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

Idalmis Milián Sardiña[1], **Cristina Boeres**[1], **Lúcia Drummond**[1]

[1]Instituto de Computação – Universidade Federal Fluminense (IC-UFF)
Rua Passo da Pátria, 156 - São Domingos - 24.210-240 - Niterói - RJ-Brasil

{isardina, boeres, lucia}@ic.uff.br

Abstract. *In large-scale distributed systems, the occurrence of faults is a problem that should be tackled. This work proposes a fault-tolerant bi-objective static scheduling strategy, in order to establish a desirable execution time, a high reliability and fault tolerance. The strategy explores a passive replication technique for a application scheduling to tolerate faults. To ensure flexibility and consistency, the proposed algorithms introduce rules for the scheduling of backups tasks, based on a weighted cost function. The results emphasize the importance of using flexible scheduling approaches with fault tolerance in heterogeneous and distributed environments.*

Resumo. *Em sistemas distribuídos, a propensão à ocorrência de falhas é um problema que deve ser atacado. Este trabalho propõe uma estratégia de escalonamento estático bi-objetivo e tolerante a falhas, que tem como finalidade permitir uma execução eficiente e confiável de uma aplicação mesmo na presença de falha. Na estratégia explora-se uma técnica de replicação passiva para escalonar aplicações com tolerância a falha. Para garantir flexibilidade e consistência, o algoritmo introduz critérios para o escalonamento de tarefas backups, considerando uma função de custo ponderada. Os resultados destacam a importância de usar novas abordagens de escalonamento flexíveis com tolerância a falha em ambientes distribuídos com recursos heterogêneos.*

1. Introdução

Em plataformas heterogêneas de larga escala, falhas de recursos podem ocorrer inviabilizando a execução das aplicações. Consequentemente, existe uma grande necessidade de desenvolver técnicas para alcançar tolerância a falha. Muitas heurísticas de escalonamento utilizadas como [Topcuouglu et al. 2002], empregam modelos onde aspectos sobre confiabilidade e tolerância a falha não são considerados. Diferentes trabalhos abordam confiabilidade mas não toleram falhas. [Sardina et al. 2009] por exemplo, propõe um escalonamento bi-objetivo ponderado que integra a minimização do *makespan* e a maximização da confiabilidade, classificando as soluções de compromisso produzidas.

Diversos algoritmos de escalonamento que consideram tolerância a falha, empregam replicação de tarefas baseadas no esquema primária-*backup* [Benoit et al. 2008, Qin and Jiang 2006, Alan Girault 2003, Liberato et al. 2000, Naedele 1999]. Para a técnica de replicação ativa [Benoit et al. 2008, Alan Girault 2003] 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 [Liberato et al. 2000, Naedele 1999, Qin and Jiang 2006], a *backup* de uma tarefa somente é ativada quando detectada falha na primária usando mecanismos. A mesma tem se mostrado na literatura uma alternativa interessante à replicação ativa, dado que não necesita de uso de recursos extras, com custos de execução de *backups* mais reduzidos.

[Naedele 1999, Liberato et al. 2000] utilizam primária-*backup* para escalonamento dinâmico. Já trabalhos com escalonamento estático para este esquema, em muitos casos são projetados para tempo real como [Qin and Jiang 2006], ou utilizam replicação ativa como [Alan Girault 2003, Benoit et al. 2008]. Em [Qin and Jiang 2006], aplicações com restrições de precedência são escalonadas em um ambiente que considera a heterogeneidade de recursos de processamento, da comunicação e da confiabilidade. Para tolerar uma falha permanente de processador (*crash*) e melhorar a qualidade do escalonamento, a heurística usa sobreposição de *backups*. A função de custo não é ponderada e prioriza a confiabilidade, só em caso de empate considera o tempo de execução.

Neste artigo é proposto um algoritmo de escalonamento estático, inicialmente baseado em [Qin and Jiang 2006] ao empregar uma heurística do tipo *list scheduling* para escalonar as tarefas primárias e *backups*, com tolerância de uma falha *crash*. A nova abordagem adiciona o escalonamento bi-objetivo de [Sardina et al. 2009, Boeres et al. 2010]. Este algoritmo não é projetado para sistemas de tempo real e acrescenta maior flexibilidade com a introdução de novos conceitos e critérios para o escalonamento de *backups*.

