108	627.0,	**627.0,**	39.7	39858.0,	10804.0,	-47.2	2628.0,	**1274.0,**	7.3
		0.8682	**0.8908**	0.4	0.9571	0.9527	0.9	0.9130	**0.9181**	0.6
Di_{529}	24	1881.0,	**1881.0,**	51.2	38617.0,	19418.0,	-63.2	29492.0,	19418.0,	-63.2
		0.8779	**0.8791**	0.2	0.9328	0.9292	0.9	0.9311	0.9292	0.9
	45	1571.1,	**1571.1,**	66.7	38617.0,	20367.0,	-52.1	31828.0,	20367.0,	-52.1
		0.8655	**0.8702**	0.3	0.9437	0.9385	0.9	0.9413	0.9385	0.9
	66	1505.1,	1505.2,	65.3	19418.0,	8833.0,	-54.6	16133.0,	8833.0,	-54.6
		0.8676	0.8747	0.3	0.9584	0.9539	0.9	0.9577	0.9539	0.9
	87	1485.1,	1485.2,	69.7	19418.0,	7081.0,	-59.6	13067.0,	7081.0,	-59.6
		0.8603	0.8711	0.5	0.9584	0.9551	0.9	0.9571	0.9551	0.9
	108	1485.1,	1485.2,	66.1	38617.0,	10293.0,	-57.8	22630.0,	10293.0,	-57.8
		0.8634	0.8754	0.5	0.9584	0.9541	0.9	0.9568	0.9541	0.9

e dominadas em cada heurística, para os distintos grafos. As outras soluções, por ser incomparáveis ou idênticas, não foram consideradas.

O Método de Normalização de BSA

Tabela 4.9: Relação de Dominância de MRCD sobre MRCD_{BSA} nos três cenários para G_n, R_n and Di_n sobre 24 processadores nos três cenários

	CPC			CMC			CPH			
Soluções	G_n	R_n	Di_n	G_n	R_n	Di_n	G_n	R_n	Di_n	Total
MRCD Dominantes	12	11	8	15	14	2	18	11	9	100
MRCD_{BSA} Dominadas	11	15	8	18	14	2	21	13	9	111

Para analisar a normalização da função em BSA, foi implementada uma versão de MRCD denotada por MRCD_{BSA}, considerando os seguintes valores normalizados:

$$EFT_n^{BSA}(v, p_j) = \frac{EFT(v, p_j)}{\max_{p \in P}\{EFT(v, p)\}}, \tag{4.9}$$

e

$$R_n^{BSA}(v, p_j) = \frac{RC(v, p_j)}{\max_{p \in P}\{RC(v, p)\}}, \tag{4.10}$$

como definidos em [39].

A Tabela 4.9 mostra as vantagens de usar a normalização proposta em MRCD que considera ambos os valores, mínimo e máximo, do tempo de fim e custo de confiabilidade, e não somente os valores máximos como em BSA. As soluções foram geradas para todos os GADs G_n, R_n e Di_n (os mesmos 15 grafos) analisados nos três cenários. A tabela mostra o número total de soluções *Dominantes* (100) de MRCD sobre MRCD$_{BSA}$, e o número total de soluções *Dominadas* (111) de MRCD$_{BSA}$ por MRCD. Note que o número total de soluções dominantes e dominadas não é igual para as heurísticas na tabela, a diferença são soluções incomparáveis que não foram consideradas. É importante ressaltar que as soluções para MRCD não foram nunca dominadas (Tabela 4.9).

Foi observado que, como a normalização de BSA divide somente pelo valor máximo do objetivo dado, BSA alcança os piores resultados. Portanto, o procedimento de normalização da função de custo para escalonar as tarefas deve considerar ambos valores mínimo e máximo dos respectivos objetivos para uma melhor precisão na priorização dos objetivos.

Comparação entre MRCD e BSAMod

Comparando MRCD com BSAMod nos três cenários, pode ser observado na Tabela 4.10 que MRCD obteve um total de 72 soluções dominantes, enquanto BSAMod obteve somente 11 dominantes. Em relação a quantidade de soluções dominadas, MRCD teve um desempenho semelhante, BSAMod atinge 78 soluções dominadas por MRCD, para somente 11 soluções de MRCD dominadas por BSAMod. Como em todos os casos MRCD é bem mais dominante que dominada, a tabela mostra a vantagem de se usar a função de custo e o método de normalização de MRCD para especificar um escalonamento apropriado. Note que, mesmo nos casos onde não domina, MRCD é raramente dominada por BSAMod, sendo a maioria das soluções incomparáveis.

Note que os melhores resultados foram obtidos nos cenários, homogêneo para velocidade (CPH) e conflitante (CPC), onde MRCD se destaca pela vantagem na dominância. No caso do cenário CMC que se caracteriza por ser mais equilibrado, as soluções de BSAMod e MRCD ficam mais próximas e portanto aumenta o número de soluções incomparáveis, diminuindo a relação de dominância. Mesmo assim, neste cenário MRCD obtém um número maior de soluções dominantes e menor de dominadas que BSAMod. Observe que em todos os cenários, G_n se comporta melhor que os outros GADs, seguido de R_n e por último Di_n.

Em Di_n pela topologia destes grafos e a homogeneidade das tarefas, aumenta o custo de confiabilidade e a solução de confiabilidade se prejudica mais que nos outros grafos G_n e R_n. Em Di_n um número elevado de tarefas adjacentes são alocadas sobre um mesmo processador, incrementando o custo de confiabilidade do processador. Isto

diminui a possibilidade da solução MRCD dominar BSAMod, embora fiquem próximas. Entretanto os *makespan*s para as soluções de MRCD são significativamente melhores, o que será analisado nos próximos testes.

Tabela 4.10: Relação de Dominância entre MRCD e BSAMod nos três cenários para G_n, R_n and Di_n sobre 24 processadores

		Dominantes		**Dominadas**	
	GADs	BSAMod	MRCD	BSAMod	MRCD
CPC	G_n	0	14	15	0
	R_n	1	8	10	1
	Di_n	1	4	4	1
CMC	G_n	2	5	5	2
	R_n	3	4	4	3
	Di_n	0	3	3	0
CPH	G_n	0	15	19	0
	R_n	2	10	11	2
	Di_n	2	9	7	2
Total		**11**	**72**	**78**	**11**

Tabela 4.11: Comparação entre BSAMod e MRCD para G_{702} no cenário de pior caso CPC

w	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0	Average
M(%)	0.0	18.5	49.0	75.0	106.0	142.0	113.0	10.0	17.0	13.0	0.0	49.4
R_T (%)	0.0	2.8	3.9	5.6	7.8	8.2	5.2	-0.8	0.3	0.2	0.0	3.01
D	-	49.1	45.7	42.8	41.0	38.7	27.8	-4.80	-36.3	-57.3	-	-

No segundo conjunto de experimentos, um conjunto de soluções S foi gerado dividindo o intervalo $(0, 1)$ de w na forma $W = \{0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1\}$, para cada uma das heurísticas MRCD e BSAMod na Classe 2 no pior cenário CPC. Note que neste caso não foram selecionadas as soluções em S usando os conceitos de dominância (o conjunto S'). A solução de MRCD é comparada somente com a solução de BSAMod para o mesmo valor de w. A Tabela 4.11 mostra um exemplo (G_{702}) de como foram calculadas as percentagens das diferenças para cada objetivo, *makespan* e confiabilidade, em cada valor de w. A linha com $\mathcal{M}$ apresenta a percentagem da melhora do *makespan* obtido por MRCD sobre BSAMod, e para R_T, a percentagem de deteriorização para a confiabilidade de MRCD sobre BSAMod. Uma variação de n (Tabela 4.12) e outra m (Tabela 4.13), foram também consideradas nestes experimentos no pior cenário CPC.

Na Tabela 4.12 são apresentadas as médias finais, variando n nos três tipos de aplicações diferentes G_n, R_n e Di_n. Note que na maioria dos casos, uma melhoria notável do *makespan* é alcançado por MRCD, enquanto a confiabilidade permanece quase a mesma que BSAMod. Somente R_n apresenta uma melhoria menor que os outros grafos, e alguns casos onde o *makespan* de MRCD não supera o de BSAMod, mas por valores pequenos. Em casos raros (R_n) o *makespan* de BSAMod é melhor a MRCD, porém nestes casos se observou que MRCD teve um número maior de soluções dominantes nos experimentos anteriores. Ademais, este teste apresenta em média um número considerável de percentais (%) com melhora no *makespan*. Um conjunto similar de experimentos considerando

Tabela 4.12: Comparação das médias dos percentais (%) entre MRCD e BSAMod para G_n, R_n e Di_n sobre 24 processadores nos três cenários.

		G_n					R_n				
	n	152	252	377	527	702	80	98	152	256	364
CPC	Med. $\mathcal{M}$ (%)	31.5	38.6	44.4	48.0	49.4	9.77	8.01	3.66	10.0	-4.26
	Med. R_T (%)	0.28	0.60	1.06	1.91	3.01	0.15	0.0	0.11	-4.65	-0.92
CMC	Med. $\mathcal{M}$ (%)	0.07	1.2	1.99	2.72	4.58	-2.76	-2.06	-0.82	5.52	1.08
	Med. R_T (%)	0.07	-0.18	0.32	1.1	2.65	0.05	-0.2	0.19	0.81	1.78
CPH	Med. $\mathcal{M}$ (%)	1.41	3.8	38.1	9.19	40.7	-2.13	-1.24	-1.32	12.4	2.80
	Med. R_T (%)	0.1	0.36	0.82	0.58	1.94	0.20	-0.11	0.22	-0.62	1.66

		Di_n				
	n	81	100	144	256	361
CPC	Med. $\mathcal{M}$ (%)	35.6	75.8	53.0	138.6	96.3
	Med. R_T (%)	0.99	5.92	2.23	4.94	8.07
CMC	Med. $\mathcal{M}$ (%)	1.83	1.73	3.06	15.1	13,2
	Med. R_T (%)	0.09	0.21	0.48	2.32	4.79
CPH	Med. $\mathcal{M}$ (%)	13.1	17.5	24.8	68.5	51.4
	Med. R_T (%)	0.33	0.52	1.32	4.77	9.66

a variação de m aparece na Tabela 4.13 para G_{1034}, R_{546}, e Di_{529} em CPC chegando a conclusões semelhantes. Em geral as melhorias no *makespan*, fornecidas por MRCD foram significativas, a um custo de uma pequena deterioração da confiabilidade.

Tabela 4.13: Comparação entre as médias dos percentais (%) entre BSAMod e MRCD variando o número de processadores no cenário de pior caso (CPC)

	G_{1034}			R_{546}			Di_{529}		
m	24	87	213	24	87	213	24	87	213
Med. $\mathcal{M}$(%)	47.2	41.1	41.1	-3.34	5.55	8.25	105	71.5	68.3
Med. R_T (%)	0.61	1.41	1.28	-0.13	2.04	0.52	1.23	3.08	2.79

Em resumo, foram apresentados os beneficios da aplicação do algoritmo *list scheduling* implementado junto com a função de custo $f(v, p_v)$ em MRCD e a classificação das soluções. Foi observado também que, como o procedimento de normalização de BSAMod não considera uma distribuição entre os verdadeiros valores extremos, esta escolha conduz a resultados piores que MRCD. A normalização dos objetivos como proposta neste trabalho, traz vantagens em relação a maior dominância das soluções, com uma melhoria significativa nas soluções do *makespan*.

4.5 Conclusões

No algoritmo proposto, ao escalonar as tarefas nos recursos, inicialmente nenhum dos objetivos tem uma prioridade específica. Diferentemente da função hierárquica de [61], o algoritmo deste trabalho escolhe o processador que apresenta menor custo para escalonar a tarefa de acordo com uma função integrada que incorpora ambos os objetivos simultaneamente, tanto minimização do tempo de execução quanto maximização da confiabilidade. A função é ponderada para indistintamente valorizar seus objetivos no problema

de escalonamento bi-objetivo.

Através da função de custo e uma abordagem flexível, a estratégia consegue gerar escalonamentos mais promissores. Para ajudar na busca de melhores soluções de escalonamentos, foi proposta uma metodologia para ajustar os pesos na função de custo. A partir da informação adicional D proposta, e uma classificação usando conceitos de dominância, o algoritmo sugere possíveis soluções de compromisso convenientes para o problema bi-objetivo.

Comparado com outras heurísticas, as melhorias no *makespan* fornecidas por MRCD foram significativas, a um custo de uma pequena deterioração da confiabilidade, como observado nos experimentos. Além do mais, as soluções MRCD em muitos casos foram dominantes ou incomparáveis, e em um número desprezível foram soluções dominadas. Particularmente, MRCD obteve um número maior e considerável de dominâncias que as heurísticas comparadas.

Os experimentos realizados destacam a importância do uso de uma estratégia flexível e os resultados se mostram favoráveis, quando comparados com outras heurísticas bi-objetivo existentes em ambientes distribuídos heterogêneos. Na maioria dos casos foi possível achar uma solução MRCD muito próxima das soluções das outras heurísticas, sendo que em um número maior de casos MRCD domina a outra solução.

Para o escalonamento mono-objetivo HEFT, os resultados mostram que é possível considerar a confiabilidade sem prejudicar o desempenho. Por outro lado, embora as outras heurísticas considerem a confiabilidade além do tempo, os resultados gerados com RCDMod e com RHEFT mostram a importância de se usar nestes ambientes um escalonamento com uma função de custo ponderada, que permita mudar as prioridades dos objetivos do problema de acordo com distintos interesses. A flexibilidade de MRCD fornece mais opções de escalonamento para uma mesma aplicação e sistema computacional, com uma probabilidade maior de encontrar soluções próximas as esperadas pelo usuário.

Este trabalho mostra que existem maneiras de se ajudar o usuário a achar um compromisso entre a confiabilidade e o *makespan*. Segundo pesquisas realizadas [28], isto é uma questão pouco explorada no escalonamento de aplicações paralelas sobre sistemas distribuídos.

Em resumo, a estratégia de escalonamento:

1. Propõe uma função de custo ponderada e uma normalização dos objetivos diferente das outras propostas, cujos resultados mostram ser superiores em relação à dominância e com soluções de compromisso mais eficientes, na maioria dos casos; MRCD oferece melhoras no *makespan* enquanto mantém resultados similares de confiabilidade.

2. Propõe uma métrica no algoritmo, denotada Diferença Média dos objetivos D, cujo propósito principal é auxiliar no ajuste dos pesos na função de custo. A partir da mesma, o usuário pode tomar diferentes decisões para escolher uma solução de escalonamento conveniente.

3. Propõe uma classificação dos pesos aplicada a função de custo para achar soluções de compromisso de mais qualidade (não dominadas). Em particular usando conceitos de dominância, é definida uma metodologia para ajustar os pesos, que utiliza a classificação e a métrica propostas. O método sugere possíveis soluções de escalonamento mais adequadas dentre soluções de maior qualidade.

4. Apresenta um estudo de casos com variações dos pesos que mostra a importância de utilizar uma abordagem flexível (múltiplas soluções) para escalonar aplicações em sistemas distribuídos heterogêneos. Na maioria dos casos, MRCD pode encontrar soluções eficientes e dominantes quando comparadas com outras heurísticas.

Os resultados [65, 66] mostram a superioridade da proposta identificada em item 1 sobre outras heurísticas comparadas. É importante ressaltar que em trabalhos correlatos, os pontos 2, 3 e 4 não são nem considerados. Desta forma, diferentemente da literatura, a proposta responde a questão de *como* achar determinada solução de escalonamento que represente um compromisso conveniente entre o *makespan* e a confiabilidade. A abordagem pode ser aplicada a trabalhos existentes, desde que se utilize o mesmo processo de normalização, e o cálculo de uma informação adicional como D para medir a diferença ou desequilíbrio dos objetivos no escalonamento.

No próximo capítulo, MRCD será utilizado junto com uma abordagem tolerante a falhas baseada no esquema primária-*backup*, com o objetivo de recuperar múltiplas falhas e executar eficientemente aplicações paralelas sobre sistemas distribuídos heterogêneos.

Capítulo 5

Escalonamento Tolerante a Falhas Proposto

Em plataformas heterogêneas de larga escala como grades computacionais e computação nas nuvens, as falhas de recursos podem ocorrer [63], inviabilizando a execução das aplicações. Consequentemente, existe uma grande necessidade de desenvolver técnicas para alcançar tolerância a falhas nestes sistemas. Muitas heurísticas de escalonamento da literatura, como [40, 46, 71, 75], empregam simples modelos onde aspectos sobre confiabilidade e tolerância a falhas não são considerados. Geralmente estes modelos pressupõem que os processadores e os canais de comunicação são seguros. Com o objetivo de garantir a execução confiável e eficiente das aplicações, estudos realizados [8, 9, 14, 23, 34, 48, 60] mostram a importância de revisar as heurísticas de escalonamento existentes quando utilizadas em ambientes que possuem uma variedade de recursos interligados.

Alguns trabalhos em escalonamento de aplicações para ambientes distribuídos já apresentam diferentes formas de melhorar o tempo de execução das aplicações, incluindo aspectos de confiabilidade e tolerância a falhas [8, 9, 33, 34, 48, 53, 60]. Mesmo assim, muitos algoritmos são baseados em modelos que não representam certas características importantes tanto em relação à aplicação quanto ao sistema heterogêneo. Ao escalonar tarefas com heurísticas com mecanismos para tratamento de falhas, o tempo de execução total da aplicação tende a aumentar, mesmo quando nenhuma falha ocorre durante a execução da aplicação. Assim, um custo está associado a adição destas técnicas no escalonamento, sendo fundamental um estudo aprofundado de métodos que permitam minimizá-lo.

Diversos algoritmos de escalonamento que consideram tolerância a falhas, empregam técnicas de replicação baseadas no esquema primária-*backup* [8, 9, 60, 61], sendo um dos mais importantes esquemas empregados em escalonamento com tolerância a falhas.

Nesta tese é também proposta uma metodologia de tolerar múltiplas falhas baseada em um modelo arquitetural real que considera não somente custos computacionais e de comunicação, mas também de confiabilidade. Note que para o esquema primária-*backup* da técnica de replicação ativa [8, 9, 34] não existem mecanismos para detectar e tratar as falhas, ambas cópias da tarefa são executadas simultaneamente, o que pode sobrecarregar bastante o sistema distribuído. Já com a replicação passiva [33, 48, 53, 60], justificando a

proposta, a *backup* da tarefa somente é ativada quando detectada falha na cópia primária, por mecanismos de tratamento de falhas.

Neste trabalho, é proposto um algoritmo de escalonamento estático tolerante a falhas que utiliza o esquema primária-*backup* para executar aplicações paralelas sobre ambientes distribuídos heterogêneos de larga escala. O escalonamento agrega a técnica de replicação passiva para conseguir a execução completa e consistente da aplicação na presença de falhas. A estratégia proposta inicialmente é baseada no algoritmo de [60, 61] que emprega uma heurística do tipo *list scheduling* para escalonar as tarefas primárias e *backups*, com tolerância de somente uma falha permanente de processador. Para reduzir o custo de tolerância a falhas, este trabalho de tese propõe uma nova abordagem que adiciona também o escalonamento bi-objetivo proposto no capítulo anterior.

Diferentemente de [60], a abordagem de escalonamento proposta aqui não é projetada para sistemas de tempo real e acrescenta maior flexibilidade com a introdução de novos conceitos e critérios para o escalonamento de *backups*, sendo possível tolerar múltiplas falhas. O objetivo principal deste trabalho é garantir uma execução eficiente e consistente da aplicação na ocorrência de falhas em sistemas distribuídos, que permita reduzir o custo do mecanismo tolerante a falha durante a recuperação. Técnicas de escalonamento como a sobreposição de *backups* [33, 53, 60] são reformuladas para funcionar no ambiente computacional considerado e melhorar seu desempenho.

5.1 Estratégia de Escalonamento Proposta

A abordagem proposta neste trabalho escalona estaticamente as tarefas primárias e também as *backups*, sendo que para estas últimas, técnicas são especificadas para oferecer tolerância a falhas e ao mesmo tempo reduzir o *makespan* da aplicação. Para atribuir prioridades diferentes aos objetivos do escalonamento, a abordagem bi-critério proposta no Capítulo 3 é também utilizada para as *backups*. O algoritmo de forma geral apresenta os seguintes objetivos iniciais:

1. encontrar uma alocação para as primárias das tarefas da aplicação sobre a arquitetura do modelo proposto;

2. de acordo com a alocação das primárias, encontrar uma alocação para as *backups* das tarefas da aplicação sobre a arquitetura do modelo proposto que permita tolerar uma falha permanente de processador;

3. minimizar o *makespan* e maximizar a confiabilidade da aplicação mesmo na presença de falhas.

O algoritmo de escalonamento estático tolerante a falha proposto é chamado neste trabalho *Fault Tolerant Makespan and Reliability Cost Driven* (FTMRCD) e representa a última etapa conforme visto no Algoritmo 3 que é denominado *FTframework*. O Algoritmo 3 divide-se em três etapas principais: ordenação das tarefas segundo um critério de prioridade, escalonamento das tarefas primárias ordenadas no processador que otimiza a função

de custo ponderada e, logo depois o escalonamento das *backup*s, ordenadas pela prioridade no processador que otimiza a função de custo ponderada.

Algoritmo 3 : *FTframework(G,P,w_1,w_2)*

1 $V_{ordG} = \langle v_0, \ldots, v_{n-1} \rangle / blevel(v_i) \leq blevel(v_{i+1}), i = 0, \ldots, n-2$;
2 $\langle$ *Sch*, *S*(*Sch*), $\rangle$=MRCD(V_{ordG},*P*,w_1);
3 $\langle$ *SchBck*, *LSucFalha* $\rangle$=FTMRCD(V_{ordG},*P*,*Sch*, w_2);

O *FTframework* é preparado para a tolerância de uma falha permanente de processador, onde inicialmente as tarefas são ordenadas em V_{ordG}, de acordo com a prioridade de *blevel*() especificada no Algoritmo 2 do Capítulo 4. A seguir as cópias primárias são escalonadas conforme o algoritmo bi-objetivo ponderado MRCD(V_{ordG},*P*,w_1), onde w_1 é o parâmetro de ponderação da função de custo. *Sch* é o escalonamento obtido por MRCD e $S(Sch) = (\mathcal{M}, R_T)$ a solução de *Sch*. A seguir, as cópias *backups* são escalonadas de acordo com o escalonamento das primárias *Sch*. Assim, o escalonamento de *backups* é realizado por FTMRCD(V_{ordG},*P*,*Sch*,w_2) logo depois de escalonar as primárias, a partir de um conjunto de critérios e métodos para tolerância a falha definidos com detalhes na próxima seção.

Na última etapa, o algoritmo denotado FTMRCD(V_{ordG},*P*,*Sch*,w_2) utiliza também uma heurística do tipo *list scheduling* similar a MRCD(V_{ordG},*P*,w_1), onde w_2 é o parametro de ponderação usado na função de custo para escalonar as *backups*. O objetivo principal deste escalonamento é garantir uma execução consistente e eficiente da aplicação na ocorrência de falhas. Em particular, a sobreposição de *backups*, usado na literatura em esquemas primária-*backup* para replicação de tarefas, é um dos critérios considerados no algoritmo para melhorar o escalonamento. O escalonamento gerado como saída é representado por *SchBck*. A lista de tarefas *LSucFalha* é uma informação gerada também por FTMRCD, a qual deve ser utilizada posteriormente para recuperar a aplicação da falha. O algoritmo FTMRCD, assim como os mecanismos propostos para tolerância a falha, são apresentados separadamente na próxima seção.

5.2 Critérios para Escalonar as Tarefas *Backups*

Esta seção faz uma análise detalhada da terceira etapa do $FTframework$ no Algoritmo 3, onde são definidos os principais aspectos do tratamento de falhas no escalonamento das *backups* de FTMRCD.

Para alcançar a tolerância a falha em FTMRCD, primeiramente é definida uma serie de critérios que permitem escalonar as *backup*s da aplicação, de forma a garantir um escalonamento tolerante a falhas consistente e com bom desempenho. Neste capítulo, as cópias primária e *backup* de uma tarefa v_i são denotadas como v_i^P e v_i^B, respectivamente. A precedência entre duas tarefas primárias v_j^P e v_i^P, tal que $v_j^P \in Pred(v_i^P)$ é denotada por $v_j^P \rightarrow v_i^P$.

O conceito ***primária forte*** para o esquema primária-*backup* foi introduzido em [60] para classificar a tarefa primária v^P durante o escalonamento das *backups*. A tarefa

primária v_i^P é ***classificada como primária forte*** se:

1. não tem predecessores, ou
2. tem predecessores e dada a precedência $v_j^P \rightarrow v_i^P$,
   (a) v_i^P e v_j^P estão alocados no mesmo processador e v_j^P é uma tarefa primária forte, ou
   (b) v_i^P e v_j^P estão alocados em processadores diferentes e a *backup* v_j^B está escalonada antes de v_i^P, tal que v_i^P consegue receber as mensagens de v_j^B.

Por definição, uma tarefa primária forte sempre poderá executar, exceto quando seu processador falha sem ela ter finalizado sua execução. Neste caso, a *backup* correspondente deve ser ativada para executar. Note que a classificação em [60] limita-se a uma mesma classificação da primária v_i^P em relação a todos os predecessores. Diferentemente, neste trabalho a classificação proposta da primária varia de acordo com a relação com os distintos predecessores. Portanto, a seguir, é reformulada e estendida com a adição de novos conceitos.

5.2.1 Classificação de Primárias: Critério C_1

Para classificar uma primária v^P, é definido neste trabalho o critério C_1, que utiliza as relações R_1 e R_2 entre duas tarefas primárias que são introduzidas a seguir. O objetivo principal é oferecer maior flexibilidade na formulação dos distintos critérios de escalonamento das *backups*, para assim conseguir maior desempenho e consistência durante a execução da aplicação em ambientes distribuídos.

A aplicação do critério C_1 deve ser realizada a cada iteração do algoritmo FTMRCD, onde as tarefas do GAD G são visitadas de acordo com a ordem previamente estabelecida em V_{ordG}. Desta forma a classificação das tarefas primárias usando C_1 percorre V_{ordG}, cumprindo a ordem de precedência das tarefas da aplicação, e assim, no caso de v_i^P ter predecessores, v_i^P só poderá ser classificada com C_1 depois de terem sido classificados todos seus predecessores v_j^P.

Relação R_1: Dada a precedência entre duas primárias $v_j^P \rightarrow v_i^P$, v_i^P é ***forte para*** v_j^P se:

1. v_j^P e v_i^P estão alocados no mesmo processador, e
   (a) v_j^P não tem predecessores (caso de uma tarefa origem em G), ou v_j^P é *forte para* todos seus predecessores, ou
   (b) v_j^B está escalonada antes de v_i^P, tal que v_i^P consegue receber a mensagem de v_j^B, ou
2. v_j^P e v_i^P estão alocados em processadores diferentes, e

(a) v_j^B está escalonada antes de v_i^P, tal que v_i^P consegue receber a mensagem de v_j^B.

Pela relação anterior R_1, se a primária v_i^P em G não é ***forte para*** v_j^P, então v_i^P é ***fraca para*** v_j^P, veja a seguir quando uma primária é *fraca* em relação a seu predecessor.

Relação R_2: Dada a precedência entre duas primárias $v_j^P \rightarrow v_i^P$, v_i^P é ***fraca para*** v_j^P se:

1. v_j^P e v_i^P estão alocados no mesmo processador, e
   (a) v_j^P é *fraca para* algum predecessor e
   (b) v_i^P não consegue receber a mensagem de v_j^B, ou
2. v_j^P e v_i^P estão alocados em processadores diferentes, e
   (a) v_i^P não consegue receber as mensagem de v_j^B.

As relações ***forte para*** ou ***fraca para*** são utilizadas para classificar as tarefas primárias em C_1, e por outros critérios de escalonamento que serão definidos a seguir.

Critério C_1: A primária v_i^P é ***classificada*** em:

1. ***totalmente forte***, se
   (a) não tem predecessores, ou
   (b) tem predecessores e dada a precedência $v_j^P \rightarrow v_i^P$, v_i^P é *forte para* todos seus predecessores v_j^P;
2. ***parcialmente forte***, se v_i^P não é *totalmente forte*, mas pelo menos é *forte para* algum predecessor v_j^P;
3. ***totalmente fraca***, se v_i^P é *fraca para* todos seus predecessores v_j^P.

Por C_1, de acordo com a ordem em V_{ord}, as tarefas origens do grafo (sem predecessores) da aplicação G são classificadas em primárias *totalmente fortes.*

Para ilustrar a classificação considerada, o GAD $G = (V, E)$ da Figura 5.1 e o escalonamento também visto na figura, v_1^P e v_2^P são primárias *totalmente fortes* e v_3^P é uma *primária parcialmente forte.* A tarefa v_1^P é o caso 1 (a) do critério C_1, onde a tarefa não tem predecessores. Já v_2^P é o caso 1(b) do critério C_1, onde a tarefa tem somente v_1 como predecessor e não deixa de receber a mensagem da *backup* v_1^B em caso de falha de $p(v_1^P)$. Como v_1 é o único predecessor de v_2, v_2^P é totalmente forte. Se v_1^P não finaliza com sucesso, é necessário executar v_1^B que ao terminar envia a mensagem para v_2^P. Já v_3^P é uma primária *parcialmente forte* por C_1, é *forte para o predecessor* v_1^P mas é *fraca para* v_2^P, pois v_2^B termina depois do tempo de inicio de v_3^P.

