Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Abordagem Adaptativa de Monitoramento para Escalonamento de Grafos Dirigidos Acíclicos em Ambientes Distribuídos

Jorge Schtoltz

Orientador Prof. Dr. Ing. Gerson Henrique Pfitscher

Brasília (DF)
2007

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Jorge Schtoltz

Abordagem Adaptativa de Monitoramento para Escalonamento de Grafos Dirigidos Acíclicos em Ambientes Distribuídos

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre
Programa de Pós-Graduação em Informática
Departamento da Ciência da Computação
Universidade de Brasília.

Orientador: Prof. Dr. Ing. Gerson Henrique Pfitscher.

Brasília (DF), 21 de setembro de 2007

Universidade de Brasília - UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Mestrado em Informática

Coordenadora: Profª Drª Alba Cristina Magalhães Alves de Melo

Banca examinadora composta por:
Prof. Dr. Ing. Gerson Henrique Pfitscher (orientador) – CIC/UNB
Profª Drª Alba Cristina Magalhães Alves de Melo – CIC/UNB
Prof. Dr. Mario Antonio Ribeiro Dantas – INE/UFSC

CIP – CATALOGAÇÃO INTERNACIONAL DA PUBLICAÇÃO

Jorge Schtoltz

Abordagem Adaptativa de Monitoramento para Escalonamento de Grafos Dirigidos Acíclicos em Ambientes Distribuídos/ Jorge Schtoltz.
Brasília : UnB, 2007.
84p. : Il. ; 29,5 cm.

Dissertação (Mestrado) – Universidade de Brasília, Brasília, 2007.

1. Algoritmos de Escalonamento, 2.Grafos de Tarefas, 3. Ambientes Distribuídos, 4. *Clusters* de Workstations.

CDU 004

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro – Asa Norte
CEP 70.910-900
Brasília – DF – Brasil

DEDICATÓRIA

Dedico este trabalho aos meus Pais, a Paula e a todos meus amigos
que ajudaram de alguma forma na sua realização.

AGRADECIMENTOS

Agradeço a Deus e a minha esposa Paula pela força e suporte que me prestou nesta jornada, a minha família, aos meus amigos e colegas de trabalho por entenderem as minhas ausências durante a realização deste curso.

A todos os meus colegas de mestrado, pela ajuda mútua, especialmente ao pessoal *lambda-lambda-lambda*.

Ao professor Gerson, pela sua compreensão e paciência durante este período de grandes dificuldades e conquistas da minha vida.

E por fim aos professores e professoras do mestrado pela oportunidade e dedicação ao seu trabalho.

RESUMO

O escalonamento estático de um programa, representado por um grafo dirigido acíclico de tarefas, em um ambiente multiprocessado, tem o objetivo de minimizar o tempo de conclusão do programa. Apesar das pesquisas nesta área terem obtido heurísticas eficientes, encontrar um escalonamento ótimo é um problema NP - Completo.

Clusters de *workstations* podem ser utilizados no processamento paralelo de programas e devido à complexidade de integração entre os programas, o monitoramento e a simulação são efetuados através de ferramentas que gerenciam o *cluster* e a execução dos programas paralelos. Dentre as ferramentas analisadas, o PM^2P, será utilizado como base de estudo devido ao conhecimento da ferramenta. O objetivo deste trabalho é o desenvolvimento e implementação na ferramenta citada um monitoramento periódico para verificar a disponibilidade das máquinas do *cluster* e re-escalonar os programas se houver necessidade ou ganho de desempenho. Para testar este monitoramento foram implementados, devido à carência de algoritmos na ferramenta, três algoritmos estáticos voltados para o escalonamento de tarefas que possam ser representados por um GDA (Grafo Dirigido Acíclico). Estes algoritmos funcionam de forma similar, gerando uma lista em ordem topológica das tarefas e àquelas pertencentes ao mesmo nível, portanto, concorrentes entre si, são ordenadas pelo maior ou menor tempo de execução. As tarefas são distribuídas de acordo com a disponibilidade das máquinas no *cluster* e o objetivo do algoritmo de escalonamento é manter o *makespan* gerado igual ao tempo do caminho crítico do grafo.

Os resultados dos testes, as conclusões sobre o monitoramento e re-escalonamento serão demonstradas através de tabelas e mapas de Gantt para facilitar a visualização e o entendimento. Foram testados conjuntos de tarefas distintas que representam aplicações exemplo. Foi observado que aplicações rápidas, que finalizam sua execução concomitante ou logo após o tempo gasto para verificar a disponibilidade das máquinas no *cluster*, o monitoramento e o re-escalonamento não são necessários e neste caso é recomendável o reinício da aplicação. Ao contrário, aplicações que demandam mais tempo para sua execução, o monitoramento e o re-início de algum programa no caso de indisponibilidade de uma máquina são importantes, uma vez que a aplicação continua sua execução com a nova arquitetura de máquinas, a partir do ponto de detecção da falha. Apesar do custo adicional para execução da atividade, conclui-se que há vantagens em se ter um *cluster* monitorado quando da execução dos programas paralelos utilizando a biblioteca MPI.

ABSTRACT

Static scheduling of a program represented by a directed acyclic graph task on a multiprocessor environment to minimize the program completion time is the goal. It is a well-known problem of concurrent processing. Although the researches in this area already have reached heuristic efficient to find an optimal scheduling is a NP-Complete problem.

The Cluster of Workstations can be used to the parallel processing of programs and due to the program interaction complexity, the monitoring and the simulation are made through frameworks that manage the cluster and the parallel program execution. Among the frameworks analyzed, the framework PM^2P, will be used as the base study. The objective of this work is the development and implements a periodic monitoring to verify the machines availability at the cluster and rescheduling the programs whether is necessary or to improve the performance. It has been implemented, to test this periodic monitoring due to the framework algorithms lack new three static algorithms toward the scheduling of programs of tasks that could be represented by DAG (*Directed Acyclic Graph*). These three algorithms working in a similar way, each one create a topological order list of the tasks and they scheduling the tasks that belongs at the same level of the graph and ordering these tasks by the bigger or smaller execution time criteria. The tasks are sorted and distributed through the cluster machines in accordance with the availability of them. The scheduling goal is getting the graph makespan like the critical path time.

The tests results, monitoring conclusions and the scheduling algorithms proposals will be demonstrated with tables and Gantt charts to a best exhibition and comprehension. The tests were over sets of different tasks that representing some real application. It was observed that applications with small execution time finish their at the same time or as soon as the framework check the cluster machines availability. In these cases are not necessary to rescheduling the applications and is recommended restart them. By the other hand, applications with big execution time, the programs monitoring and the restart of some program is important if any machine become unavailability. So, the application continues this execution at the restart point using the other workstations. Nevertheless the additional monitoring overhead, the conclusions show that exist advantages of monitoring the cluster when are executing a parallel programs using the MPI library with a big execution time.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

1 INTRODUÇÃO

Um algoritmo de escalonamento é composto por procedimentos que tem por objetivo ordenar as tarefas para serem processadas observando o atendimento dos requisitos de cada tarefa. O escalonamento de sistemas de tempo real envolve a determinação de um ordenamento temporal de tarefas, alocadas num conjunto de processadores sob restrições de alguma especificação de tempo, precedência e requisitos de recursos [1]. Os algoritmos de escalonamento utilizados em ambientes multiprocessados, observando as dependências entre as tarefas, custo de comunicação e tempos de execução diversos não apresentam um escalonamento ótimo em todos os casos. A aplicabilidade dos algoritmos de escalonamento é ampla e está presente no atendimento das necessidades específicas de sistemas embarcados [2], atendimento de QoS (*Quality of Service*) na transmissão de dados [3], atendimento de sistemas de tempo real *soft*, por algoritmos de tempo real *hard* para a garantia do *deadline* de determinadas tarefas [4], atendimento dos *deadlines* das tarefas baseadas em taxas de periodicidade [5], detecção dinâmica das periodicidades das tarefas [6] e nas mais diversas necessidades devido às particularidades dos sistemas paralelos [7].

A diminuição do custo de aquisição dos equipamentos do tipo *workstation,* faz com que seja aumentada a utilização de *clusters* e conseqüentemente o multiprocessamento tornou-se uma prática comum [8]. Além da alta capacidade de processamento a um menor custo, esta arquitetura proporciona características de escalabilidade, confiabilidade e robustez. Entretanto, a implementação desta solução é complexa, pois exige que as tarefas que compõem a aplicação sejam processadas em paralelo, tornando a organização e atribuição das tarefas aos processadores uma prática demorada [9]. O escalonamento de tarefas, considerando as suas dependências e o custo de execução efetuado num ambiente multiprocessado, é um problema NP - Completo [10]. Alguns trabalhos que tratam deste assunto propõem a utilização de algoritmos adaptativos nos quais, a partir de um algoritmo de escalonamento conhecido são inseridos novos parâmetros, especializando este algoritmo para atender de forma eficiente um determinado conjunto de requisitos.

Estudos demonstram que os algoritmos *Earliest Deadline First* - EDF e Least Laxity First - LLF são considerados ótimos em ambientes monoprocessados, entretanto, em ambientes multiprocessados nem todos os sistemas escalonados por estes algoritmos apresentam resultados ótimos [11]. Estes algoritmos são amplamente utilizados para o desenvolvimento de algoritmos adaptativos [6, 12] e também como parâmetro para avaliar os

resultados de testes de novos algoritmos propostos [7, 13]. Serão apresentados cinco algoritmos adaptativos baseados nos algoritmos EDF e LLF para demonstrar as variações que podem ser implementadas. Este trabalho faz uma análise dos algoritmos de escalonamento RIPF (*Risk Incursion Potential Function*) [14], A-GREEDY (*Adaptive Scheduling with Parallelism Feedback*) [15, 16], SHARP (S*ecurity and Heterogeneity-Driven Scheduling Algorithm*) [17], ELLF (*Enhanced Last Laxity First*) [18] e A-STEAL (A*daptive Work-Stealing Thread Scheduler*) [16, 20, 21].

Diferente da programação tradicional, a programação paralela pode proporcionar maior *performance*, entretanto o seu desenvolvimento é mais complexo. Para subsidiar estas tarefas são utilizados *frameworks* que simulam e executam os programas auxiliando na determinação da sua viabilidade, melhor solução e acurácia. A partir do levantamento de simuladores de escalonamento de tarefas [22, 23, 24, 25], foram escolhidos para análise os *frameworks* PM^2P (*Performance Monitoring of Message Passing*) [8, 26], YASA (*Yet Another Schedulability Analyser*) [18, 27] e *Cheddar* [28, 29]. O critério para determinação e escolha de um *framework* dentre aqueles analisados foi a facilidade de uso e o suporte às características necessárias para o desenvolvimento deste trabalho como a possibilidade de se incluir um algoritmo para monitoração e controle das máquinas, novos algoritmos de escalonamento, disponibilidade de todos os códigos fonte e ser de domínio público.

O *framework* escolhido, o PM^2P, está voltado para monitoração e controle de *cluster Workstations.* A infra-estrutura disponível para implementação deste trabalho são oito máquinas definidas como escravas as quais executam o processamento das tarefas e uma máquina *front end*, que coordena a distribuição e colhe as informações dos processamentos para apresentação dos resultados através de mapa de Gantt. O PM^2P*,* após distribuir as tarefas do sistema no *cluster* de máquinas de acordo com o escalonamento gerado pelo algoritmo, monitora os resultados do processamento de cada tarefa. Entretanto, como o escalonamento do GDA (Grafo Dirigido Acíclico) é efetuado de forma estática, de acordo com a disponibilidade do *cluster*, caso a quantidade de máquinas do *cluster* se altere durante a execução do sistema, o escalonamento poderá ficar prejudicado, uma vez que as tarefas não são re-escalonadas dinamicamente considerando a nova arquitetura.

Um algoritmo adaptativo pode ser interpretado como uma coleção de cláusulas do tipo *if-then*, as quais testam a situação corrente do dispositivo em que atuam em relação a uma configuração específica, e levam o dispositivo à sua próxima situação. Esses dispositivos

podem ser modelados como abstrações matemáticas capazes de modificar-se dinamicamente, constituindo assim formalismos capazes de auto modificar-se autonomamente [30]

O objetivo desta dissertação de mestrado é propor e implementar uma abordagem adaptativa de monitoramento em ambientes distribuídos para gerenciar a disponibilidade das máquinas que compõem um *cluster* e re-escalonar o conjunto de tarefas representadas por um GDA, caso a disponibilidade de processamento se altere durante a execução da aplicação. As propostas de escalonamento adaptativo abordam a disponibilidade das tarefas considerando os requisitos da aplicação, re-escalonando as tarefas de acordo com os novos dados recebidos, não levando em conta a disponibilidade da infra-estrutura existente [14, 15, 16, 17, 18]. Será implementado no PM^{2}P um algoritmo para monitorar o *cluster* e adaptar a execução do sistema a nova infra-estrutura, re-escalonando as tarefas pendentes de execução. Este monitoramento é executado de forma concorrente na máquina *front end*. A fim de não aumentar o número de mensagens trafegando no sistema, o monitoramento é efetuado através de um mecanismo que conecta as máquinas escravas a máquina *front end* via porta paralela [26]. O monitoramento das máquinas do *cluster* mostrou-se uma técnica relevante quando o GDA apresenta tarefas com tempos de execução longos, neste caso a alteração da disponibilidade dos equipamentos é determinante no tempo de execução dos programas paralelos. Os resultados dos testes são apresentados na forma de mapas de Gantt, a fim de facilitar a visualização do escalonamento e a comparação com os demais escalonamentos.

Para testar o algoritmo de monitoramento proposto nesta dissertação foi implementado no PM^{2}P três novos algoritmos estáticos voltados para o escalonamento de tarefas representadas por um GDA. O funcionamento dos algoritmos desenvolvidos dá-se em duas etapas realizadas de forma seqüencial. Na primeira parte é criada uma lista dos nós do GDA em ordem topológica, dividido em níveis de execução e respeitando as dependências entre as tarefas. Na segunda parte todas as tarefas concorrentes que pertencem ao mesmo nível do GDA e estão prontas para execução, são alocadas aos processadores disponíveis, utilizando os critérios de menor ou maior tempo de execução. Os algoritmos consideram como tempo máximo para execução de cada tarefa, pertencentes ao conjunto de tarefas que estão no mesmo nível do grafo, o tempo de início de suas tarefas filhas, e definem este tempo máximo como o *deadline* das tarefas. Os algoritmos de escalonamento dinâmicos tipo EDF e LLF são semelhantes aos algoritmos desenvolvidos quando o processo entra na segunda fase e são feitos os escalonamentos das tarefas pertencentes ao mesmo nível do grafo, uma vez que este grupo de tarefas, prontas para execução, não possuem dependência entre si e estão

limitadas a um *deadline*. Os algoritmos desenvolvidos apresentam os melhores resultados de escalonamento quando o *makespan* gerado é igual ao caminho critico deste grafo. Estes tempos se alteram de acordo com o grafo de tarefas utilizado e disponibilidade do *cluster*. Um dos algoritmos implementados na ferramenta, além de ordenar as tarefas pelos tempos de execução mais longos, considera outro parâmetro para priorizar as tarefas no escalonamento. Este parâmetro é inserido pelo usuário de acordo com a semântica da aplicação e neste caso o escalonamento prioriza estas tarefas em detrimento do outro crtério de ordenação.

Esta dissertação de mestrado está dividida em sete capítulos. Após esta introdução, o capítulo dois apresenta os conceitos de sistemas de tempo real, sistemas operacionais de tempo real e políticas de escalonamento de sistemas de tempo real.

O capítulo três detalha o conceito de algoritmo de escalonamento, a taxonomia dos algoritmos estáticos e dinâmicos, conceito de tarefas, conceitos e definições de grafos acíclicos dirigidos utilizados para desenvolvimento dos algoritmos de escalonamento propostos, os conceitos e definições de mapa de Gantt, os algoritmos EDF e LLF e o detalhamento dos cinco algoritmos de escalonamento que garantem resultados ótimos, dentro do escopo do problema a que se destinam.

No capítulo quatro detalhamos os três *frameworks* utilizados para análise e auxílio no desenvolvimento de programas concorrentes. Foi dada maior atenção ao *framework* PM^2P por ser a ferramenta sobre a qual este trabalho está baseado.

No capítulo cinco estão detalhados os algoritmos desenvolvidos para auxílio nos testes do algoritmo adaptativo proposto, como está implementado na ferramenta e seu funcionamento.

No capítulo seis está o roteiro e os resultados dos testes realizados durante a execução de um sistema paralelo fictício utilizando os algoritmos propostos e simulando quedas das máquinas do *cluster*.

No capitulo sete a conclusão sobre os trabalhos efetuados.

2 SISTEMAS DE TEMPO REAL

Um sistema de tempo real é um conjunto de equipamentos e aplicativos, onde a importância não está somente na capacidade e na corretude de entrega dos resultados, mas também, no tempo em que esses resultados são disponibilizados [1]. Esta é a principal diferença entre sistemas de tempo real e sistemas não tempo real.

Uma possível classificação dos sistemas de tempo real é quanto à relevância da violação do seu *deadline*, sendo denominados de sistemas de tempo real *hard* ou *soft* [31]. Nos sistemas *hard,* as violações das restrições do tempo causam sérios problemas, como grandes perdas econômicas ou acidentes fatais devido ao não cumprimento das restrições de tempo. Nos sistemas *soft,* as perdas não são tão significativas e portando podem ser aceitas, como nas aplicações de s*tream* de vídeo.

Outra classificação foi proposta por Nissanke [1] e refere-se à periodicidade na execução das tarefas, denominadas de eventuais ou periódicas. Tarefas periódicas possuem um período de execução, com tempos de chegada, de processamento e *deadline* conhecidos, e não se alteram entre estes períodos. Tais tarefas ocorrem tipicamente nos equipamentos de monitoramento de sistemas de tempo real. As tarefas eventuais ou esporádicas possuem tempos de chegada, tempos de processamento e *deadlines* arbitrários. São típicas em equipamentos de controle de sistemas de tempo real. São mais freqüentes na inicialização de sistemas e recuperação de estados (no caso de erros) e geralmente ocorrem com menor freqüência.

2.1 Conceito de Tarefa

Uma tarefa é uma entidade de *software* ou um programa especializado na execução de uma ação específica e pode ser considerada, do ponto de vista de escalonamento, como a menor entidade que pode ser despachada por um sistema operacional [1].

Sob o ponto de vista do escalonamento, qualquer tarefa pode ser representada e especificada pelos seguintes parâmetros [32]:

- Tempo de chegada da tarefa, ou tempo de chamada da tarefa, *a*: É o instante que a tarefa entra na fila de execução.

- *Ready time* (primeiro tempo de início permitido), *r*: É o instante em que a tarefa já dispõe de todos os recursos necessários a sua execução.
- Tempo de execução (ou tempo de processamento), *c*: É o intervalo de tempo gasto para a execução completa da tarefa.
- *Deadline* (prazo para completar a tarefa), *d*: é o último instante em que a tarefa pode ser finalizada sem falhas.
- Regularidade de chegada da tarefa. É o ciclo de apresentação da tarefa para execução e podem ser classificadas em:

a) Tarefas periódicas, com *T* representando o período. Tais tarefas ocorrem tipicamente no monitoramento em tempo real;

b) Tarefas esporádicas, com tempos de chegada, tempos de processamento e *deadlines* arbitrários. São típicas em controle de tempo real. São mais freqüentes na inicialização de sistemas e recuperação de estados (erros), pode ocorrer com menor freqüência.

As dependências entre as tarefas existem por diversas razões, dentre as quais se destacam [1]:

- Relação de precedência – referem-se às tarefas que possuem restrições de dependência entre si. Tarefas sem nenhuma restrição de precedência entre si são ditas independentes. Estas restrições podem ser induzidas pelo ordenamento de processos dependente da aplicação e a comunicação entre tarefas.
- Dependência de recursos – referem-se às tarefas que interagem com recursos da máquina e são traduzidas normalmente como restrições de exclusão mútua ou *overlaping:*
  - Uso de recursos ativos refere-se àqueles com capacidade de processamento, normalmente são requisitados em modo exclusivo.
  - Uso de recursos passivos refere-se aos arquivos do sistema e podem ser acessados tanto no modo exclusivo como compartilhado.

Pode-se adotar uma definição alternativa aplicável ao escalonamento de tarefas [1]. Seja TID um conjunto de identificadores de tarefas para distingui-las entre si de modo único e $\mathcal{T}$ o conjunto de comprimentos de tempo de relógio $\mathbb{T}$ e 0 um comprimento zero. É adotado o seguinte conjunto como uma abstração matemática de todas as possíveis tarefas esporádicas (ou aperiódicas):

$$\text{TASK} \triangleq \{(tid, a, c, d) \bullet tid : \text{TID}; c : \mathcal{T}; a, d : \mathbb{T} \mid c \geq 0 \wedge a + c \leq d\} \quad (2.1)$$

Assim, cada tarefa é representada por uma quadra de valores, *tid, a, c, d*, cujos elementos formam um subconjunto dos parâmetros de tarefas, satisfazendo certas relações matemáticas.

2.2 Sistemas Operacionais de Tempo Real

Os sistemas operacionais de tempo real são partes integrantes dos sistemas de tempo real. As funções principais são similares às dos sistemas operacionais convencionais. Entretanto, a maneira como estas funções são implementadas é o que difere dos demais sistemas [33].

Além das funções necessárias ao correto funcionamento, é desejável que os sistemas operacionais de tempo real apresentem outras características que o qualificam em relação aos demais sistemas [31, 33, 34]:

- **Ser *multithreaded* e preemptivo.** Os recursos existentes necessitam ser alocados a uma *thread* e controlados tempestivamente para o atendimento de uma requisição. A tarefa em execução pode ser interrompida.

- **Cada *thread* deve ter uma prioridade exclusiva**. Neste caso, é importante possuir um grande número de prioridades, porque os projetistas convertem os requisitos de *deadline* das tarefas em prioridades. Se duas *threads* possuem a mesma prioridade, elas executam em regime de *time slicing*, comprometendo o atendimento dos requisitos do sistema.

- **Possuir mecanismos determinísticos de sincronização de *threads***. Estes mecanismos incluem soluções para os problemas da exclusão mútua, gerenciamento de eventos, comunicação entre *threads,* etc.

- **Prevenir a inversão de prioridades.** Possuir um sistema de herança de prioridades e ter um comportamento temporal previsível, apresentando um limite superior conhecido para os tempos de latência, de interrupção, escalonamento, etc.

- **Possuir Atividades assíncronas.** Ser capaz de temporizar atividades assíncronas em relação ao *clock* da CPU e escalonar tarefas periódicas nesta base de tempo. Aplicações críticas requerem o uso de temporizadores precisos, com boa repetição de comportamento. Na ocorrência de uma interrupção de *timer,* uma ISR (*Interrupt Service*

Routine) será escalonada, ou uma *thread* será despertada para realizar alguma atividade crítica.

De forma similar aos sistemas de tempo real, os sistemas operacionais de tempo real podem ser caracterizados como [31, 34, 35]:

- **Sistemas operacionais de tempo real *soft.*** Possuem em sua implementação características que não oferecem garantias temporais, do tipo melhor esforço. Exemplo: OS9 [36].

- **Sistema operacional de tempo real *hard***. Diferem basicamente no aspecto temporal, pois os sistemas *hard* oferecem mecanismos adicionais para não perderem as garantias temporais estabelecidas. Exemplo: RTAI [37], *SHaRK* [38] e *RTLinux* [39].

2.3 Políticas de Escalonamento de Sistemas de Tempo Real

O escalonamento de tarefas de tempo real, definido no capítulo 3, segue esquemas de ordenamento e distribuição das tarefas entre os processadores de acordo com as políticas definidas.

Existem vários esquemas de escalonamento de tarefas de tempo real, as políticas mais comuns são as de escalonamento preemptivo com prioridade fixa e escalonamento preemptivo com prioridade dinâmica [40].

Para demonstrar as duas políticas mais utilizadas, necessitamos entender o funcionamento da preempção. Conforme Cottet et al. [31], a preempção é definida como a suspensão e substituição de uma tarefa em execução, com menor prioridade, por outra de maior prioridade que necessite da utilização do processador. Este conceito é aplicado aos algoritmos de escalonamento os quais podem ser ou não preemptivos.

2.3.1 Políticas de Escalonamento preemptivo com prioridade fixa

É suportada pela maioria dos sistemas operacionais de tempo real. Cada tarefa possui uma prioridade fixa e que não é alterada, a menos que a aplicação modifique-a. Devido à preempção, uma tarefa de prioridade mais alta interrompe uma tarefa de prioridade inferior. Estes escalonamentos proporcionam fácil adaptação às tarefas e também acomodam facilmente tarefas esporádicas. Possuem um comportamento determinístico em sobrecargas (apenas as tarefas menos prioritárias são afetadas). Comparando com os escalonamentos não

preemptivos, sua implementação é mais complexa e demanda uma sobrecarga de execução mais elevada (*scheduler e dispatcher*). Em casos de sobrecarga no sistema, devido a erros de projeto ou programação, os níveis de prioridades mais elevados podem bloquear o sistema [40]. Como exemplo, temos o escalonamento *Rate Monotonic* [31], que é uma política de escalonamento preemptivo ótima de prioridades fixas, na qual existe uma relação direta entre a freqüência de ativação de uma tarefa e sua prioridade. Assume que o *deadline* de uma tarefa periódica é igual ao seu período. Pode ser implementado em sistemas operacionais que suportam escalonamento preemptivo com prioridade fixa ou generalizada para tarefas aperiódicas. Liu e Layland [41] provaram que este escalonamento é ótimo para tarefas com prioridades estáticas, desde que, se um conjunto de tarefas periódicas é possível ser escalonado por algum método de atribuição de prioridades, então também será com o algoritmo *Rate Monotonic*. Outro exemplo é o escalonamento *Deadline Monotonic* [31]. Trata-se da generalização da política de escalonamento *Rate Monotonic*, no qual existe uma relação direta entre o *deadline* de uma tarefa e sua prioridade. Quando o tempo do *deadline* se iguala o período da tarefa, esta política é idêntica ao esquema de escalonamento *Rate Monotonic.* Nos demais casos, o *deadline* de uma tarefa é um ponto fixo no tempo, relativo ao início do período. Quanto mais próximo este *deadline e*stiver, maior será a sua prioridade. Conforme Sha et al [42], Leung provou que este escalonamento é ótimo, considerando a possibilidade de tarefas que possuam *deadlines* menores que seus períodos.

2.3.2 Políticas de Escalonamento preemptivo com prioridade dinâmica

Poucos sistemas operacionais de tempo real trazem implementado de forma nativa este tipo de escalonamento. O sistema altera a prioridade das tarefas a qualquer instante, ao longo de sua execução, para atender a um objetivo específico de resposta temporal [31]. Devido à preempção, uma tarefa de prioridade mais alta interrompe uma tarefa de prioridade inferior. De forma análoga à política de escalonamento com prioridades fixas, este escalonamento é de fácil adaptação, permitindo que alterações no número de tarefas possam ser imediatamente levadas em conta pelo escalonamento e também acomodam facilmente tarefas esporádicas. Em relação aos escalonamentos de prioridades fixas, estes escalonamentos possuem implementação mais complexa e exigem que o sistema operacional tenha implementado em seu *kernel* suporte ao escalonamento com prioridades dinâmicas de tarefas. A sobrecarga na execução do algoritmo também é mais elevada devido à reordenação

dinâmica da fila das tarefas prontas, e esta sobrecarga varia de acordo com o algoritmo. Dentre as dificuldades apresentadas, a instabilidade em sobrecargas é mais crítica, uma vez que não se pode determinar a *priori*, quais os subconjuntos de tarefas que vão, ou não vão cumprir o *deadline* [40]. Como exemplo tem-se o escalonamento EDF [7, 31], cuja descrição encontra-se na seção 3.6.1. É uma política de escalonamento preemptivo com prioridade dinâmica, onde os *deadlines* são calculados e a tarefa com o *deadline* mais próximo é escolhida para executar primeiro, interrompendo uma tarefa com um *deadline* posterior. Esta política visa minimizar o atraso máximo de um conjunto de tarefas. Conforme Sha et al [42], Dertouzos provou que EDF é um algoritmo ótimo entre todos os algoritmos de escalonamentos preemptivos dinâmicos no sentido de que, se existe um escalonamento viável para um conjunto de tarefas, então o escalonamento produzido por EDF também será viável. Outro exemplo é o escalonamento LLF [31, 43], cuja descrição encontra-se na seção 3.6.2. Da mesma forma que o EDF, trata-se de uma política de escalonamento preemptivo com prioridade dinâmica. Utiliza como critério de ordenamento a folga ou relaxação de uma tarefa que é calculada pela diferença entre o seu *deadline* e o tempo necessário para sua execução, a partir do instante atual. O escalonamento atribui maior prioridade às tarefas que apresentam menor tempo de relaxação executando-as primeiro e interrompendo, se necessário, uma tarefa com uma relaxação maior. Esta política maximiza o atraso mínimo de um conjunto de tarefas. Conforme Sha et al [42], Mok provou a optimalidade deste algoritmo, considerando um ambiente monoprocessado.