2. Modelo do Sistema

Neste trabalho as aplicações são modeladas por *Grafos Acíclicos Direcionados* (GAD), com $G = (V, E, e, c)$, onde V é o conjunto de vértices (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). 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 $Succ(v_j)$ são os sucessores imediatos de v_j.

Entre os GADs utilizados neste trabalho, G_n é uma paralelização do método Eliminação de Gauss para solucionar sistemas de equações lineares. A estrutura de G_n possui pesos de computação variável, uma vez que são maiores na parte superior do GAD e diminuem a cada nível. G_n serve como modelo para estudar aplicações heterogêneas em relação as tarefas. O segundo GAD é o diamante Di_n, que paraleliza a multiplicação de matrizes, e caracterizado por ter uma estrutura regular e homogêneo em relação as tarefas.

Em relação a arquitetura, $P = \{p_0, p_1, ..., p_{m-1}\}$ é o conjunto de m processadores heterogêneos, sendo que a cada p_j é associado o índice de retardo (*computational slowdown index*), denotado por $csi(p_j)$, sendo esta métrica inversamente proporcional ao poder computacional de p_j. 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).

São consideradas falhas permanentes de processador, eventos independentes entre si, e que ocorrem de acordo com uma distribuição de *Poisson* com probabilidade de falha

$FP(p_j)\ \forall\ p_j \in P$ e valor constante, conforme [Qin and Jiang 2006]. $FP(p_j)$ representa a quantidade de falhas por unidade de tempo que podem ocorrer em p_j. O custo de confiabilidade de execução de v em p_j é definido como $RC(v, p_j) = FP(p_j) \times eh(v, p_j)$ e deve ser minimizado. Sendo $ltask(p_j)$ a lista de tarefas atribuídas a p_j, o custo associado à p_j é $RC_p(p_j) = \sum_{\forall v \in ltask(p_j)} RC(v, p_j)$. Para um sistema P com m processadores, o custo de confiabilidade de escalonar uma aplicação pode ser definido como $RC(G, P) = \sum_{\forall p_j \in P} RC_p(p_j)$. Assim, a confiabilidade da aplicação G é dada por $R = e^{-RC(G,P)}$.

3. Estratégia de Escalonamento Proposta

O escalonamento proposto de forma geral apresenta os seguintes objetivos: encontrar uma alocação para as primárias das tarefas da aplicação sobre a arquitetura do modelo proposto; de acordo com a alocação das primárias, encontrar uma alocação para as *backups* que permita tolerar uma falha permanente de processador; e minimizar o *makespan* e maximizar a confiabilidade da aplicação mesmo na presença de falha. O algoritmo estático tolerante a falha proposto é chamado *Fault Tolerant Makespan and Reliability Cost Driven* (FTMRCD), representando a última etapa do Algoritmo 1:*FTframework*.

Algoritmo 1 :*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 tolerar uma falha *crash*. Inicialmente as tarefas são ordenadas em V_{ordG} e de acordo com a prioridade de $blevel()$ conforme [Topcuouglu et al. 2002]. O escalonamento de primárias Sch é feito pelo algoritmo MRCD(V_{ordG},P,w_1) ([Sardina et al. 2009]), onde w_1 é o parâmetro de ponderação da função de custo e $S(Sch) = (\mathcal{M}, R_T)$ a solução de Sch. FTMRCD(V_{ordG},P,Sch,w_2) escalona as *backups* de acordo com Sch e utiliza uma heurística do tipo *list scheduling* similar a MRCD, sendo w_2 o parâmetro para as *backups*. FTMRCD utiliza critérios para tolerância a falha definidos a seguir e gera como saída o escalonamento $SchBck$ e a lista de tarefas $LSucFalha$, informações que serão utilizadas para recuperar a aplicação.

3.1. Critérios para Escalonar as Tarefas *Backups*

Para alcançar a tolerância a falha com FTMRCD, é definida uma série de critérios que permitem escalonar as *backup*s da aplicação, de forma a garantir consistência e o bom desempenho. As cópias primária e *backup* de v_i são denotadas como v_i^P e v_i^B, respectivamente e a precedência entre 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 [Qin and Jiang 2006] para classificar a tarefa primária v^P durante o escalonamento das *backups*. A 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 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 recebe as mensagens de v_j^B.