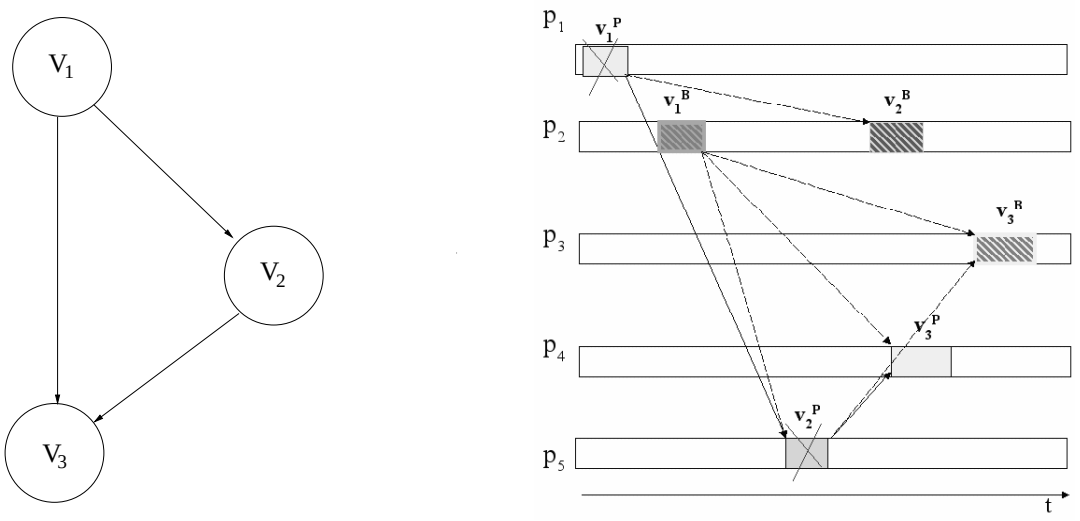


Figura 5.1: Critério de classificação C_1 aplicado no GAD

C_1 será utilizado por outros critérios ainda a serem especificados neste capítulo para escalonar as *backups*. Esta classificação permitirá também determinar as tarefas predecessoras e sucessoras de uma tarefa escalonada em caso de falha, assim como reduzir o *makespan*. O objetivo principal é garantir o funcionamento e reduzir o custo do mecanismo que trata a falha, facilitando a formulação dos critérios de escalonamento de *backups*.

Uma falha de processador provoca a falha das tarefas primárias escalonadas nele que ainda não executaram, e pode também provocar a impossibilidade de execução de tarefas primárias sucessoras que deviam executar em outros processadores. Para definir as classes de falha para uma tarefa primária, diferente da falha de processador, são propostos neste trabalho a seguir dois tipos de falha para v_i^P:

1. ***falha por hardware***: é provocada pela falha de seu processador $p(v_i^P)$ (onde v_i^P foi escalonada).

2. ***falha induzida***: é provocada por outra primária v_j^P, predecessora de v_i^P, que não envia a mensagem a v_i^P no tempo previsto e, portanto, v_i^P não pode executar.

Note que no segundo caso (*falha induzida*), v_i^P é *fraca para* v_j, logo por C_1 v_i^P é *parcialmente forte* ou *totalmente fraca*. A partir destas classes de falhas de uma tarefa primária, podem ser especificadas as seguintes classificações:

Dada a precedência $v_j^P \rightarrow v_i^P$ em G, se v_i^P é *fraca para* v_j^P, então v_i^P é ***candidata a falha induzida por*** v_j^P, e v_j^P é ***candidata a provocar falha induzida de*** v_i^P.

De maneira recorrente, dada a precedência $v_k^P \rightarrow v_j^P \rightarrow v_i^P$, se v_i^P é *candidata a falha induzida por* v_j^P, então v_i^P é também *candidata a falha induzida pelos predecessores* v_k^P de v_j^P, se v_j^P é fraca para v_k^P. Ao mesmo tempo esses predecessores v_k^P são denotados *candidatos a provocar a falha induzida de* v_i^P.

Por outro lado, quando o processador $p(v_i^P)$ falha, se v_i^P não executa completamente, então v_i^P sofre *falha por hardware*. Assim qualquer primária da aplicação G, independente

da classificação com C_1, é sempre ***candidata a falha por hardware***. No caso da falha de $p(v_i^P)$, as outras tarefas v_k^P escalonadas nos outros processadores $p(v_k^P)$ diferentes de $p(v_i^P)$, se não classificadas totalmente fortes (*parcialmente fortes* ou *totalmente fracas*) então são *candidatas a falha induzida*.

5.2.2 Exclusão Mútua: Critérios de Escalonamento C2 e C3

Os critérios C_2 e C_3, utilizados para o escalonamento de *backups* são formulados a seguir, baseados nas restrições de exclusão espacial e temporal do esquema primária-*backup* para a técnica de replicação passiva. Estes critérios garantem o funcionamento do mecanismo de tolerância a falhas e são comumente utilizados na literatura por algoritmos de escalonamento tolerantes a falhas que usam replicação passiva [33, 48, 53, 60]. Com as definições de tempo de ínicio $EST(v_i, p(v_i))$ e tempo de fim $EFT(v_i, p(v_i))$, sendo $p(v_i)$ o processador que tem escalonada v_i, as cópias primária e *backup* de v_i devem cumprir os critérios de exclusão mútua C_2 e C_3.

Critério C_2: A *backup* v_i^B ***deve ser escalonada em processador diferente*** da sua respectiva primária v_i^P (exclusão mútua no espaço), ou seja $p(v_i^B) \neq p(v_i^P)$.

O critério C_2 garante que, no caso de uma falha de processador, uma das cópias da tarefa v_i seja primária ou *backup*, possa executar.

Critério C_3: A *backup* v_i^B ***deve ser escalonada depois da sua primária*** v_i^P. O tempo de início de v_i^B deve ser superior a soma do tempo de fim da primária v_i^P mais o tempo estimado gasto para detectar a falha (exclusão mútua no tempo), ou seja, $EST(v_i^B, p(v_i^P)) \geq EFT(v_i^P, p(v_i^P)) + tDetFalha(p(v_i^P))$.

Os critérios de exclusão mútua, tanto no espaço (C_2) quanto no tempo (C_3) para escalonar as *backups* são obviamente necessários para tolerar a falha permanente de processador. Em caso de falhar o processador $p(v_i^P)$ antes de v_i^P finalizar, a *backup* v_i^B correspondente será ativada em outro processador $p(v_i^B)$, em um momento mais tarde $EST(v_i^B)$.

5.2.3 Seleção de Processadores: Critérios de Escalonamento C4 e C5

O resto dos critérios de escalonamento apresentados neste capítulo, de C_4 a C_7, são definidos neste trabalho baseados no critério de classificação C_1. Eles são propostos com o objetivo principal de garantir a consistêcia do funcionamento do mecanismo e achar os processadores onde é possível escalonar a *backup*, de forma a oferecer uma execução da aplicação completa e eficiente, mesmo na ocorrência de falhas. Em particular, os critérios C_4 e C_5 permitem selecionar os processadores disponíveis que podem ser utilizados para escalonar as *backups*.

Critério C_4: Dada a precedência $v_j^P \rightarrow v_i^P$, a *backup* v_i^B ***não pode ser escalonada no mesmo processador que o predecessor*** v_j^P, se as primárias v_i^P e v_j^P:

1. estão alocadas no mesmo processador, ou
2. estão alocadas em processadores distintos e, v_i^P é *fraca para* v_j^P.

O critério C_4 (2) quer dizer que, se v_j^P (*candidata a provocar a falha induzida* de v_i^P) falhar, v_j^B será executada e provocará a falha de v_i^P. Logo é necessário que v_i^B seja executada no lugar de v_i^P, portanto v_i^B não pode ser escalonada em $p(v_j^P)$.

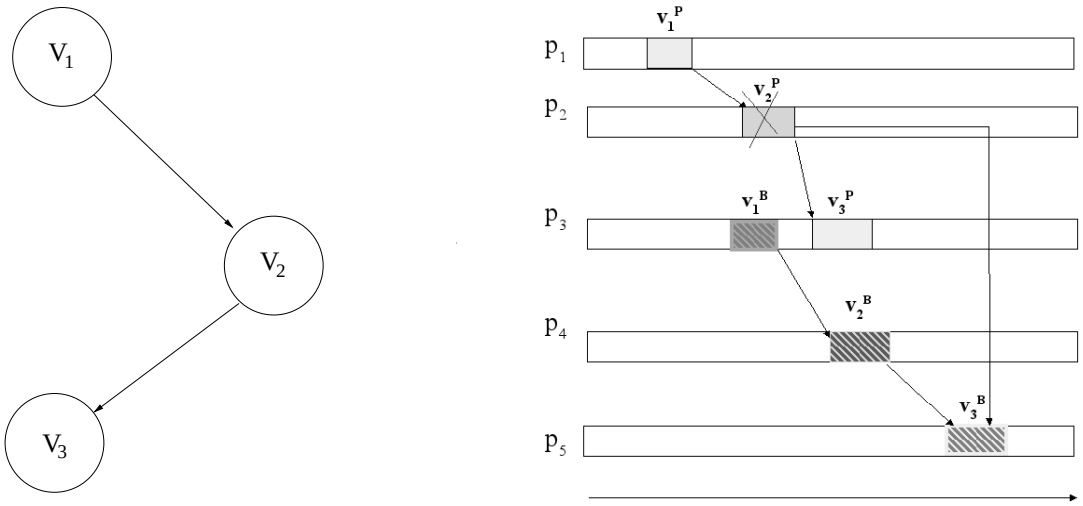


Figura 5.2: Critérios C_4 e C_5 aplicados no GAD

Na Figura 5.2, v_3^P e v_2^P estão escalonadas em processadores diferentes e v_3^P é *fraca para* v_2^P pois v_2^B foi escalonado depois de v_3^P (caso 2 de C_4). Se p_2 ($p(v_2^P)$) falhar, é necessário que v_2^B execute. Neste caso v_3^B não pode ser escalonada no mesmo processador p_2 que v_2^P, já que em caso de falha de p_2, v_3^B seria obrigada a executar.

Critério C_5: Dada as precedências $v_k^P \rightarrow v_j^P \rightarrow v_i^P$, a *backup* v_i^B ***não pode ser escalonada no mesmo processador que*** v_k^P se:

1. v_i^B não pode ser escalonada no processador do predecessor v_j^P, por C_4, e
2. v_j^P é *fraca para* seu predecessor v_k^P.

Assim, C_5 se aplica recorrentemente também a todos os predecessores de v_k^P que são *não totalmente fortes*, voltando pela precedência no grafo da aplicação G até chegar a tarefas *totalmente fortes*.

A Figura 5.2 é o caso de C_5, onde v_3^B não pode ser escalonada em $p(v_1^P)$, pois além de v_3^P ser *fraca para* v_2^P, também v_2^P é *fraca para* v_1^P. Se $p(v_1^P)$ falhar é necessário que v_1^B execute. Neste caso v_3^B não pode ser escalonada no mesmo processador que v_1^P, já que em

caso de falhar $p(v_1^P)$ também provoca a falha induzida de v_2^P e consequentemente a falha de v_3^P, assim v_3^B estaria obrigada a executar.

Note que, a classificação C_1 em C_4 e C_5, oferece flexibilidade e consistência ao definir as possíveis opções de escalonamento de uma *backup* sobre os diferentes processadores. A *backup* de uma tarefa v sendo escalonada, deixa de ser alocada somente nos processadores dos predecessores aos quais ela é *fraca*, e tem disponíveis para utilizar os processadores dos outros predecessores que ela é *forte*. Os critérios C_4 e C_5 têm que ser aplicados para manter a consistência do escalonamento das *backups*, senão podem vir a ocorrer falhas induzidas durante a execução da aplicação devido a falha de um processador.

5.2.4 Sobreposição de *Backups*: Critérios de Escalonamento C6 e C7

Os critérios C_6 e C_7 estabelecem no algoritmo FTMRCD as regras para sobreposição de *backups* com outras tarefas, sejam *backups* ou primárias, durante o escalonamento.

Critério C_6: A *backup* v_i^B ***não pode ser escalonada sobreposta com outra backup*** v_j^B já escalonada no processador $p(v_j^B)$, se:

1. v_j^P e v_i^P estão escalonadas no mesmo processador, ou seja, $p(v_j^P) = p(v_i^P)$, ou
2. existe a precedência $v_j^P \rightarrow v_i^P$, e v_i^P é *fraca para* v_j^P, ou
3. não existe nenhuma relação de precedência entre v_j^P e v_i^P, mas pelo menos uma das primárias, por exemplo v_i^P, sem perda de generalidade, foi classificada como *parcialmente forte* ou *totalmente fraca*, e se:
   (a) existe v_k^P predecessora de v_i^P escalonada com v_j^P, ou seja $p(v_k^P) = p(v_j^P)$, tal que v_k^P é *candidata a provocar falha induzida de* v_i^P, ou
   (b) ambas v_i^P e v_j^P não são classificadas *totalmente fortes*, e existe uma tarefa v_k^P, *candidata comum a provocar as falhas induzidas* de v_i^P e v_j^P, ou
   (c) ambas v_i^P e v_j^P não são classificadas *totalmente fortes*, e existem v_k^P e v_l^P, $v_k^P \neq v_l^P$, escalonadas no mesmo processador ($p(v_k^P) = p(v_l^P)$), tal que v_k^P é *candidata a provocar a falha induzida de* v_i^P, e v_l^P é *candidata a provocar a falha induzida* de v_j^P.

Obviamente, se o processador $p(v_j^P) = p(v_i^P)$ falhar, por C_6 (1) as correspondentes *backups* v_j^B e v_i^B terão que executar, assim não poderiam ficar sobrepostas. A restrição C_6 (2) é necessária, pois a mensagem da predecessora v_j^B não chegaria a v_i^P, visto que v_i^P é *fraca para* v_j^P, e assim em caso de falhar $p(v_j^P)$, as *backups* tanto v_j^B como v_i^B devem ser ativadas no lugar das primárias e não podem ter suas execuções sobrepostas.

Observe que se existe relação de precedência entre v_i^P e v_j^P, v_i^B não ficará escalonada sobreposta com sua predecessora v_j^B, pois pelo critério C_3 de exclusão mútua

$EFT(v_i^P, p(v_i^P)) \leq EST(v_i^B, p(v_i^B))$, e como v_i^P é *forte para* v_j^P, $EFT(v_j^B, p(v_j^B)) \leq EST(v_i^P, p(v_i^P))$ assim, $EFT(v_j^B, p(v_j^B)) \leq EST(v_i^B, p(v_i^B))$.

O critério C_6 (3) é importante para destacar os casos onde v_i^B e v_j^B não podem ficar sobrepostas, quando suas primárias não têm relação de precedência. Nestes casos (a), (b) e (c), ambas as *backups*, v_i^B e v_j^B, terão que executar. Note que, nos casos distintos a estes será possível a sobreposição, até de *backups* de primárias não classificadas totalmente fortes, o que pode melhorar ainda mais o *makespan* do escalonamento.

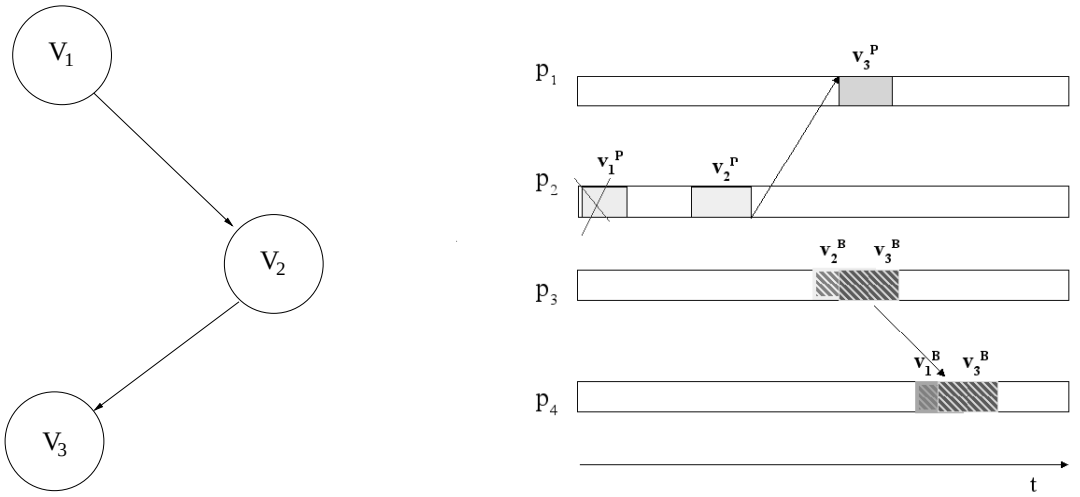


Figura 5.3: Critério C_6 aplicado no GAD

No exemplo da Figura 5.3, v_3^B não pode ser escalonada em p_3 sobre v_2^B pelo critério C_6 (2), pois existe relação de precedência entre v_2^P e v_3^P, v_3^P é *fraca para* seu predecessor v_2^P e então, em caso de falha do processador $p_2 = p(v_2^P)$, v_2^B e v_3^B têm que executar por ocorrer a falha induzida de v_3^P. Por C_6 (3), v_3^B também não pode ser escalonada sobre v_1^B, porque mesmo que não exista relação de precedência entre v_1^P e v_3^P, v_3^P foi classificada como *totalmente fraca* (devido a v_2^P). Por C_6 (3)(a), caso $p_2 = p(v_1^P)$ falhe durante a execução de v_1^P, as *backups* v_1^B e v_2^B devem executar, e como v_2^B não consegue enviar mensagem a tempo para v_3^P (v_3^P é *fraca para* v_2^P), v_3^B também deve executar. Portanto v_3^B e v_1^B não podem ser escalonadas sobrepostas.

Critério C_7: A *backup* v_i^B ***pode ser escalonada sobreposta a outra primária*** v_k^P, já escalonada no processador $p(v_k^P)$ e tal que v_k^P não está sobreposta com nenhuma outra *backup* já escalonada, se:

1. existe a precedência $v_i^P \rightarrow v_k^P$, ou
2. não existe nenhuma relação de precedência entre v_i^P e v_k^P, e:
   (a) os processadores das primárias são diferentes $p(v_i^P) \neq p(v_k^P)$, e v_i^P é *totalmente forte*, e existe v_l^P escalonada depois de v_i^P ($EST(v_l^P) > EFT(v_i^P)$) no mesmo processador $p(v_i^P) = p(v_l^P)$, tal que v_l^P é uma tarefa *candidata a provocar falha*

induzida de v_k^P (v_k^P é *parcialmente forte* ou *fraca*).

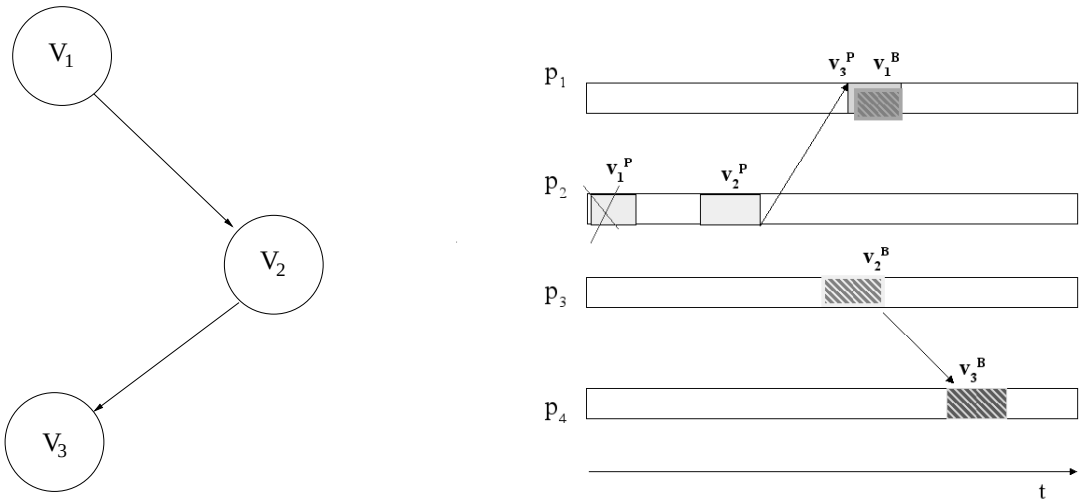


Figura 5.4: Critério C_7 aplicado no GAD

Com C_7 (1), v_i^B pode sobrepor a sucessora primária v_k^P, pois quando v_k^B for escalonada, v_k^P será classificada *fraca para* v_i^P, pois a mensagem de v_i^B não chegará a tempo para v_k^P. Desta forma, se v_i^P falhar durante sua execução, provocará a falha induzida de v_k^P e consequentemente, as *backups* v_k^B e v_i^B terão que executar. Portanto as execuções de v_k^P e v_i^B podem ficar sobrepostas.

Por C_7 (2), sem relação de precedência entre v_j^P e v_i^P, a *backup* v_i^B de uma tarefa totalmente forte pode ser escalonada sobre uma primária v_k^P parcialmente forte ou fraca, se v_l^P, candidata a provocar a falha induzida de v_k^P, está escalonado no mesmo processador $p(v_i^P)$ e depois de v_i^P. Se $p(v_i^P)$ falhar, v_i^B deve executar e a primária sobreposta v_k^P não poderá executar, pois como v_l^P está escalonado no mesmo processador $p(v_i^P) = p(v_l^P)$, v_l^P também falha e provoca a falha induzida de v_k^P.

A restrição C_7 (2) pode ser ilustrada com a Figura 5.4, neste caso v_1^B pode ser escalonada sobre a tarefa v_3^P, onde v_1^P e v_3^P não têm relação de precedência e seus processadores são distintos. Além disso, v_1^P foi classificada *totalmente forte* (não têm predecessores) e v_3^P tem o predecessor v_2^P tal que foi escalonado no mesmo processador que v_1^P (p_1), e ainda v_3^P é *fraca para* v_2^P. Logo, v_1^B e v_3^P podem ter suas execuções sobrepostas.

Note que em [61] a sobreposição sobre primárias não foi considerada. Já em [60] a sobreposição de *backups* sobre primárias foi abordada, mas somente para tarefas com relação de precedência. Diferentemente, neste trabalho o critério é proposto também para tarefas primárias que não têm relação de precedência em G, o que permite melhorar ainda mais o desempenho da execução em caso de falha.

Os critérios de C_1 a C_7 apresentados nesta seção, são as regras utilizadas por FTMRCD para escalonar as *backups*. O cumprimento destes critérios devem garantir a consistência e o devido funcionamento do mecanismo de tolerância a falhas durante a execução posterior no ambiente real. Assim, espera-se também uma diminuição no custo do

mecanismo e consequentemente, do *makespan* da aplicação.

5.3 Algoritmo de Escalonamento para Tolerar uma Falha

A descrição do algoritmo FTMRCD proposto para escalonar as *backups* com o objetivo de tolerar uma falha permanente de processador segue nesta seção. De forma geral, FTMRCD escalona as *backups* seguindo a mesma estrutura do algoritmo MRCD descrito no Capítulo 4, mas com a adição dos critérios acima definidos.

O Algoritmo 4 mostra os passos com as principais funções propostas para FTMRCD. Seja a lista V_{ordG} com todas as tarefas $v \in V^B$ ordenadas pela prioridade de seleção $blevel()$. Inicialmente, a partir de V_{ordG}, a primária v_i^P de cada *backup* v_i^B é classificada de acordo com o critério de classificação C_1. Para cada tarefa $v \in V_{ordG}$, a função $ClasPrimaria$ na linha 3 determina se v_i^P é *totalmente forte* ou não, conforme C_1. Esta função fornece também os predecessores ($LPredFalha$) e sucessores ($LSucFalha$) de cada primária que serão utilizados posteriormente para recuperar aplicação em caso de falha. Logo depois na linha 4 com a função $AchaCjtoProcs$ são escolhidos os processadores onde v^B pode ser escalonada (critérios C_2, C_4 e C_5).

O conjunto dos processadores escolhidos P^*, para escalonar v_i^B é percorrido na linha 5, e para cada p_j e calculado o tempo disponível mais cedo $EAT(v_i^B, p_j)$ em que v_i^B deve começar em p_j (linha 6). Para isto calcula-se o máximo entre o tempo disponível mais cedo das primárias predecessoras $EATP(Pred(v_i^P))$, o tempo disponível mais cedo das *backups* predecessoras $EATB(Pred(v_i^P))$ e o tempo de fim de v_i^P, $EFT(v_i^P, p_j)$ (C_3). Para cada p_j, na linha 7 a função $CriaLNaoSobreposta$ cria uma lista $LNSob$ das tarefas que não podem sobrepor com v^B, baseado nos critérios de sobreposição C_6 e C_7. Similar a MRCD, o algoritmo FTMRCD emprega uma heurística do tipo *list scheduling*, com uma política de inserção de tarefas para escalonar as *backups*, e com a mesma função de custo ponderada que em MRCD. Na linha 8 a função $InsTarefas$ calcula o melhor tempo de início $EST(v_i^B, p_j)$ de v_i^B em p_j. Desta forma, checando os intervalos desocupados em $LNSob$, aloca-se v^B em espaços ociosos entre tarefas primárias e *backups* já escalonadas, e no melhor processador que minimiza a função de custo f ($F = \min f()$ na linha 19). Os valores dos objetivos EFT e RC são calculados antes e agregados na linha 18 dentro de f para todos os processadores de P^*.

Para concluir, caso P^* seja diferente de vazio, atualiza-se na linha 22 o escalonamento de *backups* $SchBck$ com a nova *backup* v_i^B escalonada. Se P^* é vazio, então não foi possível achar processadores para escalonar v_i^B e o algoritmo termina na linha 24 considerando indisponibilidade de recursos.

Algoritmo 4 : *FTMRCD(V_{ord},P,w,Sch)*

1 for $i = 0, \ldots, n-1$
2 $\quad F = \infty$;
3 $\quad \langle TForte, Forte, LPredFalha, LSucFalha\rangle = ClasPrimaria(v_i^P)$; /*$C_1$*/
4 $\quad P^* = AchaCjtoProcs(v_i^B, P, LPredFalha)$; /*$C_2, C_4, C_5$*/
5 $\quad \forall p_j \in P^*$
6 $\quad\quad EAT(v_i^B, p_j) = \max\{EATP(Pred(v_i)), EATB(PredB(v_i)), EFT(v_i^P, p_j)\}$ /*C_3*/
7 $\quad\quad LNSob = CriaLNaoSobreposta(v_i^B, p_j)$; /*$C_6$, C_7 */
8 $\quad\quad EST(v_i^B, p_j) = InsTarefas(v_i^B, p_j, LNSob, EAT)$;
9 $\quad\quad EFT(v_i^B, p_j) = EST(v_i^B, p_j) + eh(v_i^B, p_j)$;
10 $\quad\quad RC(v_i^B, p_j) = FP(p_j) \times eh(v_i^B, p_j)$;
11 $\quad EFT_{min} = \min_{p_j \in P}\{EFT(v_i^B, p_j)\}$;
12 $\quad EFT_{max} = \max_{p_j \in P}\{EFT(v_i^B, p_j)\}$;
13 $\quad RC_{min} = \min_{p_j \in P}\{RC(v_i^B, p_j)\}$;
14 $\quad RC_{max} = \max_{p_j \in P}\{RC(v_i^B, p_j)\}$;
15 $\quad \forall p_j \in P^*$
16 $\quad\quad EFT_n(v_i^B, p_j) = norm(0, 100, EFT_{min}, EFT_{max}, EFT(v_i^B, p_j))$;
17 $\quad\quad RC_n(v_i^B, p_j) = norm(0, 100, EFT_{min}, RC_{max}, RC(v_i^B, p_j))$;
18 $\quad\quad f(v_i^B, p_j) = (1-w) \times EFT_n(v_i^B, p_j) + w \times RC_n(v_i^B, p_j)$;
19 $\quad\quad$ if $(f(v_i^B, p_j) < F)$
20 $\quad\quad\quad F = f(v_i^B, p_j)$; $p_{v_i^B} = p_j$;
21 $\quad$ if $(P^* \neq \varnothing)$
22 $\quad\quad SchBck = SchBck \cup \langle v_i^B, p_{v_i^B}, EST(v_i^B, p_{v_i^B})\rangle$;
23 $\quad$ else
24 $\quad\quad SchBck = \varnothing$; $LSucFalha = \varnothing$; $sair()$;
25 $\quad retorna(SchBck, LSucFalha)$;

5.3.1 Classificação de Primárias

O Algoritmo 5 mostra o pseudocódigo da função *ClassPrimaria* que implementa o critério C_1, onde $TForte(v^P)$ indica se a tarefa v^P é totalmente forte ou não, e $Forte(v^P, v_j)$ denota se a tarefa v^P é forte para a predecessora v_j^P ou não. Na linha 3, o conjunto de predecessores $Pred(v^P)$ de v^P é percorrido e caso seja vazio, a tarefa v^P é classificada como *totalmente forte*, sinalizado por $TForte(v^P) = 1$. Por R_1, se os processadores de $v_j^P \in Pred(v^P)$ e de v^P são iguais, ou seja, $p(v_j^P) = p(v^P)$ e v_j^P é totalmente forte, então v^P é *forte para* v_j^P com $Forte(v^P, v_j^P) = 1$ na linha 6. Da mesma forma, se v^P está escalonada depois de v_j^B e recebe a mensagem de v_j^B a tempo, como aparece na linha 5,

então v^P é *forte para* v_j^P, ou seja, $Forte(v^P, v_j^P) = 1$. Caso contrário, v^P é *fraca para* v_j^P ($Forte(v^P, v_j^P) = 0$) e v^P deixa de ser *totalmente forte* ($TForte(v_j^P = 0$ na linha 8). Para este caso, v^B é inserida a seguir na lista de sucessores *backups* de v_j^P $LSucFalha(v_j^P)$, e deve executar em caso de falha de v_j^P. Na linha 10, v_j^P é inserida na lista de primárias predecessoras $LPredFalha(v^P)$ de v^P, por ser *candidata a provocar a falha induzida* de v^P.