2.3.3 Outras políticas correlacionadas

As políticas de escalonamento de tarefas podem ser correlacionadas com as políticas de carga balanceada, Ferrari et al [44] define carga balanceada como um esquema cujo objetivo é minimizar o tempo de resposta de um comando de usuário, considerando uma possível execução remota. Uma das classificações de carga balanceada podem ser preemptiva ou não preemptiva, seguindo a mesma definição apresentada para escalonamento na seção 2.3. Outra classificação de carga balanceada é quando o esquema envolve controle centralizado definindo onde serão tomadas as decisões, podendo ser através de uma iniciativa ou por meio de um tomador de decisões e nesta perspectiva, o controle pode estar na máquina origem ou destino. A máquina local toma a iniciativa quando um novo processo está para ser executado ou a carga está além do limite tolerável. A máquina que contém o controle

centralizado irá iniciar as tomadas de decisões de balanceamento de carga, necessitando selecionar recebedores e destinatários. Esta seleção pode ser efetuada de forma dependente ou independente. Entre as políticas de carga independente está o esquema randômico, o qual escolhe de forma randômica o destinatário do processo para ser executado de forma remota. No caso de carga dependente a máquina destino é escolhida pelo menor tempo [44].

3 ESCALONAMENTO DE TEMPO REAL

Escalonamento de tarefas de tempo real refere-se a um ordenamento das tarefas para serem alocadas a um conjunto de processadores levando em conta determinadas restrições de tempo, prioridades, precedência e necessidade de recursos [33].

3.1 Taxonomia

Os escalonamentos podem ser classificados em estáticos, dinâmicos ou mistos [1, 45]. Os escalonamentos estáticos possuem todas as informações necessárias para o escalonamento das tarefas, e são utilizados em situações bem conhecidas, como nos processamentos *batch*. Para qualquer alteração em algum dos parâmetros das tarefas é necessária a revisão de toda a fila de escalonamento, portanto este escalonamento não suporta tarefas não previstas. Ao contrário do estático, o escalonamento dinâmico é projetado para escalonar tarefas sem previsão de chegada (indeterminadas). Há uma total flexibilidade na alocação das tarefas, inclusive com preempção das mesmas para atendimento de requisições emergenciais. O escalonamento misto tende a combinar os dois escalonamentos, ou seja, são ordenadas aquelas tarefas cujas características são conhecidas a *priori* de forma estática e são acomodadas dinamicamente as demais tarefas conforme sua chegada.

Kwok e Ahmad [10] e Gambier [34] apresentam classificações dos mais conhecidos algoritmos de escalonamento estáticos e dinâmicos respectivamente, para sistemas multiprocessados e monoprocessados. A classificação destes algoritmos está representado na figura 3.1. As siglas com os respectivos nomes dos algoritmos encontram-se na tabela 3.1 para os algoritmos dinâmicos e na tabela 3.2 para os algoritmos estáticos.

Como pode ser observado na figura 3.1, os escalonadores podem ser divididos em estáticos e dinâmicos. O segmento de escalonadores estáticos está baseado em GDA e a classificação utilizada divide-os de acordo com sua capacidade, diferente do segmento de escalonadores dinâmicos, os quais são divididos em outros subgrupos. As tabelas 3.1 e 3.2 contêm a descrição dos escalonadores montada a partir das siglas inseridas na figura 3.1.

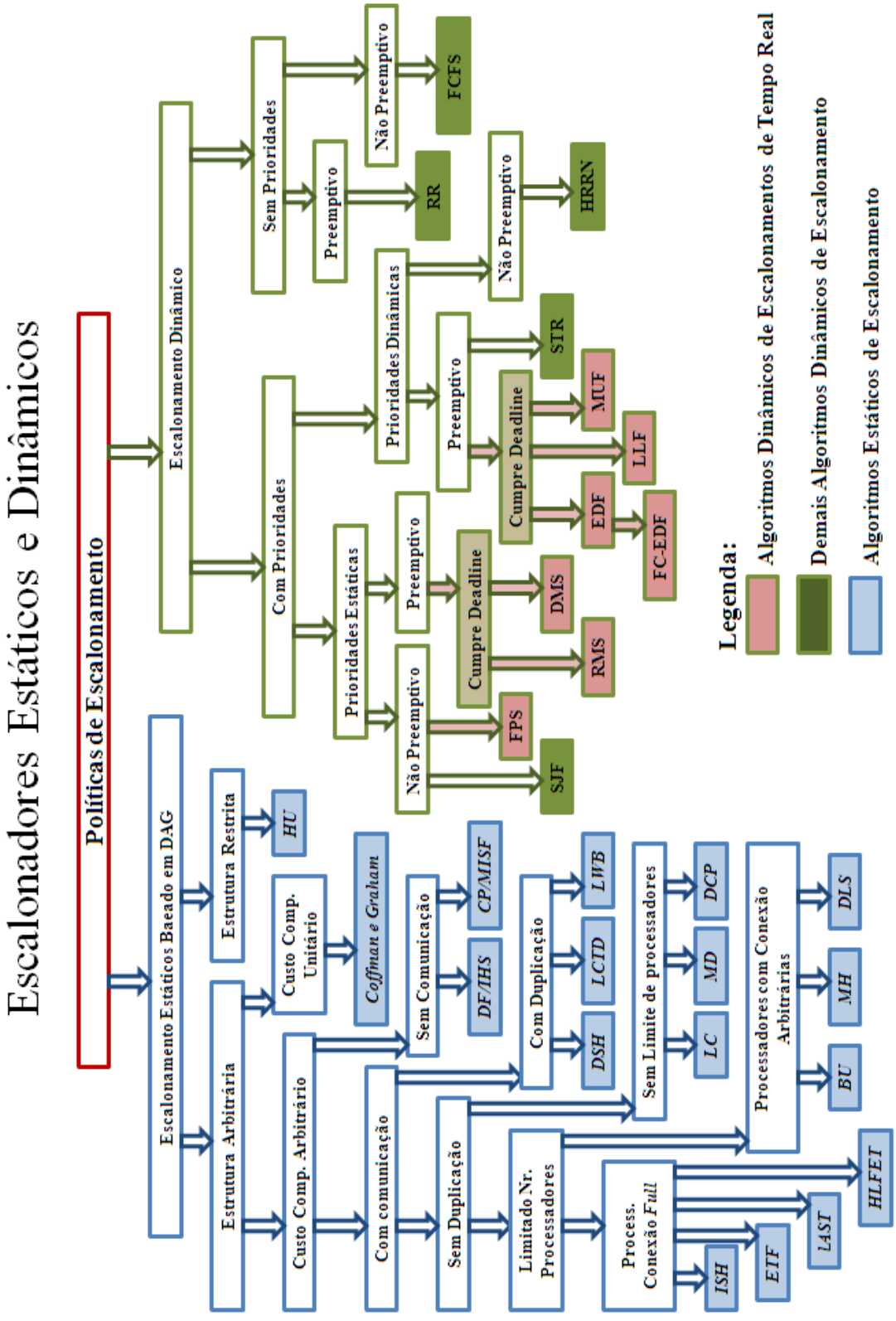

Figura 3.1: Taxonomia com os principais escalonamentos estáticos e dinâmicos [10, 34].

Tabela 3.1: Algoritmos de escalonamento dinâmicos [34].

RR	*Round Robin*
FCFS	*First Come First Served*
HRRN	*Highest Response Ratio Next*
SRT	*Shortest Remaining Time*
MUF	*Maximum Urgency*
LLF	*Least Laxity First*
EDF	*Earliest Deadline*
FC-EDF	*Feedback-Earliest Deadline*
DMS	*Dealine Monotonic Scheduling*
RMS	*Rate Monotonic Scheduling*
FPS	*Fixed Priority*
SJF	*Shortest Job First*

Tabela 3.2: Algoritmos de escalonamento estáticos [10].

HU	*Hu's Algorithm for Tree-Structured DAGs*
Coffman and Graham	*Coffman and Graham's Algorithm for Two-Processor Scheduling*
DF/HIS	*Depth-First with Implicit Heuristic Search*
CP/MISF	*Critical Path/ Most Immediate Successors First*
DSH	*Duplication Scheduling Heuristic*
LCTD	*Linear Clustering with Task Duplication*
LWB	*Lower Bound*
LC	*Linear Clustering*
MD	*Mobility Directed*
DCP	*Dynamic Critical Path*
BU	*Bottom Up*
MH	*Mapping Heuristic*
DLS	*Dynamic Level Scheduling*
ISH	*Insertion Scheduling Heuristic*
ETF	*Earliest Time First*
LAST	*Localized Allocation of Static Tasks*
HLFET	*Highest Level First with Estimated Times*

3.2 Grafos Dirigidos Acíclicos - GDA

Um programa paralelo pode ser representado por um GDA, *G*= (*V, E*), onde *V* são os nós e *E* são as arestas do grafo. Um nó do GDA representa uma tarefa definida na seção 2.1. O peso do nó representa o custo de computação da tarefa e é denotado por *w*. As arestas representam as trocas de informações entre as tarefas, sendo que o direcionamento da aresta indica a dependência. O nó fonte de uma aresta é denominado nó pai e o nó destino, denominado nó filho. Um nó sem entradas é chamado de nó de entrada e um nó sem arestas de saídas é chamado nó de saída. O peso de cada aresta representa o custo de comunicação (tempo) entre as mesmas. Este peso pode chegar a zero se as tarefas estão sendo executadas na mesma máquina uma vez que a comunicação intra-processador é desprezível comparado com o custo de comunicação entre processadores de máquinas distintas. As ligações de precedência de um GDA indicam que os nós fontes necessitam ser finalizados para que o nó destino inicie sua execução [10]. A figura 3.2 ilustra um exemplo de um GDA onde temos uma aplicação composta por sete tarefas, onde T1 e T7 são os nós de entrada e saída, respectivamente.

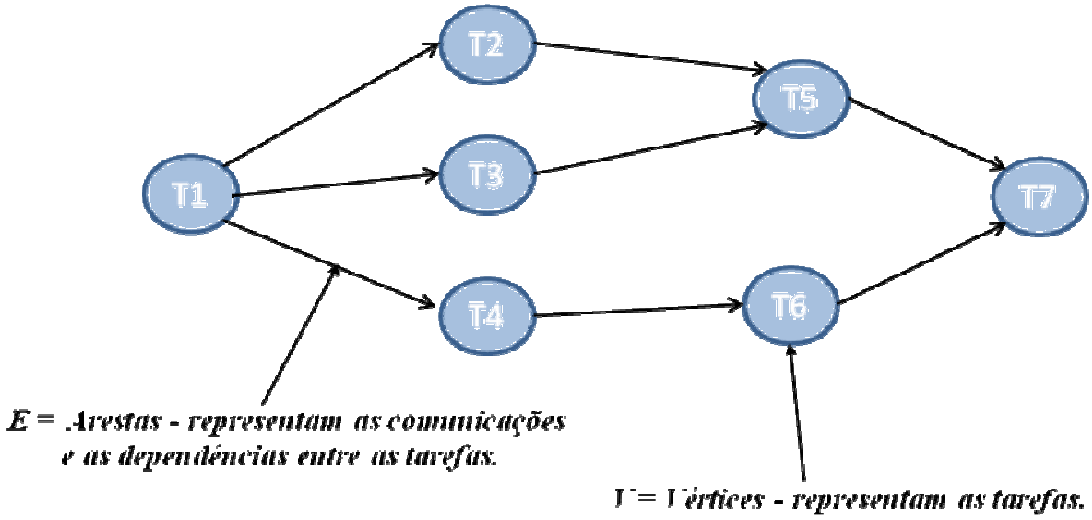

Figura 3.2: GDA – Grafo dirigido acíclico [46].

O tempo de finalização de um programa paralelo é normalmente chamado de comprimento de escalonamento ou *makespan*. Existem ainda algoritmos de escalonamento de grafos dirigidos que utilizam algumas variações do objetivo de minimização do tempo de finalização da tarefa, como por exemplo, a minimização do tempo médio de finalização de cada tarefa.

Algumas variações do modelo GDA genérico são descritas a seguir [46]:

a) Modelo exato - em um modelo exato, o peso de um nó é a soma dos seguintes instantes: o momento da computação, o momento de receber mensagens antes da computação, e o momento de emitir mensagens após a computação. O peso de uma aresta é uma função que depende da distância entre a fonte e os nós de destino e conseqüentemente da alocação dos nós e da topologia da rede. Depende também da disputa por banda de rede e esta característica pode ser difícil de modelar. Quando dois nós são atribuídos a um único processador, o peso da aresta torna-se zero para o envio e recebimento de mensagens.

b) Modelo aproximado 1 - neste modelo, o peso da aresta é aproximado por uma constante [46], independente da distância da transmissão da mensagem e pela disputa por banda de rede. Uma rede completamente conectada sem disputa se ajusta neste modelo.

c) Modelo aproximado 2 - neste modelo, o tempo de envio e recebimento de mensagens é ignorado, além disso, o peso da aresta é aproximado por uma constante [46]. Estes modelos que efetuam a aproximação são mais adaptáveis às seguintes situações:

- O tamanho do grão do processo é muito maior do que o tempo de envio ou recebimento da mensagem;
- As comunicações são garantidas por algum *hardware* dedicado de modo que o processador gaste uma quantidade de tempo insignificante em uma comunicação;
- O tempo da transmissão da mensagem tem pouca variação em relação a distância da transmissão da mensagem;
- A rede não está sobrecarregada.

No geral, os modelos aproximados (modelo 1 e modelo 2) podem ser usados para granularidade do tipo mediana a grande, desde o maior tamanho do grão do processo, até a menor comunicação, e conseqüentemente a rede não está sobrecarregada. A segunda razão para usar os modelos aproximados é que os pesos do nó e da aresta são obtidos através de estimativas, possuindo uma exatidão fraca. Assim, um modelo exato não é inútil quando os pesos dos nós e das arestas não são exatos [46].

Um GDA pode ser escalonado numa rede de processadores de duas formas [46]:

a) Mapeamento direto – O GDA é mapeado diretamente na rede de processadores de acordo com a topologia da mesma, usando o modelo exato. A classe de algoritmos APN (*arbitrary processor network)* [10] foram projetados para mapeamento direto. Ex.: MH.

b) Mapeamento indireto – O escalonamento do GDA nos processadores é efetuado utilizando um dos modelos de aproximação e ignorando a topologia de rede dos processadores. Neste caso é assumido que os processadores estão totalmente conectados.

Na técnica de mapeamento por topologia, os processadores são mapeados de acordo com topologia dada. Esta técnica tem por objetivo diminuir o impacto da disputa por banda de rede. Entretanto, a maioria dos algoritmos de escalonamento está baseada na técnica de escalonamento através de lista. O escalonamento através de lista é uma classe de escalonamento heurístico na qual as prioridades são atribuídas aos nós e estes inseridos em uma lista cuja ordem de prioridade é descendente. O nó com a prioridade mais elevada será escalonado antes do nó com a prioridade mais baixa. Se mais de um nó possuir a mesma prioridade, esta dependência de prioridades, indicando a ordem da lista é quebrada usando algum critério definido pelo algoritmo. Dois principais métodos podem ser utilizados para designar a prioridade a um GDA [46]: o *nível-t (top level)* e o *nível-b (bottom level)*. O *nível-t* de um nó n_i é o tamanho mais longo do trajeto de um nó de entrada e o nó n_i no GDA excetuando-se o nó n_i. Neste caso, o comprimento do caminho é a soma de todos os pesos dos nós e das arestas ao longo do caminho. O *nível-t* de n_i é altamente correlacionado com o tempo de início mais cedo de n_i, denotado (pelo que é determinado) depois que o n_i é alocado a um processador. O *nível-b* de um nó n_i é comprimento do maior caminho de n_i a um nó de saída. O *nível-b* de um nó n_i é limitado pelo tamanho do caminho crítico. Um caminho crítico (CP – *critical path*) de um GDA é um caminho de um nó de entrada a um nó de saída, cujo comprimento seja o máximo [46].

3.3 Optimalidade

Um algoritmo de escalonamento dinâmico só é aceitável se ele sempre produzir um escalonamento viável para um conjunto qualquer de tarefas, desde que um algoritmo estático com conhecimento prévio das mesmas tarefas também o fizer [41].

Em sistemas de tempo real *hard*, onde as restrições de tempo não podem ser relaxadas, o desempenho algumas vezes é erroneamente relacionado com o atendimento destes requisitos. Embora as restrições de tempo real muitas vezes expressem níveis de desempenho desejados, não são diretamente relevantes para a avaliação e comparação de projetos [47].

Nos algoritmos estáticos de escalonamento de GDAs, são conhecidos somente três casos especiais em que se obtêm o resultado do escalonamento em tempo polinomial de execução [10]:

a) Escalonamento de um grafo de tarefas com estrutura em árvore com custos de computação uniformes e um número arbitrário de processadores.

b) Escalonamento de um grafo de tarefas arbitrárias com custos de computação uniforme em dois processadores.

c) Escalonamento de um grafo de tarefas *interval-ordered* [10] com pesos uniformes nos nós e um número arbitrário de processadores.

Em todos os três casos citados são desprezados o tempo de comunicação entre as tarefas (assumido como zero).

Existem diversas maneiras de se mensurar a eficiência dos escalonamentos de tempo real e as algumas das formas são:

- Comprimento total do escalonamento ou *makespan* – É o intervalo de tempo necessário para executar o conjunto de tarefas escalonadas [9].
- Número de processadores necessários – Calculado de acordo com a Potência Computacional Excedente [11] que determina a quantidade mínima de processadores para atender a necessidade de processamento de acordo com os requisitos das tarefas mapeadas no GDA.
- Níveis de utilização dos processadores – Apresentam os índices de uso dos processadores.
- *Throughput* – Calculado de acordo com número de tarefas processadas por unidade tempo.

3.4 Relaxação

A relaxação de uma tarefa, por definição, indica sua urgência de processamento [11]. O seu valor é expresso pela diferença temporal entre o *deadline* e o tempo de execução da tarefa. O algoritmo LLF utiliza o critério da menor folga ou relaxação para priorização das tarefas.

Se for utilizada a informação da folga ou relaxação no escalonamento de tarefas, num programa representado por um grafo dirigido acíclico, este conceito limita-se às tarefas pertencentes ao mesmo nível e que estejam prontas para execução. Neste ambiente, o limite

de tempo ótimo para execução de um conjunto de tarefas (pertencentes ao mesmo nível do grafo) é dado pelo tempo do caminho crítico do grafo para aquele nível. Então, o algoritmo que consiga finalizar a execução das tarefas sem atrasar o início de suas tarefas filhas, ou seja, atender ao tempo de *deadline* das tarefas para todos os níveis do GDA [1] irá atender aos requisitos do conjunto de tarefas. As tarefas com a menor folga, considerando o tempo de execução da tarefa em relação ao *deadline* do respectivo nível do GDA, são priorizadas para execução.

A relaxação $l(t)$ da tarefa T, no instante t é definida com a seguinte fórmula:

$$l(t) = d - c(t) - t \tag{3.1}$$

onde:

- d – *deadline* da tarefa.
- c(t) – tempo de execução da tarefa a partir do instante t.
- t – instante t refere-se ao início da contagem do tempo.

Sendo D o intervalo de tempo entre o instante atual ou o *arrival time*, aquele que for o mais tarde, até o *deadline* da tarefa, temos que $D = d - t$. Então a relaxação pode ser também representada por:

$$l(t) = D - c(t) \tag{3.2}$$

A partir do cálculo da relaxação da tarefa, temos os seguintes possíveis resultados para o valor da relaxação:

- $l(t) > 0$: Há tempo suficiente para a tarefa ser executada e terminada no limite especificado, salvo contratempos.
- $l(t) = 0$: A tarefa está em execução ou vai iniciar imediatamente para completar sem interrupção.
- $l(t) < 0$: O *deadline* da tarefa fatalmente será desrespeitado.

3.5 Mapa de Gantt

Em 1910, Gantt desenvolveu uma ferramenta para mostrar o progresso de um projeto na forma de um mapa especializado. A aplicação representada foi a construção de um

navio. Atualmente, esta ferramenta é muito utilizada na representação de projetos de uma forma geral. O mapa de Gantt é bidimensional sendo que no eixo horizontal está representada a grandeza de tempo, que pode ser de forma absoluta ou relativa e no eixo vertical está representado um conjunto de atividades cujo progresso é demonstrado por barras, respeitando o tempo indicado no eixo horizontal [36].

A definição do mapa de Gantt pode ser apresentada através da representação abstrata de processadores [1]. Considerando PID (*Processor Identification*) como um conjunto de identificadores de processadores e PROC (*Processor*) como um conjunto de todos os possíveis processadores, onde cada elemento do conjunto é uma trinca (*pid*, T-SET, SCH):

onde:

pid: é o identificador único de um determinado processador, e *pid* $\subset$ PID;

T-SET: é o conjunto de tarefas alocado ao processador *pid.* T-SET: $\mathbb{P}$ TASK ($\mathbb{P}$ é o *power set* de TASK e TASK é um conjunto de todas as possíveis tarefas).
SCH: é uma função parcial de TASK em $\mathbb{O}_f$, SCH:TASK $\mapsto \mathbb{O}_f$.

$\mathbb{O}_f$: é um conjunto finito de intervalos finitos de Tempo Real, disjuntos, abertos à direita, isto é, $\mathbb{O}_f = \{ \theta \in \mathbb{O} \mid card(\theta) \in \mathbb{N} \}$, $\mathbb{O}_f \subset \mathbb{O}$;

θ: um intervalo de tempo real aberto à direita, $[t_1, t_2)$;
$\mathbb{O}$: é o conjunto de subconjuntos de $\mathbf{\Gamma}$, $\mathbb{O} \subset \mathbb{P}\,\mathbf{\Gamma}$;
card(θ): é a cardinalidade de θ.

A figura 3.3 mostra uma representação de SCH que é o mapeamento do escalonamento de um conjunto de tarefas numa linha de tempo.

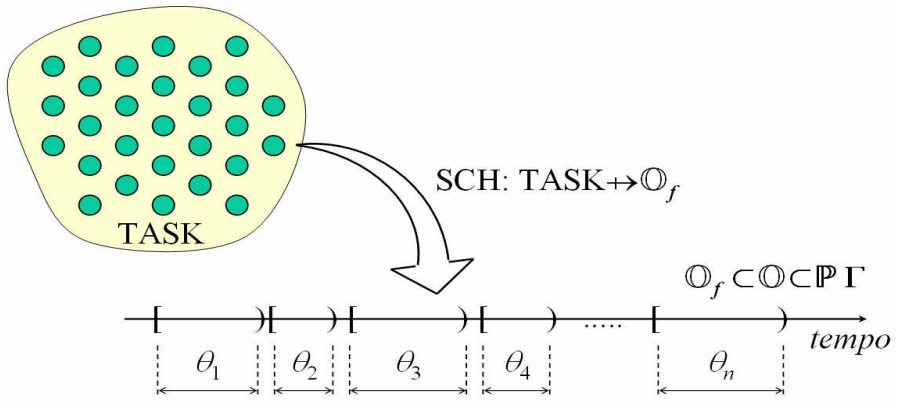

Figura 3.3: Representação de SCH.

O conjunto de números reais $\mathbb{R}$ é indicado como conjunto ideal para representar intervalos de tempo real. Seja $\mathbf{\Gamma}$ o subconjunto de todos os possíveis intervalos abertos à direita nesta representação, então:

$$\mathbf{\Gamma} : \mathbb{P}(\mathbb{PR}) \tag{3.3}$$

$$\mathbf{\Gamma} \triangleq \{ [t_1,t_2) \bullet t_1,t_2 : \mathbb{R} \} \tag{3.4}$$

onde :

$\mathbb{P}$ indica o *power set* do conjunto;

$$[t_1,t_2) = \{ t \bullet t : \mathbb{R} \mid \{t_1 \leq t < t_2 \} \} \tag{3.5}$$

A função SCH também pode ser representada de forma diagramática conforme a figura 3.4 que apresenta um escalonamento envolvendo duas tarefas τ_i e τ_j, num único processador.

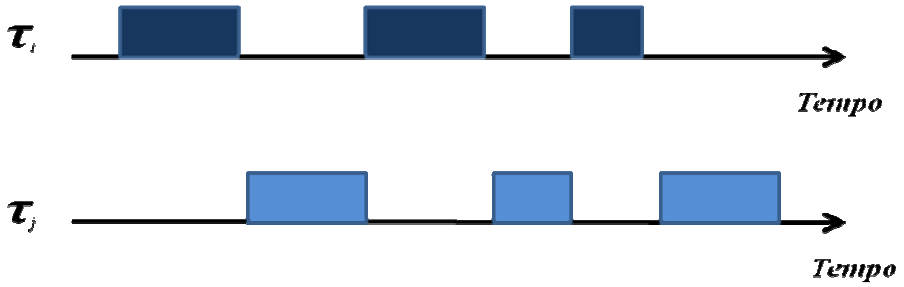

Figura 3.4: Representação do escalonamento de duas tarefas em um processador

Com base nas informações constantes na figura 3.4, está representada na figura 3.5 uma forma alternativa de conhecer um mapa de Gantt de um processador. Um mapa de Gantt pode ser definido como uma função relacionada a SCH do seguinte modo:

$$\text{G-CHT} : \mathbf{\Gamma} \mapsto \text{TASK} \tag{3.6}$$

$$\text{G-CHT} = \{(x,\tau) \bullet x : \mathbf{\Gamma}; \boldsymbol{\tau}\text{: TASK} \mid \exists \theta : \mathbb{O}_f \bullet (\tau,\theta) \in \text{SCH} \wedge x \in \theta\} \tag{3.7}$$

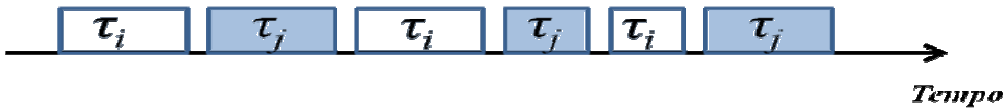

Figura 3.5: Representação de um mapa de Gantt

As trincas da forma (*pid*, T-SET, SCH) assumem PROC como sendo:

$$\text{PROC} \subseteq \text{PID} \times (\mathbb{P}\ \text{TASK}) \times (\text{TASK} \times \mathbb{O}_f)$$

Cada trinca em PROC é uma representação abstrata do processador identificado por *pid*. Dado um processador p : PROC, p.T-SET representa o conjunto de tarefas a ser escalonado em p.

3.6 Algoritmos de Escalonamento

Um algoritmo de escalonamento tem por objetivo realizar o ordenamento temporal de um conjunto de tarefas, gerenciando a atualização da fila de execução, dinamicamente ou não, dependendo da técnica utilizada.