Por definição, uma primária forte sempre poderá executar, exceto quando seu processador falha sem ter finalizado sua execução. Assim, a *backup* correspondente deve ser ativada para executar. Note que esta classificação limita-se a uma mesma classificação de 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, esta classificação é reformulada e estendida com a adição de novos conceitos.

3.1.1. Classificação de Primárias: Critério C_1

Para classificar v^P é definido o critério C_1, que utiliza as relações R_1 e R_2 entre duas tarefas primárias, introduzidas a seguir. A aplicação de C_1 deve ser realizada a cada iteração de FTMRCD, onde as tarefas do GAD G são visitadas de acordo com a ordem previamente estabelecida em V_{ordG}. A classificação usando C_1 percorre V_{ordG}, cumprindo a ordem de precedência das tarefas, e assim, no caso de v_i^P ter predecessores, só poderá ser classificada com C_1 depois de serem classificados todos seus predecessores v_j^P.

Relação R_1: Dada a precedência $v_j^P \rightarrow v_i^P$, 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 recebe 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 recebe a mensagem de v_j^B.

Pela relação R_1, se 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 $v_j^P \rightarrow v_i^P$, 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.

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, na Figura 1 são apresentados o GAD $G = (V, E)$ e o seu escalonamento, onde v_1^P e v_2^P são *totalmente fortes* e v_3^P é *parcialmente forte*. 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) de 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 *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 início de v_3^P.

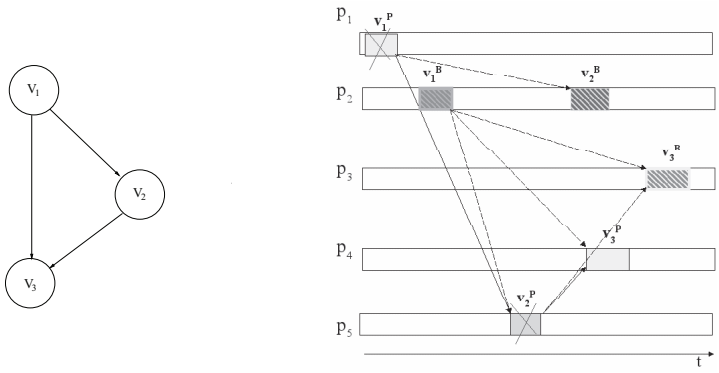

Figura 1. Critério de classificação C_1 aplicado no GAD

Uma falha de processador provoca a falha das primárias escalonadas nele que ainda não executaram, e pode também provocar a falha de execução de primárias sucessoras escalonadas em outros processadores. Então, para cada v_i^P são definidas: a ***falha por hardware***, provocada pela falha de seu processador $p(v_i^P)$ e; a ***falha induzida***, 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. No caso da *falha induzida*, v_i^P é *fraca para* v_j^P, logo por C_1 v_i^P é *parcialmente forte* ou *totalmente fraca*. Dada $v_j^P \rightarrow v_i^P$ 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 $v_k^P \rightarrow v_j^P \rightarrow v_i^P$, 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. Esses predecessores v_k^P são denotados *candidatos a provocar a falha induzida de* v_i^P.

Quando $p(v_i^P)$ falha, se v_i^P não executa completamente, então v_i^P sofre *falha por hardware*. Assim, qualquer primária 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 são totalmente fortes então são *candidatas a falha induzida*.

3.1.2. Exclusão Mútua: Critérios de Escalonamento C2 e C3

Os critérios C_2 e C_3 são formulados a seguir, baseados nas restrições de exclusão espacial e temporal do esquema primária-*backup*. Eles garantem o funcionamento do mecanismo de tolerância a falhas e são utilizados por algoritmos de escalonamento

tolerantes a falhas que usam replicação passiva [Liberato et al. 2000, Naedele 1999, Qin and Jiang 2006]. Com as definições de tempo de ínicio $EST(v_i, p(v_i))$ e tempo de fim $EFT(v_i, p(v_i))$, sendo que $p(v_i)$ tem escalonada v_i, a *backup* de v_i deve 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 $p(v_i^B) \neq p(v_i^P)$.

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, $EST(v_i^B, p(v_i^P)) \geq EFT(v_i^P, p(v_i^P)) + tDetFalha(p(v_i^P))$, onde $tDetFalha(p(v_i^P))$ é o tempo de detecção.