Algoritmo 5 : $ClasPrimaria(v^P)$

```
1  /*por critério C1*/
2  TForte(v^P) = 1; LSucFalha = ∅; LPredFalha = ∅
3  ∀v_j ∈ Pred(v^P)
4      if (p(v_j^P) = p(v^P) ∧ TForte(v_j^P)) ∨
5      (EST(v^P, p(v^P)) > (EFT(v_j^B, p(v_j^B)) + c(v_j^B, v^P) ×
       L(p(v_j^B), p(v^P))))
6          Forte(v^P, v_j^P) = 1;
7      else
8          Forte(v^P, v_j^P) = 0; TForte(v^P) = 0;
9          LSucFalha(v_j^P) = LSucFalha(v_j^P) ∪ v^B;
10         LPredFalha(v^P) = LPredFalha(v^P) ∪ v_j^P;
11     retorna(TForte, Forte, LPredFalha, LSucFalha);
```

A lista $LSucFalha(v_j^P)$ é uma lista formada pelas *backups* sucessoras de v_j^P, que devem executar caso v_j^P falhar, pois as primárias das tarefas desta lista são *candidatas a falha induzida* pelas predecessoras v_j^P. Da mesma forma, v_j^P é uma *candidata a provocar falha induzida* das primárias de $LSucFalha(v_j^P)$. Esta informação definida por FTMRCD é importante, pois será utilizada depois pelo mecanismo tolerante a falhas no ambiente real para permitir a continuidade da execução da aplicação em caso de falha, conforme será descrito no Capítulo 6. Em caso de falha, se v_j^P, não finaliza por falha do processador onde estava executando $p(v_j^P)$, então devem ser ativadas a sucessoras *backups* v^B contidas em $LSucFalha(v_j^P)$ para assim recuperar a aplicação. Por outro lado, a lista $LPredFalha(v^P)$ de cada v^P é formada pelas primárias predecessoras $v_j^P \in Pred(v^P)$, tal que a falha de v_j^P pode provocar também a falha de v^P (*candidata a falha induzida*), pois nem a mensagem de v_j^B consegue chegar a primária v^P, e portanto a *backup* v^B deve executar. As tarefas de $LPredFalha(v^P)$ são predecessoras *candidatas a provocar falha induzida* de v^P, e seus processadores não formarão parte dos processadores onde v^B pode ser escalonada.

5.3.2 Seleção de Processadores

A próxima função *AchaCjtoProcs* na linha 4 de FTMRCD no Algoritmo 4, procura o conjunto de processadores P^*, onde é possível escalonar cada v_i^B ($i = 1...n$), a partir dos critérios C_2, C_4 e C_5. No pseudocódigo no Algoritmo 6, por C_2 na primeira linha o processador de v^P não faz parte de P^*, pois v^B não pode ser escalonada nele. Logo

depois, o conjunto de predecessores de v^P é percorrido para verificar se satisfazem C_4. Se os processadores das primárias são iguais ou v^P é *fraca para* v_j^P na linha 3, então o processador de v_j^P não fará parte de P^* (linha 5). Apartir da função $AtualizaP^*$ na linha 6 (pseudocódigo no Algoritmo 7), é atualizado o conjunto P^*. Em $AtualizaP^*$ a lista $LPredFalha(v^P)$ é percorrida para retirar de P^* (linha 1) todos os processadores que alocam os predecessores de v_j^P, *candidatos a provocar falha induzida* de v_j^P. Note que quando a função atinge primárias v^P totalmente fortes $LPredFalha(v^P) = \varnothing$.

Algoritmo 6 : $AchaCjtoProcs(v^B, P, LPredFalha)$

```
1 P* = P − p(v^P); /*C2*/
2 ∀v_j^P ∈ Pred(v^P) /*C4*/
3     if ((p(v_j^P) = p(v^P)) ∨ (¬Forte(v^P,v_j^P)))
4         if (p(v_j^P) ∈ P*)
5             P* = P* − {p(v_j^P)}; /*não escalonar v^B em p(v_j^P)*/
6     P* = AtualizaP*(P*, v_j^P, LPredFalha) /*C5*/
7 retorna(P*)
```

Algoritmo 7 : $AtualizaP^*(P^*, v^P, LPredFalha)$

```
1 ∀v_j^P ∈ LPredFalha(v^P) /*C5*/
2 while (∃p(v_j^P) ∈ P*)
3     P* = P* − {p(v_j^P)}; /*não escalonar v^B em p(v_j^P)*/
4 retorna(P*)
```

Note que a função $AchaCjtoProcs(v^B, P, LPredFalha)$ pode não achar processadores disponíveis para escalonar certa *backup* v^B. Uma forma de garantir que sempre existam processadores disponíveis para escalonar as *backups*, é garantir que o sistema distribuído seja grande o suficiente dependendo da aplicação, ou que todas as primárias sejam *totalmente fortes* por C_1. Porém o primeiro caso pode não ser possível e o segundo caso é praticamente impossível pela abordagem de escalonamento utilizada. Nesta abordagem, para garantir o desempenho da execução quando não há falha, o algoritmo escalona todas as primárias antes de escalonar as *backups*, o que provoca mais primárias *totalmente fracas* ou *parcialmente fortes*. Dependendo da topologia e tamanho da aplicação e do ambiente, para cada primária suas *backups* predecessoras podem não ser escalonadas antes que elas. Por exemplo, aplicações com mais dependências e em particular com um número maior de cadeias com tarefas dependentes entre si são mais propensas a ter mais primárias *parcialmente fortes* ou *totalmente fracas*. Isto depende de onde foram escalonados seus predecessores *backups* e se conseguem enviar as mensagens no tempo previsto. Assim, por C_4 e C_5, enquanto as primárias sejam *fortes para* um número maior de predecessores, menos chances terão de ser *candidatas a falhas induzidas*, havendo mais recursos para ser escalonadas. Por isso, técnicas como sobreposição de tarefas ou inserção em espaços

ociosos são estratégias importantes, adotadas e reformuladas neste trabalho. O algoritmo proposto FTMRCD considera uma função de custo ponderada, aumentando a probabilidade de achar uma solução possível dentre múltiplos escalonamentos que existem para uma mesma aplicação e ambiente de execução. Assim, com a variação de w é possível buscar soluções de escalonamento com tolerância a falhas viáveis, e ainda próximas ao compromiso almejado entre os objetivos do problema. Por outro lado, para ter maior disponibilidade nos recursos, este trabalho explora os critérios de sobreposição de *backups* e é considerada uma política de inserção de *backups* em espaços ociosos entre primárias e *backups*.

5.3.3 Sobreposição de *Backups*

Para cada $p_j \in P^*$, a função $CriaListaNaoSobreposta$ (Algoritmo 8) na linha 7 de FTMRCD, constrói uma lista $LNSob$ de tarefas já escalonadas (primárias e *backups*), que v^B não pode sobrepor durante sua execução. Em particular no pseudocódigo do Algoritmo 8 de $CriaListaNaoSobreposta$ as tarefas v_i escalonadas no processador p são analisadas com C_6 e C_7. Com C_6, se v_i é uma *backup* (v_i^B na linha 3) com $p = p(v_i^B)$, e sua primária correspondente v_i^P está escalonada no mesmo processador que v^P ($p(v_i^P) = p(v^P)$ na linha 4), então v_i^B é inserida em $LNSob$ por C_6 (1). Se v_i^B é predecessora de v^P, tal que $v_i^P \in Pred(v^P)$ na linha 6, e v^P é *fraca para* v_i^B ($\neg Forte(v^P, v_i^P)$ na linha 6), então por C_6 (2) v_i^B forma parte de $LNSob$.

Para concluir C_6, se não existe nenhuma relação de precedência entre v_i^P e v^P ($v_i^P \notin Pred(v^P) \wedge v_i^P \notin Suc(v^P)$ na linha 9), e uma das tarefas primárias: v^P ou v_i^P, é *parcialmente forte* ou *totalmente fraca*, pelo critério $C_6(3)$ dependendo dos casos seguintes, v_i^B é adicionada em $LNSob$. Primeiramente por (a), se uma das primárias v^P ou v_i^P não é classificada *totalmente forte* ($\neg TForte(v^P)$ ou $\neg TForte(v_i^P)$ nas linhas 10 e 13), mas se v_k^P, *candidata a provocar falha induzida*, está escalonada no mesmo processador que a outra primária, então v_i^B é adicionada na lista não sobreposta $LNSob$ por (b). Se ambas as tarefas não são classificadas *totalmente fortes*, é adicionada v_i^B em $LNSob$ na linha 13, quando as duas tarefas apresentam uma mesma tarefa v_k^P candidata a lhes provocar falha induzida. Da mesma forma v_i^B forma $LNSob$ na linha 21 por (c), se as respectivas *candidatas a provocar falhas induzidas* de v_l^P e v_k^P, respectivamente ($\exists v_k^P \in LPredFalha(v^P)$) $\wedge$ ($\exists v_l^P \in LPredFalha(v_i^P)$), estão escalonadas no mesmo processador $p(v_l^P) = p(v_k^P)$.

Com C_7 se v_i é primária (v_i^P na linha 23), escalonada em p ($p = p(v_i^P)$) e v_i^P é predecessora de v^P ($v_i^P \in Pred(v^P)$ na linha 24) então forma $LNSob$ por C_7 (1). Por outro lado, se não existe relação de predecedência entre v_i^P e v^P ($v^P \notin Pred(v_i^P)$ e $v^P \notin Suc(v_i^P)$ na linha 27), no caso em que uma das seguintes condições é satisfeita: as primárias estão escalonadas no mesmo processador, $p(v_i^P) = p(v^P)$ na linha 28, ou v^P não é classificada *totalmente forte* ($\neg TForte(v^P)$ na linha 30), ou v_i^P não é classificada *totalmente forte* ($\neg TForte(v_i^P)$ mas não existe predecessor v_k^P ($\nexists v_k^P$ na linha 32), tal que v_i^P *é fraca para* v_k^P ($\neg Forte(v_i^P, v_k^P)$), e está escalonado no mesmo processador ($p(v_k^P) = p(v^P)$) depois de v^P ($EST(v_k^P) > EFT(v^P)$), então por C_7 (2), v_i^P é adicionada em $LNSob$.

Algoritmo 8 : $CriaListNaoSobreposta(v^B, p)$

1 $LNSob = \varnothing$;
2 $\forall v_i$ / $p(v_i) = p$
3 if v_i^B /*C_6 (se *backup*)*/
4 if $(p(v_i^P) = p(v^P))$ $LNSob = LNSob \cup \{v_i^B\}$ /*$C_6(1)$*/
5 else
6 if $(v_i^P \in Pred(v^P)) \wedge (\neg Forte(v_i^P, v^P))$ /*C_6 (2)*/
7 $LNSob = LNSob \cup \{v_i^B\}$;
8 else
9 if $(v_i^P \notin Pred(v^P) \wedge v_i^P \notin Suc(v^P))$ /*$C_6(3)$*/
10 if $(\neg TForte(v_i^P)) \wedge (\exists v_k^P \in LPredFalha(v_i^P)) \wedge (p(v_k^P) = p(v^P)))$
11 $LNSob = LNSob \cup \{v_i^B\}$;
12 else
13 if $(\neg TForte(v^P)) \wedge (\exists v_k^P \in LPredFalha(v^P)) \wedge (p(v_k^P) = p(v_i^P)))$
14 $LNSob = LNSob \cup \{v_i^B\}$;
15 else
16 if $(\neg TForte(v^P) \wedge \neg TForte(v_i^P)) \wedge$
17 $(\exists v_k^P \in \{LPredFalha(v^P) \wedge LPredFalha(v_i^P)\})$
18 $LNSob = LNSob \cup \{v_i^B\}$;
19 else
20 if $(\neg TForte(v^P) \wedge \neg TForte(v_i^P)) \wedge$
21 $(\exists v_k^P \in LPredFalha(v^P)) \wedge (\exists v_l^P \in LPredFalha(v_i^P)) \wedge (p(v_l^P) = p(v_k^P))$
22 $LNSob = LNSob \cup \{v_i^B\}$;
23 if v_i^P /*C_7 (se primária)*/
24 if $(v_i^P \in Pred(v^P))$ /*$C_7(1)$*/
25 $LNSob = LNSob \cup \{v_i^P\}$;
26 else
27 if $(v_i^P \notin Pred(v^P) \wedge v_i^P \notin Suc(v^P)$ /*$C_7(2)$*/
28 if $(p(v_i^P) = p(v^P))$ $LNSob = LNSob \cup \{v_i^P\}$;
29 else
30 if $(\neg TForte(v^P))$ $LNSob = LNSob \cup \{v_i^P\}$;
31 else
32 if $(\nexists v_l^P \in LPredFalha(v_i^P)) \wedge$
33 $(p(v_l^P) = p(v^P)) \wedge (EST(v_l^P) > EFT(v^P))$)
34 $LNSob = LNSob \cup \{v_i^P\}$;
35 $retorna(LNSob)$;

O escalonamento final gerado $Sch \cup SchBck$, descreve como será o comportamento da aplicação em caso de falha, tendo diferentes combinações ou possíveis execuções das tarefas da aplicação em dependência do processador que falha. Como uma única falha permanente de processador é tolerada, somente uma destas combinações será ativada se ocorre falha. Neste caso, serão executadas as *backups* das primárias que não terminaram de executar (*falha de hardware*) no processador com falha, mais as *backups* das primárias que não irão executar em outros processadores, por apresentar *falha induzida* provocada por seus predecessores. Portanto, para cada falha de processador, obtém-se um escalonamento de primárias e *backups* e como resultado um *makespan* diferente.

5.3.4 Cálculo do Escalonamento e da Solução com Falha

A estratégia de escalonamento bi-objetivo e tolerante a falhas proposta produz um escalonamento Sch para as cópias primárias de um GAD de entrada através de MRCD, e um escalonamento para as *backups* $SchBck$, de acordo com os critérios ja especificados em FTMRCD. No entanto, para analisar os benefícios de tal estratégia, é necessário calcular o *makespan* e a confiabilidade do escalonamento resultante em caso de uma falha permanente de um determinado processador. Assim, nesta seção é proposto um simulador da execução do GAD de acordo com os escalonamentos produzidos por MRCD e FTMRCD ($Sch \cup SchBck$), dado que a falha ocorre no processador $p(v^P)$ durante a execução de uma tarefa v^P. O objetivo é estimar os valores esperados de *makespan* $\mathcal{M}_{falha}$ e a confiabilidade $R_{T_{falha}}$, alcançados pela aplicação quando ocorre uma falha permanente de processador e gerar o escalonamento Sch_{falha} das primárias e *backups* que realmente executam em caso de falha.

O simulador desenvolvido considera uma falha permanente do processador $p(v^P)$, que tem alocada v^P, para um dado escalonamento $Sch \cup SchBck$ durante a execução de v^P. Inicialmente, v^P e as tarefas primárias escalonadas depois de v^P em $p(v^P)$, apresentam falha por *hardware*. Consequentemente, as primárias que fazem parte da lista de sucessores destas tarefas $LSucFalha$ são atualizadas no simulador para apresentar *falha induzida* de v^P e as mesmas não podem executar durante a execução da aplicação com falha, ou seja, são substituídas por suas *backups*.

Para simular a execução paralela da aplicação com falha, o conjunto P^* contendo o resto dos processadores, $P^* = P - p(v^P)$ é percorrido, sendo atualizados o *makespan* com o máximo dos tempos de fim e a confiabilidade total com a soma dos custos de confiabilidade, sempre que cada tarefa v_i finaliza sua execução. Dependendo se a tarefa v_i que executa é primária ou *backup*, o simulador utiliza a informação do escalonamento Sch ou $SchBck$, para assim determinar o processador $p(v_i)$ onde v_i deve executar, e seu tempo de computação. Antes de começar cada v_i, verifica-se se todos seus predecessores já terminaram e, caso verdadeiro, re-calcula-se o tempo de início $EST(v_i, p(v_i))$ e o tempo de fim $EFT(v_i, p(v_i))$ em $p(v_i)$. Observe que, durante a simulação, os tempos de início das *backups* v_i^B que executam devido a uma determinada falha podem atingir valores menores do que os previstos no escalonamento inicial $SchBck$ obtido por FTMRCD, pois outras

backups escalonadas antes, no mesmo processador, $p(v_i^B)$ não executam. Finalmente, são coletados o *makespan* $\mathcal{M}_{falha}$ e a confiabilidade $R_{T_{falha}}$ da execução da aplicação com falha. O simulador gera também como saída o escalonamento Sch_{falha} com as tarefas (primárias e *backups*) que realmente devem executar para determinada falha permanente de processador. Sch_{falha} é utilizado para avaliar a consistência da execução da aplicação em caso de falha.

5.4 Extensão para Tolerar Múltiplas Falhas de Processador

O algoritmo proposto FTMRCD na seção anterior é estendido para considerar múltiplas falhas de processador, e denotado mFTMRCD como mostra o *framework* do Algoritmo 9. P é um parâmetro de entrada que representa o conjunto de todos os processadores do sistema heterogêneo e distribuído. No entanto, P pode ser visto como um conjunto de k subconjuntos de processadores tal que $P = \{P_0, ..., P_{k-1}\}$, onde cada $P_i \in P$, $i = 0, ..., k-1$, pode representar um *cluster* de processadores, ou um subconjunto de um *cluster* que tem uma certa característica, ou ainda P pode ser dividida em k subconjuntos aleatoriamente. Note que, no algoritmo proposto para tolerar múltiplas falhas mFTMRCD, $P_i \cap P_j = \varnothing$ $\forall P_i, P_j \in P$.

Em mFTMRCD, os critérios e as funções para escalonar cada *backup* v_i^B apresentados anteriormente (C_1 a C_7), são aplicados somente dentro de um grupo de processadores do sistema, $P_s \subset P$ $(0 \leq s < k)$, onde P_s contém o processador que foi alocada v_i^P pelo algoritmo MRCD, que escalona as primárias. Desta forma, em mFTMRCD é possível assumir múltiplas falhas, sendo possível só uma falha de processador por grupo. O máximo de falhas que podem ser toleradas com este algoritmo é então k falhas.

Algoritmo 9 : *mFTframework(G,P,w_1,w_2)*

1 $V_{ord} = \langle v_0, v_1, \ldots, v_{n-1} \rangle / blevel(v_i) \leq blevel(v_{i+1}), i = 0, \ldots, n-2$;

2 $\langle$ Sch, $S(Sch)$ $\rangle$=MRCD(V_{ord},P,w_1);
3 $P = \{P_0, P_1, ..., P_{k-1}\}$;
4 $\langle$ $SchBck$,$LSucFalha$ $\rangle$=mFTMRCD(V_{ord},P, Sch, w_2);

Desta forma, o procedimento FTMRCD do Algoritmo 4, é estendido para mFTMRCD no *framework* do Algoritmo 9, entretanto, a estrutura geral proposta é semelhante. A nova distribuição do ambiente P proposta na linha 3 é passada como parâmetro de entrada na linha 4, com os distintos grupos P_s de processadores obtidos.

Diferentemente, em mFTMRCD para escalonar cada *backup* v_i^B, seleciona-se de P o grupo P_s que contém o processador da primária v_i^P correspondente, tal que $p(v_i^P) \in P_s$ com $s = 0...k-1$. Desta forma, a linha 4 de FTMRCD é substituída no Algoritmo 4 pela função $P^* = AchaCjtoProcs(v_i^B, P_s, LPredFalha)$, ou seja, somente é realizada a busca dos processadores dentro do grupo P_s para escalonar cada *backup* v_i^B. As próximas linhas de mFTMRCD são exatamente as mesmas que as de FTMRCD, quando propostas para uma falha no Algoritmo 4.

Considerando uma falha de processador por grupo ou *cluster*, ao escalonar a *backup* v_i^B no grupo de sua correspondente primária v_i^P, se garante que se ocorre uma falha de processador no grupo, quando v_i^P ainda não executou, outra falha de processador no mesmo grupo não pode ocorrer. Portanto a *backup* correspondente v_i^B poderá ser ativada para executar no processador onde foi escalonada.

Backup Alternativa

Como os critérios de sobreposição foram propostos presumindo uma única falha de processador, os mesmos podem ser aplicados dentro de cada grupo, onde só um processador pode falhar. Assim, cada *backup* v_i^B, ao ser escalonada dentro de um grupo, pode sobrepor *backups* já escalonadas ali a partir dos critérios C_6 e C_7. Entretanto, o escalonamento de múltiplas falhas resultante da proposta anterior não necessariamente poderá sempre recuperar a aplicação. Dependendo dos processadores com falha nos distintos grupos P_i, tarefas primárias podem sofrer falhas induzidas, por primárias escalonadas em outros grupos. A falha de processador de um grupo P_i pode provocar falhas induzidas de primárias escalonadas em outros grupos P_j, $j \neq i$. Se coincide que as *backups* destas primárias, estão escalonadas em processadores que também sofrem falhas em seus grupos P_j, estas *backups* não poderão executar e a aplicação não finaliza.

Desta forma, uma primeira versão de escalonamento para tolerar múltiplas falhas pode ser considerada, mas não garante que a aplicação seja sempre recuperável, ou seja, vai depender de se as falhas de processador que ocorrem geram o problema anterior. Se existem casos com *backups* que devem executar (por falhas induzidas de suas primárias), escalonadas em processadores com falha, o mecanismo de tolerância a falhas não executa, e avisa que a recuperação dessas falhas não é possível. Em casos distintos funciona normalmente.

Para resolver o problema anterior, uma segunda versão para tolerar múltiplas falhas é proposta. Neste caso o algoritmo segue a mesma proposta, mas difere pela adição de outra *backup* alternativa $v_i^{B_a}$ para cada primaria v_i^P que não foi classificada *totalmente forte*. Com mFTMRCD, $v_i^{B_a}$ é escalonada depois de v_i^B (repetindo as linhas 6 a 20 do Algoritmo 4), em um processador $p(v_i^{B_a})$ do mesmo grupo de processadores do escalonamento de v_i^P e v_i^B. O processador $p(v_i^{B_a})$ é obviamente diferente ao processador da sua *backup* original $p(v_i^B)$, e consideram-se os mesmos critérios de escalonamento que foram aplicados com v_i^B dentro desse grupo.

No cálculo de $EAT(v_i^B, p_j)$ na linha 6 do Algoritmo 4, $EATB(v_i^B, p_j)$ agora considera também os tempos das *backup*s alternativas $v_k^{B_a}$ dos predecessores v_k^P que não são classificadas *totalmente fortes*, ou seja, $EAT(v_i^B, p_j) = \max\{EATP(Pred(v_i)), EATB(PredB(v_i)), EATBa(PredB_a(v_i)), EFT(v_i^P, p_j)\}$, onde $EATBa(PredB_a(v_i))$ é o tempo disponível mais cedo que v_i^B pode ser escalonada em relação a suas *backups* predecessoras alternativas $v_k^{B_a}$ (se tiver). Note que o tempo disponível mais cedo $EAT(v_i^{B_a}, p_j)$ para $v_i^{B_a}$ repete a mesma formulação.

Desta forma, quando ocorre falha de processador em um grupo, se for necessário

executar a *backup* original v_i^B de alguma primária v_i^P que sofre *falha induzida*, se v_i^B está escalonada em um processador que falha, o mecanismo de recuperação ativa $v_i^{B_a}$, que executará no processador $p(v_i^{B_a})$ onde foi escalonada. Como somente ocorre uma falha de processador por grupo, se garante que $p(v_i^{B_a})$ não falha. Note que diferente das primárias, as *backups* não podem sofrer falha induzida, elas são escalonadas depois de receber todas as mensagens predecessoras (primárias e *backups*). Se uma das *backups* (v_i^B e $v_i^{B_a}$) não pode executar por falha de seu processador, a outra poderá executar, permitindo a continuidade da execução.

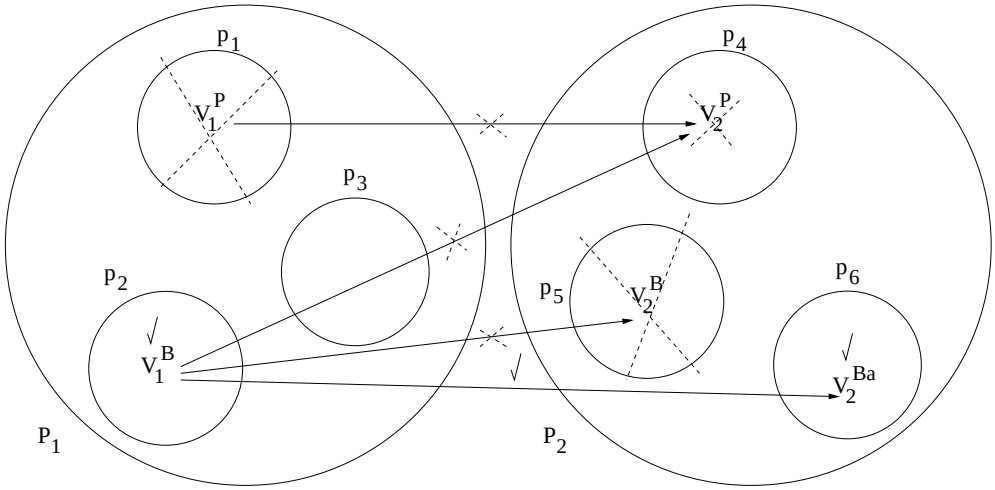


Figura 5.5: Exemplo com 2 falhas de processador

A Figura 5.5 mostra um exemplo para esta abordagem. O sistema representado têm 6 processadores distribuídos em dois grupos P_1 e P_2, tal que $P_1 \cup P_2 = P$, cada grupo com três processadores como visto na figura. Para simplificar, são analisadas só duas tarefas da aplicação v_1^P e v_2^P, com a relação de precedência $v_1^P \rightarrow v_2^P$, onde v_2^P é *fraca para* v_1^P. Por MRCD, v_1^P está alocada em $p_1 \in P_1$, e v_2^P em $p_4 \in P_2$. De acordo com mFTMRCD as *backups* originais v_1^B e v_2^B são escalonadas em outro processador do mesmo grupo que suas primárias, $p_2 = p(v_1^B) \in P_1$ e $p_5 = p(v_2^B) \in P_2$, respectivamente. Se o processador de uma primária v_i^P falhar, então o processador da correspondente *backup* v_i^B não irá falhar. No entanto, se uma primária v_i^P sofre *falha induzida*, mesmo que seu processador não falhe, sua *backup* v_i^B seria obrigada a executar. Neste exemplo, como os processadores p_1 e p_5 falham, v_1^P não executa e portanto não envia mensagem para v_2^P. Assim v_1^B executa, mas não consegue enviar mensagem a v_2^P, provocando sua *falha induzida*. v_2^B deveria executar no lugar de v_2^P, porém como $p_5 = p(v_2^B)$ falha em P_2, v_2^B também não poderá executar. Por mFTMRCD, a *backup* alternativa é escalonada se sua primária não é classificada *totalmente forte*, e assim neste exemplo, $v_2^{B_a}$ escalonada em $p_6 \in P_2$ (v_2^P é *fraca para* v_1^P) será executada. $v_2^{B_a}$ recebe a mensagem de v_1^B e executa com sucesso. Observe que isto aconteceu por coincidir que a *backup* original v_2^B foi escalonada no processador que falha em P_2, tendo que ser ativada sua alternativa $v_2^{B_a}$ para garantir a recuperação da aplicação.

5.4.1 Considerações sobre as Falhas e o Número de Subconjuntos

Note que, um custo maior em relação a solução estará associado ao mecanismo de tolerância a falha durante execução da aplicação na medida que cria-se um número maior k de grupos com mFTMRCD. Em mFTMRCD com a restrição de escalonar cada *backup* dentro de um único grupo de processadores, diminui a chance de achar processadores com melhores características tanto em relação a tempo de execução quanto em confiabilidade para escalonar as *backups*. Contudo, aumenta o número máximo de falhas que pode ser tolerado pelo algoritmo de escalonamento proposto.

Portanto, seria interessante controlar este custo relacionado ao tempo de execução e a confiabilidade, controlando de maneira flexível a relação entre a quantidade de processadores e o número de falhas toleradas, através de uma função de compromisso no algoritmo proposto. Para regular o custo de incluir tolerância a falhas, pode ser atribuído um valor diferente a k dependendo da situação. Por exemplo, caso deseja-se obter um custo menor na solução, embora a aplicação aumente seu risco de apresentar falhas, é possível especificar um valor baixo de k.