Os algoritmos são compostos por duas funções: o despachante e o escalonador. Um escalonamento estático necessita de todas as informações sobre as tarefas a *priori* e o escalonamento dinâmico não, uma vez que as configurações podem sofrer alterações em tempo de execução, são flexíveis e possuem baixo determinismo. Os escalonamentos dinâmicos ou *on-line* por demandarem mais cálculos e por possuírem menor determinismo não apresentam bom desempenho em situações de sobrecarga do sistema [1].

Para garantir os *deadlines* das tarefas, os escalonamentos exigem que o índice de utilização do processador não seja superior a 1 (um). Liu e Layland [41] demonstraram que a utilização total do processador, para um conjunto de n tarefas, é calculada pela seguinte fórmula:

$$U = \frac{\sum_{i=1}^{n} C_i}{min(D_i, T_i)} \tag{3.8}$$

onde:

- U = Índice de utilização de CPU.
- C_i = Tempo de execução da Tarefa *i.*
- D_i = *Deadline* da Tarefa *i.*
- T_i = Período da Tarefa *i.*

Algoritmos de escalonamento com prioridade fixa como *Rate Monotonic* e dinâmicos como *Earliest Deadline First* e *Least Laxity First* são eficientes se aplicados a sistemas monoprocessados com preempção [31], entretanto, podem ocorrer falhas de alocação e conseqüentemente o não atendimento do *deadline* se determinados parâmetros das tarefas não forem conhecidos a *priori*. É sabido que não são conhecidos algoritmos de escalonamento ótimos sem um conhecimento prévio das características de cada tarefa [11].

3.6.1 *Earliest Deadline First* - EDF

É um algoritmo de escalonamento dinâmico baseado na execução da tarefa com o *deadline* mais próximo. Para efetuar o escalonamento, o algoritmo ordena as tarefas numa fila de prioridades. Sempre que um novo evento ocorre (nova tarefa, término de uma tarefa), o escalonamento procura na fila de tarefas àquela com o *deadline* mais próximo e habilita essa tarefa para ser executada [48].

Nos casos onde as tarefas possuem *deadlines* iguais aos períodos de re-escalonamento, a utilização do processamento chega próxima a 100 %. Ou seja, EDF pode garantir que as tarefas são atendidas de acordo com a capacidade do ambiente. Se comparar o EDF com escalonamentos de prioridades fixas, como RM, pode-se garantir que todos os *deadlines* serão atendidos [48].

Entretanto, quando o sistema está sobrecarregado, ou seja, algumas tarefas irão ultrapassar seu *deadline,* é impossível prever quais tarefas serão penalizadas. Seria necessária uma função específica para controlar o momento exato em que ocorre a sobrecarga do sistema e descartar as tarefas com *deadline* ultrapassado e com menor prioridade. Esta é uma desvantagem considerável para a arquitetura de um escalonamento dinâmico. Por ser um algoritmo de difícil implementação em *hardware*, EDF não é um escalonamento comum em sistemas industriais embarcados de tempo real [34].

Seja um sistema de três tarefas que são executadas em um processador e são escalonadas pelo algoritmo EDF. Para certificar-se de que estas tarefas não terão seus

deadlines ultrapassados é possível calcular a taxa de utilização do processador, de acordo com os parâmetros de cada tarefa, conforme apresentado na tabela 3.3.

Tabela 3.3: Exemplo de três tarefas e seus tempos de execução e períodos.

Tarefas	Tempo de execução	Período de execução
T1	1,0	8,0
T2	2,0	5,0
T3	4,0	10,0

A taxa de utilização *U,* aplicada no conjunto de processadores com base na fórmula 3.8 apresentada neste capítulo será: (1/8 + 2/5 + 4/10) = 0,925. Como o Limite teórico de qualquer processador é 100%, concluímos que o sistema é viável, uma vez que todos os *deadlines* serão respeitados.

A figura 3.6 apresenta o resultado do escalonamento efetuado pelo algoritmo EDF, sem a utilização de preempção, simulando a execução das três tarefas periódicas descritas na tabela 3.3, em um processador.

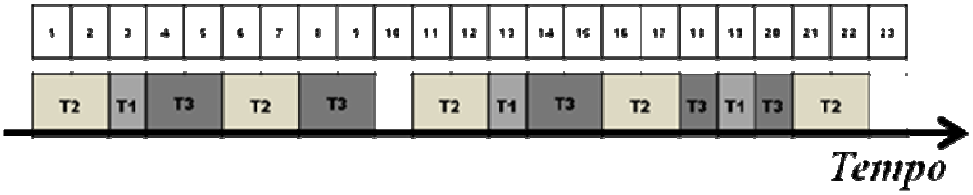

Figura 3.6: Mapa de Gantt com o resultado do escalonamento EDF, sem preempção

3.6.2 *Least Laxity First* - LLF

É o algoritmo mais usado em sistemas embarcados, especialmente com multiprocessadores. Trata-se de um escalonamento dinâmico e é baseado na execução da tarefa com a menor relaxação [31]. Para simplificar o algoritmo, os desenvolvedores definem uma restrição: cada tarefa deve ter o mesmo tempo de execução independente do processador. Este algoritmo apresenta melhor comportamento em sistemas que possuem tarefas não periódicas, uma vez que o algoritmo não trata prioridades em eventos que acontecem periodicamente. Uma das fraquezas do LLF é não ter o controle de tarefas futuras, ou seja, trabalha somente no estado atual do sistema. Por isso, podem existir situações em que o algoritmo deixa o processador ocioso durante o espaço de tempo em que o acesso a algum recurso do equipamento estiver sobrecarregado [43].

A figura 3.7 apresenta o resultado do escalonamento efetuado pelo algoritmo LLF, sem a utilização de preempção, simulando a execução das três tarefas periódicas conforme descrito na tabela 3.3, em um processador.

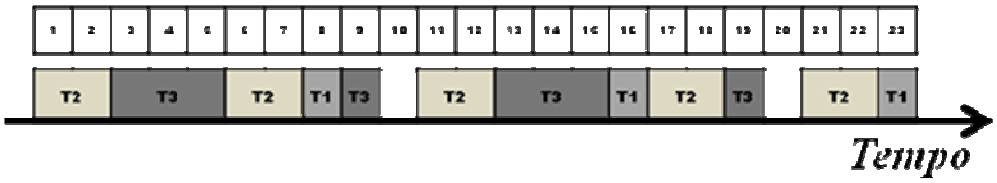

Figura 3.7: Mapa de Gantt com o resultado do escalonamento LLF, sem preempção

3.6.3 Diferenças entre os Algoritmos EDF e LLF

Embora EDF e LLF sejam considerados algoritmos ótimos em ambientes monoprocessados, podem ser considerados em ambientes multiprocessados somente sob determinadas situações [11].

Para os propósitos de implementação em *software*, o algoritmo EDF possui um desenvolvimento mais simples. Numa situação de sobrecarga no sistema, não há controle da perda dos *deadlines* das tarefas (ocorre o efeito dominó de perda de *deadline* das tarefas). Outra desvantagem do algoritmo EDF está na impossibilidade de detectar antecipadamente que uma tarefa perderá seu *deadline.*

O algoritmo de escalonamento LLF tem um custo de execução maior devido ao cálculo da relaxação e ao grande número de trocas de contextos causados por duas situações: quando ocorre alteração da relaxação durante o período de execução ou quando duas ou mais tarefas apresentarem a mesma relaxação. A vantagem deste algoritmo é que pode-se detectar a perda do *deadline* de uma tarefa, ainda durante a execução da mesma.

3.7 Algoritmos Adaptativos

Existem diversas propostas de algoritmos de escalonamento dinâmico para sistemas de tempo real multiprocessados a partir dos estudos efetuados nos algoritmos EDF e LLF. Serão apresentados cinco algoritmos de escalonamentos e analisados as vantagens e desvantagens de cada solução, citando as características positivas as possíveis falhas de escalonamento.

3.7.1 Escalonamento *Risk Incursion Potential Function* - RIPF

Trata-se de uma forma de escalonamento utilizando a noção de RIF (*Risk Incursion Function*) e é um escalonamento orientado a *deadline* o qual evoluiu do escalonamento de prioridade fixa [14]. Escalonamentos orientados a *deadline* produziram uma abordagem de programação como o TMO (*Time-triggered and Message-triggered Object*) [14]. Esta característica permite aos programadores especificar janelas de tempo com início e término de *deadline* em segmentos de computação em tempo real de maneiras mais convenientes. E é mais natural para os programadores de tempo real pensar em janela de tempo de início e término de *deadline* do que em números de prioridades. Numa abordagem diferente, os escalonamentos podem ser projetados para refletir não somente especificações de tempo associados com vários segmentos de computação de tempo real, mas também, considerar na priorização as especificações dos impactos e danos sobre as várias violações de tempo da aplicação. Este impacto pode variar muito de acordo com o tipo de aplicação utilizada.

Uma das funções básicas para manter a QoS (*Quality of Service*) solicitada pelos requisitos das aplicações é especificar o impacto dos danos causados pelas várias violações de tempo nestas aplicações. Kim e Liu [14] estabeleceram a noção de RIF como o *framework* para a especificação de tais impactos dos danos. Basicamente, o impacto potencialmente danoso incorrido pela violação do tempo é chamado de risco.

Serão apresentados dois tipos básicos de RIF e dois algoritmos de escalonamento específicos criados para refletir as especificações RIF.

Uma RIF é associada com cada função de saída do sistema e indica a quantidade de risco contraído [14]. Como por exemplo, a alocação de recursos, onde cada recurso possui

um risco de pré-avaliado de acordo com a semântica da aplicação e a alocação do *time slice* da CPU sendo, neste caso, a granulosidade muito fina e de difícil implementação.

Na figura 3.8, consideramos os tipos básicos de RIF [14]: *Hard deadline,* no qual o risco varia de zero a um nível crítico se o *deadline* não for atendido. *Soft deadline,* quando o risco inicia de zero e vai aumentando de uma maneira côncava, se o *deadline* não for atendido.

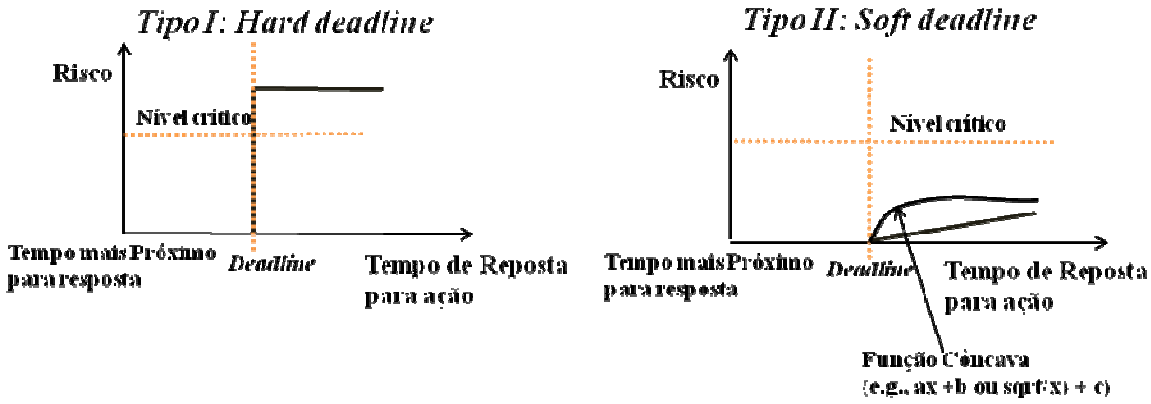

Figura 3.8: Tipos básicos de risco abordados por RIF [14].

Os *deadlines* referentes às saídas das tarefas são decididos com base nas considerações das restrições físicas e o RIF é definido de acordo com a criticidade das saídas.

A partir dos dois tipos de riscos identificados, conforme figura 3.8, foram desenvolvidos dois algoritmos RIPF-LLF e RIPF-Relaxação e ambos funcionam em dois passos, entretanto são escalonados pelo algoritmo LLF no primeiro passo. No segundo passo é verificado o potencial risco de cada tarefa, priorizando àquelas cujo risco é mais crítico. A diferença entre os dois algoritmos está na forma como o risco é aplicado no segundo passo. No primeiro algoritmo, o valor é atribuído diretamente e no segundo, o valor atribuído ao risco é divido pelo tempo de relaxação da tarefa.

O algoritmo RIPF-LLF apresentou bons resultados e o algoritmo RIPF-Relaxação apresentou resultados melhores com custo ligeiramente maior no tempo de execução. Kim e Liu [14] compararam o algoritmo RIPF-LLF e o algoritmo EDF. Em condições normais todas as tarefas foram completadas. Em condições severas onde existem perdas de *deadlines*, percebe-se que as perdas são diferentes para cada tipo de algoritmo (EDF e RIPF). A tabela 3.4 mostra os levantamentos dos tempos percebidos na execução de uma aplicação de teste

composta de cinco tarefas independentes, com riscos distintos cadastrados pelo usuário e os resultados dos tempos de execução de cada tarefa:

Tabela 3.4: Levantamento dos tempos de execução da aplicação exemplo [14].

	RIPF	*Deadline*	Tempo de execução	Violação do *deadline* com RIPF	Violação do *deadline* com. EDF	Período
Tarefa 1	1	60 *ms*	10 *ms*	141	0	100
Tarefa 2	2	60 *ms*	6 *ms*	86	4	100
Tarefa 3	150	70 *ms*	25 *ms*	0	133	200
Tarefa 4	180	40 *ms*	18 *ms*	0	0	100
Tarefa 5	100	80 *ms*	27 *ms*	5	50	500

As figuras 3.9 e 3.10 apresentam respectivamente os escalonamentos efetuados pelos algoritmos EDF e RIPF. São analisados quatro casos de composição das tarefas e com base nos diagramas, é fácil observar quais tarefas correm o risco de perder seus *deadlines.*

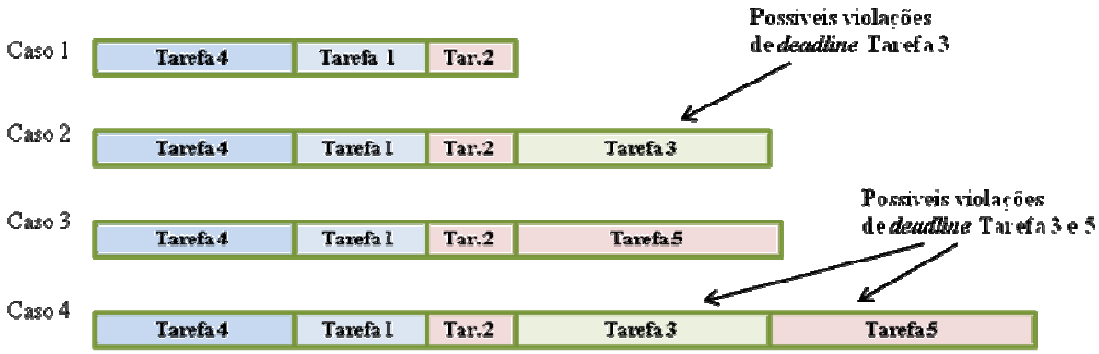

Figura 3.9: Escalonamento das tarefas utilizando o algoritmo EDF [14].

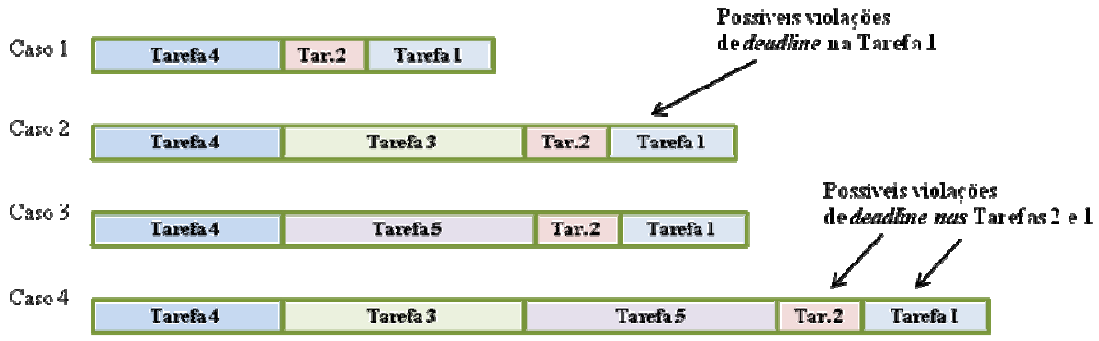

Figura 3.10: Escalonamento das tarefas utilizando o algoritmo RIPF [14].

Verifica-se na figura 3.9, que o EDF escalonou as tarefas de acordo com o *deadline*. Devido ao intervalo de execução das tarefas 3 e 5 serem respectivamente 200

milissegundos e 500 milissegundos, os mesmos não entraram no caso 1. No caso 4, é grande a possibilidade das tarefas 3 e 5 perderem seus *deadlines*.

De forma semelhante, na figura 3.10, RIPF escalonou as tarefas de acordo com o valor do risco contraído. Há grandes chances das tarefas 1 e 2 perderem seus *deadlines* no caso 4.

Verificamos que o escalonamento EDF sacrificou as tarefas 3 e 4, porque seus *deadlines* eram os maiores e o escalonamento RIPF sacrificou as tarefas 1 e 2, porque seus riscos contraídos são menores. Neste exemplo, é relevante qual tarefa será sacrificada, então, as perdas dos *deadlines* das tarefas 3 e 4 são mais catastróficas. Portanto, se não houver perdas de tarefas, ambos os algoritmos são apropriados mas, nos casos onde nem todos os *deadlines* serão cumpridos, RIPF é mais útil, porque considera o potencial risco de cada tarefa e descarta os menos importantes.

Considerando escalonamentos voltados a *deadline*, embora seja a melhor alternativa escalonar tarefas com prioridades fixas [14], não consideram muitos requerimentos de qualidade e semântica, os quais são importantes no projeto da aplicação, principalmente nos impactos das violações dos *deadlines* [14].

3.7.2 Escalonamento *Adaptive Greedy Task Scheduler* - A-GREEDY

Trata-se de um escalonamento orientado a *space-sharing* e pode ser implementado usando a estratégia de dois níveis: Um nível de *kernel*, onde os processos são atribuídos aos processadores de acordo com a disponibilidade e um nível de usuário o qual escalona as tarefas pertencentes a um dado processo. Este modelo de execução com processos possuindo tarefas concorrentes e dependentes entre si pode ser representado como um grafo dirigido acíclico, detalhado na seção 3.2, onde cada ponto do GDA representa uma tarefa de tempo unitário e as arestas representam as dependências entre as tarefas. Uma tarefa se torna pronta para execução quando todas as suas predecessoras já tenham sido executadas. Cada processo tem seu próprio escalonamento de tarefas e este escalonamento funciona de maneira *on-line*.

Neste contexto, o número de processadores atribuídos a um processo pode variar durante a sua execução e o escalonamento de tarefas precisa ajustar estas alterações de recursos de acordo com os processadores disponíveis. Para uma eficiência global, o escalonamento de tarefas deve fornecer um *feedback* sobre a utilização dos processadores para o algoritmo de alocação de processos, a fim de evitar que um processador fique ocioso

[15]. O escalonamento sugerido utiliza um algoritmo adaptativo que fornecerá um *feedback* contínuo para o algoritmo de atribuição de processos aos processadores. Este algoritmo denominado A-GREEDY utiliza pelo menos uma fração constante dos processadores distribuídos [15]. No exemplo utilizado para análise do A-GREEDY, o algoritmo de alocação de processos, troca informações com o escalonamento de tarefas e também administra as políticas de distribuição de processadores. Leiserson et al [15] mostraram que a performance do escalonamento de tarefas é boa para a maioria dos casos através de uma técnica simulando a execução do algoritmo. No nível de *kernel*, o algoritmo aloca os processos nos processadores e efetua uma espécie de blefe ao outro nível (de usuário), disponibilizando para o escalonamento de tarefas um grande número de processadores, que não estão realmente disponíveis. Os dois níveis conseguiram ajustar o problema sem apresentar falha.

A maioria dos trabalhos sobre escalonamentos de tarefas, focados em ambientes multiusuário e multitarefa, utilizam escalonamentos não adaptativos e a atribuição dos processos aos processadores distribui um número fixo de processadores em todo o tempo de vida do processo. Um processo cujo paralelismo não é conhecido antecipadamente pode consumir ciclos de processamento para alterá-lo durante a sua execução, caso um processo com baixo paralelismo seja distribuído para mais processadores do que ele possa efetivamente e produtivamente utilizá-los. Além disso, nos ambientes multiusuário e multitarefa, o escalonamento que não é adaptativo não permite o início de novos processos a qualquer instante, se todos os processadores já estiverem sido alocados, uma vez que os processos existentes e em execução podem estar utilizando ou vir a utilizar estes processadores.

Com escalonamentos adaptativos, a alocação de processos em processadores permite de alterar o número de processadores distribuídos em tempo de execução. Portanto, novos processos podem ser inseridos, uma vez que o algoritmo de alocação requisita processadores já alocados e os redistribui para os novos processos.

A solução A-GREEDY é apresentada através de um algoritmo adaptativo, onde o escalonamento de tarefas fornece um *feedback* ao algoritmo de alocação de processos a fim de que, quando um processo não possa usar muitos processadores, estes processadores possam ser realocados para outros processos que têm necessidade. Devido a este *feedback*, o escalonamento adaptativo altera a distribuição de processadores de acordo com a disponibilidade.

3.7.3 Escalonamento *Security and Heterogeneity-driven Scheduling Algorithm* - SHARP

Trata-se de um algoritmo de escalonamento de tempo real com suporte a *clusters* heterogêneos e implementa as funcionalidades de encriptação, integridade e autenticação nas trocas de informações das tarefas que são escalonadas [17].

Um dos desafios do processamento paralelo é a dificuldade de sincronização de relógios em *clusters* heterogêneos e a necessidade desse sincronismo em aplicações paralelas. Outro ponto é o aumento do número de aplicações paralelas que demandam segurança nos dados (prevenção de acesso não autorizado), e que o processamento exige alta performance para execução. *Clusters* heterogêneos executam diversas aplicações que por sua vez, permitem o acesso de diversos usuários e nem todos tem boas intenções.

Escalonamentos orientados à segurança são voltados para ambientes *grid* e o escalonamento SHARP embora tenha funções de segurança é voltado para *clusters* heterogêneos. Devido às diferenças entre o ambiente centralizado (*cluster*) e descentralizado (*grid*), os algoritmos de escalonamento possuem características diferentes: em *grid,* os algoritmos de escalonamento não são eficientes para aplicações de tempo real, enquanto que o algoritmo SHARP suporta aplicações concorrentes com restrições de tempo; em *grid,* os algoritmos não levam em consideração a quantidade de execução com falhas no escalonamento e o algoritmo SHARP propõe um modelo integrando quantidade de execução com níveis de segurança [17].

Xie et al [17] desenvolveram um modelo prático para mensurar o *overhead* que a implementação dos serviços de segurança geram no algoritmo. O algoritmo propõe uma maneira de resolver o problema da formulação do escalonamento orientado à segurança, de forma que os tempos necessários para escalonamento e execução das tarefas e o *overhead* gerado pela segurança não ultrapassassem o tempo de *deadline* de cada tarefa.

O algoritmo SHARP [17] tem como premissa o atendimento do *deadline* das tarefas e minimizar o grau de deficiência de segurança. O algoritmo calcula o tempo de início mais cedo para as tarefas e o *overhead* da segurança de cada tarefa é verificado pela medição dos tempos que cada item necessita para execução, verificando através do somatório de todos tempos levantados se os *deadlines* são respeitados. Se não for possível atingir o *deadline* das tarefas, a aplicação é rejeitada.

A complexidade de tempo do SHARP é $O(nmq)$, onde:

- n é o número de nós do *cluster;*
- m é o número de tarefas da aplicação paralela;

• q é o número de serviços de segurança implementados (autenticação, encriptação e integridade).

Xie et al [17] compararam o algoritmo SHARP com os algoritmos EDF e LLF. O objetivo foi verificar o impacto na *performance* do novo algoritmo. Foi utilizado como infra-estrutura um *cluster* com 128 nós. Os testes foram realizados durante três meses, totalizando 6.400 tarefas na simulação. Foram verificadas as informações de tempo de início, tempo de execução, número de processadores utilizados em cada tarefa, *deadline* e os requisitos de segurança das tarefas e dos dados. O levantamento dos tempos foi contado de forma separada: tempo de execução, tempo gasto com os serviços de segurança, etc. Foi observado que o algoritmo SHARP é melhor que LLF e EDF nos quesitos: de segurança, uma vez que o algoritmo SHARP é orientado à segurança e os algoritmos LLF e EDF não possuem esta característica; taxa de execução, o algoritmo SHARP também é melhor, uma vez que nos testes, foram adicionadas mais tarefas para execução e SHARP escolhe os nós com menor tempo de execução total, diferente dos algoritmos EDF e LLF que somente escolhem os nós que oferecem menor tempo de execução. Por isso o algoritmo SHARP oferece um maior número de escalonamentos viáveis para aplicações concorrentes.

3.7.4 Escalonamento *Enhanced Least Laxity First* - ELLF

Para superar os problemas encontrados nos algoritmos EDF e LLF, destalhados nas seções 3.6.1 e 3.6.2 respectivamente, Blumenthal et al [18] propuseram o algoritmo ELLF, que tem por objetivo combinar os benefícios de cada um dos algoritmos EDF e LLF. Semelhante aos algoritmos base, ELLF é considerado melhor, pois utiliza 100% de CPU e a determinação da perda dos *deadlines* é verificada antes de sua ocorrência. Para minimizar a utilização de CPU na execução do algoritmo, foi introduzido um co-processador (uma placa aceleradora). Este co-processador também pode ser usado com aplicações de interface em *RT-Linux* (*Real Time Linux*) e pelo *framework* YASA [18].

Este algoritmo é uma variação do algoritmo LLF. É mais complexo e efetua o escalonamento em dois passos: O 1º. Passo é semelhante ao LLF, ou seja, é escolhida a tarefa com a menor relaxação, entretanto, caso exista mais de uma tarefa com a mesma relaxação, passa para o 2º. Passo. Neste passo, entre as tarefas com a menor relaxação, é escolhida aquela que tiver o *deadline* mais próximo, e nas demais tarefas (com a menor relaxação) é ajustada uma nova variável denominada: estado de execução. Com este parâmetro, as demais

tarefas são impedidas de concorrer com as tarefas que estão em execução, evitando a preempção. Isso faz com que diminuam as quantidades desnecessárias de troca de contexto que é típico no LLF.

Como este algoritmo apresenta um *overhead* em relação ao original, uma maneira de melhorar a performance do sistema é mover as funções do algoritmo de escalonamento (em todo ou em parte) para um dispositivo adicional de processamento (co-processador).

Dentre as operações de escalonamento, as mais recomendadas para serem executadas no co-processador são:

I) Computação *online* das prioridades;

II) Comparação das prioridades;

III) Seleção da tarefa para ser executada. É mais produtivo manter esta última operação no co-processador, uma vez que já dispomos do resultado da comparação.

Embora as duas últimas operações não requeiram muito processamento é recomendado manter todas as três tarefas executando no co-processador para evitar a necessidade de transferir o controle da execução e comunicação das tarefas para o sistema operacional. Obviamente a introdução de um novo *hardware* só é justificável se a performance apresentada for satisfatória. A arquitetura do co-processador para algoritmos de prioridade dinâmica, como EDF, LLF, ELLF, está ilustrado na figura 3.11.