C_2 garante que, no caso de uma falha de processador, a primária ou *backup* de v_i possa executar. Em caso de falhar $p(v_i^P)$ antes de v_i^P finalizar, v_i^B correspondente será ativada em outro processador $p(v_i^B)$, em um momento mais tarde $EST(v_i^B)$.

3.1.3. Seleção de Processadores: Critérios de Escalonamento C4 e C5

Os critérios propostos de C_4 a C_7, são definidos baseados em C_1 e permitem selecionar os processadores 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.

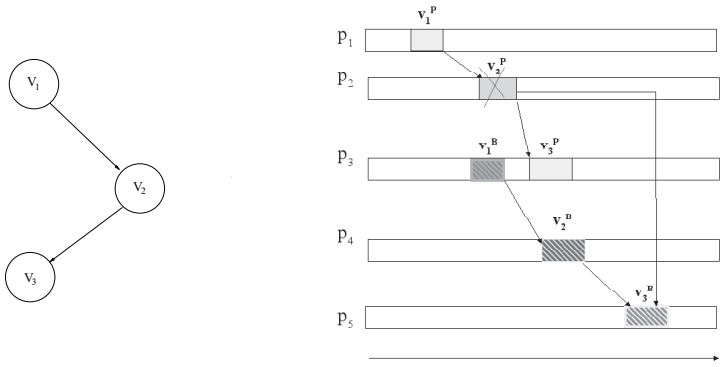

Figura 2. Critérios C_4 e C_5 aplicados no GAD

O critério C_4.2 mostra 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 execute no lugar de v_i^P, portanto v_i^B não pode ser escalonada em $p(v_j^P)$. Na Figura 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 (C_4.2). Se p_2 ($p(v_2^P)$) falhar, é necessário que v_2^B execute. v_3^B não pode ser escalonada no mesmo processador p_2, já que em caso de falha de p_2, v_3^B deve 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 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.

A classificação C_1 em C_4 e C_5, oferece flexibilidade e consistência ao definir as opções de escalonamento de uma *backup* sobre os diferentes processadores. A *backup* de v_i sendo escalonada, deixa de ser alocada apenas nos processadores dos predecessores aos quais ela é *fraca* como em [Qin and Jiang 2006], e os processadores dos outros predecessores que ela é *forte* podem ser utilizados. Portanto, C_4 e C_5 devem ser aplicados senão podem vir a ocorrer falhas induzidas durante a execução da aplicação.

3.1.4. Sobreposição de *Backups*: Critérios de Escalonamento C6 e C7

Os critérios C_6 e C_7 estabelecem em FTMRCD as regras para a 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.

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. Da mesma forma, $C_6.3$ 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.

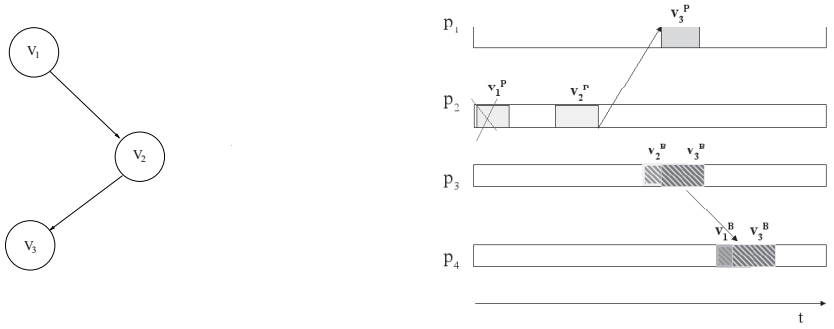

Figura 3. Critério C_6 aplicado no GAD

Na Figura 3, v_3^B não pode ser escalonada em p_3 sobre v_2^B por C_6.2, pois existe relação de precedência entre v_2^P e v_3^P, v_3^P é *fraca para* v_2^P e então, em caso de falha de $p_2 = p(v_2^P)$, 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, 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*).

Com C_7.1, v_i^B pode sobrepor a sucessora 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. Se v_i^P falhar durante sua execução, provocará a falha induzida de v_k^P e consequentemente, 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, v_i^B de uma primária totalmente forte pode ser escalonada sobre uma v_k^P parcialmente forte ou fraca, se v_l^P, candidata a provocar a falha induzida de v_k^P, está escalonada 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. Isto pode ser ilustrado com a Figura 4, v_1^B pode ser escalonada sobre 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.