Um outro aspecto que deve ser considerado é o compromisso entre o número de falhas toleradas k e o peso associado a confiabilidade da aplicação durante o escalonamento bi-objetivo. Se de certa forma, a confiabilidade do escalonamento apresenta maior prioridade no algoritmo (w na função ponderada na linha 18 do Algoritmo 4), diminuindo assim a probabilidade da aplicação falhar durante a execução. Neste caso, com o propósito de obter um custo menor em relação a ambos objetivos do escalonamento, pode ser interessante considerar um número k menor de falhas a ser toleradas pelo algoritmo mFTMRCD. Assim, uma proposta no algoritmo pode ser estabelecer uma relação direta entre estes parâmetros w e k, tal que o aumento do w gere uma diminuição de k e vice-versa.

O compromisso entre o número de falhas toleradas pela proposta de tolerar múltiplas falhas proposto e o valor da ponderação w, segue o seguinte raciocínio. Quanto maior o valor de w utilizado no algoritmo MRCD, mais confiáveis são os processadores escolhidos e então menor a probabilidade de ocorrência de falhas nestes processadores. Assim, um número menor de falhas poderia ser especificado e como resultado, um menor número de subconjuntos k de processadores pode ser definido.

Portanto, considerar a abordagem bi-objetivo ponderada proposta no capítulo anterior, e a agrupação dos processadores como parte do algoritmo proposto, podem auxiliar na redução do custo de utilizar o mecanismo tolerante a falhas para executar a aplicação e como consequência, a sobrecarga do tempo final de execução. A proposta de uma função de compromisso pode controlar este custo, ao permitir variar o número de falhas que podem ser toleradas em função da importância atribuída a confiabilidade no escalonamento ou ao número de sobconjuntos.

5.5 Análise de Desempenho

Com o objetivo de avaliar o desempenho da estratégia de escalonamento proposta, as soluções geradas por FTMRCD e mFTMRCD são analisadas e comparadas com MRCD em diferentes testes. Como em estudos anteriores, foram consideradas aplicações paralelas sintéticas representadas pelas três classes de GADs: a Eliminação Gausseana, denotada por G_n; o diamante Di_n, que representa a multiplicação de matrizes; e GADs aleatórios R_n, com topologias irregulares. Em todos os casos, n denota o número de tarefas da aplicação. Os pessos associados a cada GAD são os mesmos conforme descritos na Seção 4.4.

Inicialmente, foi utilizado o cenário de pior caso, descrito no Capítulo 4, com $m = 24$ processadores agrupados inicialmente em três grupos P_0, P_1 e P_2, cada um com $m = 8$ processadores. Relembrando das características nos diferentes grupos de processadores, as taxas de falha FP foram geradas uniformemente nos intervalos $[10^{-5}, 3.3 \times 10^{-5}]$ $\forall p_i \in P_0$, $[3.4 \times 10^{-5}, 6.6 \times 10^{-5}]$ $\forall p_i \in P_1$ e $[6.7 \times 10^{-5}, 10^{-4}]$ $\forall p_i \in P_2$, respectivamente. Em relação ao índice de retardo, $csi(p_i) = 73$ $\forall p_i \in P_0$, $csi(p_i) = 53$, $\forall p_i \in P_1$ e $csi(p_i) = 33$ $\forall p_i \in P_2$. FP e csi por grupo apresentam comportamentos de um certo modo conflitantes, ou seja, o primeiro grupo P_0 apresenta os processadores mais lentos e mais confiáveis e o último grupo P_2, os processadores mais rápidos e menos confiáveis. Esta configuração é escolhida para estudar casos onde é difícil chegar a um acordo entre os objetivos do escalonamento.

Outros dois cenários foram considerados, cenário de processadores homogêneos (CPH) e cenário de processadores aleatórios (CPA). CPH é um cenário de processadores homogêneos em relação ao indice de retardo, $csi(p_i) = 23$ $\forall p_i \in P$ com $m = 25$, parecido ao descrito na Seção 4.4.3, mas simula o ambiente computacional real de nossa intituição. As taxas de falhas dos processadores $FP(p_i)$ foram calculadas uniformente no intervalo $[1.8 \times 10^{-6}, 3.2 \times 10^{-6}]$. Já CPA é aleatório para csi e FP, com grupos de processadores que variam com o número de processadores m, dependendo da distribuição realizada $P = \{P_0, P_1, ..., P_{k-1}\}$, com k, quantidade de grupos ou falhas, variando da forma $k = (2, 4, 6, 8, 10)$. As velocidades dos processadores foram atribuídas aletóriamente entre os valores: $csi(p_i) = (33, 53, 73)$, em qualquer grupo. Neste cenário CPA, similar a CPC, os processadores mais rápidos são associados às taxas FP mais altas (menos confiáveis) e vice-versa, para gerar situações conflitantes. $FP(p_i)$ foi obtido no intervalo $[10^{-5}, 10^{-6}]$.

5.5.1 Uma Falha de Processador

O primeiro grupo de experimentos analisa a solução do escalonamento obtido pelo algoritmo FTMRCD na presença de uma falha permanente de processador, usando o simulador que calcula o *makespan* e a confiabilidade em caso de falha, proposto na Seção 5.3.4. Inicialmente, foram analisadas duas soluções para as diferentes aplicações G_n, R_n e Di_n: a solução de desempenho $S_{\mathcal{M}}$ a a solução de equilíbrio S_e, para uma falha de processador no cenário CPC. As soluções de escalonamento com falha, com FTMRCD, foram comparadas com as obtidas na execução sem falha, ou seja, as geradas por MRCD para as tarefas primárias. A coluna (%) nas distintas tabelas apresentadas, mostra o percentual

Tabela 5.1: Variação das soluções para $w_2 = 0$ em FTMRCD, pela recuperação de 1 falha de processador no inicio da execução no CPC, com $m = 24$ processadores para MRCD e FTMRCD.

	MRCD, $w_1 \neq 0$		FTMRCD, $w_2 = 0$			
GAD_n	$S_{\mathcal{M}}$	S_e	$S_{\mathcal{M}}$	% $S_{\mathcal{M}}$	S_e	% S_e
G_{152}	190.0,	449.4,	232.5,	22.3,	**434.0**,	-15.4
	0.93316	0.94287	0.91511	-1.93	0.93793	-0.52
G_{252}	318.7,	856.3,	459.8,	44.2	**834.3**,	-2.56
	0.85865	0.87844	0.82309	-3.50	0.87138	-0.80
G_{377}	480.4,	1494.7,	762.5,	58.7	**1461.2**,	-2.24
	0.75132	0.78560	0.69323	-7.73	0.77635	-1.17
G_{527}	675.1,	2272.9,	1233.6	82.7	**2234.2**	-1.70
	0.61243	0.66578	0.54766	-10.5	0.65334	-1.86
G_{702}	902.8,	3463.3,	-	-	**3431.6**	-0.91
	0.45439	0.53184	-	-	0.51773	-2.65
R_{80}	330.0,	949.0,	384.0	16.3	949.0,	0.0
	0.84159	0.86478	0.83543	-0.73	0.86400	-0.09
R_{98}	330.0,	1314.0,	330.0	0.0	**1253.0**,	-4.64
	0.87650	0.84059	0.80636	-8.00	0.82576	-1.76
R_{152}	396.0,	1533.0,	-	-	**1533.0**,	0.0
	0.69907	0.75375	-	-	0.73445	-2.56
R_{256}	528.0,	1314.0,	689.0	30.4	**1314.0**,	0.0
	0.54215	0.58315	0.53755	-0.84	0.57927	-0.66
R_{364}	759.0,	4161.0,	-	-	**3942.0**,	-5.26
	0.41804	0.51698	-	-	0.48993	-5.23
Di_{81}	580.9,	1664.9,	933.9	60.7	**1657.9**,	-0.40
	0.83218	0.87524	0.79318	-4.68	0.86995	-0.60
Di_{100}	660.0,	1957.0,	1040.0,	57.5	**1950.0**,	-0.35
	0.79665	0.84719	0.74722	-6.20	0.84150	-0.67
Di_{144}	825.0,	2468.0,	1358.0,	64.6	**2461.0**,	-0.28
	0.71903	0.78322	0.65755	-8.55	0.77614	-0.90
Di_{256}	1155.9,	3636.0,	2195.0,	89.9	**3629.0**,	-0.19
	0.54940	0.63608	0.51366	-6.5-	0.62575	-1.65
Di_{361}	1452.0,	4512.0,	3112.0	114.4	**4505.0**,	-0.15
	0.42095	0.52008	0.45527	-8.15	0.50812	-2.29

de variação de cada solução em relação aos dois objetivos *makespan* e confiabilidade.

Foram realizadas duas comparações das soluções obtidas por FTMRCD com as geradas pelo algoritmo MRCD sem falhas. Na primeira comparação (Tabelas 5.1 e 5.2), o escalonamento de MRCD aloca as tarefas nos mesmos processadores $m = 24$. Já na segunda comparação (Tabelas 5.3 e 5.4), o escalonamento MRCD aloca as tarefas para $m = 23$ processadores, desconsiderando o processador que teve a falha. A primeira comparação mostra a variação da solução de escalonamento, mudança inevitável pela tolerância da falha. Já a segunda avalia de certa forma a qualidade do escalonamento de FTMRCD, obtido com o uso de *backups*, ao se comparar com o escalonamento de primárias de MRCD sem o processador que falha.

Como especificado, tanto MRCD como FTMRCD recebem w, peso associado aos objetivos, como parâmetro $(1-w)$ associado a minimização do *makespan* $\mathcal{M}$, e w, à maximização da confiabilidade R_T. No entanto, diferentes valores de w podem ser estabelecidos para MRCD e FTMRCD. Assim, seja w_1 o peso dado como entrada a MRCD (primárias)

Tabela 5.2: Variação da solução de escalonamento com $w_2 = w_1$ em FTMRCD, pela recuperação de 1 falha de processador no inicio da execução no CPC.

	MRCD, $w_1 \neq 0$		FTMRCD, $w_1 = w_2$			
GAD_n	$S_{\mathcal{M}}$	S_e	$S_{\mathcal{M}}$	% $S_{\mathcal{M}}$	S_e	% S_e
G_{152}	190.0,	449.4	256.7	35.1	**406.0**,	-43.4
	0.93316	0.94287	0.92389	-0.99	0.93480	-0.85
G_{252}	318.7,	856.3,	479.9,	50.5	**721.6**,	-15.7
	0.85865	0.87844	0.83935	-2.24	0.86284	-1.77
G_{377}	480.4,	1494.7,	893.6,	86.0	**1182.4**,	-20.0
	0.75132	0.78560	0.72681	-3.26	0.75434	-3.97
G_{527}	675.1,	2272.9,	1362.5	101.8	**1687.0**	-25.7
	0.61243	0.66578	0.58774	-4.03	0.62366	-6.32
G_{702}	902.8,	3463.3,	-	-	**2674.3**	-22.7
	0.45439	0.53184	-	-	0.47626	-10.4
R_{80}	330.0,	949.0,	457.0	38.4	1008.0,	6.21
	0.84159	0.86478	0.83978	-0.21	0.86115	-0.41
R_{98}	330.0,	1314.0,	597.0	80.9	**1201.0**,	-8.59
	0.87650	0.84059	0.80300	-8.38	0.82510	-1.84
R_{152}	396.0,	1533.0,	-	-	1870.0,	21.9
	0.69907	0.75375	-	-	0.73478	-2.51
R_{256}	528.0,	1314.0,	803.0	52.0	**1493.0**,	13.6
	0.54215	0.58315	0.54353	-0.25	0.58202	-0.19
R_{364}	759.0,	4161.0,	-	-	4234.0,	1.75
	0.41804	0.51698	-	-	0.49564	-4.12
Di_{81}	580.9,	1664.9,	1286.9	121.5,	**1624.9**,	-2.40,
	0.83218	0.87524	0.85937	-3.26	0.84789	-3.12
Di_{100}	660.0,	1957.0,	1426.0,	116.0,	**1917.0**,	-2.04,
	0.79665	0.84719	0.82543	-3.61	0.81459	-3.84
Di_{144}	825.0,	2468.0,	1692,	105.0,	**2327.0**,	-5.71,
	0.71903	0.78322	0.75414	-4.87	0.72960	-6.84
Di_{256}	1155.9,	3636.0,	2302.0,	99.1,	**3465.0**,	-4.78,
	0.54940	0.63608	0.55771	-1.51	0.58418	-8.15
Di_{361}	1452.0,	4512.0,	3172.0	118,4,	**3319.0**,	-26.4
	0.42095	0.52008	0.46382	-5.96	0.44605	-14.2

e w_2, à FTMRCD (*backups*). Para cada par de tabelas, as seguintes situações são consideradas. Nas Tabelas 5.1 e 5.3, $w_1 \neq 0$ ($S_{\mathcal{M}}$ e S_e) e $w_2 = 0$, ou seja, as *backups* são escalonadas só considerando a minimização do *makespan*. Já nas Tabelas 5.2 e 5.4, $w_1 \neq 0$ ($S_{\mathcal{M}}$ e S_e) e $w_2 = w_1$, primárias e *backups* são escalonadas com o mesmo peso não nulo considerando em ambos os escalonamentos, o mesmo compromisso entre os objetivos.

Em todas as tabelas, o símbolo − representa o fato de que o respectivo algoritmo não consegue achar processadores disponíveis para escalonar determinadas *backups*. Isto acontece pela limitação do número de processadores para escalonar as *backups*, na medida que aumenta o tamanho da aplicação. Observe que estes casos aparecem em alguns experimentos de uma falha em soluções $S_{\mathcal{M}}$ quando o ambiente é constante, $m = 24$, e dependendo da topologia do GAD. Especificamente, ocorre para a maior aplicação G_{702}, e em duas aplicaçoes dos GADs gerados com topologias aleatórias R_{152} e R_{364}. Note que para D_i mesmo com o aumento da aplicação, o escalonamento sempre encontra solução, o grafo só tem no máximo uma largura de 19.

Nas tabelas 5.1, 5.2, 5.3 e 5.4, especificamente para R_n, de topologias irregulares

Tabela 5.3: Comparação da solução de FTMRCD com $w_2 = 0$ e 1 falha de processador, com MRCD sem incluir o processador que falha ($m = 23$) no CPC.

	MRCD, $w_1 \neq 0$		FTMRCD, $w_2 = 0$			
GAD_n	$S_{\mathcal{M}}$	S_e	$S_{\mathcal{M}}$	% $S_{\mathcal{M}}$	S_e	% S_e
G_{152}	190.0, 0.93247	502.7 0.94470	232.5 0.91511	22.3, -1.86	**434.0**, 0.93793	-13.6, -0.72
G_{252}	318.7, 0.85793	907.2, 0.87990	459.8, 0.82309	44.2, -4.06	**834.3**, 0.87138	-8.03, -0.97
G_{377}	480.4, 0.74863	1643.7, 0.78952	762.5, 0.69323	58.7, -7.40	**1461.2**, 0.77635	-11.1, -1.67
G_{527}	675.1, 0.60869	2468.6, 0.66951	1233.6 0.54766	82.7, -10.0	**2234.2** 0.65334	-9.49, -2.42
G_{702}	942.8, 0.45729	3810.7, 0.53888	- -	- -	**3431.6** 0.51773	-9.94, -3.92
R_{80}	358.0, 0.84070	1022.0, 0.86653	384.0 0.83543	7.26, -0.63	**949.0**, 0.86400	-7.14 -0.29
R_{98}	398.0, 0.80921	1241.0, 0.83904	**330.0** 0.80636	-17.0, -0.35	1253.0, 0.82576	0.96, -1.58
R_{152}	410.0, 0.69602	1752.0, 0.75749	- -	- -	**1533.0**, 0.73445	-12.5, -3.04
R_{256}	584.0, 0.53889	1387.0, 0.58463	689.0 0.53755	17.9, -0.25	**1314.0**, 0.57927	-5.26, -0.92
R_{364}	808.0, 0.41692	4380.0, 0.52088	- -	- -	**3942.0**, 0.48993	-10.0, -5.94
Di_{81}	833.9, 0.84373	1704.9, 0.87730	933.9 0.79318	11.9, -5.99	**1657.9**, 0.86995	-2.75, -0.84
Di_{100}	947.0, 0.810910	1997.0, 0.84900	1040.0, 0.74722	9.82, -7.85	**1950.0**, 0.84150	-2.35, -0.88
Di_{144}	834.0, 0.71896	2541.0, 0.78509	1358.0, 0.65755	62.8, -8.54	**2461.0**, 0.77614	-3.14, -1.14
Di_{256}	1268.0, 0.55939	3749.0, 0.63910	2195.0, 0.51366	73.1, -8.17	**3629.0**, 0.62575	-3.20, -2.09
Di_{361}	1518.0, 0.41511	4844.0, 0.52690	3112.0 **0.45527**	105.0, 9.67	**4505.0**, 0.50812	-6.99, -3.56

obtidas aleatóriamente e pesos de computação homogêneo, aparece um número maior destes casos de indisponibilidade, seguido por G_n de topologia regular, mas heterogêneo em relação aos pesos de computação das tarefas. Já Di_n com grafos de topologia regular e tarefas homogêneas em computação, sempre apresenta disponibilidade de processadores para escalonar as *backups* nos testes realizados. Dependendo da relação de precedência das tarefas a serem escalonadas, para *backups* com maior número de predecessores, R_n e G_n, o escalonamento restringue mais a disponibilidade de processadores. Para uma tarefa com mais predecessores aumenta a chance de falhar, se sua primária for classificada não totalmente forte. Neste caso, podem aumentar as restrições impostas nos critérios de escalonamento ao alocar cada *backup*, diminuindo o número de processadores possíveis para utilizar. Por exemplo, isto ocorre quando a *backup* não pode ser escalonada no processador de uma primária candidata a lhe provocar *falha induzida*.

Os grafos de Gauss (G_n) se caracterizam por ter muitas dependências entre suas tarefas e principalmente múltiplos caminhos (largura do grafo) com tarefas conectadas (dependentes) com comprimentos até a altura do grafo. Se em um caminho houver muitas

primárias não classificadas *totalmente fortes*, com predecessores por sua vez não classificados *totalmente fortes* até a tarefa origem; essas primárias podem apresentar um número elevado de *candidatas a provocar suas falhas induzidas*. Note que estes caminhos aumentam com o tamanho da aplicação, e assim dependendo do escalonamento pode aumentar o número de predecessores para os quais as primárias são fracas. Se ao mesmo tempo estas primárias estiverem escalonadas mais espalhadamente nos processadores do ambiente, pelo tamanho do ambiente (menor) e da aplicação (maior), aumenta a chance de encontrar uma *backup* próxima ao final de algum caminho no GAD, que não acha processadores disponíveis para ser escalonada. Por C_4 e C_5, as *backups* de primárias *parcialmente fortes* ou *totalmente fracas* não podem ser escalonadas junto com primárias *candidatas a provocar suas falhas induzidas*.

Da mesma forma as topologias de R_{152} e R_{364} coincidem com grafos, onde cada primária têm muitos predecessores para os que ela é fraca, tendo suas correspondentes *backups* menos opções de escalonamento. Já D_i pela sua topologia se comporta diferente, uma mesma tarefa tem menos caminhos com dependências de primárias *parcialmente fortes* ou *totalmente fracas*, com menos probabilidade de provocar suas falhas. A regularidade e a homogeneidade da aplicação durante o escalonamento também é favorável neste sentido.

Por outro lado, como a solução $S_{\mathcal{M}}$ prioriza o *makespan*, para aplicações menores de G_n, inicialmente a maioria das primárias são escalonadas no grupo de processadores mais rápido em CPC. Porém na medida que aumenta a aplicação e como o número de processadores se mantém igual, os processadores com menores *csi* não determinam mais o escalonamento. Assim, a partir de certo número de tarefas as primárias podem ser alocadas em processadores, não necessariamente mais rápidos, tendo disponíveis um número maior deles. Por isto e pela topologia, aumenta a chance de G_n ter alguma *backup* de primária não classificada totalmente forte que não pode ser mais escalonada, ou seja, não pode usar os processadores de primárias candidatas a provocar sua *falha induzida* (por C_4 e C_5). Já com S_e, como o escalonamento busca um equilíbrio, influência também a confiabilidade, sendo mais difícil no cenário conflitante CPC escolher qualquer processador para alocar as primárias. Assim o escalonamento têm mais processadores disponíveis (geralmente os menos confiáveis) para alocar as *backups*.

Mesmo assim, em casos como estes onde acontece indisponibilidade de processadores uma variante deste algoritmo, o valor de w deve mudar para achar outra solução com disponibilidade de processadores que possa gerar um escalonamento tolerante a falhas.

Nos primeiros testes com FTMRCD, a falha de processador é simulada no último grupo P_2 (mais rápido), para um processador que apresenta a menor taxa de falha $FP(p_i) = 1.8 \times 10^{-6}$ deste grupo. Como o objetivo é simular um caso crítico, com estas características este processador apresenta um número elevado de primárias escalonadas e a falha ocorre logo no inicio executando a primeira tarefa da aplicação v_0^P. Observe que neste primeiro teste é considerado o algoritmo FTMRCD proposto somente para 1 falha de processador, onde as *backups* são escalonadas considerando o número total de processadores em P (sem grupos). As quatro primeiras Tabelas 5.1, 5.2, 5.3 e 5.4 mostram os resultados obtidos neste conjunto inicial de testes.

Tabela 5.4: Comparação da solução de FTMRCD com $w_2 = w_1$ e 1 falha de processador, com MRCD sem incluir o processador que falha ($m = 23$) no CPC.

	MRCD, $w_1 \neq 0$		FTMRCD, $w_1 = w_2$			
GAD_n	$S_{\mathcal{M}}$	S_e	$S_{\mathcal{M}}$	% $S_{\mathcal{M}}$	S_e	% S_e
G_{152}	190.0,	502.7	256.7	35.1,	**406.0**,	-19.2,
	0.93247	0.94470	0.92389	-0.92	0.93480	-1.04
G_{252}	318.7,	907.2,	479.9,	50.5,	**721.6**,	-20.4,
	0.85793	0.87990	0.83935	-2.16	0.86284	-1.93
G_{377}	480.4,	1643.4,	893.6,	86.0,	**1182.4**,	-28.0
	0.74863	0.78952	0.72681	-2.91	0.75434	-4.44
G_{527}	675.1,	2468.6,	1362.5	101.8,	**1687.0**	-31.6,
	0.60869	0.66951	0.58774	-3.44	0.62366	-6.86
G_{702}	942.2,	3810.7,	-	-	**2674.3**	-29.8,
	0.45729	0.53888	-	-	0.47626	-11.6
R_{80}	358.0,	1022.0,	457.0	27.6,	**1008.0**,	-1.36,
	0.84070	0.86653	0.83978	-0.10	0.86115	-0.62
R_{98}	398.0,	1241.0,	597.0	50.0	**1201.0**,	-3.22,
	0.80921	0.83904	0.80300	-0.76	0.82510	-1.66
R_{152}	410.0,	1752.0,	-	-	1870.0,	6.73,
	0.69602	0.75749	-	-	0.73478	-2.99
R_{256}	584.0,	1387.0,	803.0	37.5,	1493.0,	7.64,
	0.53889	0.58463	**0.54353**	0.86	0.58202	-0.44
R_{364}	808.0,	4380.0,	-	-	**4234.0**,	-3.33,
	0.41692	0.52088	-	-	0.49564	-4.84
Di_{81}	833.9,	1704.9,	1286.9	54.3,	**1624.9**,	-4.74,
	0.84373	0.87730	**0.85937**	1.85	0.84789	-3.35
Di_{100}	947.0,	1997.0,	1426.0,	50.5,	**1917.0**,	-4.00,
	0.81091	0.84900	**0.82543**	1.79	0.81459	-4.05
Di_{144}	834.0,	2541.0,	1692.0,	50.7,	**2327.0**,	-8.42,
	0.71896	0.78509	**0.75414**	4.89	0.72960	-7.06
Di_{256}	1268.0,	3749.0,	2302.0,	81.5,	**3465.0**,	-7.57
	0.55939	0.63910	0.55771	-0.30	0.58418	-8.60
Di_{361}	1518.0,	4844.0,	3172.0	108.9,	3319.0,	4.63,
	0.41511	0.52690	**0.46382**	11.7	0.44605	-15.3

Em particular nas Tabelas 5.1 e 5.3 como as *backups* são escalonadas com prioridade total para o *makespan* $w_2 = 0$, a confiabilidade é menor do que nas soluções obtidas nas Tabelas 5.2 e 5.4. No entanto para $S_{\mathcal{M}}$ obviamente o *makespan* gerado por FTMRCD é maior do que o gerado por MRCD. Como visto no capítulo anterior, $S_{\mathcal{M}}$ para MRCD obtêm a melhor solução para o *makespan* embora estabeleça um compromisso com a confiabilidade, portanto não é melhorada por FTMRCD, devido ao fato de que as *backups*, mesmo com $w_2 = 0$, são escalonadas em processadores diferentes aos de suas primárias, com características piores de *csi*. A confiabilidade em $S_{\mathcal{M}}$ para MRCD também é melhor que FTMRCD por considerar um compromisso entre os objetivos, o que não acontece com a solução de FTMRCD com $w_2 = 0$. Note que para FTMRCD, a confiabilidade também piora porque no caso do experimento em questão o processador que falha é o processador mais confiável do grupo P_2, assim as *backups* correspondentes as primárias que estão nele, quando escalonadas em P_2 têm o FP maior. Observe que a maioria destas *backups* estão em P_2 para priorizar o *makespan* $w_2 = 0$.

Diferentemente, para a outra solução analisada S_e de FTMRCD o *makespan* chega

a ser até menor que o de MRCD (% com valores negativos). Observe que as primárias (MRCD) foram escalonadas considerando o equilíbrio entre os objetivos, enquanto as *backups* manipuladas por FTMRCD só priorizam o tempo de execução. Assim, as *backups* foram escalonadas em processadores diferentes dos processadores das primárias, mais rápidos, por priorizar o *makespan* (geralmente em P_2). Já a confiabilidade de S_e para FTMRCD, embora pior que a de MRCD, é maior obviamente que a da solução $S_{\mathcal{M}}$.

Para o caso onde $w_2 = w_1$ para FTMRCD visto nas Tabelas 5.2 e 5.4, as soluções são melhores em confiabilidade. Note que as *backups* foram escalonadas considerando um compromisso entre os objetivos de MRCD. Como esperado, a solução de $S_{\mathcal{M}}$ produzida por MRCD com $w_1 > 0$ tem *makespan*s maiores do que aquelas produzidas por FTMRCD com $w_2 = 0$. Já para S_e o equilíbrio entre os objetivos é considerado para escalonar as *backups*, e da mesma forma que $w_2 = 0$, as soluções FTMRCD para o *makespan* alcançam valores menores do que MRCD, dependendo da aplicação (G_n e D_n). Tanto em $w_2 = 0$ como em $w_1 = w_2$, a maioria das *backups* das primárias do processador que falha foram escalonadas ainda no grupo mais rápido (P_2) garantindo a melhora do *makespan*. Em relação à confiabilidade, como falha o processador mais confiável em P_2, as *backups* são escalonadas em processadores com maior taxa de falha dentro de P_2. Especificamente, com $w_2 = w1$ o *makespan* de FTMRCD é um pouco melhor que em MRCD, ao aumentar a taxas de falha dos processadores pelas *backups*, diminui a confiabilidade, e assim o escalonamento tende a priorizar mais o *makespan* para manter o equilíbrio de S_e.

5.5.2 Duas Falhas de Processador

Nos experimentos agora analisados, cujos resultados podem ser vistos nas Tabelas 5.5, 5.6, 5.7 e 5.8, as soluções do algoritmo de múltiplas falhas mFTMRCD considerando 2 falhas de processador, para os mesmos casos anteriores $w2 = 0$ e $w2 = w1$ no CPC. As duas primeiras Tabelas 5.5 e 5.6 mostram a variação da soluções em relação ao escalonamento sem falha (MRCD). Logo depois, nas Tabelas 5.7 e 5.8 é feita a comparação com o escalonamento de MRCD sem os 2 processadores que falham para desta forma avaliar o escalonamento com o emprego de *backups* ($m = 23$). As falhas foram simuladas, cada uma nos grupos extremos P_0 e P_2. O processador que falha em P_0, é mais lento com $csi = 73$ mas apresenta baixa taxa de falha, $FP = 2.5 \times 10^{-5}$. Já o processador que falha em P_1 é rápido ($csi = 33$), no entanto é dos menos confiáveis em P ($FP = 6.7 \times 10^{-5}$). Para simular casos mais críticos de falha, estes processadores apresentam a falha no inicio da execução e têm um número elevado de tarefas escalonadas.

Neste grupo de experimentos, a solução de equilíbrio S_e é comparada para duas distribuições diferentes de P no CPC. A primeira distribuição mantém o mesma configuração do cenário proposto inicialmente com três grupos P_s, $s = 0...2$, cada um de 8 processadores. Já a nova distribuição considera só dois grupos de maior tamanho (12 processadores), de forma que o grupo P_1 (do meio) da distribuição inicial é agora dividido em duas partes iguais, e cada parte é agregada nos outros dois grupos extremos P_0 e P_2, respectivamente.

Como esperado, para $w_2 = 0$ (Tabelas 5.5 e 5.7) as soluções de escalonamento se

Tabela 5.5: Variação da solução de escalonamento com $w_2 = 0$ em mFTMRCD pela recuperação de 2 falhas de processador no inicio da execução no CPC.