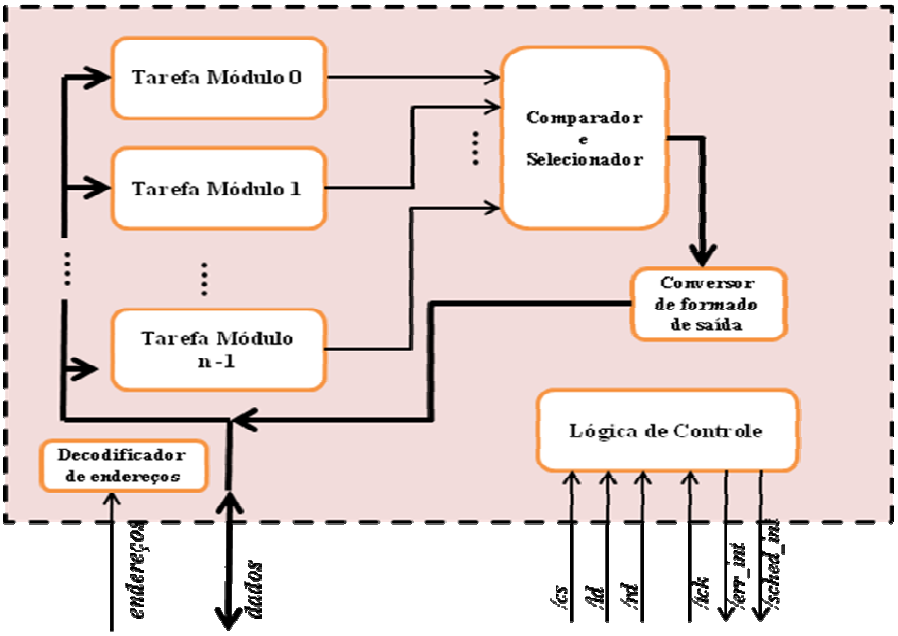

Figura 3.11: Arquitetura de um co-processador [17].

Cada tarefa é representada por um bloco funcional que determina os valores das prioridades e armazena-os inclusive com os estados das tarefas. Essas informações podem ser modificadas durante o tempo de execução. Exemplo: EDF utiliza o parâmetro *deadline* para escalonamento, este dado é calculado automaticamente (decrementado) enquanto a tarefa estiver ativa. A perda do *deadline* acontece quando este valor chega a zero e a tarefa ainda está ativa. Em algoritmos mais complexos que utilizam a relaxação da tarefa, como o LLF e o ELLF, estes parâmetros também são calculados automaticamente enquanto a tarefa é executada. A relaxação da tarefa é utilizada para ordená-las na fila de execução. Estas operações são feitas simultaneamente para todas as tarefas durante o escalonamento e via *hardware*, faz com que seja aumentada a velocidade de execução comparada com a solução implementada por *software*.

O objetivo da comparação das prioridades das tarefas é a determinação da ordem na fila de execução e a forma de comparação depende do algoritmo utilizado. Em qualquer caso o co-processador retorna uma identificação da tarefa, resultado do comparador. Embora o co-processador possa ser utilizado para mais de um tipo de algoritmo de escalonamento (EDF, LLF), a interface de retorno para o sistema operacional é sempre a mesma.

3.7.5 Escalonamento *Adaptive Work-Stealing Thread Scheduler* - A-STEAL

Este escalonamento foi projetado para ambientes multiusuário e multitarefa. A utilização destes equipamentos só os faz atraentes, se executadas cargas de trabalho multiprogramadas onde muitas aplicações paralelas compartilham a mesma máquina [20]. Podem ser implementados escalonamentos para estas máquinas usando dois níveis: um atribuindo processos nos processadores no nível de *kernel*, e outro escalonando tarefas no nível de usuário. A atribuição de processos pode ser implementada utilizando a técnica *space-sharing* (tarefas ocupam recursos de processador separados), ou *time-sharing* (tarefas diferentes podem compartilhar os mesmos recursos de processador em tempos diferentes). Além disso, os algoritmos de atribuição de processos e escalonamento de tarefas podem ser adaptativos, onde o número de processadores atribuídos a um processo pode alterar-se em tempo de execução, ou não adaptativo, onde o processo executa por completo em um número fixo de processadores.

A-STEAL [20] é um escalonamento adaptativo de tarefas do tipo *work-stealing* que trabalha de uma forma descentralizada para efetuar o escalonamento das tarefas, foi

inspirado num algoritmo A-GREEDY, detalhado na seção 3.7.2, cujo escalonamento das tarefas é efetuado de forma centralizada.

A técnica *work-stealing* provou ser um modo efetivo para projetar as atribuições de processos em processadores e efetuar os escalonamentos de tarefas tanto teoricamente como na prática. O funcionamento do escalonamento de tarefas é descentralizado e não tem conhecimento de todas as tarefas que estão executando em um determinado momento. Sempre que um processador possui disponibilidade para executar outros trabalhos, solicita o trabalho de outro processador que é escolhido de forma aleatória [18].

O escalonamento A-STEAL fornece um *feedback* sobre a situação dos processos, em intervalos regulares, para o algoritmo de atribuição de processos do tipo *space-sharing* no momento da requisição de processadores. Esta requisição é denominada de *sheduling quanta* [20]. Baseado neste *feedback* o algoritmo de atribuição de processos pode alterar a distribuição dos processos nos processadores de acordo com a nova disponibilidade. Leiserson et al [20] mostraram que A-STEAL é eficiente, minimizando tempo de execução e ciclos desperdiçados do processador comparado com o algoritmo A-GREEDY, detalhado na seção 3.7.2.

4 SIMULADORES PARA CONTROLE E MONITORAÇÃO DE ESCALONAMENTOS

São ferramentas que controlam a execução de aplicações, monitorando o escalonamento das tarefas e verificando se os seus *deadlines* serão atendidos. Permitem a visualização e um melhor entendimento dos escalonamentos e apresentam os resultados dos escalonamentos através de uma interface gráfica, amigável para o usuário.

Estes *frameworks* podem controlar a execução da aplicação ou simplesmente simular sua execução, baseando-se nas informações prestadas pelo usuário (quantidade de tarefas, tempo de execução, dependência entre as tarefas, etc). Independente da forma de funcionamento do *framework*, para uma efetiva análise do resultado de um escalonamento gerado pelo sistema, o *framework* utilizado deve fornecer as seguintes informações:

- Relatórios com os resultados da execução (ou simulação).
- Mapa de Gantt demonstrando o escalonamento das tarefas.
- GDA com a visualização das tarefas e seus relacionamentos.

A execução da aplicação possa ser efetuada por diversos escalonamentos e os respectivos resultados possam ser avaliados.

4.1 *Framework*s analisados

Foram analisados três *frameworks* que estão disponíveis para utilização sem necessidade de pagamento de direitos autorais e foi definido o PM^2P, o qual será utilizado para testes e implementação das propostas deste trabalho.

4.1.1 *Framework: Yet Another Schedulability Analyser - YASA*

O *framework* YASA [18, 27] vem sendo desenvolvido desde 2002, pela Universidade de Rostok, Alemanha. Trata-se de um *framework* versátil e de fácil utilização. Permite a seleção do escalonamento mais adequado, durante a fase de projeto, retirando do programador a necessidade de conhecer detalhes de implementação dos escalonamentos em diferentes ambientes. Por isso os escalonamentos ficam encapsulados pela API do YASA.

Esta propriedade facilita o uso do YASA em diversos sistemas operacionais [18]. A figura 4.1 detalha a arquitetura do *framework*:

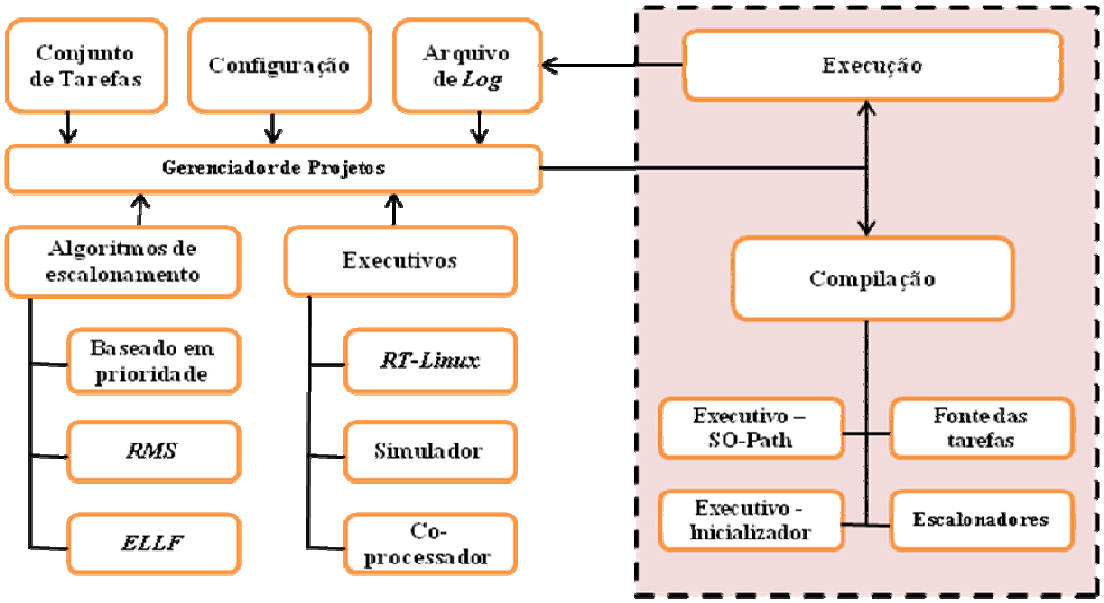

Figura 4.1: Arquitetura do *framework* YASA [18].

O ambiente é totalmente configurável por uma interface gráfica. É possível criar novas tarefas, atribuindo seus parâmetros: tempo de computação, *deadline*, período ou comportamento no caso de perda do *deadline*. Para cada tarefa, uma função de início pode estar vinculada ficando a critério do usuário sua definição. É necessário efetuar alguns ajustes como selecionar o executivo (módulo do *framework* com informações sobre o ambiente) e o escalonamento desejado para as máquinas alocadas. Após todas as informações serem inseridas no *framework* o projeto poderá ser compilado. O processo de compilação inclui todos os componentes selecionados como o executivo, escalonamentos, funções do projeto, informações das tarefas e código de início. Todas essas informações são *linkadas* de diferentes modos, dependendo do executivo utilizado. O resultado pode ser um executável monolítico ou um sistema com diversas bibliotecas compartilhadas. Em alguns casos há necessidade de compilar novamente o *kernel* do sistema operacional. Um mecanismo de *log* interno registra todos os passos como: a quantidade de vezes que o escalonamento é chamado, *deadlines*, troca de tarefas (*switches*) ou bloqueio de recursos (*locks*). Após o término da execução, o arquivo de *log* é avaliado pelo *front end* gráfico e o resultado pode ser visualizado através de uma estrutura de diagramas. Desta forma é possível gravar no *log* cada ação do RTOS (*Real Time Operating System*) [27].

O *framework* YASA é dependente das diferentes restrições de um sistema operacional. Não é possível executar ou compilar certos conjuntos de tarefas utilizando diversos executivos se estas tarefas usam recursos específicos do sistema operacional. Por exemplo: o uso da função *malloc*(), para alocar memória, é uma função de bloco. Por esta razão não está disponível no modo protegido (*kernel*) da maioria dos sistemas operacionais, diferente dos sistemas embarcados. Tarefas que usam esta função não conseguem compilar com sucesso se o executivo estiver rodando no modo protegido (*kernel*). Uma das soluções é o emprego de *kmalloc*(), que restringe o conjunto de tarefas para o tamanho máximo de páginas e assim o executivo consegue rodar no *Linux* e seus derivados. Outra restrição é dada pelo módulo de escalonamento. Se o sistema operacional suporta apenas escalonamentos baseados em prioridades, a seleção do escalonamento ficará restrita as propriedades do Sistema Operacional.

Os programas do módulo escalonamento são definidos na linguagem *C*. Estes módulos são compilados como bibliotecas compartilhadas ou *linkados* estaticamente. Os módulos são carregados e inicializados em tempo de execução durante a fase de início do executivo. Durante a chamada do módulo executivo, a função *schedule*() pertencente ao módulo escalonamento corrente, será executada cada vez que o escalonamento do sistema é chamado para determinar a próxima tarefa a ser executada. O resultado do processo de encapsulamento transforma a função real do escalonamento numa função muito pequena. Por exemplo: o escalonamento baseado em prioridades consiste de apenas 25 linhas de código fonte e é capaz de rodar em diferentes executivos.

O executivo descreve o ambiente no qual os conjuntos de tarefas e os escalonamentos escolhidos serão usados. Fornece uma API e esconde o sistema operacional do conjunto de tarefas. O acesso a diferentes estruturas e funções dos sistemas operacionais, como *threads* ou parâmetros de escalonamento só é possível via API. Além disso, funções internas adicionais como montadores de arquivos de *log* ou sincronização de conjunto de tarefas são incluídas na API.

Em sistemas operacionais reais o executivo é integrado na maioria das vezes via *path* no *kernel* original. O estado final do executivo depende do sistema operacional e dos detalhes de implementação. O *framework* possui dois executivos completamente diferentes: o simulador e o executivo *RT-Linux*.

O algoritmo de simulação é uma máquina virtual multiprocessada. Este executivo pode ser combinado com todos os escalonamentos disponíveis em software definidos no

YASA. Suporta diferentes métodos para manipular falhas de *deadline* e é capaz de usar diferentes protocolos para designar recursos como semáforos e *mutexes*. O simulador é independente do sistema operacional.

Após definir o conjunto de tarefas que compõe a aplicação, contendo as informações dos seus atributos, os escalonamentos e o conjunto de tarefas serão compilados em diversos objetos e *linkados* em um executável. Após a compilação, a aplicação gerada a partir das informações das tarefas, poderá será iniciada. Todas as informações dentro do simulador serão gravadas em *log*.

Uma das limitações do simulador é que ele não é capaz de simular a execução programas reais. A razão é que diferentes executivos podem ser executados em diferentes sistemas operacionais e equipamentos. Para executar uma aplicação real dentro do simulador, seria necessário emular o mundo real, em particular as API do sistema operacional usadas em cada conjunto de tarefas. Além disso, a interação com o ambiente e o comportamento dinâmico teriam que ser emulados.

Para obter resultados mais similares as aplicações reais durante a simulação, o YASA fornece métodos para definir os tempos de início de tarefas assíncronas e os tempos de alocação de recursos. Estes tempos são usados para simular bloqueios e tempos de *standby*. É necessário ter um conhecimento sobre o progresso das tarefas para obter resultados mais próximos da realidade. É por isso que o simulador é útil apenas para obter informações sobre o escalonamento em geral.

RT-Linux é uma extensão do *Linux* convencional para tempo real *hard*. A atual versão é a 3.0 e está disponível para diversas plataformas, como *x86* e suporta processadores SMP (*Symmetric multiprocessing*). Foi escolhido o *RT-Linux* devido ao padrão POSIX (*Portable Operating System Interface*) das API, a possibilidade de alterar o escalonamento original com pequeno esforço e pela disponibilidade do código fonte do sistema operacional.

O YASA pode ser executado no sistema operacional *RT-Linux*, que é uma extensão do *Linux* convencional para tempo real *hard.* Este sistema é iniciado com a carga de diversos módulos do *kernel*, como o *rtl_sched.o* ou *rtl_fifo.o*. O desenvolvedor pode alterar ou melhorar estes módulos se necessário. *RT-Linux* utiliza um escalonamento baseado em prioridade por *default*. Ou seja, somente as tarefas não suspensas e com a mais alta prioridade serão escolhidas para execução. Este escalonamento é muito rápido e simples, mas é estático. Nos períodos de ociosidade do sistema, se nenhuma tarefa de tempo real está executando será executado o *Linux* com a *thread* de menor prioridade. *RT-Linux* não trata as informações de

tempo de computação, tempo de execução e *deadline* das tarefas. Para resolver este problema foram adicionadas algumas variáveis para especificar escalonamentos, *threads* e estruturas de escalonamento para armazenar as informações necessárias. Isso é feito no executivo - *path* do sistema operacional do executivo *RT-Linux*. Não foram alteradas as API do *RT-Linux*, para evitar incompatibilidade com os projetos convencionais, mas foram adicionadas (*pathed*), algumas funções, como o *yasa_setscheduler*(), no módulo escalonamento original para garantir as funcionalidades dos projetos usando valores *defaults*, cujo uso é opcional.

Quando os desenvolvedores criam um novo projeto, o *framework* YASA cria um código fonte para o conjunto de tarefas, de acordo com as informações de cada tarefa. O YASA inicia o processo de compilação de todo o *kernel* do *RT-Linux* e alguns módulos como: *yp_project.o, ys_scheduler.o e logfilereader.o*. Os dois primeiros módulos são utilizados para iniciar o projeto e ajustar o escalonamento. Todas as tarefas serão iniciadas em modo suspenso. Após a iniciação com sucesso da aplicação, serão sincronizados todos os módulos do YASA e será efetuado o escalonamento de todas as tarefas de tempo real, sendo que as informações gravadas em *log* iniciarão com tempo de início igual a zero.

A maioria dos algoritmos de escalonamento usados no YASA necessitam de informações detalhadas das tarefas mas, principalmente do tempo de computação. Para simular o tempo exato de computação na aplicação é necessário verificar no equipamento: velocidade da CPU, memória, atividades de I/O, *caches,* etc. Se o tempo de computação calculado para o escalonamento for menor do que o tempo de execução real, muitos escalonamentos dinâmicos não funcionarão corretamente.

4.1.2 *Framework: Cheddar – A Flexible Framework*

Cheddar [28, 29] é uma ferramenta de escalonamento de tempo real, desenvolvida na universidade de *Brest*, *França.* O projeto encontra-se atualmente na sua versão 2.0. Tem como objetivo verificar e analisar restrições temporais de tarefas de sistemas de tempo real. O *framework Cheddar* é um simulador de tempo real composto de duas partes independentes:

a) Um editor gráfico usado para descrever aplicações de tempo real.

b) O *framework*, o qual possui uma interface para efetuar testes e a inclusão de novos algoritmos de escalonamento, permitindo a visualização dos resultados.

O *Cheddar* permite também ao usuário criar e inserir facilmente novos escalonamentos através do editor gráfico. Permite o compartilhamento de recursos. Calcula o

tempo de resposta e computação de todo o processo de execução da aplicação (fim-a-fim). Simula o escalonamento de tarefas com restrições de precedência, sendo que as tarefas podem ser cíclicas ou aperiódicas e permite a execução em diferentes processadores, compartilhando mensagens.

O *Cheddar* foi escrito na linguagem *Ada* e o editor gráfico foi construído utilizando a biblioteca *GtkAda* (*Graphical Toolkit for Ada*). É compatível com os sistemas operacionais *Solaris*, *Linux* e *Windows* [28].

O *framework Cheddar* possui dois módulos principais: um simulador e um teste de viabilidade. O teste de viabilidade permite que o usuário estude o comportamento das aplicações de tempo real, sem preocupar-se com o escalonamento. E o simulador pode ser usado inicialmente para executar um escalonamento e apresentar, de forma automática, as restrições das tarefas alocadas pelo escalonamento, como por exemplo, se todas as tarefas conseguiram ser executadas dentro das especificações de tempo exigidas.

Ferramentas de testes de viabilidade são menos complexas que *framework*s de escalonamento, mas a maioria delas é limitada, disponíveis somente para um conjunto pequeno de escalonamentos. Uma das características do *framework Cheddar* é integrar ao simulador outras ferramentas que facilitam o desenvolvimento de escalonamentos específicos e modelos de tarefas específicos, representando um sistema particular.

O *Cheddar* suporta os seguintes escalonamentos [29]:

- *Rate Monotonic* – RMA, RM ou RMS.
- *Earliest Deadline First* – EDF, detalhado na seção 3.6.1.
- *Deadline Monotonic* - DM, *Inverse Deadline*.
- *Least Laxity First* - LLF, detalhado na seção 3.6.1.
- Escalonamento *POSIX* 1003b e políticas de enfileiramento - SCHED_FIFO, SCHED_RR, SCHED_OTHERS.

A partir do resultado da simulação de uma aplicação de tempo real, obtemos as seguintes informações sobre o escalonamento:

- Pior, melhor, e média do tempo de resposta;
- Número de preempções e trocas de contexto;
- *Deadlocks e* inversões de prioridade*;*
- Perda de *deadlines*.

As características a seguir podem ser pré-definidas e aplicadas nos testes, para verificar a viabilidade do sistema analisado, nos casos de haver ou não preempção de tarefas:

- Limites para o tempo de resposta das tarefas para escalonamentos EDF, LLF, DM e RM;
- Índice de consumo de CPU para escalonamentos EDF, LLF, DM e RM;

A figura 4.2 mostra o resultado de uma simulação utilizando o *framework Cheddar*. Na parte superior desta figura, está o detalhamento do resultado do escalonamento de um sistema com três tarefas executado em um processador. Cada linha representa uma tarefa, e os segmentos em negrito representam execuções durante um período de tempo. Na parte inferior da figura temos o detalhamento das informações sobre as tarefas, como o tempo de resposta, perda de *deadline*, fator de utilização do processador, etc.

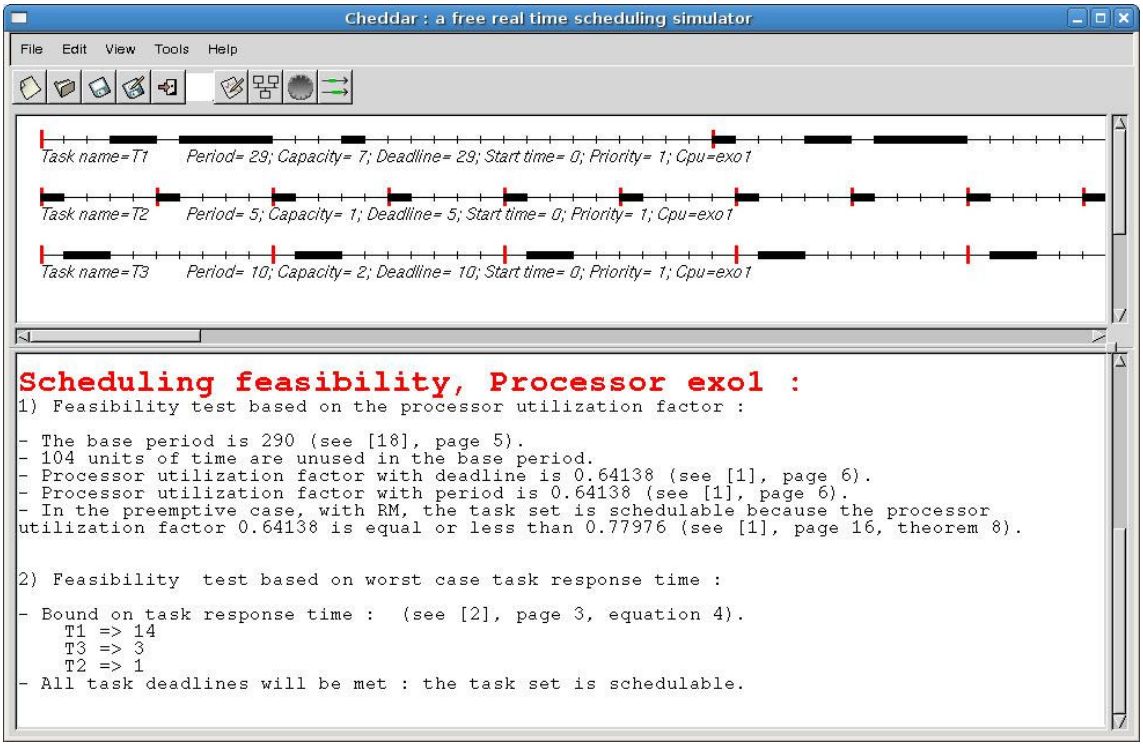

Figura 4.2: *Framework Cheddar*. Resultado de um escalonamento.

4.1.3 *Framework – Performance Monitoring of Message Passing* - PM^{2}P

O *framework* PM^{2}P [7, 26] foi projetado para simular a execução de aplicações em ambientes multiprocessados e também para acompanhar a execução de aplicações que rodam em paralelo num *cluster* de *workstations*, utilizando a biblioteca MPI (*Message Passing Interface*). Os pontos de checagem dos tempos de execução são definidos pelo usuário, através de chamadas inseridas no sistema com auxílio das portas paralelas.

Para realizar medições de intervalos de tempos entre eventos, em diferentes máquinas, foi montado um esquema utilizando as portas paralelas. Cada máquina que participa do *cluster* é conectada a máquina *front end* por uma das linhas de entrada da porta paralela. A latência da rede não afeta o monitoramento do sistema, pois não há comunicação pela rede envolvida no processo.

Para medir os intervalos de tempos entre eventos em uma aplicação distribuída, é necessário manter uma visão única da hora e o uso da porta paralela representa uma solução para o problema de ausência de relógios sincronizados entre as máquinas. Sempre que for desejado adicionar um evento de marcação de tempo em algum ponto de um programa, o programador precisa inserir uma chamada de função, que lê o valor corrente do primeiro *bit* da porta paralela e escreve de volta seu complemento. Como cada máquina é conectada a um pino diferente do conector da porta paralela do *front end*, é possível identificar qual máquina enviou o sinal pelo número do pino cujo valor foi alterado. Para captar a sinalização dos eventos enviados pelas máquinas escravas, o *front end* lê continuamente o conteúdo da sua porta paralela. Sempre que esse conteúdo mudar, o *front end* o armazena em memória juntamente com a leitura da hora do seu relógio local. Ao final do monitoramento, todos os pares de estado e hora armazenados são escritos em um arquivo cujo nome contém o índice da tarefa (ex: rank7.trc).

Esta ferramenta possui um conjunto de funcionalidades desenvolvidas com o objetivo de permitir que o usuário tome decisões a respeito de como melhor estruturar e dividir uma aplicação MPI a partir de informações de tempo de execução de sua aplicação. É possível executar e medir o tempo de execução de aplicações já existentes, simular a execução de grafos de tarefas, gerar grafos aleatórios a partir de parâmetros especificados pelo usuário, visualizar resultados em mapas de Gantt e dividir as tarefas de uma aplicação entre os processadores do *cluster*, entre outros.

É possível também editar grafos de precedência de tarefas através de uma interface gráfica que permite a definição de vértices e arestas, representando tarefas e comunicações, respectivamente. A inserção dos vértices e arestas para editar ou construir grafos pode ser removida posteriormente. A figura 4.3 apresenta um exemplo de um grafo editado utilizando o PM^2P. As informações a respeito das tarefas podem ser editadas selecionando-se a tarefa desejada através do *mouse*. A tela apresentada na figura 4.4 permite a edição do identificador da tarefa, o tempo de execução (caso seja conhecido), a localização do executável associado, *deadline*, prioridade e uma caixa de escolha para definir se é uma tarefa

inicial ou não. Nem todas as informações precisam ser obrigatoriamente preenchidas para realizar as diversas funcionalidades possíveis. A única restrição com relação às tarefas é que não é permitido atribuir o mesmo índice a duas tarefas distintas do mesmo grafo.

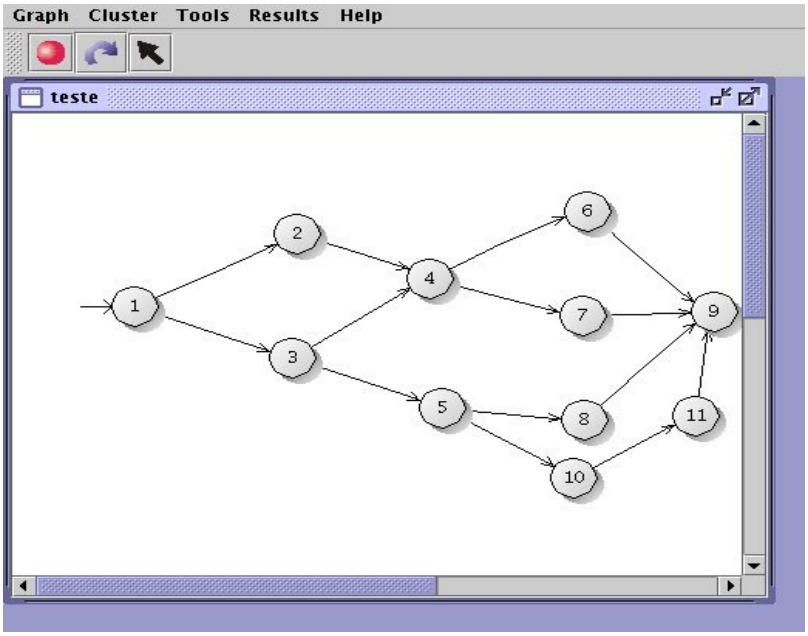

Figura 4.3: *Framework* PM^{2}P. Grafo editado pela ferramenta.