Em [Qin and Jiang 2006] a sobreposição de *backups* sobre primárias foi abordada, mas somente para tarefas com relação de precedência. Diferentemente, neste trabalho

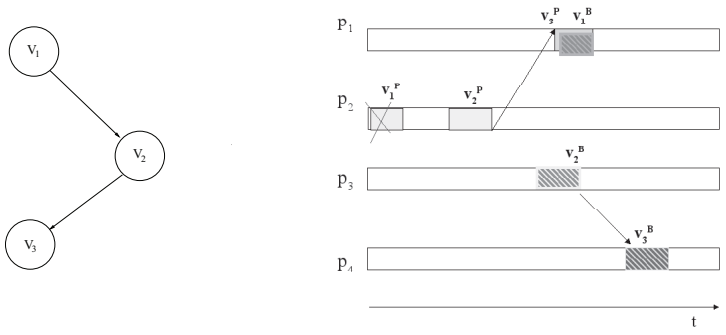

Figura 4. Critério C_7 aplicado no GAD

C_7 é proposto também para primárias sem relação de precedência em G, o que permite melhorar ainda mais o desempenho da execução em caso de falha.

3.2. Algoritmo de Escalonamento para Tolerar uma Falha

O algoritmo FTMRCD de forma geral escalona as *backups* seguindo a estrutura de MRCD ([Sardina et al. 2009]), mas com a adição dos critérios definidos anteriormente.

Algoritmo 2 : *FTMRCD(V_{ordG},P,w,Sch)*

```
1   for i = 0, ..., n − 1
2       F = ∞;
3       ⟨TForte, Forte, LPredFalha, LSucFalha⟩ = ClasPrimaria(v_i^P); /*C1*/
4       P* = AchaCjtoProcs(v_i^B, P, LPredFalha); /*C2, C4, C5*/
5       ∀p_j ∈ P*
6           EAT(v_i^B, p_j) = max{EATP(Pred(v_i)), EATB(PredB(v_i)),
            EFT(v_i^P, p_j)} /*C3*/
7           LNSob = CriaLNaoSobreposta(v_i^B, p_j); /*C6, C7 */
8           EST(v_i^B, p_j) = InsTarefas(v_i^B, p_j, LNSob, EAT);
9           EFT(v_i^B, p_j) = EST(v_i^B, p_j) + eh(v_i^B, p_j);
10          RC(v_i^B, p_j) = FP(p_j) × eh(v_i^B, p_j);
11      EFT_min = min_{p_j∈P}{EFT(v_i^B, p_j)}; EFT_max = max_{p_j∈P}{EFT(v_i^B, p_j)};
12      RC_min = min_{p_j∈P}{RC(v_i^B, p_j)}; RC_max = max_{p_j∈P}{RC(v_i^B, p_j)};
13      ∀p_j ∈ P*
14          EFT_n(v_i^B, p_j) = norm(0, 100, EFT_min, EFT_max, EFT(v_i^B, p_j));
15          RC_n(v_i^B, p_j) = norm(0, 100, EFT_min, RC_max, RC(v_i^B, p_j));
16          f(v_i^B, p_j) = (1 − w) × EFT_n(v_i^B, p_j) + w × RC_n(v_i^B, p_j);
17          if (f(v_i^B, p_j) < F)
18              F = f(v_i^B, p_j); p_{v_i^B} = p_j;
19      if (P* ≠ ∅)
20          SchBck = SchBck ∪ ⟨v_i^B, p_{v_i^B}, EST(v_i^B, p_{v_i^B})⟩;
21      else
22          SchBck = ∅; LSucFalha = ∅; sair();
23      retorna(SchBck, LSucFalha);
```

O Algoritmo 2 descreve os passos com as funções propostas. 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 a 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. Esta função fornece também a lista dos predecessores ($LPredFalha$) e a lista dos sucessores ($LSucFalha$) de cada primária, que serão utilizadas para recuperar aplicação em caso de falha. Com $AchaCjtoProcs$ é escolhido

o conjunto dos processadores P^*, onde v^B pode ser escalonada (C_2, C_4 e C_5). Assim, P^* é percorrido, e para cada p_j é calculado o tempo disponível mais cedo $EAT(v_i^B, p_j)$ em que v_i^B deve começar em p_j (linhas 5 e 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, $CriaLNaoSobreposta$ 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, FTMRCD emprega uma heurística do tipo *list scheduling* com uma política de inserção de tarefas para escalonar as *backups*, e a mesma função de custo ponderada. A abordagem bi-objetivo utiliza o parâmetro w para calcular EFT e RC na função de custo. Na linha 8, $InsTarefas$ 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 17). Os valores dos objetivos EFT e RC são calculados antes e agregados na linha 16 dentro de f para todos os processadores de P^*. Para concluir, caso P^* seja diferente de vazio, atualiza-se na linha 20 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 finaliza.