	MRCD, $w_1 \neq 0$	mFTMRCD, $w_2 = 0$ ($k = 3$)		mFTMRCD, $w_2 = 0$ ($k = 2$)	
GAD_n	S_e	S_e	% S_e	S_e	% S_e
G_{152}	449.4,	548.9,	22.1,	**404.5**,	-9.99,
	0.94287	0.93345	-0.99	0.92763	-1.62
G_{252}	856.3,	1193.3,	39.3,	**791.2**,	-7.60,
	0.87844	0.86172	-1.90	0.85210	-3.00
G_{377}	1494.7,	2490.4,	66.6,	**1391.5**	-6.9,
	0.78560	0.75717	-3.61	0.75020	-4.51
G_{527}	2272.9,	4249.2	86.9,	**2177.1**	-4.21,
	0.66578	0.62991	-5.38	0.61581	-7.51
G_{702}	3463.3,	6996.5	102.0,	**3343.9**	-3.45,
	0.53184	0.48662	-8.50	0.47988	-9.77
R_{80}	1314.0,	**1022.0**,	-22.2,	1314.0,	0.0,
	0.84059	**0.85693**	1.94	**0.84342**	-2.47
R_{98}	1314.0,	**1168.0**,	-11.1,	1314.0,	0.0,
	0.84059	0.82904	-1.37	0.81921	-2.54
R_{152}	1533.0,	1971.0,	28.5,	1533.0,	0.0,
	0.75375	0.74086	-1.71	0.72658	-3.60
R_{256}	1314.0,	-	-	1314.0	0.0
	0.58315	-	-	0.56458	-3.29
R_{364}	4161.0,	4672.0,	12.2	**4111.0**,	-1.20
	0.51698	0.50112	-3.06	0.47258	-17.0
Di_{81}	1664.9,	1843.9,	10.7,	**1657.9**,	-0.42,
	0.87524	0.84582	-3.36	0.86147	-1.57
Di_{100}	1957.0,	2461.0,	25.7,	**1950.0**,	-0.36,
	0.84719	0.80284	-5.23	0.83229	-1.76
Di_{144}	2468.0,	3264.0,	32.2,	**2328.0**,	-5.67,
	0.78322	0.72779	-7.07	0.75561	-3.53
Di_{256}	3636.0,	5567.0,	53.1,	**3629.0**,	-0.19,
	0.63608	0.56354	-11.4	0.60203	-5.35
Di_{361}	4512.0,	8593.0,	90.4	**3952.0**,	-12.4
	0.52008	0.44704	-14.0	0.43307	-16.7

comportaram melhor em relação ao *makespan* que as soluções para $w_2 = w_1$ (Tabelas 5.6 e 5.8), mostrando para a segunda distribuição (2 grupos), um resultado bem melhor, com taxas % que alcançam valores negativos na maioria dos casos. Note que o algoritmo mFTMRCD diferente de FTMRCD para tolerar múltiplas falhas, deve dividir o sistema P em grupos de processadores P_s, e para cada primária não totalmente forte escalona além da *backup*, outra *backup* alternativa que dependendo das falhas de processador, pode executar. Isto obviamente, pode gerar um custo maior de tolerância a falhas da solução em relação aos dois objetivos. Portanto, se os grupos P_s na distribuição de processadores P têm um tamanho maior, e as aplicações apresentam um número alto de primárias totalmente fortes, esta variação pela tolerância a falhas pode diminuir.

Em geral, os resultados para a segunda distribuição com dois grupos de 12 processadores, foram sempre melhores que para a de três grupos de 8 processadores. Em média a aplicação D_n apresenta o melhor comportamento, tanto para $w_2 = 0$ como para $w_2 = w_1$, seguido por G_n. As tarefas em D_n apresentam uma quantidade menor de caminhos com predecessoras que em G_n, além de sua topologia regular e homogeneidade na computação,

Tabela 5.6: Variação da solução de escalonamento com $w_2 = w_1$ em mFTMRCD pela recuperação de 2 falhas de processador no inicio da execução no CPC.

	MRCD, $w_1 \neq 0$	mFTMRCD, $w_1 = w_2$ $(k = 3)$		mFTMRCD, $w_1 \neq 0$ $(k = 2)$	
GAD_n	S_e	S_e	% S_e	S_e	% S_e
G_{152}	449.4	881.1,	96.0	731.1,	62.6
	0.94287	0.93807	-0.51	0.93658	-1.62
G_{252}	856.3,	1975.5,	130.7	1549.8,	80.9
	0.87844	0.86858	-1.12	0.86638	-1.37
G_{377}	1494.7,	3516.8,	135.2	2721.1,	82.0
	0.78560	0.76579	-2.52	0.76244	-2.95
G_{527}	2272.9,	5949.8	161.7	4808.2	111.5
	0.66578	0.63827	-4.13	0.63411	-4.76
G_{702}	3463.3,	9710.3	180.3	7524.1	117.2
	0.53184	0.49410	-7.1	0.48745	-8.35
R_{80}	949.0,	1705.0,	79.6	1314.0,	38.4
	0.86478	0.86218	-0.3	0.85828	-0.75
R_{98}	1314.0,	2628.0,	100	1825.0,	38.8
	0.84059	0.83089	-1.15	0.82642	-1.69
R_{152}	1533.0,	2920.0,	90.4	2435.0,	58.8
	0.75375	0.74899	-0.63	0.74958	-0.55
R_{256}	1314.0,	-	-	2056.0	56.4
	0.58315	-	-	0.57695	-1.06
R_{364}	4161.0,	9231.0,	121.8	6862.0,	64.9
	0.51698	0.50927	-1.49	0.50435	-11.4
Di_{81}	1664.9,	2938.9,	76.5	2029.9,	21.5
	0.87524	0.84995	-2.89	0.84368	-3.61
Di_{100}	1957.0,	3775.0,	92.9,	2023.0,	3.37,
	0.84719	0.81995	-3.22	0.81289	-4.05
Di_{144}	2468.0,	4651.0,	88.4,	2826.0,	14.5,
	0.78322	0.74554	-4.81	0.73830	-5.74
Di_{256}	3636.0,	7060.0,	94.1,	5308.0,	45.9,
	0.63608	0.58456	-8.1	0.57809	-9.12
Di_{361}	4512.0,	10053.0,	122.8,	8195.0,	81.6,
	0.52008	0.46466	-10.6	0.45776	-11.9

o que diminui a chance de ter mais primárias não classificadas *totalmente fortes*. Já em R_n, como as topologias são diferentes, os resultados variam. Um ponto importante nestas comparações é mostrar como mFTMRCD em caso de falhas pode ter um custo menor em relação a solução, na medida que mais processadores fazam parte da distribuição, ou seja, dentro dos distintos grupos. Como a proposta destes algoritmos é para ambientes distribuídos maiores, melhores resultados serão obtidos com o aumento no tamanho dos grupos.

Os resultados obtidos da análise para a solução de confiabilidade com 2 falhas foram similares aos da realizada com 1 falha durante a comparação de $w_1 = 0$ e $w_2 = w_1$. Por outro lado, neste teste se observa que para $w2 = 0$, as soluções de confiabilidade podem variar com a variação do tamanho dos grupos. Já para $w_2 = w_1$ como confiabilidade é considerada pelo algoritmo ao escalonar as *backups*, mesmo variando os grupos, as soluções de confiabilidade obtidas em grupos diferentes são parecidas com valores maiores.

Tabela 5.7: Comparação da solução de mFTMRCD com $w_2 = 0$ e 2 falhas de processador, com MRCD sem incluir os processadores que falham ($m = 22$) no CPC.

	MRCD, $w_1 \neq 0$	mFTMRCD, $w_2 = 0$ ($k = 3$)		mFTMRCD,$w_2 = 0$ ($k = 2$)	
GAD_n	S_e	S_e	% S_e	S_e	% S_e
G_{152}	490.8,	548.9,	11.84,	**404.5**,	-17.5,
	0.94246	0.93345	-0.96	0.92763	-1.57
G_{252}	1009.9,	1193.3,	18.1	**791.2**,	-21.6
	0.87881	0.86172	-1.94	0.85210	-3.04
G_{377}	1806.5,	2490.4,	37.8,	**1391.5**,	-22.9,
	0.78744	0.75717	-3.84	0.75020	-4.73
G_{527}	2734.7,	4249.2	55.3,	**2177.1**	-20.3,
	0.66697	0.62991	-5.56	0.61581	-7.67
G_{702}	3923.2,	6996.5	78.3,	**3343.9**	-14.7,
	0.53086	0.48662	-8.33	0.47988	-9.60
R_{80}	1095.0,	1022.0,	6.67,	**949.0**,	-13.3,
	0.86514	0.85693	-0.95	0.84342	-2.51
R_{98}	1387.0,	**1168.0**,	-15.7,	**1314.0**,	-5.26,
	0.83747	0.82904	-1.01	0.81921	-2.18
R_{152}	1971.0,	1971.0,	0.0	**1533.0**,	-22.2
	0.75706	0.74086	-2.14	0.72658	-4.03
R_{256}	1460.0,	-	-	**1314.0**	-10.0
	0.58151	-	-	0.56458	-2.91
R_{364}	4745.0,	**4672.0**,	-1.54,	**4111.0**,	-13.3,
	0.51540	0.50112	-2.77	0.47258	-8.31
Di_{81}	1996.9,	**1843.9**,	-7.66,	**1657.9**,	-16.9,
	0.88004	0.84582	-3.89	0.86147	-2.11
Di_{100}	2322.0,	2461.0,	5.99,	**1950.0**,	-16.0,
	0.85193	0.80284	-5.76	0.83229	-2.31
Di_{144}	3092.0,	3264.0,	5.56,	**2328.0**,	-24.7,
	0.79070	0.72779	-7.96	0.75561	-4.44
Di_{256}	4552.0,	5567.0,	22.3,	**3629.0**,	-20.2,
	0.64431	0.56354	-12.5	0.60203	-6.56
Di_{361}	5972.0,	8593.0,	43.8,	**3952.0**,	-33.8,
	0.53301	0.44704	-16.1	0.43307	-18.7

5.5.3 Múltiplas Falhas de Processador variando m

Outros testes foram realizados para múltiplas falhas variando da forma $k = (2, 4, 6, 8, 10)$, com aplicações maiores G_{1034}, Di_{529} e R_{546}, aumentando o número de total de processadores m em P para CPA (Tabela 5.9). Números de falhas diferentes (kf) foram simulados, aumentando proporcionalmente com o número de processadores m (aproximadamente 10%). A solução analisada S_e de mFTMTCD obtém, na maioria dos casos, soluções melhores (% negativos) ou iguais (em G_{1034} e Di_{529}) que as de MRCD em relação a *makespan*, mesmo com um número maior de falhas, até '10 (10f). Observe que, a maioria das *backups* que devem executar foram escalonadas em processadores mais rápidos ($w_2 = 0$) que de suas primárias (com melhores resultados com 1 e 2 falhas). Nos casos onde as soluções para o *makespan* são maiores foram geradas por R_{546} com um pequeno custo na variação da solução comparado aos outros. Este GAD apresenta topologia irregular, tarefas com maior número de predecessores e um mesmo peso de computação $e = 50$. Já a confiabilidade em geral diminui, como esperado, com $w_2 = 0$, os processadores menos confiáveis são mais

Tabela 5.8: Comparação da solução para mFTMRCD com $w_2 = w_1$ e 2 falhas de processador, com MRCD sem incluir os processadores que falham ($m = 22$) no CPC.

	MRCD, $w_1 \neq 0$	mFTMRCD, $w_1 = w_2$ ($k = 3$)		mFTMRCD,$w_1 = w_2$ ($k = 2$)	
GAD_n	S_e	S_e	% S_e	S_e	% S_e
G_{152}	490.8	881.1,	79.5,	731.0,	48.9,
	0.94246	0.93807	-0.47	0.93658	-1.57
G_{252}	1009.9,	1975.5,	95.6,	1549.8,	53.4,
	0.87881	0.86858	-1.16	0.86638	-1.41
G_{377}	1806.5,	3516.8,	94.6,	2721.1,	50.6,
	0.78744	0.76579	-2.75	0.76244	-3.17
G_{527}	2734.7,	5949.8	117.5,	4808.2	75.8,
	0.66697	0.63827	-4.3	0.63411	-4.93
G_{702}	3923.2,	9710.3	147.5,	7524.1	91.7,
	0.53086	0.49410	-6.92	0.48745	-8.18
R_{80}	1095.0,	1705.0,	55.7,	1314.0,	20.0,
	0.86514	0.86218	-0.34	0.85828	-0.79
R_{98}	1387.0,	2628.0,	89.4,	1825.0,	31.5,
	0.83747	0.83089	-0.79	0.82642	-1.32
R_{152}	1971.0,	2920.0,	48.1,	2435.0,	23.5,
	0.75706	0.74899	-1.07	0.74958	-0.99
R_{256}	1460.0,	-	-	2056.0	40.8
	0.58151	-	-	0.57695	-0.78
R_{364}	4745.0,	9231.0,	94.1,	6862.0,	44.6,
	0.51540	0.50927	-1.19	0.50435	-2.14
Di_{81}	1996.9,	2938.9,	47.1,	2022.9,	1.30,
	0.88004	0.84995	-3.42	0.84368	-4.13
Di_{100}	2322.0,	3775.0,	62.5,	**2023.0**,	-12.8,
	0.85193	0.81995	-3.75	0.81289	-4.58
Di_{144}	3092.0,	4651.0,	50.4,	**2826.0**,	-8.60,
	0.79070	0.74554	-5.71	0.73830	-6.63
Di_{256}	4552.0,	7060.0,	55.1,	5308.0,	16.6,
	0.64431	0.58456	-9.27	0.57809	-10.2
Di_{361}	5972.0,	10053.0,	68.3,	8195.0,	37.2,
	0.53301	0.46466	-12.8	0.45776	-14.1

utilizados para escalonar as *backups*.

5.5.4 Até Cinco Falhas de Processador com $m = 25$

Na Tabela 5.10 outra classe de experimentos mostra as soluções geradas pelos três algoritmos MRCD, FTMRCD e mFTMRCD no CPH. Neste caso, são apresentados os resultados de mFTMRCD para $k = (2, 3, 5)$ considerando distribuições de k grupos. A taxa % mostra a variação da solução de escalonamento em relação à execução sem falha. O ambiente em CPH é homogêneo em relação a velocidade dos processadores, enquanto as taxas de falhas FP dos processadores em P apresentam valores aleatórios próximos. Para tolerar 2 falhas (2f) o *cluster* P de $m = 25$ processadores foi dividido em dois grupos diferentes: P_0 com 15 processadores e P_1 com 10 processadores. Para 3 falhas (3f), P foi dividido em três grupos: P_0 e P_1 com 8 processadores e P_2 com 9 processadores. Na última configuração de 5 falhas, P divide-se em 5 grupos iguais P_s, $s = 0, ..., 4$, de 5 processadores. Em todas as falhas de processador simuladas, os processadores se caracterizam pode ser dos mais confiáveis de P, apresentar um número elevado de tarefas escalonadas e falhar no

Tabela 5.9: Variação da solução de escalonamento para G_{1034}, R_{546} e Di_{529} com um número de falhas diferentes (k) e com a variação de m processadores no CPA

GADs	m (n^o de falhas k)	MRCD, $w_1 \neq 0$	mFTMRCD, $w_2 = 0$	
		S_e	S_e	%
G_{1034}	24 (2f)	5308.8, 0.89017	**5097.3,** 0.87303	-3.98, -1.92
	45 (4f)	3306.8, 0.91219	3306.9, 0.87254	0.0, -4.34
	66 (6f)	3546.3, 0.92950	**2658.1,** 0.85188	-25.0, -8.35
	87 (8f)	2955.0, 0.93126	**1966.5,** 0.84376	-33.4, -9.39
	108 (10f)	2955.0, 0.92849	**1786.3,** 0.85057	-39.5, -8.39
R_{546}	24 (2f)	6351.0, 0.90604	**6278.0,** 0.89291	-1.14, -1.44
	45 (4f)	2263.0, 0.90952	2263.0 0.87710	0.0, -3.56
	66 (6f)	1533.0, 0.91426	2343.0, 0.86943	52.8 -6.12
	87 (8f)	1347.0, 0.91838	1815.0, 0.86215	25.7 -6.08
	108 (10f)	1274.0, 0.91820	1657.0, 0.86578	30.0 -5.70
Di_{529}	24 (2f)	5826.0, 0.90674	**5324.0,** 0.88221	-8.61, -2.70
	45 (4f)	2845.0, 0.91563	2982.0 0.85972	4.81, -6.10
	66 (6f)	3509.0, 0.93584	**3113.0**, 0.86950	-11.2 -8.61
	87 (8f)	3257.0, 0.94103	**3140.0**, 0.85992	-3.59 -8.61
	108 (10f)	3144.0, 0.93522	**2510.0**, 0.86454	-20.1, -7.55

início da execução.

Note que para o aumentar o número de falhas toleradas, uma quantidade maior de grupos deve ser considerada na distribuição de P, o que tende piorar os valores dos objetivos nas soluções de escalonamento. Entretanto para 2 e 3 falhas os resultados nos experimentos realizados em geral foram bons, as soluções obtidas foram próximas das soluções sem falha, e as taxas % calculadas chegam alcançar em muitos dos casos valores negativos. Já para 5 falhas, inevitavelmente o custo da solução aumenta bastante, mais em relação ao *makespan*, visto que os grupos de processadores são muito pequenos (5 processadores). É importante ressaltar que as 5 falhas de processador simuladas foram caracterizadas como críticas (ocorrem no início e com um elevado número de tarefas). Além disso, para grupos menores, o escalonamento de mFTMRCD perde a chance de achar processadores com melhores características para escalonar um número elevado de *backups*. Portanto, os experimentos finais (Tabela 5.10) mostram como aumenta o custo da tolerância a falhas na medida que a aplicação é maior e mais falhas devem ser toleradas, em um mesmo número de processadores m. Ressaltam também a importância da relação

Tabela 5.10: Variação da solução de escalonamento com $w2 = 0$ pela recuperação de até 5 falhas de processador no inicio da execução de G_n no CPH.

	MRCD, $w_1 \neq 0$	FTMRCD, $w_2 = 0$		mFTMRCD, $w_2 = 0$					
GADs	S_e	S_e (1f)	$\%S_e$	S_e (2f)	$\%S_e$	S_e (3f)	$\%S_e$	S_e (5f)	$\%S_e$
G_{152}	217.5,	**212.0**,	-2.53	**207.0**,	-4.83	**207.0**	-4.83	**201.9**,	-7.17
	0.99851	0.99787	-0.06	0.99798	-0.05	0.99795	-0.06	0.99800	-0.05
G_{256}	286.1,	288.4,	0.80	315.5	10.2	303.6	6.12	477.9	67.0
	0.99662	0.99556	-0.11	0.99555	-0.11	0.99558	-0.10	0.99566	-0.10
G_{377}	508.2,	**488.5**	-3.88	**516.1**	-1.55	518.8	2.09	898.3	76.7
	0.99376	0.99189	-0.19	0.99186	-0.19	0.99187	-0.19	0.99199	-0.18
G_{527}	822.9,	**769.1**	-6.54	**801.3**	-2.62	874.9	6.32	1325.2	61.0
	0.98963	0.98689	-0.28	0.98663	-0.30	0.98655	-0.31	0.98672	-0.29
G_{702}	1222.6,	**1095,2**	-10.4	**1182.2**	-3.30	1293.9	5.83	2610.9	113.5
	0.98396	0.97991	-0.98	0.97924	-1.05	0.97910	-1.06	0.97948	-1.03
G_{1034}	2135.3,	**1848.2**	-13.4	**2062.6**	-3.40	**2039.6**	-4.48	4737.0	121.8
	0.97131	0.96490	-0.66	0.96274	-0.88	0.96274	-0.88	0.96343	-0.81
G_{1829}	5002.5,	**4096.3**	-18.1	**4036.0**	-19.3	**4626.6**	-7.51	11513.8	130.1
	0.93323	0.92058	-1.36	0.91798	-1.63	0.91329	-2.14	0.91513	-1.94
G_{3002}	10184.8,	**8225.7**	-19.2	**8048.6**	-20.9	**9568.9**	-6.05	24334.9	138.9
	0.86366	0.84112	-2.61	0.83596	-3.21	0.82533	-4.44	0.82874	-4.04
G_{4094}	16444.0,	**13817.4**	-15.9	16713.6	1.64	**15489.5**	-5.80	38652.8	135.0
	0.79161	0.75964	-4.04	0.73660	-6.95	0.73627	-6.99	0.74041	-6.47

que existe entre a quantidade de grupos (e seus tamanhos) na distribuição de P, e as falhas a ser toleradas.

Note que em todos os conjuntos distintos de experimentos analisados foram considerado casos mais críticos de falhas de processador, que usam por exemplo cenários conflitantes como CPC, processadores que falham no começo da execução e números elevados de tarefas escalonadas. Casos deste tipo com piores cenários de falha não acontecem comumente, portanto, em ambientes reais estáveis com a mesma quantidade de processadores e grupos que os analisados, os custos em execuções com falhas devem ser menores.

Em geral, dependendo do tamanho do sistema distribuído e dos grupos considerados para tolerar falhas, assim como da classificação obtida das tarefas primárias da aplicação, a solução de escalonamento obtida com tolerância de múltiplas falhas pode variar em relação a distintos aspectos. Por exemplo, o custo associado pode mudar em função das características dos processadores que falham (csi e FP), do número de tarefas escalonadas nesses processadores, do momento que ocorre a falha, das características da aplicação (topologia, dependências e quantidade de predecessores em cada tarefa etc.), número de primárias *candidatas a provocar falha induzida*, *backups* sucessoras por primária que falha, pesos de computação de primárias que falham, entre outros aspectos.

5.6 Comparação com Trabalhos Relacionados de Replicação Passiva

Nesta seção, são enumerados a seguir os principais aspectos que foram considerados nos algoritmos de escalonamento propostos FTMRCD e mFTMRCD. Estas propostas são comparadas com os trabalhos relacionados [61] e [60], que empregam também escalona-

mento estático com replicação passiva. A Tabela 5.11 mostra um resumo comparativo relacionando os distintos aspectos analisados a seguir.

1. A função de custo é integrada, em vez de ser hierárquica, acrescentando todas vantagens analisadas no Capítulo 4.

2. A função de custo é ponderada, oferecendo flexibilidade na escolha de uma solução de compromisso tolerante a falhas, com as vantagens do Capítulo 4. Em particular oferece mais opções para achar soluções possíveis para tolerância a falhas, quando determinado escalonamento apresenta indisponibilidade de processadores.

   Comparando [60] com os algoritmos propostos, como só uma solução pode ser gerada pelo escalonamento para determinada aplicação e ambiente, o algoritmo em [60] tem mais chances de não achar um escalonamento com disponibilidade de recursos em ambientes mais limitados de processadores. Diferente de [60], os algoritmos propostos neste trabalho consideram uma função de custo ponderada que aumenta a probabilidade de achar mais soluções para tolerância a falhas próximas ao compromiso almejado, dentre múltiplos escalonamentos gerados para uma mesma aplicação e ambiente.

3. Propõe uma classificação de tarefas primárias mas flexível e consistente, com objetivo de facilitar a formulação dos critérios de escalonamento de *backups*, e melhorar o custo da variação da solução pela tolerância a falhas. Com esta classificação informações importantes são definidas para ser empregadas durante a execução da aplicação com falha.

   A classificação em [60], limita-se a uma mesma classificação da primária v_i^P em relação a todos os predecessores. Diferentemente, neste trabalho de tese são propostas as relações *forte e fraca para* utilizadas pela classificação da primária C_1 e nos distintos critérios de escalonamento de *backups*. A classificação proposta varia de acordo com a relação que existe com os distintos predecessores. Portanto, comparando C_1 com a classificação em [60], C_1 é mais flexível por ter distintas maneiras de classificar v_i^P. Assim, dependendo de cada tipo de classificação obtida *totalmente forte*, *parcialemnte forte* ou *totalmente fraca*, podem ser aplicadas ações diferentes, ou seja, determinadas regras de escalonamento para as *backups* com o objetivo de garantir o desempenho e a consistência da execução com falha.

   Em geral, os diferentes conceitos introduzidos para classificar a primária, facilitam a formulação dos critérios de escalonamento de *backups*. A nova classificação define mais opções de escalonamento para uma *backup* sobre os diferentes processadores. A *backup* sendo escalonada, deixa de ser alocada somente nos processadores dos predecessores aos quais ela é *fraca*, e tem disponíveis para utilizar os processadores dos outros predecessores que ela é *forte*.

   O critério C_1 permite também determinar as listas de tarefas predecessoras e sucessoras de uma primária em caso de falha. A lista $LSucFalha(v_j^P)$ é formada pelas

backups sucessoras de v_j^P, que devem executar caso v_j^P falhar, pois as respectivas primárias são *candidatas a falha induzida* pela predecessora v_j^P, assim como v_j^P é uma *candidata a provocar falha induzida* delas. Esta informação definida pelos algoritmos propostos é importante, pois será utilizada depois pelo mecanismo tolerante a falhas em um ambiente real para permitir a continuidade da execução da aplicação em caso de falha. Outra lista $LPredFalha(v^P)$ também gerada durante a classificação, contém as predecessoras *candidatas a provocar falha induzida* de v^P, e seus processadores não farão parte do conjunto de processadores onde a *backup* v^B pode ser escalonada.

Por outro lado, a classificação de primária forte em [60] apresenta a seguinte inconsistência. Quando v_j^P e v_i^P estão alocados no mesmo processador (item 1 da classificação na Seção 5.2), se v_j^P não é forte então v_i^P não seria classificada forte. Note que em caso de falha induzida de v_i^P ($p(v_i^P)$ não teve falha), como v_i^P não é forte, por definição em [60] a falha de v_i^P foi ocasionada por algum predecessor v_j^P que não consegue enviar mensagem. Desta forma v_i^B terá que executar. Porém, se v_j^P não é forte, mas v_j^B consegue enviar mensagem para v_i^P no tempo previsto (o que é possível), então v_i^P não poderia falhar (contradição). Neste caso, veja que v_i^P deveria ser classificada em [61, 60] como uma primária forte, podendo executar sempre em caso de falha em vez de sua *backup* v_i^B, a não ser que falhe $p(v_i^P)$. Portanto, C_1, além de ser flexível como visto anteriormente, considera este caso, que garante consistência e aumenta o número de primárias fortes da aplicação. Como resultado, melhora o desempenho da execução com falhas.

4. Propõe critérios para escalonar as *backups* e garantir a tolerância a falhas com a execução consistente da aplicação, maximizando o desempenho.

   Em relação ao escalonamento de *backups*, [60] não cobre as situações necessárias para uma correta implementação do algoritmo, gerando com a formulação diferentes inconsistências. Existem situações de escalonamento de *backups* que não são contempladas, que sem elas não se garante um correto funcionamento do escalonamento, e menos uma execução sucedida da aplicação com falhas. Por exemplo, o caso da regra colocada em [60] que diz quando as *backups* das primárias v_i^P e v_j^P com relação de precedência, não podem ser escalonadas no mesmo processador. Esta regra não é suficiente, pois não especifica o caso quando v_j^B é escalonada antes de v_i^P e não consegue enviar mensagem para v_i^P (*fraca* para v_j^P). Note que aqui, v_i^B não poderia ser escalonada no processador de v_j^P, o que geraria problemas durante a execução da aplicação com falha (não funciona). Da mesma forma, [60] não contempla os casos onde a *backup* v_i^B não pode ser escalonada em processadores de primárias que podem provocar a falha induzida de v_i^P, como proposto em C_5.

5. Propõe um critério para sobreposição entre *backups* de forma a maximizar desempenho.

   Em [61] como em outros trabalhos da literatura que consideram uma falha de pro-

cessador, a sobreposição de *backups* pode ser efetuada sempre que os processadores das respectivas primárias sejam diferentes. Porém, note que esta regra não pode ser aplicada sempre. Por exemplo, se uma das primárias v_i^P não é forte, ou seja, algum predecessor pode provocar sua *falha induzida*, então v_i^B não poderia estar escalonada nos processadores das primárias que podem provocar a falha de v_i^P, mesmo que estes processadores sejam diferentes a $p(v_i^P)$. Portanto, o uso desta regra para escalonar as *backups* gera inconsistência durante a execução da aplicação em caso de falha. Já em [60] os mesmos autores melhoram a regra, porém adicionando muitas restrições. Primeiro proíbem a sobreposição de *backups* quando as correspondentes primárias têm relação de precedência, e nos outros casos obrigam a que as primárias sejam fortes em relação a todos seus predecessores. Observe que o critério de sobreposição proposto C_6 é mais flexível ao se basear em C_1. Distintas opções de escalonamento são introduzidas onde não necessariamente só *backups* de primárias *totalmente fortes* podem ficar sobrepostas como em [60]. Desta forma melhora o desempenho da execução com falha.

6. Propõe um critério para sobreposição de *backups* com primárias de forma a maximizar o desempenho.

   Note que em [61] a sobreposição de *backups* com primárias não foi considerada. Já em [60] esta sobreposição foi abordada, mas somente para tarefas com relação de precedência. Diferentemente, neste trabalho o critério C_7 é proposto também entre tarefas primárias que não apresentam relação de precedência em G, o que permite melhorar ainda mais o desempenho da execução com falha.

7. Propõe um simulador da execução da aplicação em caso de falhas, que gera o novo escalonamento com as tarefas que realmente devem executar, e cálcula o *makespan* e a confiabilidade da solução.