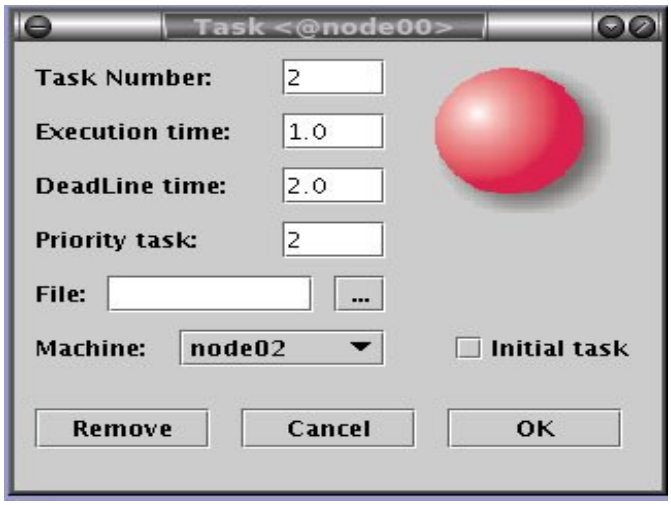

Figura 4.4: *Framework* PM^{2}P. Janela com informações sobre uma tarefa.

As comunicações entre as tarefas possuem informações associadas e é definido o número de itens e o tipo de dado a ser transmitido. O tempo de comunicação não é definido pelo usuário, mas sim calculado com informações obtidas do sistema. A figura 4.5 apresenta a

tela para entrada de informações a respeito das arestas. É possível ainda inverter o sentido de arestas e removê-las. No caso das tarefas, se uma é excluída, todas as arestas incidentes a ela também são removidas.

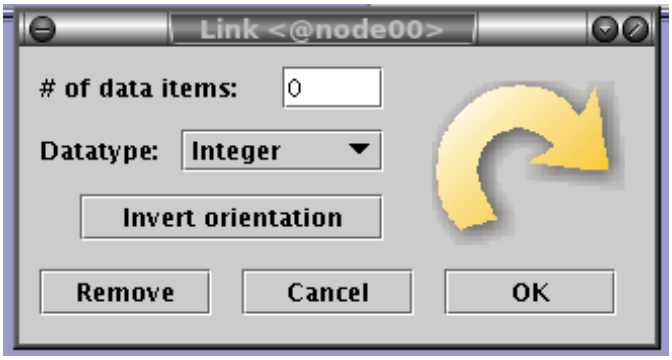

Figura 4.5: *Framework* PM^{2}P. Janela com informações sobre comunicação.

Além da entrada manual de grafos, a ferramenta gera grafos pseudo randômicos automaticamente, atendendo a especificações do usuário, para que possam ser usados em simulações e testes de algoritmos de escalonamento. Para gerar um grafo pseudo randômico, o usuário deve especificar o número de tarefas desejadas, o número máximo de tarefas por nível, o tempo máximo de computação de cada tarefa, o número máximo de arestas chegando a cada vértice e o número máximo de itens de dados enviados em uma comunicação. A partir destas informações, um grafo é gerado e apresentado em uma nova janela. A tela para entrada de informações sobre o grafo a ser gerado é apresentado na figura 4.6.

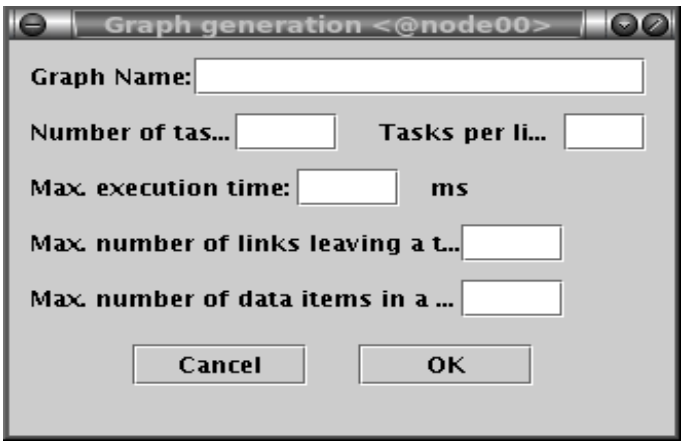

Figura 4.6: *Framework* PM^{2}P. Janela para geração randômica de grafos.

A figura 4.7 apresenta um grafo de cinqüenta tarefas gerado pela ferramenta, utilizando a opção pseudo randômica.

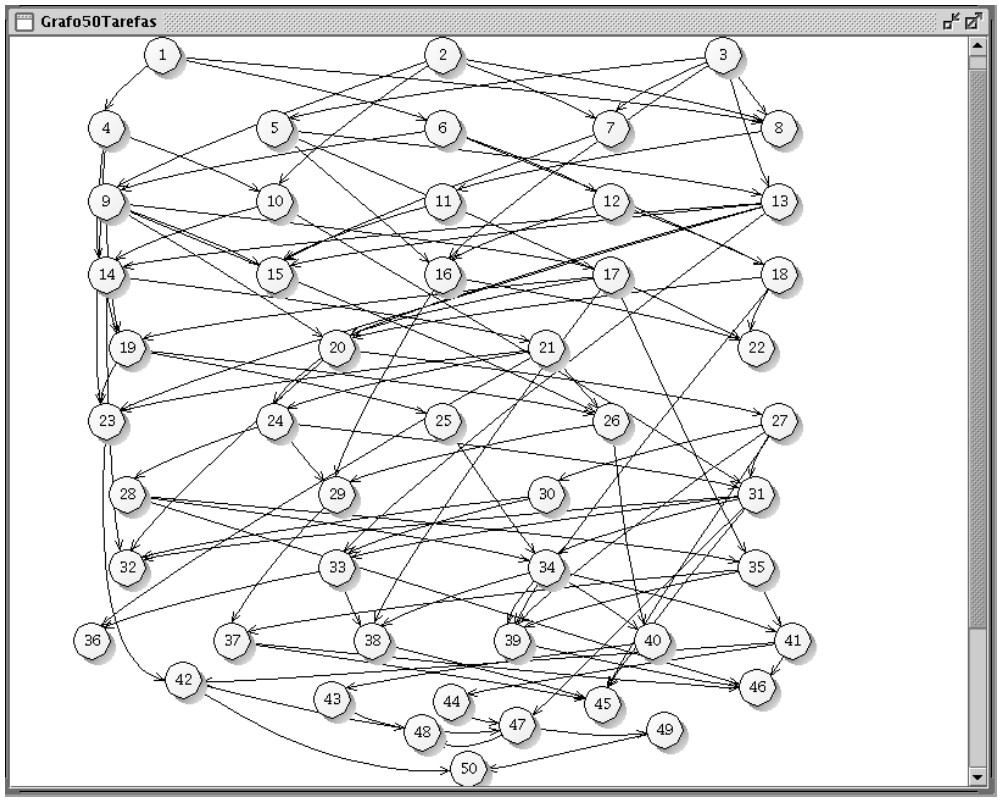

Figura 4.7: *Framework* PM^{2}P. Grafo de cinqüenta tarefas.

Para executar uma aplicação distribuída ou realizar outras operações sobre um grafo, é preciso definir um *cluster* sobre o qual a aplicação será executada. A especificação das máquinas pertencentes ao *cluster* pode ser realizada de duas maneiras: a partir de um arquivo de sistema, ou manualmente, caso tal arquivo não esteja disponível. Em qualquer momento é possível adicionar, remover, testar o seu funcionamento ou obter o número de indentificação da porta paralela à qual cada máquina escrava está conectada ao *front end*. Cada uma possui uma cor associada definida pelo usuário, e que é utilizada para preencher os vértices correspondentes às tarefas que estão mapeadas. A figura 4.8 mostra a janela de informações sobre o *cluster* com duas máquinas cadastradas. Inicialmente elas não estão ativas, pois é preciso testá-las para que a ferramenta verifique o seu funcionamento. Após a checagem das informações e o seu funcionamento, o usuário deve marcar a máquina como em uso, para que ela seja utilizada no escalonamento.

Figura 4.8: *Framework* PM^{2}P. Cluster com duas máquinas cadastradas.

Quando uma aplicação é executada através do PM^{2}P, as informações de tempo de execução de todas as tarefas são coletadas e gravadas em arquivos. Se a aplicação for escrita inteiramente pelo usuário, a ferramenta assume que as *macros* para leitura da hora tenham sido inseridas no início e término do programa. As horas marcadas no meio do programa são consideradas marcadores de eventos internos e são opcionais, enquanto que as horas marcadas no início e término do programa são marcadores obrigatórios. Para executar uma aplicação, a ferramenta cria o arquivo *procgroup* contendo uma linha por tarefa da aplicação, segundo o modelo definido pelo MPICH. A máquina *front end* não pode participar na execução das tarefas. O programa *start_machine* é então executado em todas as máquinas, criando as áreas de memória compartilhada de todos os processos e enviando sinais pela porta paralela. A aplicação é então executada a partir da tarefa inicial e do arquivo *procgroup* criado. Ao final de sua execução, o programa *end_machine* é executado em todas as máquinas, gravando as horas armazenadas na memória compartilhada em arquivos, e enviando sinais pela porta paralela. A ferramenta então analisa todos os arquivos gerados e obtém informações para apresentação dos resultados. A cada execução da aplicação, os valores dos tempos de computação da tarefa podem ou não ser atualizados segundo a vontade do usuário.

Cada tarefa pode ser alocada a uma máquina ativa do *cluster*. A atribuição de tarefas às máquinas pode ser feita manualmente pelo usuário. Entretanto, esta tarefa se torna entediante para grafos muito grandes, e, portanto, foram implementados os algoritmos de escalonamento estáticos, HLFET (*Highest Level First with Estimated Times*), ETF (*Earliest Time First*), LLFMDAG (*Least Laxity First Multiprocessor for Directed Acyclic Graph*), EDFMDAG (*Earliest Deadline First Multiprocessor for Directed Acyclic Graph*) e LLFPMDAG (*Least Laxity First with Priority Multiprocessor for Directed Acyclic Graph*). A inclusão dos *deadlines* nas tarefas, caso seja necessário, é efetuada manualmente pelo usuário ou, caso este parâmetro esteja zerado, estas informações (*deadline* e relaxação) são calculadas

de acordo com os custos de execução, comunicação e precedência entre as tarefas inseridas no grafo. Os comandos de escalonamento estão sob o menu de ferramentas (*tools*).

Os algoritmos de escalonamento utilizados assumem que não há preempção das tarefas. Entretanto, como na prática todas as tarefas são inicializadas no início da aplicação, se duas tarefas estiverem prontas para execução em um dado instante, ambas irão competir pelo processador. As únicas restrições de tempo de início de execução que podem ser impostas são aquelas impostas pelas comunicações. No caso do mapeamento de tarefas manual, não é possível criar mapas de Gantt, pois as tarefas são apenas mapeadas, e não escalonadas em cada processador.

A partir de uma execução, um mapa de Gantt é gerado contendo uma linha para cada processador (figura 4.9). Os retângulos representando cada tarefa da aplicação são preenchidos com cores aleatórias. Se duas ou mais tarefas estão em execução simultaneamente (já foram iniciadas e ainda não terminaram e estão competindo pelo processador) em certo intervalo de tempo, a linha do processador no mapa de Gantt é dividida entre as tarefas. A figura 4.9 mostra um mapa de Gantt gerado pela ferramenta.

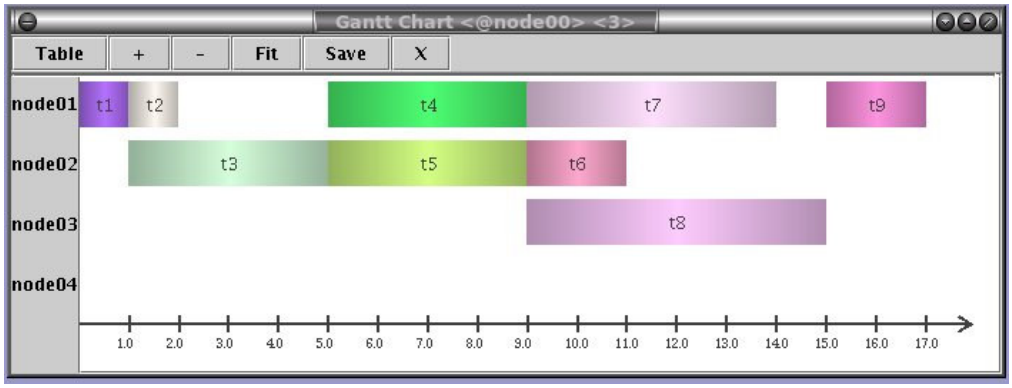

Figura 4.9: *Framework* PM^{2}P. Mapa de Gantt com nove tarefas.

5 PROPOSTA DE MONITORAÇÃO DIFERENCIADA E REESCALONAMENTO DE TEMPO REAL NO *FRAMEWORK* PM^{2}P

Conforme detalhado na seção 4.1.3, o *framework* PM^{2}P foi desenvolvido em *java* e algumas funções de controle em *C,* com o objetivo de simular e acompanhar a execução de programas paralelos utilizando a biblioteca MPI para troca de mensagens. O *framework* foi projetado para funcionar com um *cluster* arbitrário de máquinas conectadas entre si via TCP/IP (*Transmission Control Protocol/Internet Protocol*), de forma que as trocas de mensagens entre as tarefas executadas em paralelo utilizem este protocolo. Todo o controle sobre a execução das tarefas e da disponibilidade dos equipamentos no *cluster* é efetuado pela porta paralela a fim de evitar distorções na contagem de tempos e ser menos intrusivo para a aplicação paralela.

Para implementar a proposta de monitoração e re-escalonamento da aplicação caso a disponibilidade do *cluster* se altere, foi necessário codificar novas funções, inexistentes no *framework* versão 1.0 de 2002. Além do desenvolvimento da proposta foi necessário implementar três os novos algoritmos para auxilio nos testes de monitoração. A partir do desenvolvimento destas características, o *framework* PM^{2}P passou para a versão 2.0.

Por estar desenvolvido em *java*, o *framework* está estruturado em 7 pacotes (*packages*) com determinados propósitos e estes pacotes são compostos por um conjunto de classes (*class*). Os pacotes *project*, utilizado para inicializar o sistema, *ganttchart* que contém as classes para o desenho dos mapas de Gantt e *auxiliary* que contém os métodos básicos para a construção da interface gráfica, não sofreram alterações. O pacote *control*, responsável pelo controle e chamadas aos itens de *menu,* controle e criação dos grafos, *cluster* e tarefas, chamadas aos programas em *C* para colhimento e medição dos tempos e controle do simulador de tarefas, foram alterados em sua maioria. O pacote *model* o qual contém as estruturas das classes que definem o funcionamento do *cluster,* das respectivas máquinas, dos grafos com suas tarefas e conexões também necessitou ser re-estruturados para implementação da proposta de monitoração. O pacote *view,* responsável pelo desenho e apresentação da interface gráfica foi modelado nos itens de *menu* para acomodar as opções de acesso a documentação do sistema, chamada a execução de uma aplicação e chamada a execução do simulador. Foram também ajustadas as janelas de apresentação do *cluster*, máquinas e tarefas para inclusão dos novos parâmetros de prioridade e *deadline*. O pacote

scheduling, que contém os algoritmos de escalonamento, foi ajustado para acomodar os novos algoritmos de escalonamento, bem como os métodos para ordenamento das tarefas pertencentes ao grafo.

5.1 Procedimentos para utilização do *framework*

A fim de proporcionar um melhor entendimento na utilização do *framework,* especificamente quanto às novas características incluídas como a proposta de monitoração e re-escalonamento da aplicação e os novos algoritmos de escalonamento, foi montado um roteiro que define a seqüência dos principais passos, tanto aqueles que serão efetuados automaticamente pelo sistema, como aqueles que serão efetuados manualmente pelo usuário. Este roteiro está no anexo 1. Nele estão detalhadas as seqüências dos procedimentos de uma forma mais genérica, entretanto é necessário seguir a ordem sugerida para o correto funcionamento do *framework*. As novas implementações que foram incluídas neste trabalho estão referenciadas nesta seqüência de procedimentos e serão apresentadas em maiores detalhes, separadamente, sempre referenciando o item constante neste procedimento. A disposição dos passos, devido a necessidade de seguir a disposição do funcionamento da ferramenta, fez com que os itens executados automaticamente pelo *framework* e as atividades manuais a cargo do usuário, ficassem dispersos, sem uma distribuição lógica.

Os itens 9-a, 9-b, 10-a, 11-a, 12-a, 12-b, 13-a, 13-b, 14-a, 14-b, 14-c, 14-d, 14-e, 14-f, 14-g e 14-i referem-se às novas implementações efetuadas, ou seja, estes passos não existiam na versão anterior do *framework*. Já os itens 2-a, 3-a, 3-b, 4-b, 5-c, 6-a, 6-b, 6-d, 8-a, 8-b, 10-b, 10-c, 11-b, 11-c, e 14-h foram alterados com o objetivo de acomodar as implementações propostas como a proposta de monitoração e re-escalonamento da aplicação e os novos algoritmos de escalonamento.

5.2 Algoritmos implementados para auxílio no teste de monitoramento do *cluster*

A partir da análise dos algoritmos detalhados no capítulo 3, foram implementados três novos algoritmos de escalonamento de tarefas, EDFMDAG descrito na seção 5.2.1, LLFMDAG descrito na seção 5.2.2 e LLFPMDAG descrito na seção 5.2.3, para auxilio nos testes do monitoramento proposto no *cluster*, uma vez que o *framework* PM2P possui apenas dois algoritmos implementados. O desenvolvimento necessário para a inclusão destes

algoritmos refere-se aos ajustes e novas implementações dos itens 8-a, 8-b, 10-a, 10-c, 12-a, 12-b, 14-d, 14-e, 14-f, 14-g e 14-i constantes no procedimento detalhado no anexo I. Estes novos algoritmos estão voltados para o escalonamento de sistemas que possam ser representados por um GDA. Todos funcionam de forma estática e sem preempção de tarefas. Por conseguinte, os resultados podem variar na determinação da fila de execução de um sistema de tarefas para outro.

O funcionamento dos algoritmos está dividido duas fases que são realizadas seqüencialmente: A primeira fase é comum a todos e consiste na divisão do GDA em níveis, efetuando o ordenamento topológico, respeitando as dependências entre as tarefas.

A segunda fase varia de acordo com cada algoritmo, entretanto, existem alguns critérios comuns. O escalonamento é realizado somente nas tarefas que pertencem ao mesmo nível e estão prontas para execução, portanto, concorrentes e sem dependências entre si. As tarefas são alocadas aos processadores disponíveis, utilizando os critérios de menor ou maior tempo de execução. O limite de tempo para execução deste conjunto de tarefas é dado pelo tempo do caminho crítico do grafo para aquele nível. Os algoritmos consideram como tempo máximo para execução de cada tarefa, pertencentes ao conjunto de tarefas que estão no mesmo nível do grafo, o tempo de início de suas tarefas filhas, e define este tempo máximo como o *deadline* das tarefas. Os algoritmos de escalonamento dinâmicos tipo EDF e LLF são semelhantes aos algoritmos desenvolvidos quando o processo entra na segunda fase e são feitos os escalonamentos das tarefas pertencentes ao mesmo nível do grafo, uma vez que este grupo de tarefas, prontas para execução, não possuem dependência entre si e estão limitadas a um *deadline.* Os algoritmos desenvolvidos apresentam os melhores resultados de escalonamento quando o *makespan* gerado é igual ao caminho critico deste grafo. Estes tempos se alteram de acordo com o grafo de tarefas utilizado e disponibilidade do *cluster*.

O ordenamento topológico é necessário para atribuir prioridades às tarefas concorrentes e como se trata de um GDA, uma das soluções é criar uma lista de tarefas em ordem topológica ou topológica reversa. Estas duas técnicas, bastante utilizados para se atribuir prioridade são denominadas de *nível-t* (*top level*) e o *nível-b* (*bottom level*). O *nível-t* de um nó T_i qualquer é o tamanho do caminho mais longo entre um nó de entrada e T_i. O tamanho de um caminho é a soma de todos os pesos dos nós e das arestas ao longo do caminho. Assim, o *nível-t* de um nó está fortemente relacionado com o tempo de início mais antecipado de T_i, denotado por $T_s(t_i)$. O *nível-b* de um nó T_i é o tamanho do caminho mais longo entre T_i e um nó de saída. O *nível-b* de um nó é limitado superiormente pelo tamanho

de um caminho crítico do grafo. Um caminho crítico de um DAG é um dos caminhos mais longos do grafo [49].

A forma de calcular as prioridades do *nível-t* e *nível-b* dos nós de um grafo está detalhada e demonstrada em pseudocódigo. Estas duas formas de priorização de tarefas estão inseridas nos algoritmos EDFMDAG, LLFMDAG e LLFMDAG. O cálculo do *nível-t* foi modificado para acomodar o preenchimento do parâmetro *deadline* inserido nas tarefas, caso este parâmetro não tenha sido preenchido pelo usuário. Das tarefas pertencentes ao mesmo nível, o maior tempo de execução será o limite para a execução das demais tarefas. Caso este limite não seja alcançado pelo escalonamento, pode não se ter um escalonamento ótimo. Os pseudocódigos do *nível-t* e *nível-b*, bem como o detalhamento dos três algoritmos:

Cálculo do *nível-t*

Início do algoritmo

(1) Construir lista de nós em ordem topológica. Esta lista será chamada de *TopolList*.

(2) Para cada nó T_i em *TopolList* faça

max = 0

Para cada pai T_x de T_i faça

Se *nível-t*(T_x) + $w(T_x)$ + $c(T_x, T_i)$ > max então

max = *nível-t*(T_x) + $w(T_x)$ + $c(T_x, T_i)$

Fim se

Se $D(T_i)$ <= 0

$D(T_i)$ = max – (*nível-t*(T_x) + $c(T_x, T_i)$)

Fim se

Fim para

nível-t(T_i) = max

Fim Para

Fim do algoritmo

Cálculo do *nível-b*

Início do algoritmo

(1) Construir lista de nós em ordem topológica reversa. Esta lista será chamada de *RevTopolList*.

(2) Para cada nó T_i em *RevTopolList* faça

max = 0

Para cada filho T_y de T_i faça

Se $c(T_y, T_i)$ + *nível-b*(T_y) > max então

max = $c(T_y, T_i)$ + *nível-b*(T_y)

Fim se

Fim para

nível-b$(T_i) = w(T_i)$ + max

Fim para

Fim do algoritmo

5.2.1 Algoritmo EDFMDAG

O algoritmo EDFMDAG (*Earliest Deadline First Multiprocessor for Directed Acyclic Graph*), prioriza as tarefas com menor tempo de execução, utilizando o cálculo do *nível-t* para ordená-las. O algoritmo calcula para cada nível do grafo, o tempo de início de execução de todas as tarefas que já estão prontas para execução atribuindo aos processadores disponíveis. O par, tarefa-processador é ordenado em uma lista de execução em ordem crescente. Paralelamente é verificado se alguma tarefa esta com o parâmetro *deadline* superior ao valor constante no par tarefa-processador. Neste caso tenta reagrupar a tarefa em outro processador para atendimento deste parâmetro.

O detalhamento do algoritmo está descrito no português estruturado apresentado a seguir:

Algoritmo EDFMDAG

Início do algoritmo

(1) Calcular o *nível-t* do grafo, sem considerar os tempos de comunicação.

(2) Criar uma lista de nós prontos para execução chamada *ProntaExec*. Inicialmente, esta lista irá conter apenas os nós de entrada.

(3) Para todos os nós constantes na lista *ProntaExec.*

(4) Calcular o tempo de iniciação de execução mais antecipado para cada tarefa da lista *ProntaExec*. Selecionar o par, tarefa-processador que permitir a execução mais antecipada, utilizando a abordagem de não-inserção. Esta abordagem indica que tarefas são sempre escalonadas em uma máquina em um instante de tempo posterior ao de todas as tarefas já escalonadas na máquina. Isto significa que

tarefas não podem ser escalonadas em *slots* vazios anteriores (entre tarefas). Empates são decididos selecionando-se o parâmetro *deadline* e escolhendo o que tiver o menor valor. Persistindo o empate utiliza a tarefa com *nível-t* mais baixo. Escalonar o nó no processador correspondente.

(5) Atualizar a lista *ProntaExec* inserindo as novas tarefas prontas para execução.

Fim para.

Fim do algoritmo

5.2.2 Algoritmo LLFMDAG

O algoritmo LLFMDAG (*Least Laxity First Multiprocessor for Directed Acyclic Graph*), prioriza as tarefas com a menor relaxação, considerando o tempo de execução da tarefa em relação ao *deadline* do respectivo nível do GDA. O algoritmo calcula a relaxação para cada nível do grafo com o grupo de tarefas prontas e que não possuem dependência entre si. O atraso de uma tarefa implica no retardo de todas as tarefas dependentes. Fazendo uma analogia aos algoritmos de escalonamento dinâmicos, o tempo ótimo destas tarefas é dado pelo caminho critico do grafo e corresponde ao *deadline* em comum (todas as tarefas pertencentes ao mesmo nível).

O detalhamento do algoritmo está descrito no português estruturado apresentado a seguir:

Algoritmo LLFMDAG

Início do algoritmo

(1) Calcular o *nível-b* do grafo, sem considerar os tempos de comunicação.

(2) Criar uma lista de nós prontos para execução, denominada *RevProntaExec*.

(3) Ordenar *RevProntaExec* em ordem decrescente de *nível-b*. Inicialmente, esta lista irá conter apenas o nó de entrada.

(3) Repetir.

(4) Calcular a relaxação das tarefas prontas para serem executadas e ordená-las em ordem crescente na lista de nós prontos.

(5) Escalonar o primeiro nó na lista *RevProntaExec* e selecionar o par, tarefa-processador que permita a execução mais antecipada, utilizando a abordagem de não-inserção. Esta abordagem indica que tarefas são sempre escalonadas em uma

máquina em um instante de tempo posterior ao de todas as tarefas já escalonadas na máquina. Isto significa que tarefas não podem ser escalonadas em *slots* vazios (entre tarefas).

(6) Atualizar a lista *RevProntaExec* inserindo as novas tarefas prontas para execução.

Até que todos os nós tenham sido escalonados.

Fim do algoritmo

5.2.3 Algoritmo LLFPMDAG

O algoritmo LLFPMDAG (*Least Laxity First with Priority Multiprocessor for Directed Acyclic Graph*) prioriza as tarefas da mesma forma que o algoritmo LLFPMDAG, entretanto, nas tarefas com mesma folga é acrescentado outro critério de desempate. Esta nova prioridade é informada manualmente pelo usuário, nas tarefas que o mesmo julgar necessário e é dependente da semântica da aplicação. Caso este parâmetro não seja utilizado, este algoritmo fica com o mesmo escalonamento gerado pelo algoritmo LLFPMDAG.

Algoritmo LLFPMDAG

Início do algoritmo

(1) Calcular o *nível-b* do grafo, sem considerar os tempos de comunicação.

(2) Criar uma lista de nós prontos para execução, denominada *RevProntaExec*.

(3) Ordenar *RevProntaExec* em ordem decrescente conforme *nível-b*. Inicialmente, esta lista irá conter apenas os nós de entrada.

(3) Repetir.

(4) Calcular a relaxação das tarefas prontas para serem executadas e ordená-las em ordem crescente na lista de nós prontos.