3.3. Cálculo do Escalonamento e da Solução com Falha

Para analisar os benefícios da estratégia é necessário calcular o *makespan* $\mathcal{M}_{falha}$ e a confiabilidade $R_{T_{falha}}$ do escalonamento resultante em caso de falha. Assim, diferente de [Qin and Jiang 2006], é proposto um programa que simula a execução paralela do GAD de acordo com os escalonamentos de MRCD e FTMRCD ($Sch \cup SchBck$).

A falha ocorre em $p(v^P)$ durante a execução de v^P e gera o escalonamento Sch_{falha} das primárias e das *backups* que realmente executam. v^P e as 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 marcadas para ter *falha induzida* de v^P e as mesmas não podem executar, ou seja, são substituídas por suas *backups*. O conjunto $P^* = P - p(v^P)$ 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. Dependendo se v_i é primária ou *backup*, é utilizada a informação do escalonamento Sch ou $SchBck$, respectivamente, para 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 terminaram e, caso verdadeiro, re-calcula-se $EST(v_i, p(v_i))$ e $EFT(v_i, p(v_i))$ em $p(v_i)$. 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 em $SchBck$, pois outras *backups* escalonadas antes, no mesmo processador, $p(v_i^B)$ não precisaram executar. Finalmente, são coletados $\mathcal{M}_{falha}$, $R_{T_{falha}}$ da execução com falha e o escalonamento Sch_{falha}.

4. Análise de Desempenho

Para avaliar o desempenho, as soluções de FTMRCD são analisadas na presença de falha para aplicações paralelas sintéticas descritas na Seção 2. Foi utilizado um cenário de pior caso (CPC) com $m = 24$ processadores agrupados em três grupos P_0, P_1 e P_2 de

$m = 8$. As taxas FP foram geradas uniformemente em $[10^{-5}, 3.3 \times 10^{-5}]$ $\forall p_i \in P_0$, $[3.4 \times 10^{-5}, 6.6 \times 10^{-5}]$ $\forall p_i \in P_1$ e $[6.7 \times 10^{-5}$,$10^{-4}]$ $\forall p_i \in P_2$. Os índices de retardo foram $csi(p_i) = 73$ $\forall p_i \in P_0$, $csi(p_i) = 53$, $\forall p_i \in P_1$ e $csi(p_i) = 33$ $\forall p_i \in P_2$ e as latências L dos canais iguais (peso=1), conforme [Sardina 2010]. P_0 têm os processadores mais lentos e mais confiáveis e P_2, o contrário, configuração escolhida para estudar casos conflitantes onde é difícil chegar a um acordo entre os objetivos do escalonamento.

Nos experimentos calcula-se a solução do escalonamento de FTMRCD na presença de uma falha *crash* no cenário CPC, usando o simulador que gera $\mathcal{M}_{falha}$ e $R_{T_{falha}}$, como proposto na Seção 3.3. Para as aplicações G_n e Di_n foram analisadas as soluções de compromisso $S_{\mathcal{M}}$ e S_e de [Sardina et al. 2009], neste caso formadas pelo par ($\mathcal{M}_{falha}$, $R_{T_{falha}}$). $S_{\mathcal{M}}$ prioriza o *makespan* e S_e prioriza igualmente *makespan* e confiabilidade. Já S_{R_T} priorizando confiabilidade não foi considerada. As soluções ($\mathcal{M}_{falha}$, $R_{T_{falha}}$), são comparadas com as da execução sem falha ($\mathcal{M}$, R_T) de MRCD. As colunas (%) das Tabelas 1 e 2, mostram o percentual de variação da solução em relação a $\mathcal{M}$ e R_T.