   Para analisar os beneficios da estratégia de escalonamento proposta, é necessário calcular o *makespan* e a confiabilidade do escalonamento resultante quando ocorrem falhas permanentes de processador. Assim, neste trabalho de tese é proposto um simulador da execução da aplicação GAD, de acordo com os escalonamentos produzidos por FTMRCD e mFTMRCD ($Sch \cup SchBck$). O objetivo principal é estimar o *makespan* $\mathcal{M}_{falha}$ e a confiabilidade $R_{T_{falha}}$, alcançados pela aplicação quando ocorrem falhas permanentes de processador, e gerar o escalonamento Sch_{falha} das primárias e *backups* que realmente executam em caso de falha. Isto permite avaliar o custo, estudando as soluções obtidas, assim como analisar a consistência da execução com tolerância a falhas através do escalonamento resultante. Note que esta proposta não é abordada antes por trabalhos correlatos, [60] por não apresentar um simulador deste tipo, não pode avaliar o comportamento consistente da execução, nem as soluções que devem ser geradas em caso de falha. Observe que também não testa a recuperação da aplicação usando sua estratégia em um ambiente real.

8. Permite a tolerância de múltiplas falhas de processador diferente dos algoritmos de

[61] e [60], onde só é permitida uma falha permanente de processador.

Diferentemente de [60], neste trabalho com mFTMRCD podem ser toleradas múltiplas falhas de processador, sendo a execução da aplicação recuperável independente do número de falhas que podem ocorrer até o valor máximo definido. O máximo de falhas que podem ser toleradas são os k grupos obtidos ao distribuir os processadores em P antes do escalonar as *backups*, sendo possível uma falha de processador por grupo.

9. Permite estabelecer compromissos com o número de falhas toleradas. Relação com os grupos de processadores em $P(k)$, e com o ajuste dos pesos (w) no escalonamento bi-objetivo.

   Com a estratégia proposta é possível controlar o custo de tolerância a falhas relacionado ao tempo de execução e a confiabilidade, mediante uma função de compromisso no algoritmo, que controle de maneira flexível a quantidade de grupos de processadores k. Para regular este custo, pode ser atribuído um valor diferente a k dependendo da situação. Caso deseja-se obter um custo menor na solução durante a execução da aplicação, embora a aplicação aumente seu risco de apresentar falhas, o valor de k, que distribui P em subconjuntos P_i, pode diminuir e vice-versa.

   Por outro lado, é possível considerar também um compromisso entre o número de falhas toleradas ou grupos k e o peso associado à confiabilidade da aplicação w, durante o escalonamento bi-objetivo. Se a confiabilidade do escalonamento apresenta maior prioridade (maior w na função ponderada), então deve ser menor a probabilidade da aplicação falhar durante a execução. Neste caso, para diminuir o custo em relação a ambos objetivos do escalonamento, pode ser interessante considerar um número k menor de falhas a ser toleradas pelo algoritmo mFTMRCD.

   Portanto, considerar a abordagem bi-objetivo ponderada proposta no capítulo anterior, e a agrupação dos processadores como parte do algoritmo de múltiplas falhas, podem auxiliar na redução do custo, e como consequencia da sobrecarga do tempo final de execução. A proposta de uma função de compromisso pode controlar este custo, ao permitir variar o número de falhas que podem ser toleradas em função da importância atribuída, a confiabilidade no escalonamento ou ao número de subconjuntos de processadores.

5.7 Conclusões

Neste capítulo foi apresentada uma abordagem para escalonar grafos de precedência sobre um ambiente distribuído de processadores heterogêneos considerando tolerância a falhas. O escalonamento estático proposto permite recuperar falhas permanentes de processador. O escalonamento tolerante a falhas baseia-se no esquema primária-*backup* da técnica de replicação passiva, e inclui a abordagem bi-critério ponderada proposta no capítulo anterior, com o propósito de minimizar os custos da execução com tolerância a falhas

Tabela 5.11: Comparação da Proposta

	Aspectos	([Qin2002], [61])	([Qin2006], [60])	Proposta
1	Função de custo	hierárquica	hierárquica	integrada
2	Ponderação	não	não	sim
3	Classificação de primárias	sim (não flexível, inconsist.)	sim (igual [61])	sim (dif. [60])
4	Escalonamento de *backup*s	sim (inconsist.)	sim (dif. [61], inconsist.)	sim (dif. [60])
5	Sobreposição de *backups*	sim (inconsist.)	sim (dif. [61])	sim (dif. [60])
6	Sobreposição de primárias	não	sim	sim (dif. [60])
7	Múltiplas falhas	não	não	sim
8	Escalonamento e solução com falhas	não	não	sim
9	Compromisso com número de falhas	não	não	sim

em relação a tempo de execução e confiabilidade. Além disso, diferente da literatura para replicação passiva como [60, 61], múltiplas falhas de processador podem ser toleradas.

Novos conceitos foram introduzidos no algoritmo proposto, além de informações necessárias para garantir uma recuperação consistente e mas eficiente da aplicação em um sistema real com falhas. Para o escalonamento de *backups* foram propostos os critérios necessários sendo utilizados pelas distintas funções definidas no algoritmo. Uma nova classificação das tarefas primárias e das falhas que podem sofrer, foram introduzidas no capítulo, oferecendo maior flexibilidade nos critérios de escalonamento. Diferentemente da literatura, foi possível avaliar a execução da aplicação com falhas usando a estratégia, ao gerar o escalonamento real das tarefas que executam em caso de falha e calcular a solução com tolerância a falhas, *makespan* e confiabilidade. Este resultado permitiu realizar um estudo maior de como seria o comportamento da execução da aplicação na presença de falhas sobre um sistema distribuído.

Na maioria dos testes realizados, os resultados obtidos em relação aos custos da tolerância a falha ao variar a solução ($\mathcal{M}$, R_T) são razoáveis, se consideramos que falhas mais críticas de processador foram simuladas em todos os casos. Nos distintos cenários com falha, em particular a solução de equílíbrio S_e, proposta na abordagem bi-objetivo, teve seu melhor desempenho para o *makespan* chegando a alcançar valores menores que no escalonamento das primárias. Na medida que maior é o sistema distribuído, mais falhas podem ser toleradas e com menor custo. A adição da ponderação nos objetivos dentro do escalonamento de *backups*, oferece mais flexibilidade ao algoritmo tolerante a falhas, sendo possíveis distintas soluções com tolerância a falha para uma mesma aplicação e cenário da arquitetura.

Capítulo 6

Mecanismo de Recuperação para Viabilizar a Estratégia de Escalonamento em Ambientes Reais

Para avaliar a estratégia de escalonamento proposta sobre um ambiente real, foi desenvolvido um mecanismo em MPI (*Message Passing Interface*)[72] com o objetivo de executar com sucesso aplicações paralelas na presença de falhas. Neste trabalho, é utilizado um modelo de troca de mensagens implementado pela biblioteca, para executar aplicações paralelas em sistemas distribuídos. Para testar a estratégia de escalonamento tolerante a falhas em um ambiente real, os mecanismos necessários foram desenvolvidos dentro do *middleware* SGA do Projeto EasyGrid [14, 23].

Inicialmente, neste capítulo, descreve-se brevemente o Projeto EasyGrid e relacionado a ele: o princípio de funcionamento do *middleware* Sistema de Gerenciamento da Aplicação (SGA EasyGrid), a hierarquia de gerenciamento e os aspectos de tolerância a falhas considerados. Na seção seguinte é apresentado o mecanismo proposto baseado na técnica de replicação passiva para o *middleware* SGA. Em seguida, são detalhadas as funcionalidades introduzidas.

6.1 Projeto SGA EasyGrid

Para que uma aplicação paralela em um ambiente heterogêneo distribuído tenha bom desempenho, ela deve ser escalonada apropriadamente e de maneira que os recursos do sistema possam ser utilizados com eficiência e confiabilidade. Portanto, a necessidade de adaptar a execução de aplicações às diferentes características destes ambientes leva ao Projeto EasyGrid [14, 23] a investir na integração de importantes aspectos, tais como a modelagem das aplicações e do ambiente, o escalonamento das tarefas da aplicação, o monitoramento do sistema, mecanismos de tolerância a falhas, entre outros.

O *middleware* EasyGrid foi projetado com o foco principal de tornar as aplicações MPI em aplicações MPI autonômas [56] (*system-aware*) sobre uma grade computacional, transformando automaticamente os programas paralelos escritos em MPI em aplicações

system-aware. As aplicações *system-aware* são aquelas capazes de se auto-adaptar às mudanças ocorridas no ambiente computacional. Com o objetivo de definir e testar novas políticas de execução, o projeto EasyGrid leva em consideração a facilidade de uso e a portabilidade, o estudo de problemas de escalonamento estático e dinâmico, e a integração com estratégias de tolerância a falhas para a execução de aplicações *system-aware* em grades computacionais.

Uma visão centrada na aplicação é tratada na metodologia do EasyGrid, ou seja, o *middleware* de serviço é parte de cada aplicação individual. O uso eficiente dos recursos é feito por um Sistema Gerenciador da Aplicação (SGA) associado a cada aplicação, e não por um sistema gerenciador de recursos como na maioria dos casos, como por exemplo Condor-G [30], Legion [36], entre outros. Na visão centrada na aplicação é possível observar que em cada aplicação é embutido um *middleware* de serviço específico na forma de um SGA. Como conseqüência, a aplicação torna-se portável, eliminando a necessidade de *middlewares* de serviços instalados nos distintos recursos do ambiente onde serão executados processos da aplicação. Já na visão centrada nos recursos o *middleware* de serviço, por ser parte do ambiente, limita o número de recursos disponíveis para a aplicação executar.

6.1.1 *Middleware* SGA EasyGrid

No *middleware* desenvolvido SGA EasyGrid, um escalonador estático especifica um escalonamento inicial nos recursos do sistema. Com base em dados iniciais e em informações obtidas durante a execução, o escalonamento dinâmico pode ajustar a execução da aplicação a mudanças ocorridas no meio. O monitoramento de aplicações é também outra das questões importantes abordadas pelo Projeto EasyGrid para prover a execução da aplicação de mecanismos de controle e tolerância a falhas.

Partindo da lista dos recursos disponíveis, o Modelador do Sistema, cria um modelo arquitetural [52] através da obtenção de informações (por exemplo, velocidade do processador, carga da máquina, latência de comunicação, entre outros) ou de serviços de diretórios (tal como MDS Globus). Com base nas características disponíveis dos recursos e em características da aplicação, um escalonamento estático é produzido pelo escalonador estático para guiar a execução da aplicação, através da identificação dos recursos para os quais cada processo MPI deve ser alocado. Este escalonador, é também responsável por definir as máquinas onde serão disparados os processos gerenciadores do SGA EasyGrid.

Uma aplicação MPI *system-aware* é gerada incorporando o *middleware* SGA apropriado a cada aplicação, sem alterar o código fonte do usuário. As funções do SGA são embutidas nos processos MPI através de uma camada de abstração (*wrapper*) desenvolvida para agregar funcionalidades às chamadas das funções padrões do MPI. Essa informação, assim como dados da aplicação e parâmetros do SGA são armazenados em arquivos de esquema (*Schema*) utilizados para disparar a aplicação *system-aware*. A execução da aplicação do usuário é iniciada e é realizada a distribuição inicial dos dados necessários no ambiente.

No escalonamento de tarefas, uma aplicação paralela pode ser representada por um

Grafo Acíclico Direcionado (GAD). A utilização de aplicações sintéticas representadas por GADs facilita a investigação dos efeitos da granularidade e estrutura do programa em relação à eficiência do escalonamento gerado e da sua execução em ambientes de Grades.

O *middleware* EasyGrid permite que aplicação se adapte ao comportamento da grade, considerando a disponibilidade dos recursos e as características específicas de cada aplicação. Com o *middleware* embutido na aplicação se garante a portabilidade, sendo possível um número maior de recursos disponíveis, assim como uma aplicação mais eficiente e com distintos serviços disponíveis de escalonamento e tolerância a falhas. Estas vantagens fazem com que seja o SGA EasyGrid o *middleware* escolhido para agregar o mecanismo de tolerância a falhas proposto neste trabalho.

O *middleware* EasyGrid é capaz de controlar a execução de processos de uma aplicação MPI através de uma estrutura hierárquica de processos gerenciadores. A distribuição hierárquica foi adotada com o intuito de adequar os processos gerenciadores à organização dos recursos do ambiente distribuído.

A identificação de falhas é possível devido a existência de um mecanismo de monitoramento capaz de coletar dados durante a execução da aplicação MPI. Com base nesses dados é possível tomar decisões com relação à redistribuição de processos pelos recursos disponíveis na Grade, de forma a obter um ganho no desempenho da aplicação e solucionar problemas decorrentes de falhas em recursos ou processos.

O *middleware* SGA de uma aplicação EasyGrid é distribuído em três níveis de processos gerenciadores: o Gerenciador Global (GG), responsável pelo controle da execução da aplicação em todos os *sites* da Grade; os Gerenciadores dos Sites (GS), responsáveis pela execução da aplicação nas máquinas dos *sites*; e os Gerenciadores Locais das Máquinas (GM), responsáveis por gerenciar a execução de processos da aplicação atribuídos a cada máquina local. A Figura 6.1 mostra a estrutura hierárquica associada para uma aplicação executando em dois *sites*, onde cada *site* tem duas máquinas.

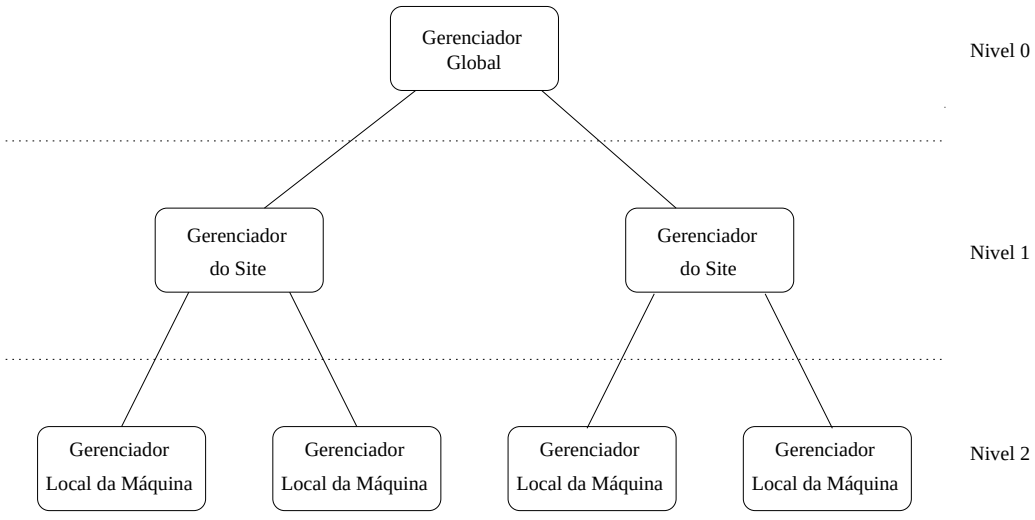


Figura 6.1: Hierarquia de gerenciadores na grade computacional (cortesia de [22]).

O *middleware* SGA EasyGrid é estruturado em camadas, como ilustrado na Fi-

gura 6.2. A camada de gerenciamento de processos é responsável pela criação dinâmica de processos MPI, tanto da aplicação como gerenciadores, e pelo redirecionamento de mensagens trocadas entre os processos criados. A coleta de informações relevantes relacionadas à execução de uma aplicação MPI é desempenhada pela camada de monitoramento da aplicação. Os dados coletados pela camada de monitoramento alimentam as camadas de escalonamento e tolerância a falhas. No topo da estrutura em camadas do SGA EasyGrid está a aplicação MPI do usuário e a camada de abstração MPI (*wrapper*).

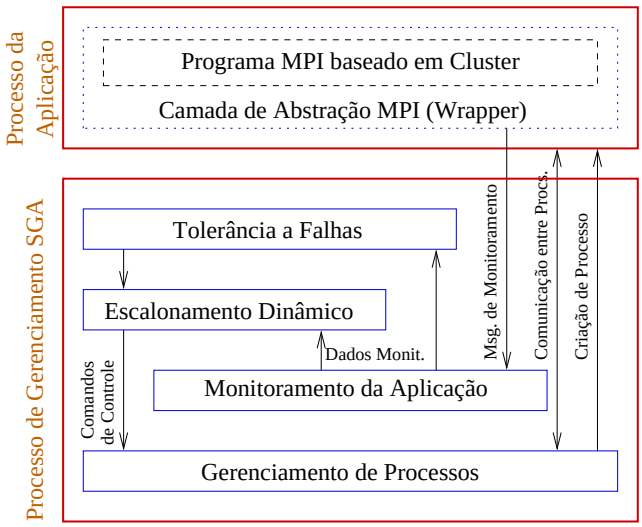


Figura 6.2: Estrutura em camadas do SGA EasyGrid (cortesia de [22]).

A estrutura de gerenciamento do SGA considera a possível implementação de novas estratégias para tratamento de falhas e mecanismos de redistribuição de tarefas da aplicação, e no momento pesquisas são realizadas neste sentido, sendo de interesse estudos que integrem novas políticas de escalonamento com mecanismos de tolerância a falhas. A filosofia dinâmica do EasyGrid SGA, tanto para o gerenciamento como para execução da aplicação GAD, facilita o uso da técnica de replicação passiva da estratégia de escalonamento proposta, para decisões de criação dinâmica, escalonamento e execução de réplicas (*backups*) das tarefas, assuntos abordados em próximas seções.

Durante a execução de uma aplicação no SGA, diferentes políticas de escalonamento dependendo do caso podem ser ativadas. A implementação de mecanismos de tolerância a falhas, em função da estrutura hierárquica, devem considerar características específicas do funcionamento efetivo dos processos gerenciadores, dependendo de seu nível hierárquico.

Ao iniciar a execução de uma aplicação EasyGrid, o Gerenciador Global é primeiramente criado, recebendo dados do Portal EasyGrid que permitem a criação dos outros processos gerenciadores da hierarquia. Entre as informações recebidas, relacionamos o número de *sites* disponíveis para executar a aplicação, dados do escalonamento estático, número máximo de processos concorrentes, tipo de política de escalonamento global, entre outros aspectos.

A partir de então, o GG cria os Gerenciadores de cada *Site*, baseado na informação dada sobre a organização dos recursos. O Gerenciador do *Site* é responsável por gerenciar

a execução de tarefas nas máquinas do seu *site* e recebe dados do portal EasyGrid que permitem desempenhar as sua funções. Nessa informação estão definidos o tipo de política de escalonamento local do *site*, por exemplo, e também a identificação das máquinas de cada *site*, para que seus respectivos Gerenciadores da Máquina sejam criados.

No último nível, o Gerenciador da Máquina dispara tarefas seguindo determinada política de escalonamento local da máquina, definindo que a ordem e o momento que um processo deve ser criado. O Gerenciador da Máquina também atende a pedidos dos outros gerenciadores, cedendo ou não tarefas segundo a política escolhida. Ainda, o GM coleta informações sobre escalonamento, tempos de fim ou métricas coletadas e cálculos e tempos, restante dos processos a serem ainda executados, e repassada para os gerenciadores de acordo com a hierarquia. A decisão de criar ou não tarefas prontas da aplicação para execução, deve ser tomada pelos processos gerenciadores com base em critérios bem definidos pelos mecanismos propostos.

6.1.2 Tolerância a Falhas

Em sistemas distribuídos é comum a ocorrência de falhas nos recursos disponíveis. O SGA EasyGrid foi desenvolvido desde o começo com a intenção de fornecer facilidades para o tratamento de falhas, prevendo a adição e implementação posterior de novas ferramentas com mecanismos tolerantes a falhas no *middleware*.

Como visto anteriormente, o SGA controla a execução dos processos da aplicação MPI através da hierarquia de processos gerenciadores, levando gerenciamento aos recursos distribuídos. Portanto, a estrutura hierárquica permite um melhor controle da execução da aplicação, uma vez que cada processo gerenciador é capaz de adotar diferentes políticas de escalonamento e tolerância a falhas específicas à máquina ou *site*. Desta forma, os mecanismos de tolerância a falhas utilizados devem funcionar distribuindo-se entre os diferentes níveis dos gerenciadores, responsáveis por detectar as anomalias, repassar e agir quando for necessário. Os Gerenciadores dos *Sites* devem ser responsáveis pela detecção e recuperação das falhas sobre o conjunto de processadores de seu *Site*, o Gerenciador Global é o líder que controla aos Gerenciadores dos *Sites* e permite a troca de informação entre eles.

O Gerenciador da Máquina implementa um esquema de gerenciamento de *log*s de mensagens, para manter a ordem das mensagens de entrada de um processo da aplicação. Cada Gerenciador da Máquina grava e gerencia o *log* de mensagens de cada processo que cria, o qual é mantido em uma ordem cronológica. Este esquema pressupõe que os processos são determinísticos, ou seja, com os mesmos dados de entrada sempre produzem os mesmos resultados. Todas a mensagens enviadas antes para o processo com falha serão recuperadas de seu *log* de mensagens. O *log* será mantido pelo Gerenciador da Máquina, somente para processos em execução ou pendentes, sendo liberado para processos que terminam de executar normalmente.

Como no SGA toda mensagem que é enviada por um processo passa antes pelo Gerenciador da Máquina, que a repassa para o nível do processo destino, dentro ou fora do

processador, dentro ou fora do mesmo *site*, podem ser atualizados a cada momento os *logs* de mensagens referentes aos processos destino com as mensagens enviadas. Para alcançar tolerância a falhas, o Gerenciador Global contém uma cópia com os *logs* de mensagens de todos os gerenciadores dos *Sites* que, por sua vez, têm uma réplica dos *logs* de todos os Gerenciadores de Máquinas de seu *site*. Portanto, durante a execução da aplicação, os *logs* de níveis superiores na hierárquia de gerenciadores são atualizados com a informação dos *logs* de níveis inferiores.

Os mecanismos para o tratamento de falhas podem adotar distintas políticas que variam dependendo da abordagem e do modelo de falhas considerados. As políticas podem ser locais, quando consideram-se apenas as máquinas pertencentes a um único *site*, ou globais quando leva-se em consideração todos os *sites* da grade. O número máximo de processos concorrentes e a ordem com que esses processos são criados estão relacionados à política local adotada por cada processo gerenciador das máquinas em que foram atribuídas tarefas da aplicação.

Em geral, os mecanismos de tolerância a falhas seguem quatro estágios para realizar o tratamento da falha: detecção da falha, localização da falha, contenção da falha e recuperação da falha. A seguir são descritos aspectos importantes considerados no SGA para o tratamento de falhas em relação a estes estágios.

No SGA a comunicação entre os processos é mediada através dos gerenciadores. O uso de diferentes intercomunicadores entre cada par de procesos gerenciadores e da aplicação, permite que falhas possam ser facilmente localizadas e isoladas. Uma vantagem disto é que além de controlar o *overhead* gerado pela comunicação entre processos da aplicação, facilita a detecção de falhas e erros de comunicação, através dos gerenciadores.

As mensagens de monitoramento são utilizadas pelos processos gerenciadores como sinais de vida de processos da aplicação e de outros processos gerenciadores. Esses sinais podem indicar se um determinado processo em execução está ou não ativo. A partir destas informações, decisões relacionadas ao tratamento de falhas podem ser tomadas pelos processos gerenciadores. Por exemplo, decisões de quando criar, escalonar ou executar processos baseados em critérios definidos pela estratégia utilizada.

Por outro lado, as funções da biblioteca MPI são também usadas para detectar a ocorrência de erros em operações de envio e recebimento entre processos da aplicação e processos gerenciadores. A detecção de falhas pelo MPI durante a execução da aplicação, é habilitada através de *error handlers* (MPI_ERRORS_RETURN) associados aos comunicadores dos processos. As funções MPI tratadas devem retornar um código de erro na ocorrência de falhas. Os processos gerenciadores analisam o valor retornado por estas operações, identificando se ocorreu ou não alguma falha durante a comunicação. Caso sejam identificadas falhas, pode ser ativado o mecanismo de recuperação proposto, e se necessário uma notificação é enviada até o Gerenciador Global desde os Gerenciadores dos *Sites*.

Portanto, o subsistema de tolerância a falhas do SGA pode detectar as falhas através de erros de comunicação, quando mensagens são enviadas ou recebidas dos processos da aplicação pelo Gerenciador da Máquina e entre os processos gerenciadores. Como a ordem, o tamanho e o tempo das mensagens são desconhecidos, os gerenciadores usam funções não

bloqueantes para minimizar a intrusão. Entretanto, a detecção de falhas em processos MPI não é sempre garantida, especialmente para funções MPI não bloqueantes que são operações locais e nem sempre identificam que um processo remoto está morto, mesmo com o *erro handler* configurado para MPI_ERRORS_RETURN.

Em [69] foi realizada uma análise sobre os problemas de detecção de erros com estas funções MPI, mostrando que existem casos que não se detecta o processo morto. Por exemplo, MPI_Send, quando usa os protocolos de comunicação *short* e *eager* é não bloqueante. O envio de duas mensagens consecutivas para um processo morto, usando o protocolo *eager* e com determinado tamanho de *bytes*, pode também causar a morte do processo origem. Já no caso de MPI_Recv, se a mensagem está disponível no *buffer* do sistema quando a função é invocada, a morte do processo remoto nem sempre é detectada.

Uma solução proposta em [69] foi desenvolvida no SGA para verificar o estado dos processos remotos antes de cada comunicação. Esta técnica é adotada para detectar falhas de processos. No Gerenciador do *Site*, uma *thread* é criada para testar se o *daemon* e o Gerenciador da Máquina remota estão vivos. A *thread* executa uma chamada de sistema com o comando *mpitask*, com o objetivo de testar os Gerenciadores das Máquinas. Uma variável compartilhada entre a *thread* e o Gerenciador do *Site* recebe o resultado da chamada de sistema. Se este resultado não está disponível quando o Gerenciador do *Site* precisa enviar outra mensagem, este gerenciador continua testando a variável numa série de tempos de espera, aumentados exponencialmente até um limite máximo configurável. Se a *thread* não está bloqueada, ou seja, *mpitask* terminou, então o *deamon* remoto está vivo. Caso contrário, depois de ultrapasar o limite máximo de espera a falha é identificada.

No SGA outra camada importante além da tolerância a falhas é o escalonamento de tarefas (Figura 6.2), que pode ser estático ou dinámico. Em [54] são propostas novas políticas de escalonamento dinâmico para o SGA garantir o desempenho da aplicação quando acontecem variações de carga e *hardware* no sistema distribuído. O trabalho de [20, 69] propõe um mecanismo para tolerar falhas de processos da aplicação e falhas de recursos, que utiliza o escalonamento dinâmico de [54]. Desta forma, uma vez detectada a falha de um processo, como analisado anteriormente, ativa-se o escalonador dinâmico para determinar onde o novo processo deve executar. Especificamente, a tolerância a falhas dos processos gerenciadores forma parte do trabalho de tese [20] que aborda um tratamento de falhas com *checkpointings* proposto para o *middleware* SGA.

Como descrito na Figura 6.2, cada processo gerenciador no SGA EasyGrid é estruturado em camadas. A funcionalidade de cada camada depende do nível do processo gerenciador na estrutura hierárquica (Figura 6.1). Neste trabalho de tese um mecanismo para tolerância a falhas é proposto no SGA. A nova abordagem propõe a implementação de novas funcionalidades nas camadas: escalonamento e tolerância a falhas.

Na próxima seção descrevem-se detalhes sobre as propostas realizadas em relação ao mecanismo de tolerância a falhas proposto no *middleware* SGA.

6.2 Mecanismo Tolerante a Falhas Proposto no SGA

Um espectro de mecanismos de tolerância a falhas pode ser encontrado na literatura [16, 29, 37, 50, 76, 78, 81]. Entretanto o custo associado, assim como a forma de manipular as falhas, pode afetar o desempenho da aplicação, sendo um fator importante a escolha do método a ser adotado. Em particular, diversos trabalhos com tolerâncias a falhas abordam o tratamento de falhas por si só, sem considerar como as tarefas da aplicação vão ser alocadas aos processadores.

O objetivo principal da implementação de um mecanismo tolerante a falhas em MPI foi viabilizar a estratégia de escalonamento proposta em capítulos anteriores, para assim avaliar seu funcionamento em um ambiente real, ou seja, avaliar a consistência e o desempenho da execução da aplicação com falhas.

O SGA EasyGrid utiliza o modelo de execução 1PTask [67], onde os programas consistem de um largo número de processos de curta duração, determinados pelo paralelismo da aplicação e não pelo número de recursos. Por esta razão, não é necessário utilizar a técnica de *checkpointing* para recuperar os processos da aplicação, pois os processos duram pouco tempo e o custo de reexecução é baixo. A idéia consiste em que os benefícios obtidos ao se evitar a implementação de sofisticados esquemas de *checkpointing* e de manter longos *log*s de mensagens, podem compensar o custo de gerenciar um maior número de processos. Desta forma para recuperar a aplicação se justifica o uso de técnicas de replicação de processos no SGA.

Cada gerenciador é capaz de adotar diferentes políticas de escalonamento e tolerância a falhas específicas a seu nível. Espera-se que com esta implementação no SGA, a estratégia proposta torne-se mais escalável e ofereça menor impacto no desempenho da aplicação.