(5) Escalonar o primeiro nó na lista *RevProntaExec* e selecionar o par, tarefa-processador que permita a execução mais antecipada, utilizando a abordagem de não-inserção. Esta abordagem indica que tarefas são sempre escalonadas em uma máquina em um instante de tempo posterior ao de todas as tarefas já escalonadas na máquina. Isto significa que tarefas não podem ser escalonadas em *slots* vazios entre tarefas. Empates são decididos pelo parâmetro *priority* das tarefas, priorizando o de maior valor.

(6) Atualizar a lista *RevProntaExec* inserindo as novas tarefas prontas para execução.

Até que todos os nós tenham sido escalonados.

Fim do algoritmo

5.3 Proposta de monitoramento diferenciada e re-escalonamento de tempo real

O *framework* escolhido, o PM^{2}P, está baseado num *cluster* de oito máquinas para efetuar o processamento das tarefas e uma máquina *front end*, que coordena a distribuição e colhe as informações dos processamentos para apresentação do resultado através de mapa de Gantt.

O processo inicia com o *framework* distribuindo as tarefas do sistema no *cluster* de máquinas, de acordo com o escalonamento gerado pelo algoritmo. Paralelamente a este processo, a máquina *front end* inicia o monitoramento dos resultados do processamento de cada tarefa. Entretanto, como o escalonamento do GDA é efetuado de forma estática, respeitando a disponibilidade de máquinas no momento da realização do escalonamento, caso alguma das máquinas do *cluster* ficar indisponível ou voltar a funcionar, durante a execução do sistema, o escalonamento ficará prejudicado, uma vez que as tarefas não são re-escalonadas considerando a nova arquitetura.

A proposta de construir o ambiente de monitoramento das máquinas veio da forma como a ferramenta PM^{2}P inicia as tarefas em paralelo para execução nas máquinas distribuídas. A utilização da biblioteca MPICH, utilizando o *device* ch_p4mpd que suporta *clusters* homogêneos de máquinas com um processador, facilita o início de cada tarefa e proporciona mais rapidez para execução do sistema. É utilizado um arquivo de configuração *procgroup*, que indica a biblioteca qual a ordem de início de cada tarefa e a respectiva máquina. Entretanto, se durante a execução das tarefas, alguma máquina ficar indisponível, o *device* aborta a execução saindo com a mensagem de erro de *timeout.* Devido a esta característica da biblioteca MPICH, utilizando o *device* ch_p4mpd, houve a necessidade de efetuar o monitoramento *on-line* do *cluster* com o objetivo de gerenciar a disponibilidade das máquinas e re-escalonar o GDA, caso esta disponibilidade se altere durante a execução das tarefas.

Após a solicitação de execução do sistema pelo usuário, dá-se o processo de escolha do melhor algoritmo de escalonamento. O processo completo é representado pelos passos 10-a, 11-a, 12-a, 12-b, 13-a, 13-b e 14-a do procedimento detalhado no item 5.1. O

monitoramento é iniciado no passo 14-b e a partir de sua iniciação, as máquinas são monitoradas de acordo com os passos 14-c, 14-d, 14-e, 14-f e 14-g e 14-i, sendo que o passo 14-h, que trata da apresentação dos resultados por meio de mapas de Gantt, independe da monitoração estar ativa ou não.

O detalhamento da implementação do monitoramento está descrito no português estruturado apresentado a seguir:

Monitoramento *on-line* do *cluster*

Início do algoritmo

(1) Iniciar monitoramento do cluster sem bloqueio do processo de escalonamento.
(2) Carregar *array* de máquinas cadastradas no *cluster.*
(3) Enquanto processo de escalonamento não termina.
(4) Repetir a cada segundo.
(5) Para cada *Id_máquina* constante no *array.*
Se a máquina não está respondendo *check_slave(id) e* marcada *em_uso.*
Se *Id_máquina* consta na lista ordenada de tarefas pendente execução.
(6) Refaz escalonamento das tarefas remanescentes.
(7) Reinicia processo com novo reescalonamento.
Fim se.
(8) Marca máquina como indisponível e libera *em_uso*.
Fim se Senão se não marcada *em_uso* (a máquina está respondendo *check_slave(id)).*
(9) Refaz escalonamento das tarefas remanescentes.
Se *Makespan* do reescalonamento < *Makespan original.*
(10) Reinicia processo com novo reescalonamento.
Fim se Senão Volta à situação original.
Fim se senão.
Fim para.
Fim repetir.
Fim enquanto.

Fim do algoritmo

6 TESTES E RESULTADOS DAS IMPLEMENTAÇÕES PROPOSTAS

O objetivo dos testes é demonstrar o desempenho de cada algoritmo, o custo necessário para efetuar o cálculo e também os ganhos na utilização do monitoramento do *cluster* e o re-escalonamento da aplicação caso a disponibilização das máquinas do *cluster* se altere.

Para realizar os testes foram pré-definidas seqüências de tarefas, com custo de processamento e *deadlines* distintos, simulando assim, situações de processamento de um sistema que possa ser representado por um grafo dirigido acíclico. As demonstrações gráficas sobre o comportamento do monitoramento de máquinas e dos reescalonamentos efetuados pelos algoritmos estão apresentados em mapas de Gantt e tabelas. Também é discutido os ganhos obtidos, bem como eventuais situações que os algoritmos porventura não contemplem.

6.1.1 Roteiro dos Testes

Os testes foram efetuados utilizando três grafos distintos, com situações de dependência e paralelismo de tarefas. O primeiro possui nove tarefas, as quais estão distribuídas de acordo com a figura 6.1 e com os respectivos custos de execução e prioridades de cada tarefa detalhadas na tabela 6.1.

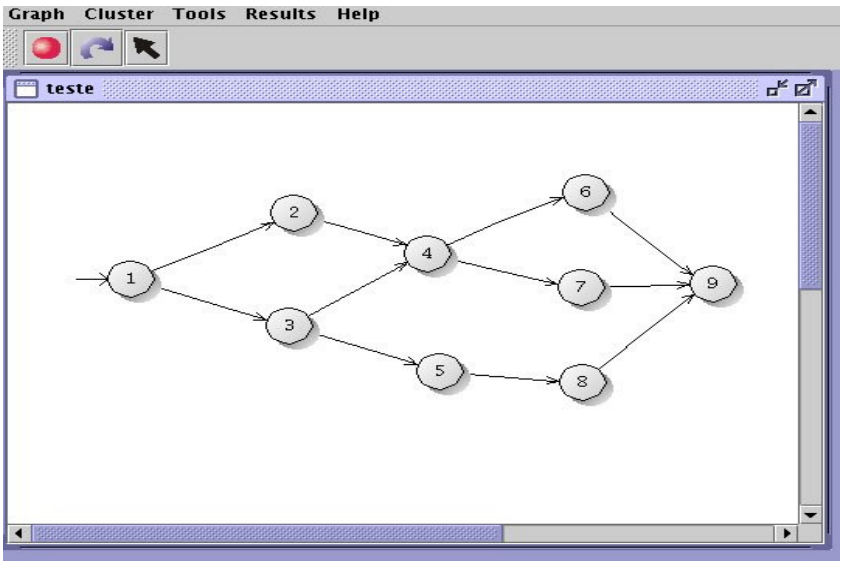

Figura 6.1: *Framework* PM^{2}P. Grafo de nove tarefas.

Tabela 6.1: Custo de execução e prioridades. Grafo de nove tarefas.

TAREFA	CUSTO EXEC.	PRIORIDADE
1	1	0
2	1	1
3	4	0
4	4	2
5	4	1
6	2	2
7	5	3
8	6	1
9	2	0

O segundo grafo possui onze tarefas, distribuídas de acordo com a figura 6.2, com os respectivos custos de execução e prioridades detalhadas na tabela 6.2.

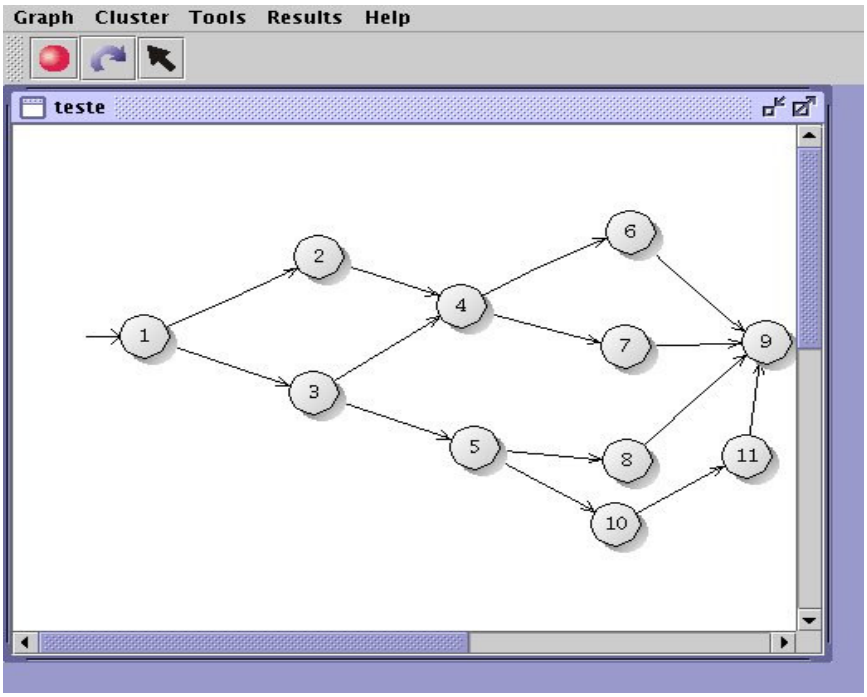

Figura 6.2: *Framework* PM^{2}P. Grafo de onze tarefas.

Tabela 6.2: Custo de execução e prioridades. Grafo de onze tarefas.

TAREFA	CUSTO EXEC.	PRIORIDADE
1	1	0
2	1	1
3	4	0
4	4	1
5	5	1
6	2	2
7	5	3
8	6	1
9	2	0
10	3	4
11	2	4

O terceiro grafo é um teste de maior esforço, com cinqüenta tarefas, distribuídas de acordo com a figura 6.3. Os algoritmos LLFMDAG, EDFMDAG e LLFPMDAG e os algoritmos já implementados na ferramenta HLFET e ETF foram avaliados quanto ao resultado do escalonamento e tempo gasto para sua execução. Todos os algoritmos de escalonamento foram avaliados nos três grafos sugeridos.

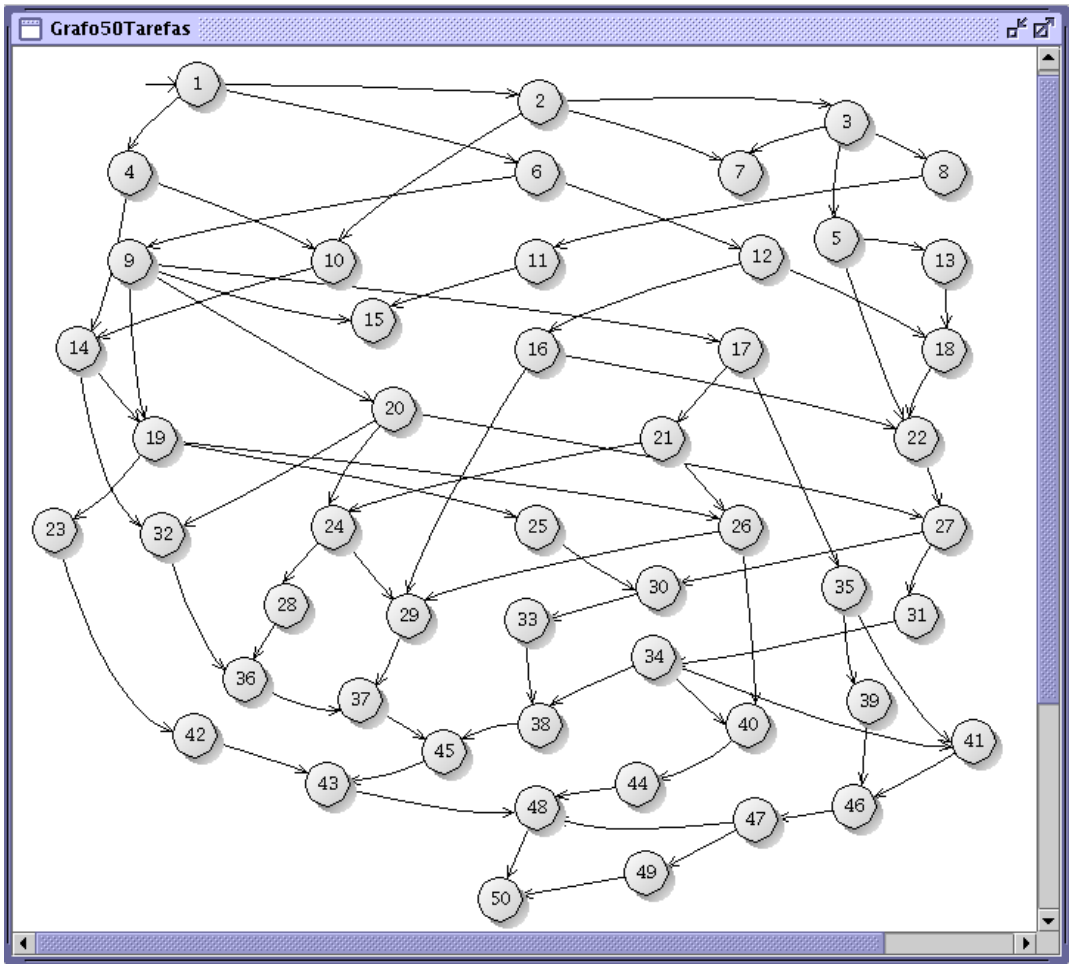

Figura 6.3: *Framework* PM^{2}P. Grafo de cinqüenta tarefas.

Quanto ao monitoramento da disponibilidade das máquinas no *cluster*, dentre as situações anteriores levantadas foi utilizado o melhor resultado do escalonamento no grafo de onze tarefas. Foi utilizado o escalonamento que tenha apresentando o menor tempo de escalonamento do sistema. Para avaliar o monitoramento foram abordados dois critérios: excluindo e adicionando máquinas no *cluster*. Inicialmente foram utilizadas todas as máquinas disponíveis no *cluster* e foram reduzidas até uma quantidade em que não seja possível escalonar o grafo dentro do tempo do seu caminho crítico. Outro critério utilizado foi a execução dos testes com duas máquinas e adicionando outras até a situação em que se encontre ociosidade no *cluster*.

6.1.2 Resultado dos Testes

Os testes a seguir apresentados foram estruturados com o objetivo de avaliar o comportamento dos algoritmos e avaliar o monitoramento do *cluster*. O *framework* PM^2P fornece os resultados dos escalonamentos em duas situações, as quais comprovarão as implementações efetuadas. Numa primeira etapa foram aferidos os escalonamentos gerados pelos algoritmos propostos baseados nos grafos demonstrativos criados e com os parâmetros informados manualmente nas tarefas. Os mapas de Gantt foram calculados de acordo com estas informações. Após a efetiva execução da aplicação, com o monitoramento do *cluster*, será apresentado um novo mapa de Gantt, elaborado a partir dos tempos reais de execução sendo estas informações guardadas em arquivos. Caso o monitoramento detecte alteração na quantidade de máquinas e esta alteração influencie na execução do sistema devido à dependência entre as tarefas que compõe o GDA, o processo é interrompido, reescalonado dentro da nova disponibilidade e reiniciado. A cada interrupção será apresentado um mapa de Gantt parcial (das tarefas que foram executadas), e outros na seqüência, até a finalização da execução.

As diferenças nos escalonamentos gerados a seguir evidenciam o critério que cada algoritmo utiliza na escolha das tarefas, considerando as prioridades de cada tarefa. Especificamente no caso do algoritmo LLFPMDAG o escalonamento é dependente das prioridades atribuídas pelo usuário e é atribuído de acordo com a semântica da aplicação. As tabelas 6.1 e 6.2 relacionam estes valores e estas informações são determinantes no escalonamento gerado.

As figuras 6.4, 6.5, 6.6, 6.7 e 6.8 mostram os mapas de Gantt com o resultado do escalonamento do grafo de nove tarefas (figura 6.1), utilizando os algoritmos HLFET, ETF, LLFMDAG, EDFMDAG e LLFPMDAG com duas máquinas ativas no *cluster*. Pode-se verificar que os algoritmos que priorizam as tarefas com maior tempo de duração apresentaram um melhor tempo de escalonamento, neste caso o HLFET e o LLFMDAG. Quanto ao tempo gasto para efetuar cada escalonamento, verifica-se que os algoritmos EDFMDAG e ETF que priorizam as tarefas mais curtas apresentam tempo ligeiramente inferior para efetuar o ordenamento. Os valores estão na tabela 6.3.

Tabela 6.3: Custo de execução e *makespan*. Grafo de nove tarefas e duas máquinas.

Algoritmo	**Custo de execução (em milissegundos)**	***Makespan* (em milissegundos)**
HLFET	31,9026	18
ETF	26,0099	19
LLFMDAG	42,6543	18
EDFMDAG	25,0289	19
LLFPDAG	33,1157	19

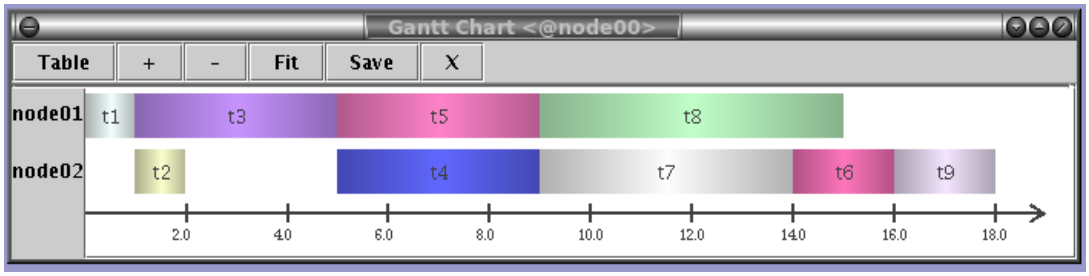

Figura 6.4: Mapa de Gantt com nove tarefas – algoritmo HLFET - duas máquinas.

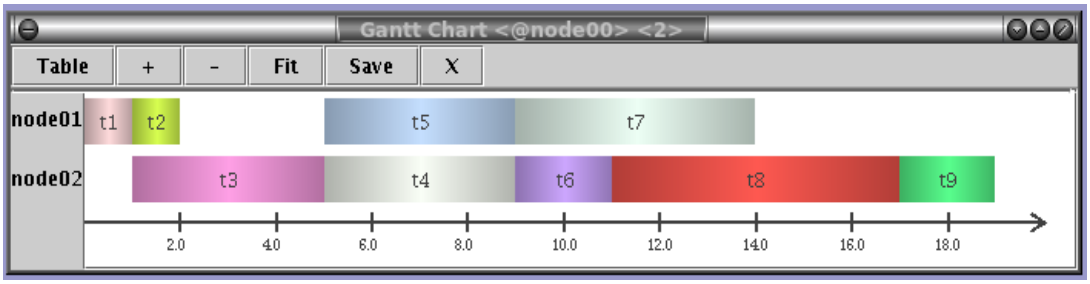

Figura 6.5: Mapa de Gantt com nove tarefas – algoritmo ETF - duas máquinas.

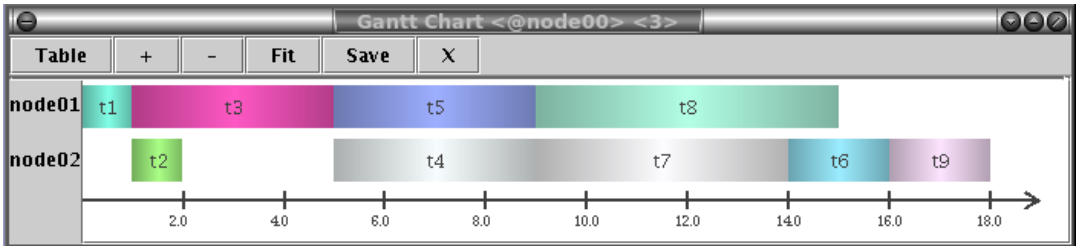

Figura 6.6: Mapa de Gantt com nove tarefas – algoritmo LLFMDAG - duas máquinas.

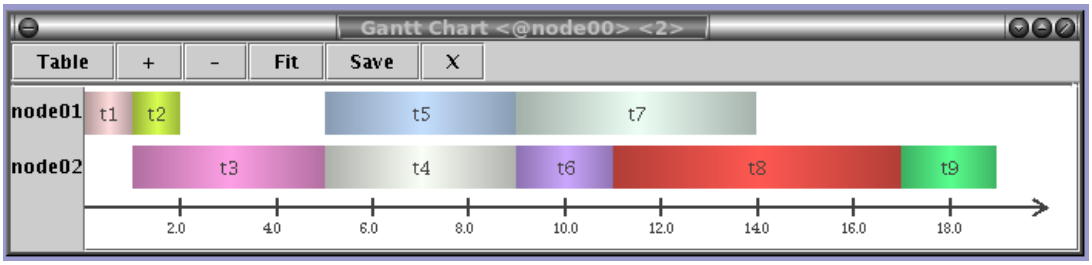

Figura 6.7: Mapa de Gantt com nove tarefas – algoritmo EDFMDAG - duas máquinas.

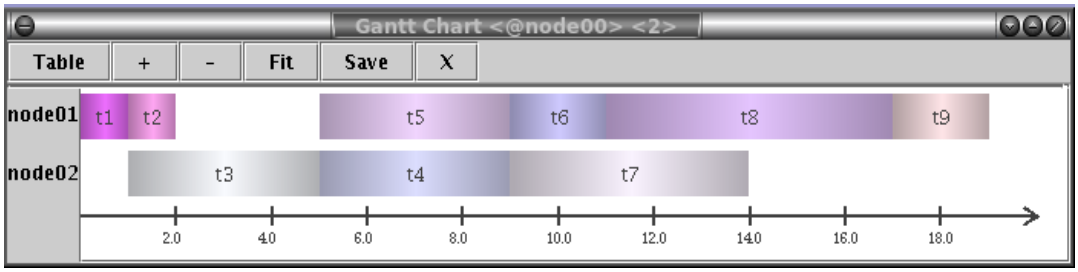

Figura 6.8: Mapa de Gantt com nove tarefas – algoritmo LLFPMDAG - duas máquinas.

As figuras 6.9, 6.10, 6.11, 6.12 e 6.13 mostram os mapas de Gantt com o resultado do escalonamento do grafo de nove tarefas (figura 6.1), utilizando os algoritmos HLFET, ETF, LLFMDAG, EDFMDAG e LLFPMDAG com quatro máquinas ativas no *cluster*. Diferente do caso anterior, dada a característica de dependência das tarefas, há máquinas ociosas e com isso todos os escalonadores apresentaram um escalonamento conforme os critérios descritos sobre cada algoritmo. Como o grafo não mudou, apenas foram acrescentadas máquinas ativas, permaneceram as mesmas características de gasto de tempo descritas no escalonamento quando havia duas máquinas. Os valores estão na tabela 6.4.

Tabela 6.4: Custo de execução e *makespan*. Grafo de nove tarefas e quatro máquinas.

Algoritmo	**Custo de execução (em milissegundos)**	***Makespan* (em milissegundos)**
HLFET	29,0779	17
ETF	25,0317	17
LLFMDAG	41,7935	17
EDFMDAG	23,1979	17
LLFPDAG	42,6766	17

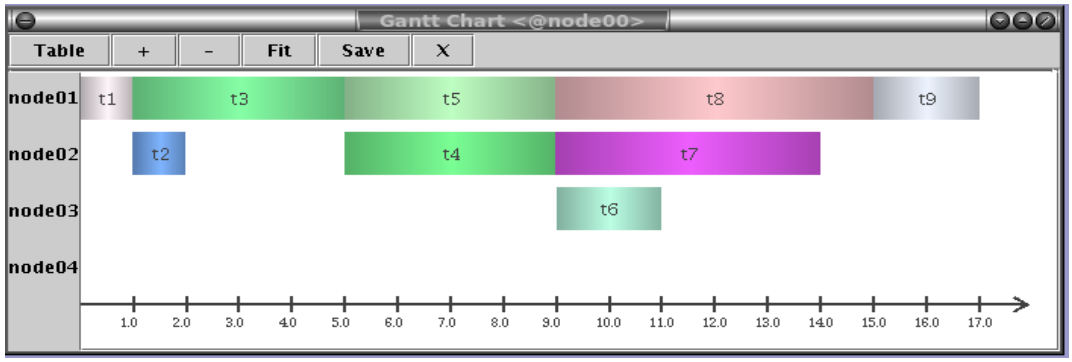

Figura 6.9: Mapa de Gantt com nove tarefas – algoritmo HLFET - quatro máquinas.

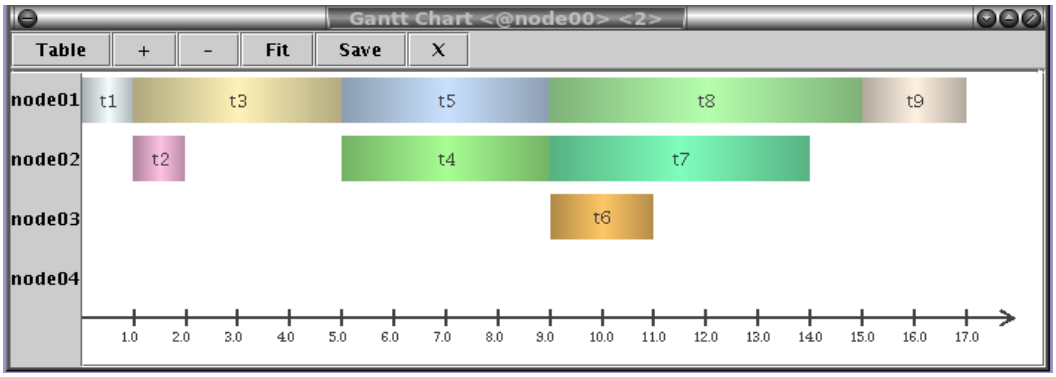

Figura 6.10: Mapa de Gantt com nove tarefas – algoritmo ETF - quatro máquinas.

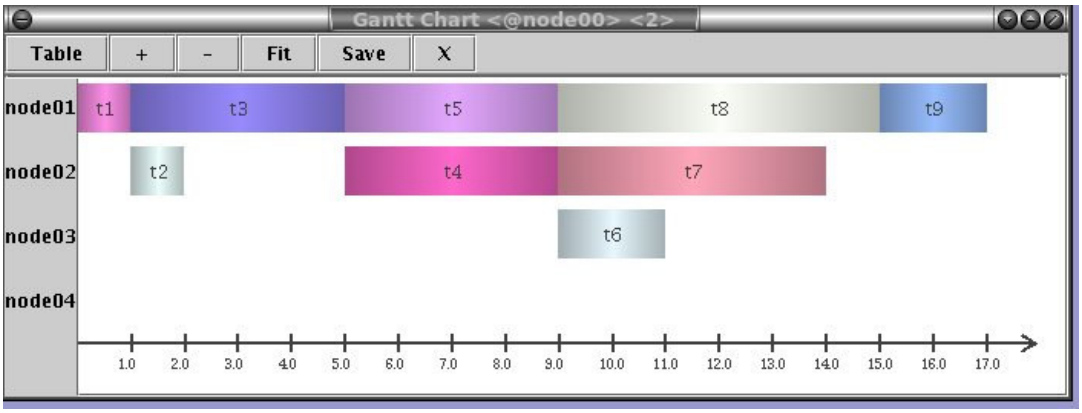

Figura 6.11: Mapa de Gantt com nove tarefas – algoritmo LLFMDAG - quatro máquinas.