MRCD aloca as tarefas para $m = 23$ processadores, desconsiderando o processador que teve a falha em FTMRCD. Com esta comparação avalia-se a qualidade do escalonamento de FTMRCD obtido com *backups*, com o escalonamento de primárias de MRCD sem utilizar o processador com falha. Com FTMRCD, a falha é simulada em P_2 (mais rápido), para um processador que apresenta a menor taxa de falha do 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 início executando a primeira tarefa da aplicação v_0^P. Em FTMRCD, as *backups* são escalonadas considerando o número total de processadores em P. As Tabelas 1 e 2 mostram os resultados obtidos.

Tabela 1. 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
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

Como especificado, tanto MRCD como FTMRCD recebem w como parâmetro, peso associado aos objetivos, $(1 - w)$ da minimização do *makespan* $\mathcal{M}$, e w à maximização da confiabilidade R_T. No entanto, diferentes w podem ser estabelecidos em cada um. Se $w = 0$, MRCD torna-se similar a HEFT [Topcuouglu et al. 2002], a confiabilidade

não é considerada na função, se $w = 1$ o *makespan*. $S_{\mathcal{M}}$ e S_e são soluções escolhidas não-dominadas em w:$\{0.1, 0.2, 0.3, ..., 0.9\}$, conforme [Sardina et al. 2009]. Assim, seja w_1 o peso dado como entrada a MRCD (primárias) e w_2, à FTMRCD (*backups*), as seguintes situações são consideradas. Na Tabela 1, $w_1 \neq 0$ e $w_2 = 0$, ou seja, as *backups* são escalonadas só considerando a minimização do *makespan*. Já na Tabela 2, $w_1 \neq 0$ e $w_2 = w_1$, primárias e *backups* são escalonadas com o mesmo w, considerando em ambos os escalonamentos (MRCD e FTMRCD), o mesmo compromisso entre os objetivos.

Nas tabelas, o símbolo $-$ representa quando FTMRCD não acha processadores para escalonar determinadas *backups*, devido a limitação de processadores na medida que aumenta a aplicação. Estes casos aparecem 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}. Para D_i, mesmo com o aumento da aplicação, sempre acha solução.

Os grafos G_n se caracterizam por ter muitas dependências entre suas tarefas nos múltiplos caminhos (largura do grafo até n) e onde as tarefas estão conectadas com comprimentos até a altura do grafo. Já em D_i pela sua topologia e pela largura máxima de 19, uma mesma tarefa tem menos caminhos e dependências. Pela relação de precedência das tarefas, escalonar *backups* com maior número de predecessores pode ser mais difícil, se suas primárias não foram classificadas *totalmente fortes* (aumenta a chance de falha). Por C_4 e C_5, estas *backups* não podem ser escalonadas junto com primárias *candidatas a provocar suas falhas induzidas*, reduzindo o número de processadores possíveis.

Por outro lado, como $S_{\mathcal{M}}$ prioriza o *makespan*, para menores G_n 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. 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. Já com S_e, como o escalonamento busca um equilíbrio, influencia também a confiabilidade, sendo mais difícil no cenário CPC escolher qualquer processador para alocar as primárias. Assim, o escalonamento tem mais processadores disponíveis (geralmente os menos confiáveis) para alocar as *backups*.

Em casos onde acontece indisponibilidade de processadores uma vantagem de FTMRCD comparado com [Qin and Jiang 2006] é que permite mudar o valor de w para achar outra solução que possa gerar um escalonamento tolerante a falha.

Na Tabela 1 como as *backups* são escalonadas com prioridade total para o *makespan* ($w_2 = 0$), a confiabilidade é menor do que nas soluções da Tabela 2. No entanto para $S_{\mathcal{M}}$ obviamente o *makespan* gerado por FTMRCD é maior que o de MRCD. $S_{\mathcal{M}}$ para MRCD obtêm a melhor solução para o *makespan* embora estabeleça um compromisso com a confiabilidade, portanto não é melhorada por FTMRCD, devido a 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