Os mecanismos de tolerância a falhas desenvolvidos no SGA, distribuím-se entre os diferentes níveis dos gerenciadores, responsáveis por detectar as anomalias, e agir quando for necessário. Em particular, os Gerenciadores dos *Sites* apresentam as funcionalidades principais dos mecanismos de tratamento de falhas, são responsáveis pela recuperação das falhas sobre o conjunto de processadores de seu *site*, o Gerenciador Global é o líder que controla aos Gerenciadores dos *Sites* e permite a troca de informação entre eles.

Como a comunicação entre os processos é mediada através dos gerenciadores do *middleware* SGA, eles podem detectar falhas e erros de comunicação, e aplicar as soluções necessárias de acordo com o mecanismo utilizado. Os tipos de falhas considerados pelo mecanismo são baseados no modelo de confiabilidade proposto neste trabalho de tese, ou seja, falhas permanentes de processador e falhas dos processos: induzida e de *hardware*. A seguir serão descritos aspectos considerados durante a implementação do mecanismo proposto referentes aos dois estágios principais para realizar o tratamento: detecção e recuperação da falha.

Para ser utilizado o mecanismo, a aplicação MPI insere o SGA nela durante a compilação, e assim pode rodar independente do sistema com todas as facilidades que o *middleware* lhe oferece, desempenho e tolerância a falhas. No começo da execução, o escalonamento estático tolerante a falhas proposto é lido pelos processos gerenciadores, e as

tarefas que executam seguem este escalonamento. O escalonador estático utiliza os dados gerados pelos algoritmos de escalonamento tolerantes a falhas propostos (FTMRCD ou mFTMRCD) no capítulo anterior, para aplicar uma estratégia de recuperação em caso de falha.

Na versão utilizada do SGA, existe uma estrutura vetor de tarefas que armazena os dados específicos de cada processo da aplicação. Com o mecanismo, o SGA também atualiza no vetor de tarefas os dados referentes a *backup* de cada processo da aplicação (informação lida do escalonamento estático proposto). Por exemplo, são armazenados: a máquina que deve ser usada para executar a *backup* em caso de falha, a ordem de execução e a lista de tarefas *backups* sucessoras que devem ser ativadas.

Para a aplicação utilizar o mecanismo proposto no SGA, o escalonamento dinâmico implementado por [54] é desativado e insere-se o conhecimento produzido pelo escalonamento estático proposto neste trabalho, que vai funcionar como um escalonamento dinâmico predefinido. Desta forma a execução da aplicação é realizada somente de acordo com a estratégia estática, ou seja, assume-se que o ambiente é semi-controlado e que outras aplicações não estão executando ou compartilhando os recursos.

Novas funcionalidades relacionadas à estratégia de tolerância a falhas são adicionadas nos processos gerenciadores do SGA (Local, do *Site* e Global). Primeiramente, os Gerenciadores dos *Sites* adotam funções para recuperar falhas permanentes de processador. O Gerenciador do *Site* detecta uma falha de processador, identificando falha em um Gerenciador Local da Máquina. Logo em seguida ativa o mecanismo de recuperação, replicando os processos *backups* nas máquinas previstas pelo escalonamento estático e dá continuidade a execução da aplicação.

Dependendo do nível hierárquico do processo gerenciador que está agindo, a falha é detectada e tratada, e decisões diferentes são tomadas como criação dinâmica de *backups*, re-alocações sobre determinadas máquinas previstas e re-envio de mensagens. Para tratar falhas, os gerenciadores aproveitam as mensagens de monitoramento do *middleware* e acrescentam um controle coordenado maior com o Gerenciador Global. Os Gerenciadores dos *Sites* também estabelecem comunicação com gerenciadores dos outros níveis da hierarquia (Figura 6.1) para re-alocar processos e redirecionar as mensagens. As mensagens são lidas desde os *logs* de mensagens dos distintos processos. Os Gerenciadores Global e Local da Máquina também sofrem modificações para implementar o mecanismo tolerante a falhas. Detalhes sobre o funcionamento do mecanismo proposto são apresentados na próxima seção.

Para funcionar o mecanismo proposto como parte do SGA *EasyGrid*, o escalonamento estático tolerante a falhas é lido inicialmente para alocar as tarefas da aplicação nos recursos heterogêneos, considerando a possível ocorrência de falhas. As funções MPI para a intercepção e tratamento de erros, formam parte do conjunto de bibliotecas que o SGA utiliza para detectar e tratar falhas em função das políticas de tolerância a falhas adotadas no portal do SGA. As novas funcionalidades como a decisão de executar processos *backups* durante a recuperação das falhas, são atribuídas no gerenciamento hierárquico do *middleware* SGA.

Desta forma, ao compilar o programa MPI do usuário é gerada uma aplicação *User's EasyGrid System Aware MPI Application*, onde fica embutido o SGA implicitamente com as novas políticas de escalonamento e os mecanismos de tratamento de falhas. Esta aplicação apresenta a capacidade de recuperar-se em caso de falha com a nova abordagem proposta.

Com a proposta do mecanismo tolerante a falhas no *middleware* SGA, primeiramente o mecanismo local cresce, e adquire uma estrutura distribuída com comportamento hierárquico, preparado para funcionar sobre ambientes e aplicações maiores. Por outro lado o *middleware* EasyGrid acrescenta uma nova estratégia de escalonamento tolerante a falhas, que pode ser ativada ou não em função das características da aplicação e do ambiente de execução distribuído. Desta forma o SGA EasyGrid pode executar a aplicação considerando múltiplos objetivos, a confiabilidade além do tempo da execução, e ainda a possibilidade de recuperar-se em caso de falhas.

A seguir serão descritos maiores detalhes sobre o funcionamento do mecanismo de recuperação proposto, para executar os processos da aplicação na presença de falhas permanentes de processador.

6.2.1 Recuperação com Replicação Passiva

Com o mecanismo tolerante a falhas no SGA, a aplicação executa seguindo o escalonamento estático de FTMRCD ou mFTMRCD, lido como entrada. Inicialmente, os processos gerenciadores dos sites, depois de criados, leem dos arquivos as informações das tarefas primárias e suas *backups*, que são necessárias para o tratamento em caso de falha. Estes dados são atualizados em uma estrutura associada aos processos da aplicação (vetor de tarefas), que armazenam informações que os processos precisam para executar. Para cada processo é armazenada por exemplo, uma lista com os processos *backups* sucessores que devem ser replicados em outros processadores, caso o processo falhar. Estas informações ficam armazenadas, e somente são acessadas quando ocorre falha. Se um processo finaliza com sucesso, seus dados no vetor de tarefas são automaticamente liberados.

Logo que se detecta uma falha de processador, os processos gerenciadores no SGA gerenciam a recuperação, a partir do mecanismo proposto baseado na estratégia de escalonamento estático com replicação passiva. Este mecanismo é proposto para tolerar falhas permanentes de processador, que como resultado, provocam também falhas nos processos da aplicação: *falha por hardware* e *falha induzida*, definidas no capítulo anterior. Note que as falhas dos processos gerenciadores são toleradas por mecanismos próprios implementados no SGA, e estão fora do escopo deste trabalho de tese.

O mecanismo de recuperação proposto implementa uma técnica de replicação passiva em conjunto com o uso de *logs* de mensagens, onde um *log* é armazenado para cada processo da aplicação. Com a adição deste mecanismo, o SGA apresenta o uso de técnicas de replicação de tarefas nas camadas de escalonamento e tolerância a falhas dos processos gerenciadores.

Em particular, a técnica de replicação passiva proposta requer da tarefa primária e de

suas *backup*s, para em caso que a primária falhar, se ative a *backup* e a aplicação continue a execução. A criação dinâmica de processos no SGA é uma das facilidades oferecidas pelo *middleware* e utilizada pelo mecanismo de recuperação durante a replicação passiva, para re-alocar os processos com falha. Um processo *backup* somente é criado quando seu correspondente processo primário falha.

Como a criação de processos no SGA garante que toda a comunicação sejá feita através de intercomunicadores entre pares de processos, caso falha um dos processos, os demais processos envolvidos na execução da aplicação não são informados. Isso evita a sincronização entre os processos espalhados pelas máquinas do sistema, e ajuda a não sobrecarregar muito o desempenho da aplicação.

As novas funcionalidades do mecanismo são introduzidas nos gerenciadores de acordo com a hierarquia. O gerenciador Global, deve ser responsável por garantir a execução livre de falhas em todos os *sites* do sistema distribuído; o do *Site*, deve ser responsável por garantir a recuperação nos recursos de seu *site*; e o da Máquina, responsável pelas funções associadas aos processos da aplicação alocados nessa máquina. Por exemplo: mudança de status (pronto, pendente, execução ou concluído), criação dinâmica, execução, liberação, etc. O gerenciador da Máquina é sempre alocado naquele processador que gerencia, o do *Site* pode ser alocado em qualquer processador de seu *site* por algum critério de desempenho.

Quando um Gerenciador do *Site* detecta uma falha de processador no seu *site*, o mesmo age para fazer a recuperação da aplicação com o mecanismo proposto. Se for necessário, dependendo das ações que devem ser executadas, o Gerenciador do *Site* se comunica com o Gerenciador Global, e então a mensagem de falha é repassada para os outros gerenciadores do sistema. Depois da falha passar pelas etapas detecção-localização, o Gerenciador de *Site* ativa o mecanismo de recuperação. No Algoritmo 10 são enumerados os principais passos executados pelo Gerenciador do *Site*, caso o sistema apresente falha permanente de processador durante a execução da aplicação.

Algoritmo 10 : *Recupera (p,* v_i^P*,vetorT)*

1 $\forall v_j^P \in p(v_i^P) / status(v_i^P) \neq concluido$

2 /* *Replica processos por falha de hardware* */

3 $\langle lMaq_{Reesc}, lMaq_{Rem}, vetorT \rangle = Replica(v_j^P, vetorT);$

4 $\forall p_j \in lMaq_{Reesc}$

5 $envia(Ger_{p_j}, vetorT_{Reesc}, tagFalha)$

6 $\forall p_j \in lMaq_{Rem}$

7 $envia(Ger_{p_j}, vetorT_{Rem}, tagLibera)$

Se um Gerenciador de *Site* detecta a falha de um processador $p(v_i^P)$, executando o processo v_i^P, o mesmo inicia a recuperação pelo Algoritmo 10. Neste caso, v_i^P apresenta *falha por hardware*. Inicialmente são identificados e isolados na linha 1 os processos da aplicação v_j^P, que foram alocados no processador da falha $p(v_i^P)$, e que ainda não executaram ($status(v_i^P) \neq concluido$). Considera-se que estes processos como consequência da

falha de $p(v_i)$ apresentam também *falha por hardware*. Portanto, para cada v_j é chamada a função $Replica(v_j^P, vetorT)$ detalhada no Algoritmo 11.

$Replica(v_j^P, vetorT)$ reescalona e prepara a execução dos processos v_j^P em outros processadores, atualizando o vetor de tarefas $vetorT$ com as informações dos processos *backups* correspondentes, processador do *backup* e ordem de execução. A função gera como saída a lista de máquinas que tiveram seus escalonamentos redefinidos $lMaq_{Reesc}$ com processos *backups*, e a lista de máquinas $lMaq_{Rem}$, que devem retirar processos primários por *falha de harware* ou *falha induzida*.

Depois de replicar os processos com falha (escalonar *backups*), o Gerenciador do *Site* envia para os Gerenciadores de Máquina dos processadores em $lMaq_{Reesc}$, informação dos processos que foram re-escalonados $vetorT_{Reesc}$, na máquina p_j. Os novos processos replicados devem executar nestes processadores. O gerenciador do *Site* também envia a Gerenciadores de Máquina em $lMaq_{Rem}$ os processos que devem ser removidos por apresentar falha induzida. Estes processos primários não podem executar e portanto devem ser liberados em seus processadores.

A função $Replica(v^P, vetorT)$ no Algoritmo 11, verifica primeiro se o processo v ainda não foi re-escalonado (linha 1) antes, devido a outras falhas. Desta forma, atualiza-se a lista de máquinas $lMaq_{Reesc}$ que devem receber processos re-escalonados com a adição da máquina p_j (p_j aloca o *backup* v^B seguindo a estratégia de escalonamento estático). Logo depois é atualizada com $p(v^P)$ a lista das máquinas que devem remover os processos com falha ($lMaq_{Rem}$). Os diferentes dados para replicar v^B são atualizados no vetor $vetorT$ na linha 4, sendo alterados os campos: máquina do processo e a ordem de execução.

No Algoritmo 11, $ListSucFalha$, proposto no Capítulo 5 contém a lista dos processos *backups* sucessores de v^P que devem executar se v^P falha. Estes processos v_j^P sofrem *falha induzida* pela falha de v^P. Assim, depois do Gerenciador do *Site* replicar v^P, $ListSucFalha(v^P)$ é percorrida na linha 7 para desta forma replicar também os processos v_j^P da lista.

Algoritmo 11 : *Replica(v^P, vetorT)*

```
1  if ¬replicado(v^P)
2      lMaq_Reesc = lMaq_Reesc ∪ p(v^B);
3      lMaq_Rem = lMaq_Rem ∪ p(v^P);
4      atualizaVetor(vetorT, v^B);
5      replicado(v^P) = 1;
6      /* Replica processos sucessores de v^P por falha induzida */
7      ∀v_j^B ∈ ListaSucFalha(v^P)
8          ⟨lMaq_Reesc, lMaq_Rem, vetorT⟩ = Replica(v_j^P, vetorT);
```

Depois do Gerenciador do Site executar o Algoritmo 10 para iniciar a recuperação da aplicação em caso de falha, uma função de recebimento de mensagens MPI_Recv, nos correspondentes Gerenciadores das Máquinas em $lMaq_{Reesc}$ e $lMaq_{Rem}$, recebe a mensagem enviada desde o Gerenciador do *Site*, com a *tag* correspondente. Dependendo do caso

(linhas 5 e 7 do Algoritmo 10), $tagFalha$ indica que a máquina recebe a informação dos processos ($vetorT_{Reesc}$) que devem ser adicionados na máquina, e com $tagLibera$ recebe informação dos processos que devem ser removidos ($vetorT_{Rem}$).

Algoritmo 12 : *RecebeTarefasF(vetorT$_{Reesc}$, tagFalha)*

1 $\forall v_j^P \in vetorT_{Reesc}$
2 if $\exists v^P \in Pred(v_j^P) / status \neq concluido$
3 $\quad lPendente = lPendente \cup v_j^P$;
4 else $lPronto = lPronto \cup v_j^P$;
5 $numProc = numProc + 1$;

No caso em que os processos devem ser re-alocados o Gerenciador da Máquina, que recebe $tagFalha$, executa a função $RecebeTarefasF$ no Algoritmo 12, que percorre os procesos em $vetorT_{Reesc}$. Para cada processo v_j^P atualiza seu status (pendente ou pronto) para execução e o coloca na lista correspondente. Se o processo ainda apresenta dependências com processos predecessores que ainda não terminaram de executar, o status é pendente para executar (lista $lPendente$), caso contrário o status é pronto (lista $lPronto$). Para concluir, a função atualiza o número total de processos que devem executar agora nesta máquina.

Algoritmo 13 : *LiberaTarefasF(vetorT$_{Rem}$, tagLibera)*

1 $\forall v_j^P \in vetorT_{Rem}$
2 if $\exists v^P / status \neq concluido$
3 $\quad listaP = listaP - v_j^P$;
4 $\quad numProc = numProc - 1$;
5 else $envia(Erro,' status = concluido')$;

Caso em que o Gerenciado da Máquina recebe $tagLibera$, os processos devem ser removidos do processador. A função $LiberaTarefasF$ do Algoritmo 13 percorre a lista destes procesos e para cada processo v_j^P, dependendo de seu status (pendente, pronto, execução ou concluído) executa um tratamento diferente. Se o status é diferente de concluído, então busca e retira o processo da lista correspondente (pendente, pronto ou execução), e logo depois libera o processo. Se concluído então emite um erro informando sobre o status desse processo. O número total de processos que devem executar é também atualizado nesta máquina.

Quando necessário o Gerenciador do *Site* comunica a falha ao Gerenciador Global, para este também agir durante a recuperação. Através do Global a mensagem de falha é repasada para outros Gerenciadores de *Sites*, dependendo de onde os processos, a serem re-escalonados ou removidos, estão localizados (fora do *site* da falha).

O mecanismo de recuperação, pela estratégia proposta no capítulo anterior, pode tolerar múltiplas falhas permanentes de processador a partir do escalonamento gerado por mFTMRCD. Entretanto, mesmo que o sistema fique confiável, o tratamento de múltiplas

falhas usando o mecanismo pode gerar certa sobrecarga no tempo de execução da aplicação. Por esta razão, resultados obtidos são avaliados na próxima seção, analisando o desempenho do mecanismo de recuperação sobre um ambiente real.

6.3 Avaliação de Desempenho

Nesta seção, é realizada uma avaliação do desempenho do mecanismo tolerante a falhas, funcionando em um ambiente computacional real e usando aplicações de maior escala. Algumas métricas como erro do *makespan*, variação do tempo de execução e sobrecarga do mecanismo são calculadas, assim como são empregados outros cenários para testar o novo modelo de escalonamento com falhas sobre um sistema real. Para realizar os experimentos, foi escolhido o GAD da aplicação Gauss G_n.

Para avaliar o desempenho do mecanismo no SGA foram realizados vários testes usando o sistema computacional Sinergia da UFF com 25 processadores Pentium IV 2.6 GHz com 512 Mb de RAM, executando Linux Fedora Core2, Globus toolkit 2.4 e MPI LAM 7.0.6. Os processadores estão interconectados por uma rede *Gigabit Ethernet* em ambiente semi-controlado, onde só executa a aplicação analisada embora o servidor e a rede podem ser acessados por múltiplos usuários. Os parâmetros da arquitetura, tais como índice de retardo *csi* e latência L, foram obtidos de acordo com as métricas coletadas por [52] e [55] com o objetivo de modelar o ambiente computacional real das máquinas de nossa instituição. O sistema distribuído é homogêneo em relação a velocidade dos processadores, modelado com índice de retardo $csi(p_i) = 23$ no cenário denotado CPH. As taxas de falha dos processadores, $FP(p_i)$, foram uniformemente geradas no intervalo $[1.8 \times 10^{-6}, 3.2 \times 10^{-6}]$, a partir de [60]. Os valores de taxa de falha e de tempo de execução são calculados em unidades de segundos.

Neste estudo como está sendo analisado o desempenho do mecanismo e da execução da aplicação para viabilizar a estratégia tolerante a falhas proposta, somente foi calculado o objetivo tempo de execução final da aplicação para comparar com o *makespan* $\mathcal{M}$ do par solução ($\mathcal{M}$,R_T) gerado pelo escalonamento estático.

6.3.1 Uma Falha de Processador

A precisão do modelo de escalonamento utilizado para a execução sem falha, foi analisada na Tabela 6.1, com o cálculo dos erros dos *makespans* em relação aos tempos de execução sem falha. Para distintos GADs de Gauss (G_N), primeiro foi calculado o *makespan* para a solução de equilíbrio S_e no cenário Cen1, usando o escalonamento de primárias de MRCD. O *makespan* de S_e foi comparado com o tempo de execução $TempExec$ obtido no SGA sem falha. O erro foi calculado como $ErroMakespan = \frac{(TempExec - makespan) \times 100\%}{makespan}$, obtendo valores menores que 10% em média e diminuindo com o tamanho da aplicação (3% aproximadamente). Em todos os experimentos foi considerada a média de três execuções. Estes resultados mostram que o modelo usado apartir de [51] pelo algoritmo de escalonamento proposto proporciona uma boa estimativa do tempo de execução da aplicação, e que

a execução da aplicação se comporta próximo do previsto.

Tabela 6.1: Erro do modelo de escalonamento em relação ao tempo de execução sem falha

GADs	*Makespan*	*TempExec*	*ErroMakespan*
G_{152}	217.5	232.0	6.70%
G_{252}	286.1	316.2	10.5%
G_{377}	508.2	561.6	10.5%
G_{527}	822.9	892.3	8.44%
G_{702}	1222.6	1285.7	5.16%
G_{1034}	2135.3	2253.9	5.56%
G_{1829}	5002.5	5153.9	3.03%
G_{3002}	10184.8	10493.4	3.03%

Analisando estes erros, mesmo que o ambiente seja semi-dedicado durante a execução outros parâmetros que não são considerados estaticamente no modelo afetam o tempo de execução real da aplicação. Sabe-se que o servidor e a rede de comunicação são acessados por múltiplos usuários, o que de maneira indireta influencia no tempo de execução final da aplicação. Por outro lado, o modelo proposto neste trabalho não considera aspectos internos de implementação específicos do SGA. Por exemplo, as mensagens que são trocadas entre os gerenciadores para controlar a execução da aplicação, onde processamentos internos são realizados para criação dinâmica de processos, armazenamento e redirecionamento de mensagens. Da mesma forma mecanismos para tratamento de falhas, ficam lendo sinais de monitoramento para assim detectar falhas a qualquer momento.

Por outro lado, no modelo é considerado como parâmetro de comunicação apenas a latência do canal de comunicação. No SGA existem três tipos distintos de comunicação entre as tarefas da aplicação. A comunicação entre tarefas que se encontram no mesmo processador, entre tarefas que se encontram em processadores distintos dentro do mesmo *site* ou entre tarefas alocadas em processadores de *sites* diferentes. Como os processos da aplicação se comunicam através dos processos gerenciadores, a sobrecarga para que uma mensagem seja recebida por um processo gerenciador existe, além do tempo que esta mensagem gasta para ser enviada en um canal de comunicação. Entretanto, o tempo de comunicação gasto por cada mensagem sobre a rede *Gigabit* é muito pequeno, e, em geral, pouca sobrecarga na comunicação é gerada.

Nos experimentos das Tabelas 6.2 e 6.3, foi considerada a ocorrência de uma falha permanente de processador em CPH. Uma falha crítica foi simulada no processador com $FP = 1.8 \times 10^{-6}$, a menor taxa de falha do cenário. Este processador apresenta o número maior de tarefas escalonadas comparado com os outros processadores, e a falha se produz logo no inicio da execução. Em particular, a Tabela 6.2 mostra a variação do tempo de execução de S_e, em relação a execução de S_e sem falha no mesmo cenário. Para aplicações de menor tamanho se observa um leve aumento do tempo de execução com falha. Quando a aplicação tem poucas tarefas existe uma sobrecarga maior, pelo tempo que se consome na criação de processos e na comunicação entre eles, sobre o tempo de computação. Porém

à medida que aumenta o tamanho da aplicação esta sobrecarga diminui, o tempo de execução com falha chega a ser menor que no cenário sem falha na maioria dos casos (valores negativos na taxa % de variação).

Tabela 6.2: Variação do tempo de execução pela recuperação de 1 falha de processador

GADs	$TempExec$	$TempExec1Falha$	$Variação$	$Makespan1Falha$	$ErroMakespan$
G_{152}	232.0	265.2	14.2%	212.0	25.1%
G_{252}	316.2	350.0	10.6%	288.4	21.3%
G_{377}	561.6	571.7	1.79%	488.5	17.0%
G_{527}	892.3	869.6	-2.55%	769.1	13.0%
G_{702}	1285.7	1224.7	-4.74%	1095.2	11.8%
G_{1034}	2253.9	2000.5	-11.2%	1848.2	8.24%
G_{1829}	5153.9	4434.8	-13.9%	4096.3	8.24%
G_{3002}	10493.4	8867.0	-15.5%	8225.7	7.80%

Observe que, neste exemplo as tarefas primárias que sofrem falha por *hardware* estavam escalonadas no processador mais confiável. Portanto, pela estratégia de escalonamento as *backups* correspondentes só podem estar escalonadas em processadores com maiores FP que o processador que falha. Como se trata da solução S_e, o escalonamento busca um equilíbrio entre os objetivos. Ao diminuir a confiabilidade das *backups*, o escalonamento tende a minimizar o tempo de fim para manter o equilíbrio, obtendo uma solução com *backups* de menor $\mathcal{M}$ e pior R_T que na solução S_e de MRCD, como analisado também no capítulo anterior.

A Tabela 6.2 mostra também o *makespan* obtido com o simulador que gera o escalonamento e a solução em caso de falha. Esta solução para o *makespan* é comparada com o tempo de execução com falha para assim analisar o erro da simulação. Note que o erro do *makespan* com uma falha é um pouco maior que o obtido sem falha (Tabela 6.1), devido a que agora se adiciona também a este erro, a sobrecarga que produz o mecanismo de recuperação proposto funcionando no SGA para permitir a execução sucedida da aplicação com falha. Especificamente, os valores calculados desta sobrecarga para 1 falha aparecem na Tabela 6.3, percentual de acréscimo do tempo de execução do escalonamento com uma falha em Cen1 em relação a execução do mesmo escalonamento sem ocorrer falha.

Para calcular a sobrecarga do mecanismo de execução com falha na Tabela 6.3, FTMRCD é comparada com uma nova execução sem falha ($TempExecEscBack$), mas que executa desde o começo as *backups* do escalonamento FTMRCD como se fossem tarefas primárias. Como os escalonamento das tarefas para ambas as execuções são iguais (sem falha e com falha), desta forma pode ser só medido o tempo que consome o mecanismo para recuperar a aplicação em caso de falha. Note que as taxas de acréscimo calculadas neste caso são pequenas. Como esperado as mais altas aparecem em aplicações menores por terem uma menor relação computação-comunicação que as outras. À medida que aumenta a aplicação, estas taxas diminuem, chegando a valores próximos de 0.

Tabela 6.3: Sobrecarga do Mecanismo de Recuperação. Acréscimo do tempo de execução de 1 falha em relação a execução sem falha considerando o mesmo escalonamento de tarefas.

GADs	$TempExecEscBack$	$TempExec1Falha$	$taxaAcresc$
G_{152}	237.7	265.2	11.5%
G_{252}	300.8	350.0	16.3%
G_{377}	531.3	571.7	7.60%
G_{527}	939.6	869.6	3.57%
G_{702}	1189.2	1224.7	2.99%
G_{1034}	1960.0	2000.5	1.60%
G_{1829}	4413.5	4434.0	0.46%
G_{3002}	8861.7	8867.0	0.06%

6.3.2 Duas Falhas de Processador

Com as Tabelas 6.4 e 6.5 para 2 falhas obtém-se resultados similares aos apresentados nas Tabelas 6.2 e 6.3. Porém, se observa um leve aumento na variação do tempo de execução e no erro do *makespan*, devido ao aumento do número de falhas neste experimento. Para simular 2 falhas, o sistema foi dividido em 2 *sites* com 12 processadores. Estas falhas foram simuladas em Cen1 usando os processadores mais confiáveis de cada *site*, com maior número de tarefas, e considerando que eles falham logo no início da execução da aplicação. Em relação à variação do tempo de execução na Tabela 6.4, como visto na análise de desempenho do capítulo anterior a estratégia de escalonamento mFTMRCD se sacrifica para tolerar múltiplas falhas, diminuindo assim o número de processadores que podem ser utilizados ao escalonar cada *backup* dentro do *site*. Entretanto, note que o acréscimo de tempo obtido, taxa de variação (%), é pequeno chegando a desaparecer à medida que aumenta-se a aplicação, até atingir valores negativos na Tabela 6.4.

Tabela 6.4: Variação do tempo de execução pela recuperação de 2 falhas de processador

GADs	$TempExec$	$TempExec2Falha$	$Variação$	$Makespan2Falha$	$ErroMakespan$
G_{152}	232.0	283.2	22.0%	207.0	36.8%
G_{252}	316.2	402.7	27.3%	315.6	27.6%
G_{377}	561.6	621.4	10.6%	516.1	20.4%
G_{527}	892.3	927.6	3.96%	801.3	15.7%
G_{702}	1285.7	1299.8	1.10%	1182.2	9.95%
G_{1034}	2253.9	2275.2	0.95%	2062.6	10.3%
G_{1829}	5153.9	4483.9	-13.0%	4036.0	11.1%
G_{3002}	10493.4	8857.2	-15.5 %	8048.6	10.0%

O erro do *makespan* calculado na Tabela 6.4 para duas falhas é um pouco maior na maioria dos casos, que o erro calculado na Tabela 6.2 para uma falha, dependendo dos processadores e do momento de falha. Nestes casos, no tempo de execução obtido para duas falhas acrescenta-se uma sobrecarga maior ao utilizar o mecanismo de recuperação,

ao ter que recuperar a aplicação do dobro de falhas comparado com a Tabela 6.2, sendo também quase o dobro de tarefas primárias a ser re-escalonadas em novos processadores. Portanto, como duas falhas críticas de processador são tratadas pelo mecanismo, há um acréscimo maior na sobrecarga, contudo este aumento é só de 1 a 2% em média (Tabela 6.5), em relação ao acréscimo da tabela 6.3. Observe também que as taxas de erro, variação e sobrecarga diminuim com o aumento da aplicação, favorecido pelo aumento da granularidade, ou seja, a relação entre computação e comunicação.