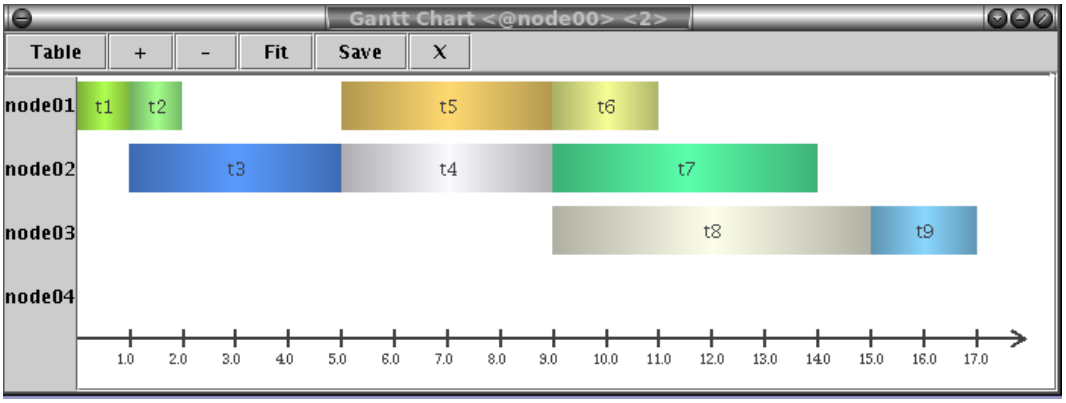

Figura 6.12: Mapa de Gantt com nove tarefas – algoritmo EDFMDAG - quatro máquinas.

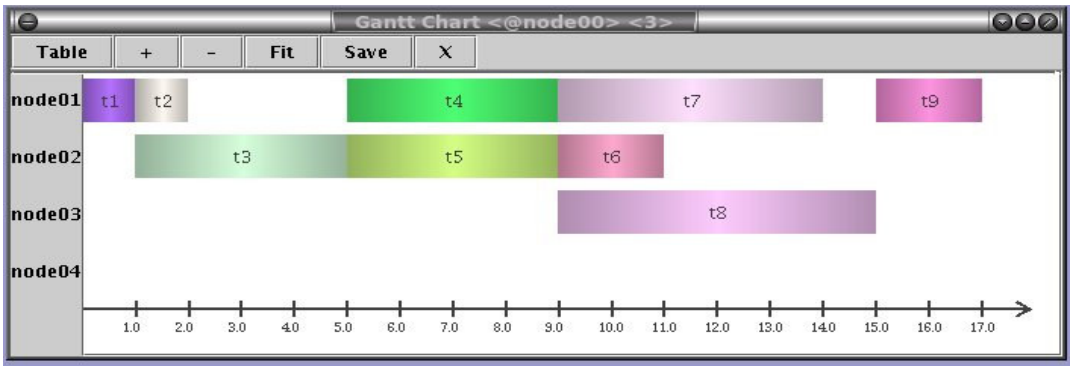

Figura 6.13: Mapa de Gantt com nove tarefas – algoritmo LLFPMDAG - quatro máquinas.

As figuras 6.14, 6.15, 6.16, 6.17 e 6.18 mostram os mapas de Gantt com o resultado do escalonamento do grafo de onze tarefas (figura 6.2), utilizando os algoritmos HLFET, ETF, LLFMDAG, EDFMDAG e LLFPMDAG com duas máquinas ativas no *cluster*. De acordo com os mapas de Gantt gerados, verifica-se que apesar do grafo com onze tarefas possuir um maior paralelismo, o critério de priorização das tarefas mais longas permanece com o melhor resultado no escalonamento. Os algoritmos ETF e LLFMDAG apresentaram os melhores escalonamentos. Quanto ao tempo de execução gasto para efetuar o escalonamento, os resultados são similares ao escalonamento de nove tarefas com duas máquinas, permanecendo os algoritmos que priorizam as tarefas mais curtas com menor tempo. A diferença em relação aos tempos anteriores é que neste processo os tempos ficaram mais homogêneos, não havendo diferenças significativas. Os valores estão na tabela 6.5.

Tabela 6.5: Custo de execução e *makespan*. Grafo de onze tarefas e duas máquinas.

Algoritmo	**Custo de execução (em milissegundos)**	***Makespan* (em milissegundos)**
HLFET	25,7978	22
ETF	25,8141	21
LLFMDAG	29,6680	21
EDFMDAG	21,9511	23
LLFPDAG	24,3764	22

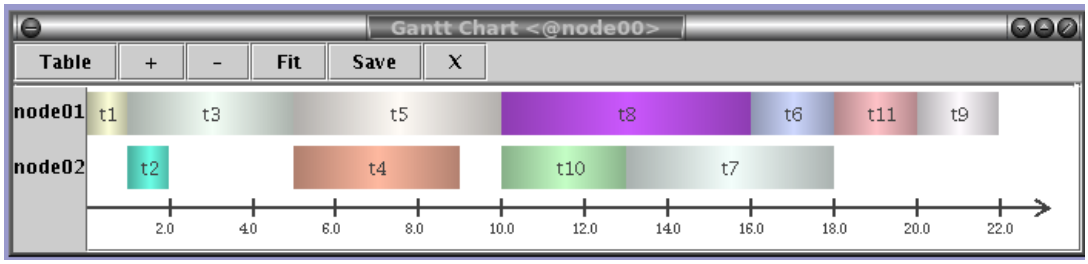

Figura 6.14: Mapa de Gantt com onze tarefas – algoritmo HLFET - duas máquinas.

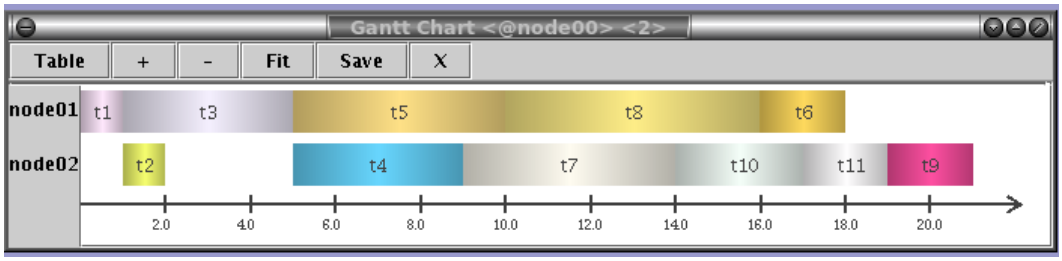

Figura 6.15: Mapa de Gantt com onze tarefas – algoritmo ETF - duas máquinas.

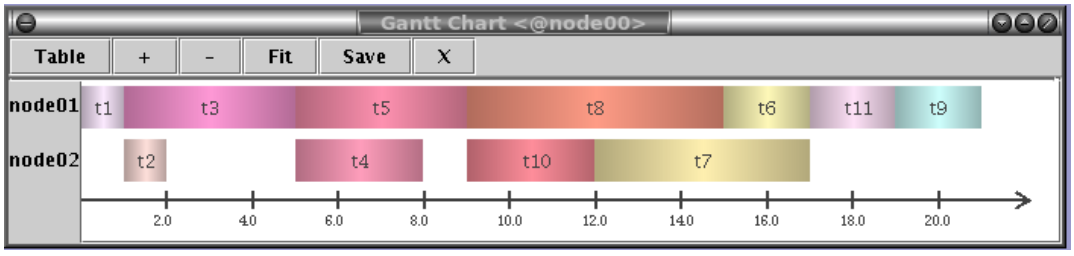

Figura 6.16: Mapa de Gantt com onze tarefas – algoritmo LLFMDAG - duas máquinas.

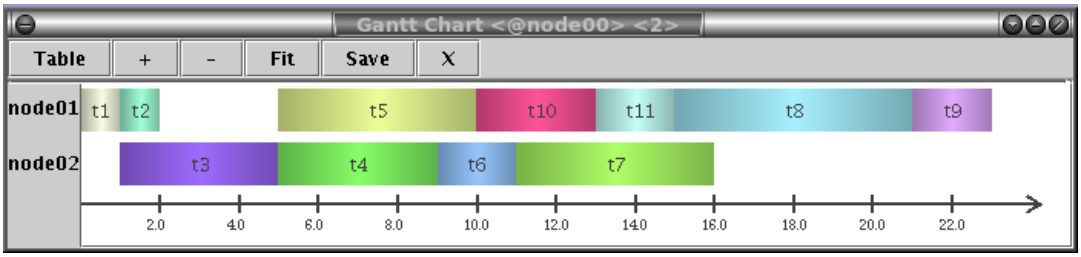

Figura 6.17: Mapa de Gantt com onze tarefas – algoritmo EDFMDAG - duas máquinas.

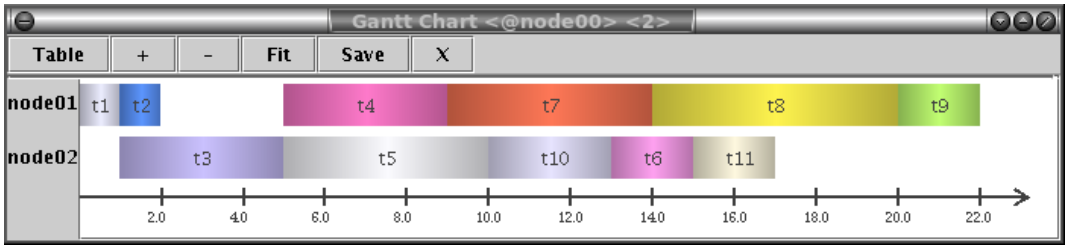

Figura 6.18: Mapa de Gantt com onze tarefas – algoritmo LLFPMDAG - duas máquinas.

As figuras 6.19, 6.20, 6.21, 6.22 e 6.23 mostram os mapas de Gantt com o resultado do escalonamento do grafo de onze tarefas (figura 6.2), utilizando os algoritmos HLFET, ETF, LLFMDAG, EDFMDAG e LLFPMDAG com quatro máquinas ativas no *cluster*. Diferente do caso anterior, dada a característica de dependência das tarefas, há máquinas suficientes para efetuar o escalonamento e com isso os escalonadores apresentaram um escalonamento ótimo, conforme os critérios descritos sobre cada algoritmo de escalonamento. Quanto aos tempos de execução para efetuar o escalonamento, os dados colhidos estão mais similares aos resultados levantados com o grafo de nove tarefas e quatro máquinas ocorrendo homogeneidade nos tempos. Permanece o algoritmo EDFMDAG com o menor custo de execução. Os valores estão na tabela 6.6.

Tabela 6.6: Custo de execução e *makespan*. Grafo de onze tarefas e quatro máquinas.

Algoritmo	**Custo de execução (em milissegundos)**	***Makespan* (em milissegundos)**
HLFET	25,4805	18
ETF	23,9375	18
LLFMDAG	40,8135	18
EDFMDAG	21,4148	19
LLFPDAG	30,9644	18

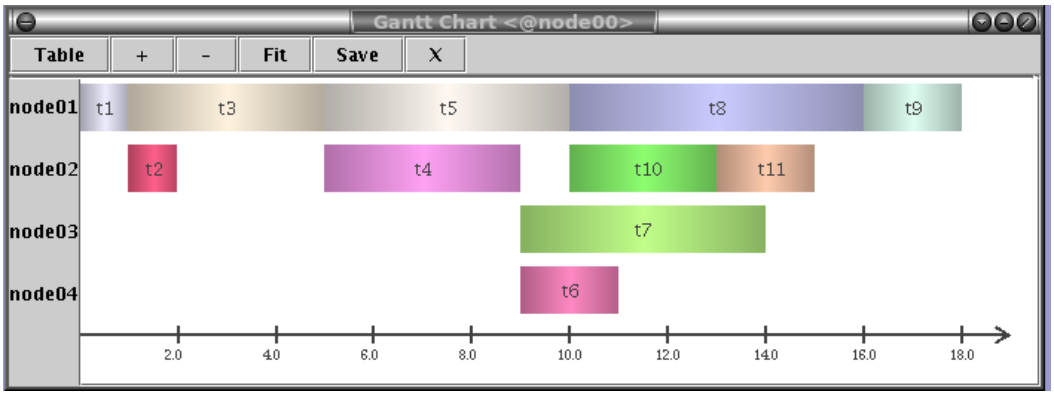

Figura 6.19: Mapa de Gantt com onze tarefas – algoritmo HLFET - quatro máquinas.

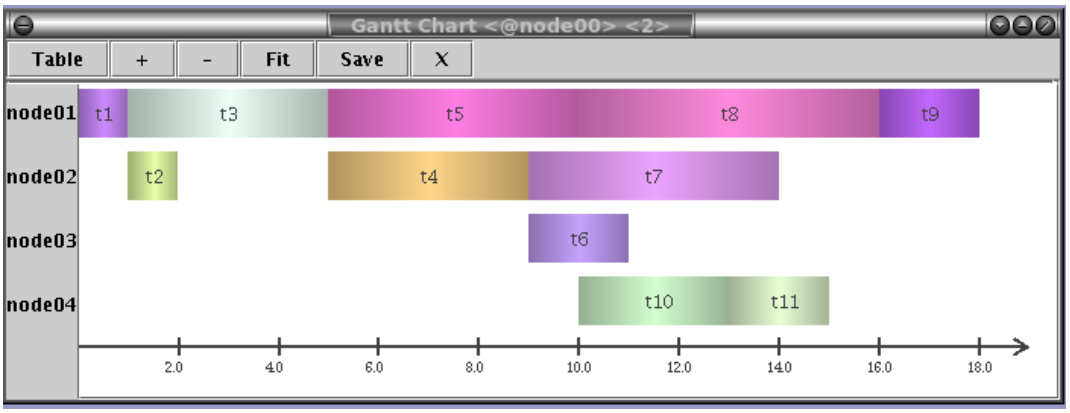

Figura 6.20: Mapa de Gantt com onze tarefas – algoritmo ETF - quatro máquinas.

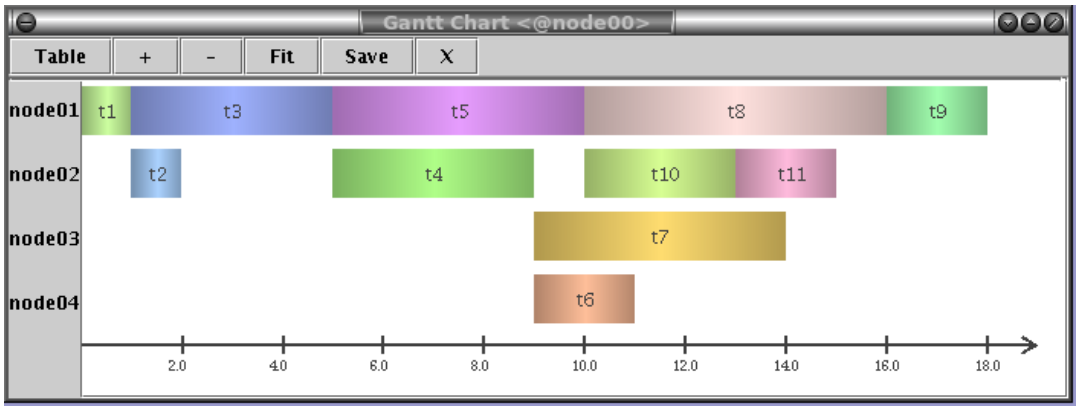

Figura 6.21: Mapa de Gantt com onze tarefas – algoritmo LLFMDAG - quatro máquinas.

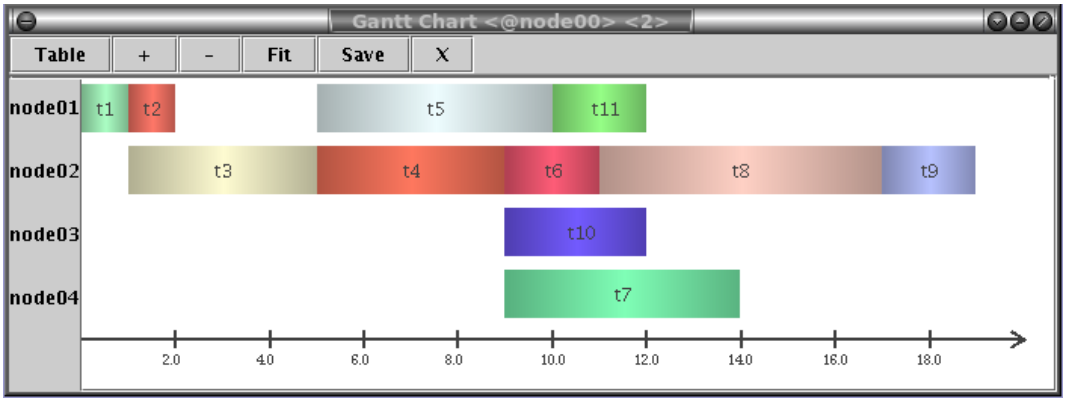

Figura 6.22: Mapa de Gantt com onze tarefas – algoritmo EDFMDAG - quatro máquinas.

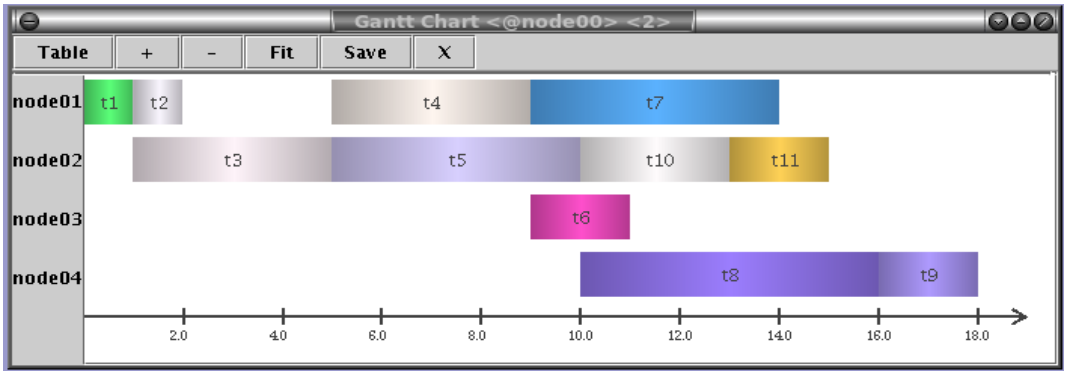

Figura 6.23: Mapa de Gantt com onze tarefas – algoritmo LLFPMDAG – quatro máquinas.

As figuras 6.24, 6.25, 6.26, 6.27 e 6.28 mostram os mapas de Gantt com o resultado do escalonamento do grafo de cinqüenta tarefas (figura 6.3), utilizando os algoritmos HLFET, ETF, LLFMDAG, EDFMDAG e LLFPMDAG com seis máquinas ativas no *cluster*. Uma maior quantidade de tarefas e máquinas permitiu avaliar a similaridade de comportamento dos algoritmos, uma vez que os tempos aferidos para execução do escalonamento ficaram proporcionais aos tempos aferidos nos testes efetuados com os grafos de nove e onze tarefas. Quanto aos tempos de escalonamento do sistema, constantes na tabela 6.7, ficaram similares a exceção dos algoritmos ETF e HLFET (tabela 6.7). As características do grafo foram atribuídas automaticamente pela ferramenta. Os tempos de execução de cada tarefa e a relação de precedência é definido aleatoriamente pela ferramenta e devido a esta característica, o *makespan* deste gráfico ficou menor que 0,1 milissegundo. Para que este tempo seja maior, haverá necessidade de rever o processo, entretanto não há necessidade. Quanto ao custo de execução dos algoritmos, as diferenças se apresentaram proporcionais aos

custos de execução avaliados para os grafos de nove e onze tarefas. Exceto que neste caso o algoritmo HLFET apresentou o menor custo de execução.

Tabela 6.7: Custo de execução e *makespan*. Grafo de cinqüenta tarefas e seis máquinas.

Algoritmo	**Custo de execução (em milissegundos)**	***Makespan* (em milissegundos)**
HLFET	30,4997	0,049
ETF	44,2494	0,049
LLFMDAG	45,2878	0,047
EDFMDAG	32,1787	0,047
LLFPDAG	41,1139	0,047

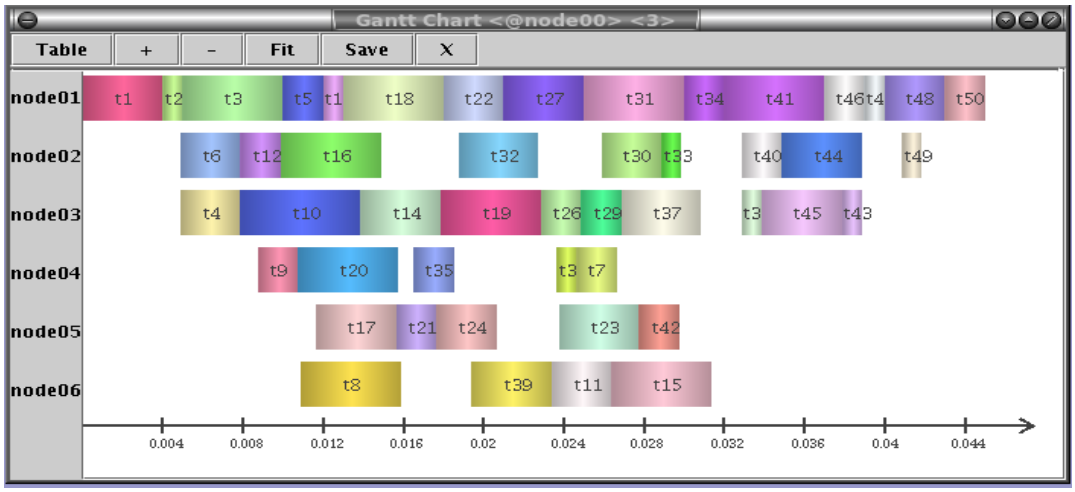

Figura 6.24: Mapa de Gantt com cinqüenta tarefas – algoritmo HLFET – seis máquinas.

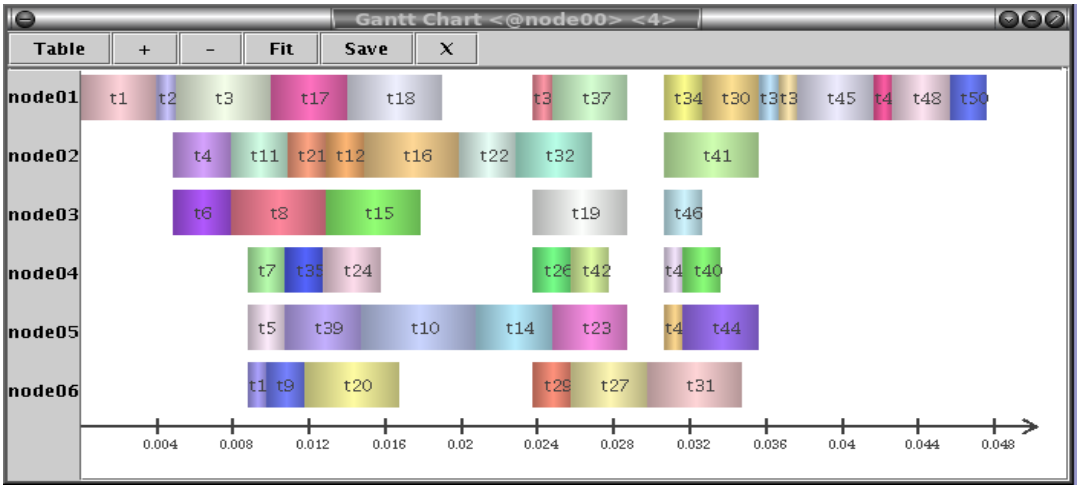

Figura 6.25: Mapa de Gantt com cinqüenta tarefas – algoritmo ETF – seis máquinas.

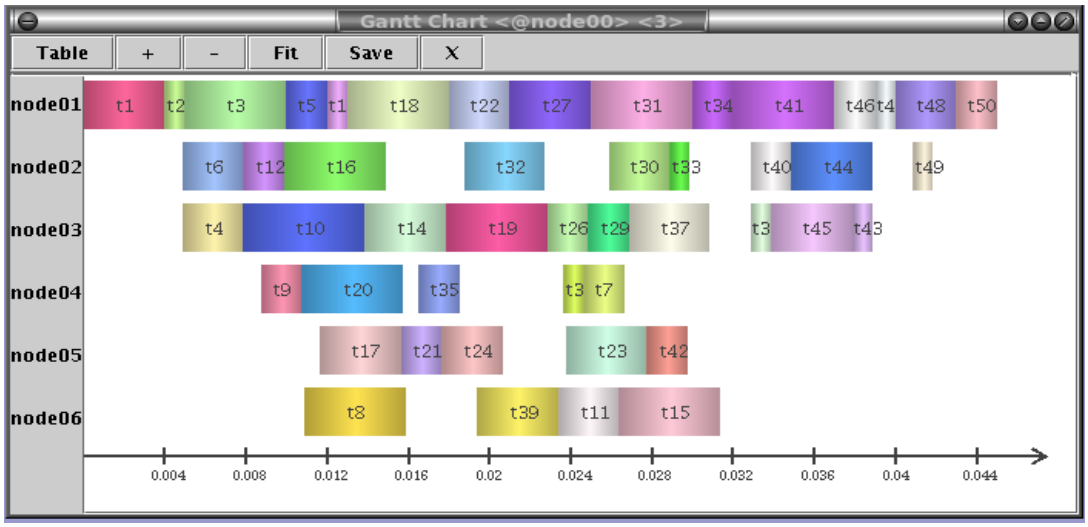

Figura 6.26: Mapa de Gantt com cinqüenta tarefas – algoritmo LLFMDAG – seis máquinas.

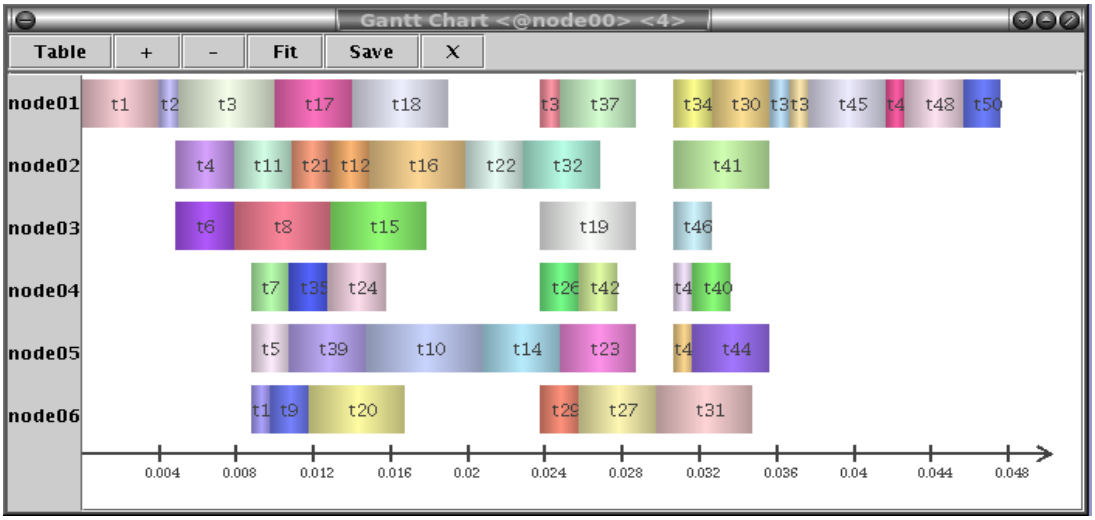

Figura 6.27: Mapa de Gantt com cinqüenta tarefas – algoritmo EDFMDAG – seis máquinas.