Tabela 2. 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, 0.93247	502.7, 0.94470	256.7, 0.92389	35.1, -0.92	**406.0**, 0.93480	-19.2, -1.04
G_{252}	318.7, 0.85793	907.2, 0.87990	479.9, 0.83935	50.5, -2.16	**721.6**, 0.86284	-20.4, -1.93
G_{377}	480.4, 0.74863	1643.4, 0.78952	893.6, 0.72681	86.0, -2.91	**1182.4**, 0.75434	-28.0 -4.44
G_{527}	675.1, 0.60869	2468.6, 0.66951	1362.5, 0.58774	101.8, -3.44	**1687.0** 0.62366	-31.6, -6.86
G_{702}	942.2, 0.45729	3810.7, 0.53888	- -	- -	**2674.3**, 0.47626	-29.8, -11.6
Di_{81}	833.9, 0.84373	1704.9, 0.87730	1286.9, **0.85937**	54.3, 1.85	**1624.9**, 0.84789	-4.74, -3.35
Di_{100}	947.0, 0.81091	1997.0, 0.84900	1426.0, **0.82543**	50.5, 1.79	**1917.0**, 0.81459	-4.00, -4.05
Di_{144}	834.0, 0.71896	2541.0, 0.78509	1692.0, **0.75414**	50.7, 4.89	**2327.0**, 0.72960	-8.42, -7.06
Di_{256}	1268.0, 0.55939	3749.0, 0.63910	2302.0, 0.55771	81.5, -0.30	**3465.0**, 0.58418	-7.57, -8.60
Di_{361}	1518.0, 0.41511	4844.0, 0.52690	3172.0, **0.46382**	108.9, 11.7	**3319.0**, 0.44605	-4.63, -15.3

processador mais confiável do grupo P_2, assim as *backups* correspondentes as primárias dele, quando escalonadas em P_2 (para priorizar o *makespan*) têm o FP maior.

Para S_e de FTMRCD o *makespan* chega a ser até menor que o de MRCD (% com valores negativos). As primárias (MRCD) foram escalonadas considerando o equilíbrio entre os objetivos, enquanto as *backups* de FTMRCD só priorizam o tempo. Assim, as *backups* foram escalonadas em processadores diferentes dos processadores das primárias, os 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 de $S_{\mathcal{M}}$.

Para $w_2 = w_1$ com FTMRCD (Tabela 2), as soluções são melhores em confiabilidade. As *backups* foram escalonadas considerando um compromisso entre os objetivos de MRCD. Como esperado, $S_{\mathcal{M}}$ de MRCD com $w_1 > 0$ têm *makespan*s maiores as de 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 melhor *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.

Por outro lado, a execução da aplicação com FTMRCD nos experimentos demonstra consistência, mesmo na presença de falha. Como [Qin and Jiang 2006] considera um sistema de tempo real com restrições e características diferentes, de forma prática a comparação não foi possível. Entretanto, se observa que alguns critérios necessários para garantir a execução completa e consistente em caso de falha não são considerados, como

por exemplo C_5. A ausência deste pode provocar falhas induzidas dependendo do processador que falha. Além do mais, a classificação da tarefa primária é feita em função de todos os predecessores e não por cada predecessor, diminuindo opções de escalonamento, flexibilidade e eficiência. Uma descrição mais detalhada aparece em [Sardina 2010].

5. Conclusões

Neste trabalho, uma abordagem bi-objetivo e novos conceitos foram introduzidos para garantir uma recuperação consistente e eficiente da aplicação em um sistema com falha. Para o escalonamento de *backups*, foram propostos distintos critérios e uma classificação das tarefas primárias e de suas falhas. A adição da ponderação nos objetivos ofereceu mais flexibilidade, sendo possíveis distintas soluções com falha para uma mesma aplicação e arquitetura. Foi possível avaliar a execução da aplicação com falha, gerando o escalonamento *backups* que realmente executam. Os resultados em relação aos custos da tolerância a falha são favoráveis e considerando que sempre foram simuladas falhas de processador mais críticas. No cenário com falha, a solução S_e, teve seu melhor desempenho para o *makespan* chegando a alcançar valores menores que no escalonamento das primárias. Avaliações do algoritmo proposto em ambientes reais de execução utilizando MPI, são apresentados em [Sardina 2010] com bons resultados.

Referências

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

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

Boeres, C., Sardina, I. M., and Drummond, L. M. A. (2010). An efficient weighted bi-objective scheduling algorithm for heterogeneous systems. *Parallel Computing*.

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

Naedele, M. (1999). 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*, page 392, Washington. IEEE Computer Society.

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

Sardina, I. M. (2010). *Escalonamento Estático de Tarefas Bi-objetivo e Tolerante a Falhas para Sistemas Distribuídos*. PhD thesis, Instituto de Computação, Universidade Federal Fluminense, Niterói, RJ, Brasil.

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

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