Tabela 6.5: Sobrecarga do Mecanismo de Recuperação. Acréscimo do tempo de execução com 2 falhas, em relação a execução sem falha e considerando o mesmo escalonamento de tarefas

GADs	$TempExecEscBack$	$TempExec2Falha$	$taxaAcresc$
G_{152}	239.3	283.2	18.3%
G_{252}	348.8	402.7	15.4%
G_{377}	573.1	621.4	8.43%
G_{527}	873.8	927.6	6.16%
G_{702}	1272.6	1299.8	2.14%
G_{1034}	2230.4	2275.2	2.01%
G_{1829}	4441.5	4483.9	0.95%
G_{3002}	8721.3	8857.2	1.56%

6.3.3 Cinco Falhas de Processador

A mesma classe de testes para 1 e 2 falhas, foram também realizados considerando 5 falhas críticas em CPH de $m = 25$. Neste caso, na Tabela 6.6, as taxas de variação aumentam proporcionalmente com o aumento da aplicação, como analisado anteriormente no modelo com falha no capítulo anterior. Observe que neste caso o cenário foi dividido igualmente em 5 sites, cada um com 5 processadores, sendo uma quantidade muito pequena para que as *backup*s a ser escalonadas durante o escalonamento ache com mFTMRCD, processadores com os melhores tempos de fim. Com a diminuição do tamanho do *site* esta busca fica restringida podendo até ocorrer indisponibilidade de processadores para escalonar, o que não aconteceu neste experimento. Note também que para tolerar um número maior de falhas, aumenta-se também o uso de *backups* alternativas com mFTMRCD (casos onde as *backups* originais não podem executar). Isto gera escalonamentos de maior *makespan* que os das *backups* originais, o que é necessário para garantir a execução completa da aplicação na presença de múltiplas falhas em tais condições mais críticas.

As mesmas observações que foram feitas na análise realizada para as falhas anteriores (1 e 2) se aplicam também nas Tabelas 6.6 e 6.7 de 5 falhas, observando obviamente um acréscimo na sobrecarga do mecanismo, e na variação do tempo de execução com falhas. Obviamente, isto se deve ao fato de que um número maior de falhas são tratadas, que como consequência aumenta o número de falhas de tarefas primárias por *hardware* e induzida, o que provoca um número maior de re-escalonamentos e atualizações no sistema.

Tabela 6.6: Variação do tempo de execução pela recuperação de 5 falhas de processador

GADs	$TempExec$	$TempExec5Falha$	$Variação$	$Makespan5Falha$	$ErroMakespan$
G_{152}	232.0	280.6	20.9%	201.9	39.0%
G_{252}	316.2	561.4	77.5%	477.9	17.4%
G_{377}	561.6	967.1	72.2%	898.3	7.67%
G_{527}	892.3	1430.4	60.3%	1325.2	7.94%
G_{702}	1285.7	2770.5	115.4%	2610.9	6.11%
G_{1034}	2253.9	5062.0	124.5%	4737.0	6.80%
G_{1829}	5153.9	11790.3	128.7%	11513.8	2.40%
G_{3002}	10493.4	25211.4	140.2%	24334.9	1.60%

Tabela 6.7: Sobrecarga do Mecanismo de Recuperação. Acréscimo do tempo de execução de 5 falhas em relação a execução sem falha considerando o mesmo escalonamento de tarefas

GADs	$TempExecEscBack$	$TempExec5Falha$	$taxaAcresc$
G_{152}	229.6	280.6	22.2%
G_{252}	490.7	561.4	14.3%
G_{377}	914.1	967.1	5.80%
G_{527}	1381.6	1430.4	3.53%
G_{702}	2696.7	2770.5	2.74%
G_{1034}	5017.7	5062.0	0.88%
G_{1829}	11741.6	11790.3	0.42%
G_{3002}	25098.1	25211.4	0.45%

Entretanto, nestes experimentos todos os grafos de G_n, mesmo com poucos processadores por *site* acham disponibilidade para escalonar as *backups*, tornando possível a solução de escalonamento com a execução sucedida da aplicação na presença de 5 falhas. Porém, é importante lembrar que cada falha simulada dentro de cada site, considera o processador mais confiável e com o número maior de tarefas, o que gera uma maior sobrecarga. Da mesma forma, as falhas acontecem logo no começo da execução representando casos piores de recuperação. Portanto para $m = 25$, sistema distribuído igualmente em *sites* de 5 processadores e considerando grandes aplicações, o desempenho do mecanismo proposto para 5 falhas é razoável. Estudos de casos não tão críticos, considerando outros momentos de falhas, por exemplo falhas na metade ou final da execução e os processadores com menor quantidade de tarefas, em geral devem produzir resultados melhores.

Em geral nas tabelas obtidas para a execução de Gauss no ambiente real CPH com múltiplas falhas, mostram a precisão dos modelos adotados em relação a execução no SGA. As sobrecargas do tratamento de falhas realizado pelo mecanismo, também alcançam valores pequenos, obtendo como esperado taxas levemente mais altas nas aplicações de menor tamanho. Á medida que aumenta-se a aplicação, estas percentagens diminuem, chegando a alcançar valores próximos de zero. Foi analisada também a variação do tempo

de execução para a solução S_e, obtendo, neste ambiente de $m = 25$ processadores, taxas muito pequenas para uma e duas falhas críticas de processador. Nestes casos, as taxas até atingem valores negativos, melhorando o tempo de execução final. Para um número maior de falhas críticas (cinco), para o cenário CPH com $m = 25$, e executando o GAD G_n de muitas dependências e computação heterogênea, já a taxa de variação aumenta. Porém os resultados são razoáveis considerando que o ambiente fica limitado a um número reduzido de processadores, e que foram consideradas falhas de processador para casos mais críticos.

6.3.4 Conclusões

Neste capítulo foi avaliada a estratégia de tolerância a falhas proposta neste trabalho de tese sobre um ambiente real. Para isto foi proposto e desenvolvido um mecanismo tolerante a falhas em MPI que utiliza o escalonamento estático proposto baseado na técnica de replicação passiva. Os dados gerados pelos algoritmos propostos FTMRCD ou mFTMRCD, para uma ou múltiplas falhas respectivamente, são lidos inicialmente e utilizados diretamente pelo mecanismo em MPI, para recuperar a aplicação em caso de falhas.

Para recuperar grandes aplicações sobre sistemas computacionais de maior escala, o mecanismo tolerante a falhas foi adicionado ao SGA EasyGrid da UFF, com a implementação de novas funcionalidades explorarando o gerenciamento hierárquico do *middleware*. A integração acrescentou ao SGA EasyGrid uma nova abordagem para tolerância a falhas com replicação passiva, e o mecanismo ganhou um funcionamento descentralizado e de maior escala. No SGA, o mecanismo pode ser ativado ou não em função das características da aplicação e do ambiente de execução distribuído. Desta forma, pode-se executar a aplicação considerando o objetivo, da confiabilidade além do tempo da execução, e ainda é possível a recuperação em caso de falhas.

Os experimentos realizados executando aplicações com o mecanismo implementado no SGA EasyGrid, mostraram os seguintes resultados. Primeiramente o simulador da solução de escalonamento, proposto em caso de falha no Capitulo 5, gera uma boa estimativa da execução real da aplicação na presença de falhas. Comparada à execução sem falha, a variação do tempo de execução para múltiplas falhas é razoável, chegando em muitos casos a melhorar o tempos de execução para a solução de equilíbrio proposta, e considerando-se que foram sempre simuladas falhas críticas. Por outro lado, um resultado importante foi que a sobrecarga de se usar o mecanismo de recuperação proposto no *middleware* SGA em caso de falhas é muito pequena, diminuindo com o tamanho da aplicação e alcançando valores próximos de zero para aplicações maiores.

Portanto, este capítulo mostra o emprego de técnicas de escalonamento estático com replicação passiva para recuperar aplicações com falhas em sistemas distribuídos, neste caso com o uso da biblioteca MPI no SGA EasyGrid. Desta forma, demostra-se a viabilidade da abordagem tolerante a falhas proposta neste trabalho para recuperar aplicações maiores sobre uma arquitetura real distribuída. Isto é importante porque muitos trabalhos correlatos não testam suas propostas em ambientes de execução reais, sendo muitas vezes inviáveis quando aplicadas nos mesmos.

Capítulo 7

Conclusões Finais e Trabalhos Futuros

Foi proposta uma abordagem de escalonamento bi-objetivo e tolerante a falhas que permite a execução efetiva de aplicações paralelas em sistemas distribuídos heterogêneos. A estratégia consegue oferecer confiabilidade e tolerância a falhas durante a execução das tarefas da aplicação.

Diferentemente da literatura, esta abordagem proposta se mostra mais flexível e eficiente, considerando a heterogeneidade destes ambientes, e também o aumento do número de recursos e quantidade de tarefas das aplicações. Uma função de custo ponderada minimiza tempo de execução e custos de confiabilidade, permitindo escolher múltiplas soluções de escalonamento para uma mesma aplicação e um mesmo ambiente de execução. A função proposta obtém escalonamentos com maior desempenho e dominância, comparada com propostas anteriores.

Neste trabalho foi apresentado um estudo comparativo de variação dos pesos, que ressalta a importância da escolha da função de custo e de uma abordagem de escalonamento flexível e eficiente sobre sistemas distribuídos com recursos heterogêneos. Para auxiliar o usuário, uma metodologia proposta para o ajuste dos pesos consegue classificar soluções de compromisso de maior qualidade, dentre múltiplas soluções de escalonamento possíveis. A metodologia combina informações adicionais no algoritmo e conceitos de dominância.

Através da execução simulada da aplicação em caso de falhas, verificou-se a tolerância de uma e múltiplas falhas de processador dos algoritmos de escalonamento propostos, além da minimização dos custos de tempo de execução e confiabilidade. Uma classificação diferente das tarefas e das falhas com uma formulação de critérios para o escalonamento de tarefas *backups*, garantiram a consistência e o desempenho da execução com falhas. Ao mesmo tempo, foi possível estabelecer compromissos entre os objetivos e o número de falhas, para ganhar maior flexibilidade no controle dos custos.

A execução de aplicações autonômicas tolerantes a falhas foi possível através da implementação de mecanismos de recuperação com replicação passiva no *middleware* SGA EasyGrid. Os mecanismos desenvolvidos consideram o escalonamento estático bi-objetivo e tolerante a falhas produzido. Os experimentos mostraram a viabilidade da estratégia

proposta, com a recuperação eficiente de múltiplas falhas no ambiente real distribuído da nossa instituição. Os experimentos geraram resultados razoáveis considerando a ocorrência de múltiplas falhas críticas em piores cenários de execução. A sobrecarga gerada pelos mecanismos de recuperação propostos para manipular as falhas, com o emprego de técnicas de replicação passiva e escalonamento estático, foi mínima com valores próximos de zero. Nestes experimentos também foi possível avaliar a precisão e robustez dos modelos adotados, junto a estratégia de escalonamento proposta.

7.1 Propostas Futuras

O escalonamento estático de tarefas da aplicação, diferente do escalonamento dinâmico garante melhor planejamento e previsão dos custos antes de executar a aplicação, mesmo com a ocorrência de falhas. Decisões podem ser tomadas com um tempo maior, empregando mais técnicas de escalonamento que garantem maior desempenho e consistência da execução. O escalonamento dinâmico não têm uma visão global da execução, de maneira que uma decisão em determinado momento pode prejudicar decisões futuras. Contudo, o escalonamento estático não pode contemplar as variações no sistema, sendo inevitável a tomada de novas decisões durante a execução da aplicação. Assim, a estratégia de escalonamento estático proposta neste trabalho pode ser combinada com uma estratégia de escalonamento dinâmico para explorar as vantagens de ambas abordagens.

A metodologia e classificação propostas para o ajuste dos pesos, podem ser aplicadas em outros algoritmos de escalonamento da literatura, com o emprego de um método de normalização e informações adicionais semelhantes ao deste trabalho. Um estudo maior dos pesos assim como das soluções de compromisso pode ser realizado em relação ao escalonamento de *backups*, de forma que aproveite a flexibilidade da função de custo com o emprego da técnica de replicação passiva.

A estratégia de escalonamento proposta pode também ser aplicada a outras plataformas distribuídas de execução diferentes de MPI, que considerem os modelos adotados neste trabalho. Em particular, para ambientes *multicores*, o modelo da arquitetura pode ser estendido. Nos modelos adotados, outros tipos de falhas como falhas transientes e de canais de comunicação, podem ser considerados. Neste sentido novos estudos podem ser realizados.

Referências Bibliográficas

[1] Ahmad, I., Kwok, Y.-K. On exploiting task duplication in parallel program scheduling. *IEEE Transactions Parallel Distributed Systems 9*, 9 (1998), 872–892.

[2] Alan Girault, Hamoudi Kalla, M. S. An algorithm for automatically obtaining distributed and fault-tolerant static schedules. In *International Conference on Dependable Systems and Networks, 2003. DSN 2003.* (2003).

[3] Allen, G., Dramlitsch, T., Foster, I., Karonis, N. T., Ripeanu, M., Seidel, E., Toonen, B. Supporting efficient execution in heterogeneous distributed computing environments with cactus and globus. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing (CDROM)* (New York, NY, USA, 2001), ACM Press, p. 52–52.

[4] Amin, A., Ammar, R., Rajasekaran, S. Maximizing reliability while scheduling real-time task-graphs on a cluster of computers. *Proceedings of the 10th IEEE Symposium on Computers and Communications (ISCC)* (2005), 1001–1006.

[5] Assayad, I., Girault, A., Kalla, H. A bi-criteria scheduling heuristic for distributed embedded systems under reliability and real-time constraints. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks* (Washington, DC, USA, 2004), IEEE Computer Society, p. 347.

[6] Baker, M., Buyya, R., Laforenza, D. Grids and grid technologies for wide-area distributed computing. *Software Practice Experience 32*, 15 (2002), 1437–1466.

[7] Barnard, S., Biswas, R., Saini, S., der Wijngaart, R. V., Yarrow, M., Zechtzer, L., Foster, I., Larsson, O. Large-scale distributed computational fluid dynamics on the information power grid using globus. In *FRONTIERS '99: Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation* (1999), IEEE Computer Society, p. 60.

[8] Benoit, A., Hakem, M., Robert, Y. Fault tolerant scheduling of precedence task graphs on heterogeneous platforms. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on* (Miami, Florida, USA, April 2008), p. 14–18.

[9] Benoit, A., Hakem, M., Robert, Y. Realistic models and efficient algorithms for fault tolerant scheduling on heterogeneous platforms. In *Parallel Processing, 2008. ICPP '08. 37th International Conference on* (Portland, Oregon, USA, Sept. 2008), p. 8–12.

[10] Boeres, C., Lima, A., Rebello, V. Hybrid task scheduling: Integrating static and dynamic heuristics. In *Proceedings of the 15th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'03)* (São Paulo- Brazil, November 2003), IEEE Computer Society, p. 199–206.

[11] Boeres, C., Nascimento, A. P., Rebello, V. E. F., Sena, A. C. Efficient hierarchical self-scheduling for MPI applications executing in computational grids. In *MGC '05: Proceedings of the 3rd international workshop on Middleware for grid computing* (New York, NY, USA, 2005), ACM, p. 1–6.

[12] Boeres, C., Rebello, V. On solving the static task scheduling problem for real machines. In *Models for Parallel and Distributed Computation: Theory, Algorithmic Techniques and Applications*, R. Correa, I. Dutra, M. Fiallos, and F. Gomes, Eds. Kluwer Academic Publishers, 2002, ch. 3, p. 53–84. Applied Optimization Series.

[13] Boeres, C., Rebello, V., Skillicorn, D. Static scheduling using task replication for LogP and BSP models. In *The Proceedings of the 4th International Euro-Par Conference on Parallel Processing (Euro-Par'98)* (Southampton, UK, September 1998), D. Pritchard and J. Reeve, Eds., LNCS 1470, Springer, p. 337–346.

[14] Boeres, C., Rebello, V. E. F. Easygrid: towards a framework for the automatic grid enabling of legacy MPI applications: Research articles. *Concurrency And Computation : Practice And Experience 16*, 5 (2004), 425–432.

[15] Canon, L.-C., Jeannot, E. Scheduling strategies for the bicriteria optimization of the robustness and makespan. In *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS08), in (NIDISC'08)* (April 2008), p. 1–8.

[16] Chanchio, K. *Efficient Checkpointing for Heterogeneus Collaborative Environments, representations, coordination, and automation.* PhD thesis, Faculty of the Lousiana 9, December 2000.

[17] Chen, T., Chang, C., Sheu, J., Yu, G. Fault tolerant model for replication in distributed file systems. *National Science Council, Republic of China, Part A: Physical Science and Engineering 23*, 3 (1999), 402–4102.

[18] Cohon, J. L. *Multiobjective Programming and Planning.* Academic Press, 1978.

[19] Cosnard, M., Trystram, D. *Parallel Algorithms and Architectures.* Thomson Learning, 1994.

[20] da Silva, J. A. *Tolerância a Falhas para Aplicações Autônomas em Grades Computacionais.* PhD thesis, Instituto de Computação, Universidade Federal Fluminense, Niterói, RJ, Brasil, Junho 2010.

[21] da Silva Martins Jr., A., Gonçalves, R. A. L. Extensões na LAM/MPI para automatizar o checkpoint e tolerar falhas em cluster de computadores. In *VI WSCAD 2005* (Rio de Janeiro, 2005).

[22] de Campos Vianna, D. Q. Um sistema de gerenciamento de aplicações MPI para ambientes grid. Dissertação de Mestrado, Instituto de Computação, Universidade Federal Fluminense, 2005.

[23] de P. Nascimento, A., da C. Sena, A., da Silva, J. A., de C. Vianna, D. Q., Boeres, C., Rebello, V. E. F. Managing the execution of large scale MPI applications on computational grids. *17th. International Symposium on Computer Architecture and High Performance Computing* (October 2005).

[24] Deb, K. *Multi-Objetive Optimization using Evolutionary Algorithms.* Wiley and Sons, 2001.

[25] Dima, C., Girault, A., Lavarenne, C., Sorel, Y. Off-line real-time fault-tolerant scheduling. In *Euromicro Workshop on Parallel and Distributed Processing* (Mantova, Italy, February 2001), p. 410–417.

[26] Dogan, A., Ozguner, F. Matching and scheduling algorithms for minimizing execution time and failure probability of applications in heterogeneous computing. *IEEE Transactions Parallel Distributed Systems 13*, 3 (2002), 308–323.

[27] Dogan, A., Ozguner, F. Bi-objective scheduling algorithms for execution time-reliability trade-off in heterogeneous computing systems. *Jornal of Computing 48*, 3 (2005), 300–314.

[28] Dongarra, J., Jeannot, E., Saule, E., Shi, Z. Bi-objective scheduling algorithms for optimizing makespan and reliability on heterogeneous systems. In *Proc. 19th Annual ACM Symp. on Parallelism in Algorithms and Architectures (SPAA '07)* (2007), ACM Press.

[29] Elnozahy, E. N. M., Alvisi, L., Wang, Y.-M., Johnson, D. B. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys 34*, 3 (2002), 375–408.

[30] Frey, J., Tannenbaum, T., Livny, M., Foster, I., Tuecke, S. Condor-g: A computation management agent for multi-institutional grids. *Cluster Computing 5*, 3 (2002), 237–246.

[31] Gal, T., Hanne, T., Stewart, T., Eds. *Multicriteria Decision Making: Advances in MCDM Models, Algorithms, Theory, and Applications.* Kluwer Academic, 1999.

[32] Garey, M. R., Johnson, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1979.

[33] Ghosh, S., Melhem, R., Mosse;, D. Fault-tolerance through scheduling of aperiodic tasks in hard real-time multiprocessor systems. *IEEE Trans. Parallel Distrib. Syst. 8*, 3 (1997), 272–284.

[34] Girault, A., Saule, i., Trystram, D. Reliability versus performance for critical applications. *Jornal of Parallel and Distributed Computing 69*, 3 (2009), 326–336.

[35] Gonzalez, O., Shrikumar, H., Stankovic, J. A., Ramamritham, K. Adaptive fault tolerance and graceful degradation under dynamic hard real-time scheduling. In *RTSS '97: Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS '97)* (1997), IEEE Computer Society, p. 79.

[36] Grimshaw, A., Wulf, W. Legion-a view from 50,000 feet. *High Performance Distributed Computing (hpdc).*

[37] Guerraoui, R., Schiper, A. Fault-tolerance by replication in distributed systems. In *Ada-Europe '96: Proceedings of the 1996 Ada-Europe International Conference on Reliable Software Technologies* (London, UK, 1996), Springer-Verlag, p. 38–57.

[38] Hagras, T., necek., J. A high performance, low complexity algorithm for compile-time task scheduling in heterogeneous systems. In *18th International Parallel and Distributed Processing Symposium (IPDPS'04)- Workshop 1* (2004).

[39] Hakem, M., Butelle, F. Reliability and scheduling on systems subject to failures. In *Proceedings of the International Conference on Parallel Processing (ICPP)* (2007), p. 38.

[40] Hwang, J.-J., Chow, Y.-C., Anger, F. D., Lee, C.-Y. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal on Computing 18*, 2 (1989), 244–257.

[41] Jeannot, E., Saule, E., Trystram, D. Bi-objective approximation scheme for makespan and reliability optimization on uniform parallel machines. In *Proceedings of the 14th international Euro-Par conference on Parallel Processing* (Berlin, Heidelberg, 2008), Springer-Verlag, p. 877–886.

[42] Johnston, W. E. Using computing and data grids for large-scale science and engineering. *Jornal of High Performance Computing Applications 15*, 3 (2001), 223–242.

[43] Kee, Y.-S., Casanova, H., Chien, A. A. Realistic modeling and synthesis of resources for computational grids. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing* (Washington, DC, USA, 2004), IEEE Computer Society, p. 54.

[44] Kwok, Y.-K. Fault-tolerant parallel scheduling of tasks on a heterogeneous high-performance workstation cluster. *Journal Supercomputing 19*, 3 (2001), 299–314.

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

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

[47] Li, D., Iwahori, Y., Ishii, N. Exploiting heterogeneous parallelism in the presence of communication delays. In *ICS '98: Proceedings of the 12th international conference on Supercomputing* (New York, NY, USA, 1998), ACM Press, p. 157–164.

[48] Liberato, F., Melhem, R., Mosse;, D. Tolerance to multiple transient faults for aperiodic tasks in hard real-time systems. *IEEE Transactions Computing 49*, 9 (2000), 906–914.

[49] Macey, B. A performance evaluation of CP list scheduling heuristics for communication intensive task graphs. In *IPPS '98: Proceedings of the 12th. International Parallel Processing Symposium* (Washington, DC, USA, 1998), IEEE Computer Society, p. 538.

[50] Maehle, E., Markus, F. J. Fault-tolerant dynamic task scheduling based on data-flow graphs. In *IEEE Work-shop on Fault-Tolerant Parallel and Distributed Systems* (Geneva, Switzerland, April 1997).

[51] Mendes, H. A. HLogP: Um modelo de escalonamento para a execução de aplicações MPI em grades computacionais.

[52] Mendes, H. A. HLogP: Um modelo de escalonamento para a execução de aplicações MPI em grades computacionais. Dissertação de Mestrado, Instituto de Computação, Universidade Federal Fluminense, 2004.

[53] Naedele, M. Fault-tolerant real-time scheduling under execution time constraints. In *RTCSA '99: Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications* (Washington, DC, USA, 1999), IEEE Computer Society, p. 392.

[54] Nascimento, A. P. *Escalonamento Dinâmico para Aplicações Autonômicas MPI em Grades Computacionais*. PhD thesis, Instituto de Computação, Universidade Federal Fluminense, Niterói, Brazil, Maio 2008.

[55] Nascimento, A. P., Sena, A. C., Boeres, C., Rebello, V. E. F. Distributed and dynamic self-scheduling of parallel MPI grid applications. *Concurrency and Computation: Practice and Experience 19*, 14 (2007), 1955–1974.

[56] Nascimento, A. P., Sena, A. C., Silva, J. A., Vianna, D. Q. C., Boeres, C., Rebello, V. E. F. Autonomic application management for large scale MPI programs. *International Journal of High Performance Computing and Networking 5*, 4 (2008), 227–240.

[57] Papadimitriou, C., Yannakakis, M. Towards and archicteture independent analysis of parallel algorithms. *SIAM Journal on Computing 19* (1990), 322–328.

[58] Pearlman, L., Kesselman, C., Gullapalli, S., B. F. Spencer, J., Futrelle, J., Ricker, K., Foster, I., Hubbard, P., Severance, C. Distributed hybrid earthquake engineering experiments: Experiences with a ground-shaking grid application. In *HPDC '04: Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC'04)* (Washington, DC, USA, 2004), IEEE Computer Society, p. 14–23.

[59] Qin, X., Jiang, H. A dynamic and reliability-driven scheduling algorithm for parallel real-time jobs on heterogeneous clusters. *Journal of Parallel and Distributed Computing 65* (2005), 885–900.

[60] Qin, X., Jiang, H. A novel fault-tolerant scheduling algorithm for precedence constrained tasks in real-time heterogeneous systems. *Parallel Computing 32*, 5 (2006), 331–356.

[61] Qin, X., Jiang, H., Swanson, D. R. An efficient fault-tolerant scheduling algorithm for real-time tasks with precedence constraints in heterogeneous systems. In *Proceeding International Conference on Parallel Processing (ICPP'02)* (2002), IEEE Computer Soc., p. 360.

[62] Qin, X., Jiang, H., Xie, C., Han, Z. Reliability driven scheduling for real-time tasks with precedence constraints in heterogeneous distributed systems. In *In Proceedings of the 12 th International Conference Parallel and Distributed Computing and Systems 2000* (November 2000), p. 617–623.

[63] Reed, D. A., da Lu, C., Mendes, C. L. Reliability challenges in large systems. *Future Generation Computer Systems 22*, 3 (2006), 293–302.

[64] Sardina, I. M., Boeres, C., Drummond, L. M. A. Escalonamento tolerante a falhas na recuperacao de aplicacoes em MPI. In *24 Simposio Brasileiro de Redes de Computadores. IV Workshop de Grids Computacionais e Aplicações* (Curitiba, PR, maio-junho 2006), p. 27–38.

[65] Sardina, I. M., Boeres, C., Drummond, L. M. A. An efficient weighted bi-objective scheduling algorithm for heterogeneous systems. In *The seventh International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms (HeteroPar 2009)* (Delft, 2009), Lectures Notes in Computer Science. New York: Springer-Verlag, p. 1–10.

[66] Sardina, I. M., Boeres, C., Drummond, L. M. A. Escalonamento bi-objetivo de aplicações paralelas em recursos heterogêneos. In *27 Simposio Brasileiro de Redes de Computadores e Sistemas Distribuídos* (Recife, maio 2009), p. 467–480.

[67] Sena, A. C., Nascimento, A. P., Silva, J. A., Vianna, D. Q. C., Boeres, C., Rebello, V. E. F. On the advantages of an alternative MPI execution model for grids. In *Proceedings of the 7th IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2007), 14-17 May 2007, Rio de Janeiro, Brazil* (Los Alamitos, CA, USA, 2007), IEEE Computer Society, p. 575–582.

[68] Silva, J., e L.M.A. Drummond, A. S. STATS: Uma ferramenta para escalonamento estático de tarefas em programas MPI. Dissertação de Mestrado, Instituto de Computação, Universidade Federal Fluminense, 2002.

[69] Silva, J. A., Rebello, V. E. F. Low cost self-healing in MPI applications. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 14th European PVM/MPI User's Group Meeting, Paris, France, September 30 - October 3, 2007, Proceedings* (2007), Franck Cappello and Thomas Hérault and Jack Dongarra, Ed., vol. 4757 of *Lecture Notes in Computer Science*, Springer, p. 144–152.

[70] Sinnen, O., Ed. *Task Scheduling for Parallel Systems.* John Wiley & Sons, 2007.

[71] Sinnen, O., Sousa, L. List scheduling: Extension for contention awareness and evaluation of node priorities for heterogeneous cluster architectures. *Parallel Computing 30*, 1 (janeiro de 2004), 81–101.

[72] Snir, M., Otto, S. W., Walker, D. W., Dongarra, J., Huss-Lederman, S. *MPI: The Complete Reference.* MIT Press, Cambridge, MA, USA, 1995.

[73] Souto, H., da Silveira Filho, O., Moyne, C., Didierjean, S. Thermal dispersion in porous media: Computations by the random walk method. *Journal of Computational and Applied Mathematics 21*, 2 (2002), 513–544.

[74] Srinivasan, S., Jha, N. K. Safety and reliability driven task allocation in distributed systems. *IEEE Transactions Parallel Distributed Systems 10*, 3 (1999), 238–251.

[75] Topcuouglu, H., Hariri, S., Wu, M. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions Parallel Distributed Systems 13*, 3 (2002), 260–274.

[76] Treaster, M. A survey of fault-tolerance and fault-recovery techniques in parallel systems. *ACM Computing Research Repository (CoRR 501002* (2005), 1–11.

[77] von Laszewski, G., Westbrook, M., Foster, I., Westbrook, E., Barnes, C. Using computational grid capabilities to enhance the ability of an x-ray source for structural biology. *Cluster Computing 3*, 3 (2000), 187–199.

[78] Weissman, J. B. Fault tolerant wide-area parallel computing. In *IPDPS '00: Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing* (London, UK, 2000), Springer-Verlag, p. 1214–1225.

[79] Wrzesinska, G., van Nieuwpoort, R. V., Maassen, J., Kielmann, T., Bal, H. E. Fault-tolerant scheduling of fine-grained tasks in grid environments. *The International Journal of High Performance Computing Applications 20*, 1 (2006), 103–114.

[80] Zeleny, M. *Multiple Criteria Decision Making*. MacGraw-Hill, 1982.

[81] Ziv, A., Bruck, J. *Checkpointing in Parallel and Distributed Systems*. Parallel and Distributed Computing Handbook. McGraw-Hill, 1996, chapter 10, p. 274–302.