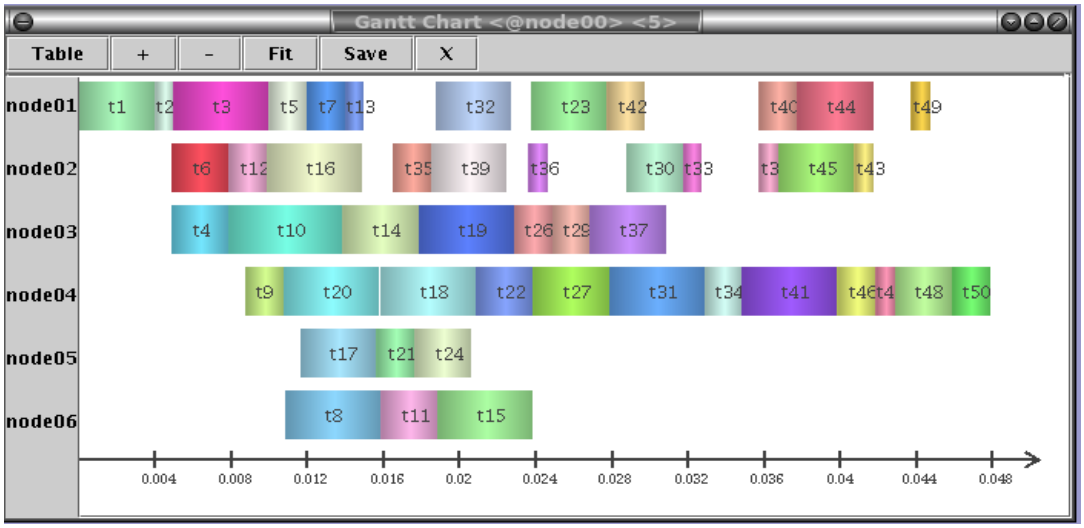

Figura 6.28: Mapa de Gantt com cinqüenta tarefas – algoritmo LLFPMDAG – seis máquinas.

As figuras 6.29, 6.30 e 6.31 mostram três mapas de Gantt complementares, representando situações distintas evidenciadas na execução do escalonamento do grafo de onze tarefas utilizando o algoritmo com menor custo de execução e com menor *makespan*. A seqüência dos três grafos evidencia a ocorrência de perda de máquinas ativas no *cluster*. A execução em paralelo da aplicação é iniciada com oito máquinas ativas no *cluster* e são desconectadas seis. A ferramenta verifica se o escalonamento sofre alguma alteração. Verificou-se que até quatro máquinas disponíveis, os re-escalonamentos efetuados no *front end* não provocaram perda de performance e, portanto não houve reinício do processo. Os três mapas gerados são independentes entre si, uma vez que eles demonstram apenas o período entre a situação atual e a detecção de um problema no *cluster* que apontou a necessidade do re-escalonamento das tarefas faltantes e reinício do processo. E são considerados complementares, uma vez que são necessários os três grafos para representar a execução completa da aplicação. O primeiro re-escalonamento ocorreu no momento em que ficaram três máquinas disponíveis.

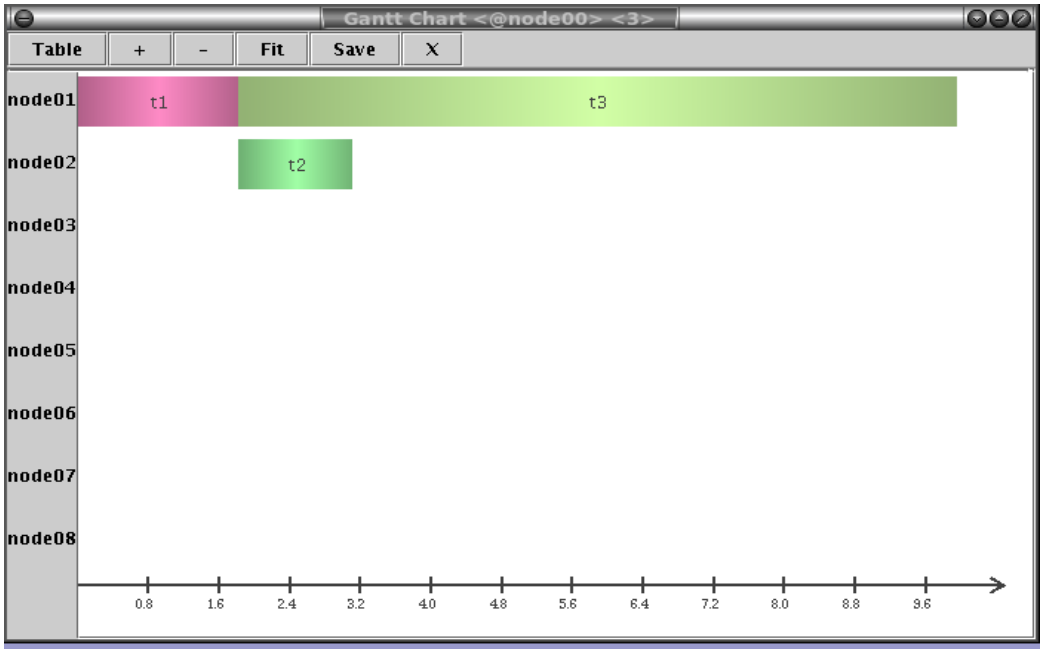

Figura 6.29: Monitoramento com perda de máquinas – oito máquinas.

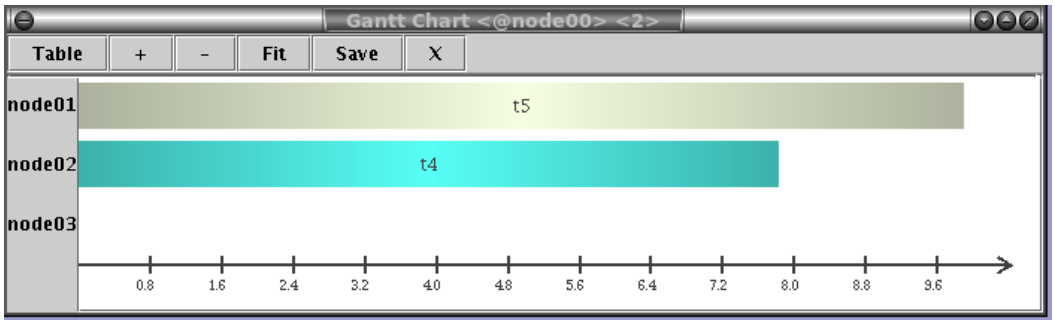

Figura 6.30: Monitoramento com perda de máquinas – três máquinas.

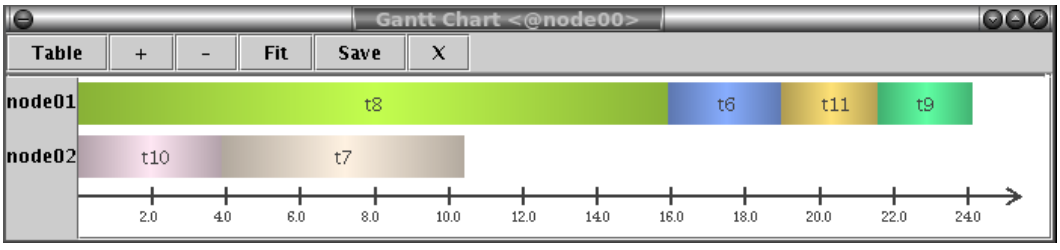

Figura 6.31: Monitoramento com perda de máquinas – duas máquinas.

As figuras 6.32 e 6.33, mostram os mapas de Gantt complementares, representando situações distintas evidenciadas na execução do escalonamento do grafo de onze tarefas utilizando o algoritmo com menor custo de execução e com menor *makespan*. A

execução do sistema iniciou com duas máquinas ativas no *cluster* e posteriormente houve a detecção de outra máquina. Após esta detecção, a verificação de ganho de performance é verificada através do reescalonamento das tarefas não marcadas como executadas. Constatado este ganho o processo é interrompido e reiniciado com a nova configuração de máquinas. De forma similar a perda de máquinas, os dois mapas de Gantt representam dois processos de execução distintos, entretanto eles são complementares, uma vez que compõem a execução completa da aplicação.

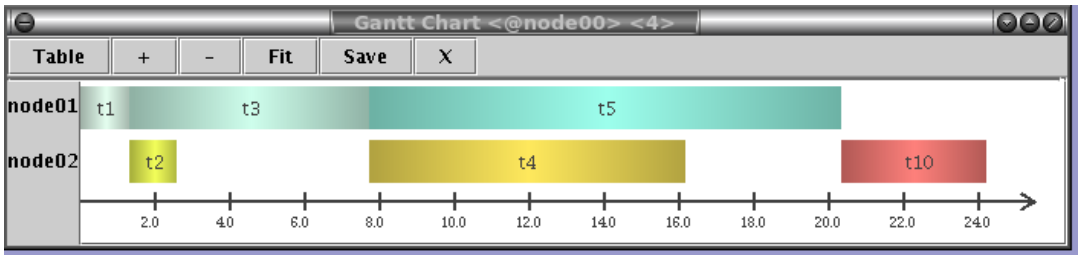

Figura 6.32: Monitoramento com ganho de máquinas – duas máquinas.

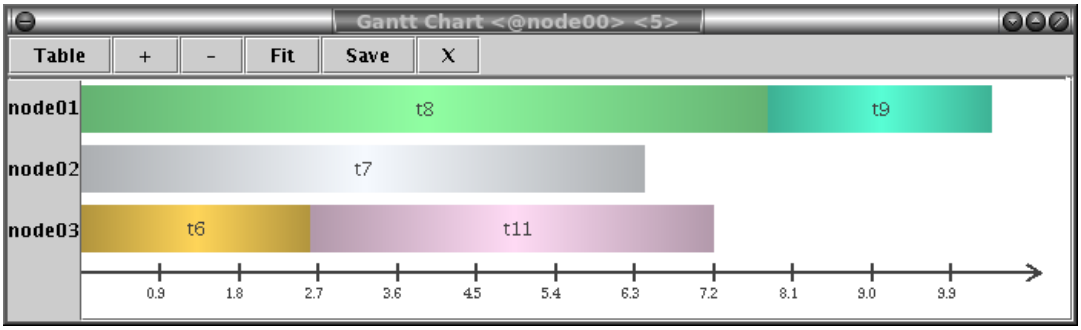

Figura 6.33 Monitoramento com ganho de máquinas – três máquinas.

7 CONCLUSÃO

O presente trabalho avaliou algoritmos de escalonamento estáticos e dinâmicos e a partir destes algoritmos, foi proposto o monitoramento das máquinas do *cluster* e no caso de alteração na situação das máquinas é efetuado o re-escalonamento da aplicação. Foram implementados três algoritmos de escalonamento, EDFMDAG, LLFMDAG e LLFPMDAG com as devidas adaptações para o escalonamento de sistemas que possam ser representados por um GDA, considerando o conhecimento antecipado das tarefas que compõem este sistema e sua relação de precedência. Se já dispomos do GDA que representa as tarefas antecipadamente e sabe-se que as tarefas não serão alteradas, conclui-se que não há necessidade de implementar algoritmos dinâmicos e preemptivos respeitando o escopo deste trabalho.

Uma das necessidades deste trabalho foi a definição de um *framework* para a implementação da proposta de monitoramento do *cluster*. Como não há similaridades entre os *frameworks* quanto à forma que cada algoritmo e outras características são implementadas, foi necessária a escolha de apenas um. Os itens mais relevantes para a definição do PM^2P foi a disponibilidade de todos os fontes, estar construído na linguagem *java* e ter acesso a infra-estrutura onde a ferramenta foi construída.

Considerando o escalonamento de um sistema, que pode ser representado por um GDA, entende-se que o objetivo deste escalonamento é obter um *makespan* do grafo igual ao seu caminho crítico. Para atingir este resultado, além das características do algoritmo, a disponibilidade de máquinas no *cluster* é outro quesito determinante.

Os algoritmos de escalonamento utilizados neste trabalho apresentam resultados ótimos quando a disponibilidade de processamento é suficiente para efetuar o escalonamento. Entretanto, quando não há disponibilidade de processamento, a determinação do melhor escalonamento é dependente da estrutura do grafo de tarefas e da relação de precedência. Os resultados dos testes indicam que se houver disponibilidade suficiente de máquinas, todos apresentam o tempo de escalonamento da aplicação igual ao caminho crítico do GDA correspondente. Mas, quando não houver processamento suficiente para acomodar todas as tarefas que estão prontas para serem executadas, os algoritmos que priorizam as tarefas mais longas, conseguem apresentar melhores resultados, considerando os testes abordados neste trabalho.

Sistemas que possuem determinadas tarefas com alto índice de criticidade necessitam que sejam priorizadas, em detrimento das demais. Nestes casos o algoritmo de escalonamento deve priorizar estas tarefas (mais relevantes), colocando em segundo plano, a busca pelo menor custo de execução. Se o fator determinante para a priorização de tarefas da aplicação está na sua semântica, a incumbência de definir as prioridades recai ao usuário. Este critério poderá aumentar o tempo de escalonamento da aplicação, penalizando as demais tarefas. Considerando estas necessidades foi implementado o algoritmo LLFPMDAG que permite ao usuário incluir maiores prioridades em determinadas tarefas, para que o algoritmo possa priorizá-las no escalonamento. Nos testes efetuados em laboratório, este algoritmo apresentou um escalonamento viável atendendo aos requisitos da aplicação.

A proposta de construir o ambiente de monitoramento das máquinas veio da forma como a ferramenta PM^{2}P utiliza a biblioteca MPI com o *driver* ch_p4mpd. As tarefas são iniciadas em paralelo de acordo com os parâmetros indicados o arquivo *procgroup.* Após iniciado este processo, se alguma máquina ficar indisponível durante a execução do sistema as tarefas subseqüentes serão penalizadas, uma vez que as tarefas definidas para executarem naquela máquina retornam erro de *timeout* e não executam. Este trabalho apresenta uma proposta para esta pendência, proporcionando alta disponibilidade e flexibilizando o escalonamento das tarefas de acordo com as máquinas ativas no *cluster*.

O monitoramento das máquinas do *cluster* mostrou-se uma técnica relevante quando o GDA apresenta tarefas com tempos de execução longos, neste caso, a alteração da disponibilidade dos equipamentos é determinante no tempo de execução das tarefas paralelas. A cada detecção de ganho ou perda de equipamento com implicações na execução da aplicação, antes do reinício da execução, todos os algoritmos são checados para verificar qual é o mais adequado para ser utilizado a partir daquele ponto. Esta proposta, apesar da sobrecarga que gera em monitoração, verificação do impacto na execução das tarefas e a escolha do melhor algoritmo de escalonamento para o reinicio do sistema, proporciona maior rapidez na finalização do processo devido ao fato de serem desconsideradas as tarefas já executadas. Entretanto como existe um custo na monitoração, em aplicações com tempo de processamento reduzido, em caso de indisponibilidade de alguma máquina do *cluster*, é recomendado que seja reiniciado todo o processo sem ponto de *restart*.

A operacionalização do monitoramento não ficou intrusiva ao sistema, uma vez as tarefas que controlam e monitoram esta atividade estão na máquina *front end* e por características da ferramenta, esta máquina fica dedicada ao controle dos tempos das tarefas.

Não ocorre intrusão na rede porque a verificação da presença das máquinas dá-se através da porta paralela, não onerando o tráfego.

Como extensão a este trabalho, sugere-se a implantação no *framework* de novas características, como a utilização da porta USB para aferição dos tempos e o suporte a máquinas heterogêneas conectadas ao *cluster*. Na máquina *front end*, onde os tempos são mostrados em tela, sugere-se a utilização de RAM *Disk Drive*, a fim de armazenar as informações em memória. Com a inclusão destas características sugere-se implementar o armazenamento, no *front end,* das trocas de dados entre as tarefas para recuperação, em caso de reescalonamento da aplicação.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] NISSANKE, N.; **Realtime Systems**. Prentice Hall. London. 1997. 441p.

[2] RICHARDSON, P.; ELKATEEB, A. M. Fault-Tolerant Adaptive Scheduling for Embedded Real-Time Systems. In: IEEE Micro. **IEEE Computer Society.** 2001, v.21, n.5, p.41 – 51.

[3] SAYENKO, A. et al. Adaptive Scheduling Using the Revenue-Based Weighted Round Robin. In: ICON'04. **Proceedings on 12th IEEE International Conference**. 2004, v.2, p.743 - 749.

[4] SRINIVASAN, A.; ANDERSON, J. H. Efficient Scheduling of Soft Real-time Applications on Multiprocessors. In: ECRTS'03. **Proceedings on 15th IEEE Euromicro Conference on Real-Time System.** 2003. p.51-54.

[5] SRINIVASAN, A.; ANDERSON, J. H. Optimal Rate-based Scheduling on Multiprocessors. In: STOC'02. **Proceedings of the 34th ACM Symposium on Theory of Computing**. 2002. p.189-198.

[6] BRANDT, S. A. The Case for Dynamic real-time Task Timing in Modern Real-time Systems. In: PDPS'04. **Proceedings 18th on International Parallel and Distributed Processing Symposium**. 2004. p.126.

[7] ANDERSON, J. H.; BUD, V.; DEVI, U. M. C. An EDF-based Scheduling Algorithm for Multiprocessor Soft Real-Time Systems**. In: ECRTS'05. 17th Euromicro Conference on Real-Time Systems**. 2005. p. 199 - 208.

[8] HARIDASAN, M.; PFITSCHER, G. H. Use of the parallel port to measure MPI intertask communication costs in COTS PC clusters. In: IPDPS'2003. **Proceedings of the International Parallel and Distributed Processing Symposium.** 2003. 6pp.

[9] WU, M. Y.; SHU, W.; GU, J. Efficient Local Search for DAG Scheduling. IEEE Transactions on Parallel and Distributed Systems. 2001, v.l2, n.6, p.617 – 627.

[10] KWOK, Y. W.; AHMAD, I. Static scheduling algorithms for allocating directed task graphs to multiprocessors. In: CSUR. **ACM Computing Surveys.** 1999, v.31, n.4, p.406 - 471.

[11] DERTOUZOS, M. L.; MOK, A. K. L. Multiprocessor On-Line Scheduling of Hard-Real-Time Tasks. IEEE Transactions on Software Engineering. 1989, v.15, n.12, p.1497 - 1506.

[12] ZHU, H.; LEHOCZKY, J. P.; HANSEN, J. P.; RAJKUMAR, R. Diff-EDF: A Simple Mechanism for Differentiated EDF Service. In: RTAS'05. **11th IEEE Real-Time and Embedded Technology and Applications Symposium**. 2005. p. 268 - 277.

[13] KREUZINGER, J.; BRINKSCHULTE, U.; PFEFFER, M.; UNGERER, T. A Scheduling Technique Providing a Strict Isolation of Real-time Threads. In: WORDS'02. **17th IEEE International Workshop on Object-oriented Real-time Dependable Systems.** 2002.

[14] KIM, K. H. K.; LIU, J. Going Beyond Deadline-Driven Low-Level Scheduling in distributed Real-Time Computing Systems. In: IFIP'02. **Proceedings 17th World Computer Congress**. 2002. p. 205-215.

[15] LEISERSON, C. E.; AGRAWAL, K.; HE, Y.; HSU, W. J. Adaptive Scheduling with Parallelism Feedback. In: PPoPP'06. **Proceedings of the Annual ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming**. 2006.

[16] HE, Y.; HSU, W. J. Provably Efficient Two-Level Adaptive Scheduling. In: JSSPP'06. **Proceedings of the 12th Workshop on Job Scheduling Strategies for Parallel Processing**. 2006.

[17] XIE, T.; NIJIM, M.; QIN, X. SHARP: A New Real-time Scheduling Algoritm to Improve Security of parallel Aplication on Heterogeneous Clusters. In: IPCCC'06. **25th IEEE International Performance, Computing and Comunications Conference**. 2006.

[18] BLUMENTHAL, J.; GOLATOWSKI, F.; HILDEBRANDT, J.; TIMMERMANN, D. YASA – A Framework for Validation, Test and Analysis of Real-Time Scheduling Algorithms. **Proceedings of 5th Real-Time Linux Workshop**. 2003. p. 197-204.

[20] LEISERSON, C. E.; AGRAWAL, K.; HE, Y. Adaptive Work Stealing with parallelism feedback**.** In: PPoPP'07. **Proceedings of the 11th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming**. 2007. p. 112 – 120.

[21] LEISERSON, C. E.; AGRAWAL, K.; HE, Y. An Empirical Evaluation of Work Stealing with Parallelism Feedback. In: ICDCS'06. **Proceedings of the International Conference on Distributed Computing Systems**. 2006.

[22] BARRETO, L. P.; MULLER G. A. Framework for simplifying the development of kernel schedulers: design and performance evaluation. **Technical report 02/8/INFO**, Ecole des Mines de Nantes. França, 2002.

[23] FU, Y. et al. SHARP: An Architecture for Secure Resource Peering. In: ACM Symposium on Operating Systems Principles**. Proceedings of the 19th ACM symposium on Operating systems principles**. 2003. p. 133 – 148.

[24] MEJIA-ALVAREZ, P.; LEVNER, E. Adaptive Scheduling Server for Power-Aware Real-Time Tasks. ACM Transactions on Embedded Computing Systems. 2004, v.3, n.2, p.284 - 306.

[25] ALDEA, M. et al. FSF: A Real-Time Scheduling Architecture Framework. In: RTAS'06. **Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium**. 2006. p. 113 - 124.

[26] HARIDASAN, M.; PFITSCHER, G. H. PM^2P: A tool for performance monitoring of message passing applications in COTS PC clusters. **Proceedings 15th Symposium on Computer Architecture and High Performance Computing**. 2003. p. 218 – 225.

[27] BLUMENTHAL, J.; GOLATOWSKI, F.; HILDEBRANDT, J.; TIMMERMANN, D. Framework for Validation, Test and Analysis of Real-Time Scheduling Algorithms and Scheduler Implementations. **Proceedings 13th IEEE International Workshop on Rapid System Prototyping**. 2002. p.146-152.

[28] SINGHOFF, F.; LEGRAND, J.; NANA L.; MARCÉ, L. Cheddar: A Flexible Real Time Scheduling Framework. ACM Ada Letters. ACM Press. **Proceedings of the ACM SIGADA International Conference**. 2004. p.15-18.

[29] SINGHOFF, F.; LEGRAND, J.; NANA, L.; MARCÉ, L. Cheddar 1.3p6 user's guide. Published as a LISYC technical report number singhoff-01-03. 2006.

[30] JOSÉ NETO, J. Tecnologia Adaptativa. Laboratório de Linguagens e Tecnologias Adaptativas, Escola Politécnica da Universidade de São Paulo**,** Set. 2004. Disponível em: <http://www.pcs.usp.br/~lta/roteiro_estudo/roteiro_estudo.html>. Acesso em: 21 ago. 2007.

[31] COTTET, F., DELACROIX, J.; KAISER, C.; MAMMERI, Z. **Scheduling in Real-Time Systems**. John Wiley & Sons Ltd. England, 2002.

[32] RAMAMRITHAM, K.; STANKOVIC, J. A.; SHIAH, P. F. Efficient scheduling algorithms for real-time multiprocessor systems. In: IEEE TPDS. **IEEE Transactions on Parallel and Distributed Systems**. 1990, v.1, n.2, p.184 – 194.

[33] RAMAMRITHAM, K.; STANKOVIC, J. A. Scheduling Algorithms and Operating Systems Support for Real-Time Systems. **Proceedings of the IEEE.** 1994, v.82, n.1, p.55 - 67.

[34] GAMBIER, A. Real-time Control Systems: A Tutorial. **Proceedings of the Asian Control Conference**. 2004. p.1023-1030.

[35] IGNAT, N.; NICOLESCU, B.; SAVARIA, Y.; NICOLESCU, G. Soft-Error Classification and Impact Analysis on Real-Time Operating Systems. Design, Automation, and Test in Europe. **Proceedings of the conference on Design, automation and test in Europe**. 2006. p.182 – 187.

[36] RadSys Corporation. MicroWare OS-9. p. 1, 2007. Disponível em: <http://www.radisys.com/products/Microware-OS-9.cfm >. Acesso em: 25 set. 2007.

[37] RTAI. The RealTime Application Interface for Linux from DIAPM. p. 1, 2007. Disponível em: <http://www.rtai.org >. Acesso em: 25 set. 2007.

[38] S.Ha.R.K. Soft Hard Real-Time Kernel. p. 1, 2007. Disponível em: <http://shark.sssup.it/>. Acesso em: 25 set. 2007.

[39] RTLINUX. Real Time Linux Foundation, Inc. p. 1, 2007. Disponível em: <http://www.realtimelinuxfoundation.org/>. Acesso em: 25 set. 2007.

[40] TIMESYS Corporation. **The Concise Handbook Of Real-Time Systems.** Version 1.3, Pittsburgh, PA, USA. 2002.

[41] LIU, C. L.; LAYLAND, J. W. Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment. **Journal of the Association for Computing Machinery.** 1973, v.20, n.1, p.44 - 61.

[42] SHA, L. et al. Real Time Scheduling Theory: A Historical Perspective. **Computer Science Real Time Systems Journal.** 2004, v.28, n.2-3, p.101 - 155.

[43] SAEZT, S.; VILA, J.; CRESPO A. A Dynamic Real-Time Scheduler for Shared Memory Multiprocessors. In: Real-Time Systems'96. **Proceedings of the 8th Workshop on Euromicro.** 1996. p.158 – 163.

[44] FERRARI, D. ZHOU, S. A load Index for Dynamic Load Balancing. In: ACM'86. **Proceedings of 1986 ACM Fall joint computer conference**. 1986, p. 684-690.

[45] CASAVANT, T. L.; KUHL, J. G. A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. IEEE Transactions on Software Engineering. 1988, v.l, se-14, n.2, p.141 - 154.

[46] AHMAD, I.; KWOK, Y.-K.; WU, M.-Y. Performance Comparison of Algorithms for Static Scheduling of DAGs to Multiprocessors. **Proceedings of the 2nd Australian Conference on Parallel and Real-Time Systems.** 1995. p.185-192.

[47] RAMAMRITHAM, K., STANKOVIC, J. A. Dynamic Task Scheduling in Hard Real-Time Distributed Systems. Software IEEE. 1984, v.1, n.3, p.65 - 75.

[48] FUNK, S.; BARUAH, S. Characteristics of EDF Schedulability on Uniform Multiprocessors. In: IEEE Computer Society ECRTS'03. **Proceedings 15th Euromicro Conference on Real-Time Systems**. 2003. p.211.

[49] EL-REWINI, H.; LEWIS, T. G. **Distributed and Parallel Computing.** Manning Pubs Co. January, 1998.

ANEXO I – ROTEIRO COM AS CARACTERÍSTICAS DO SISTEMA

*** Início do Processo ***

(1) Conexão do hardware e início da ferramenta.

a) *Verificar se todas as máquinas estão em funcionamento e devidamente conectadas via porta paralela e porta TCP/IP.*

b) *Iniciar o framework.*

c) *Aguardar comandos do usuário.*

d) *Ir para o passo* (16) *se for enceramento da ferramenta, senão, executar o comando solicitado.*

(2) Carga ou Cadastramento das máquinas no cluster.

a) *Se solicitada opção para carregar um cluster de máquinas gravado em arquivo, carregar todas as informações do cluster a partir do arquivo e se houver grafo de tarefas vinculado, carregar também as informações do grafo, ir para o passo* (4-a).

b) *Se solicitada opção para criar novo cluster de máquinas para vinculação a um grafo, disponibilizar interface para cadastramento das máquinas, ir para o passo* (3-a).

d) Se s*olicitada a gravação das modificações do cluster, ir para o passo* (3-b).

(3) Criação das máquinas no cluster.

a) *Gerar o array de máquinas com as informações sobre as máquinas do cluster. Solicitar ao usuário quais máquinas serão cadastradas e as demais informações, ir para o passo* (4-a).

b) *Gravar as informações do cluster de máquinas em arquivo. Se existir algum grafo de tarefas em uso, vincular grafo ao cluster, gravando-o também em arquivo, ir para o passo* (1-c).

(4) Verificação da conectividade dos equipamentos cadastrados.

a) *Efetuar a chamada ao programa test-slave(id_machine) para todas as máquinas cadastradas no cluster.*

b) *Efetuar o teste de presença das máquinas cadastradas e alterar o status e as respectivas imagens (retirar ou colocar o traço em x indicando a indisponibilidade), ir para o passo* (1-c).

(5) Carga ou criação de grafo de tarefas.

a) *Se solicitada a opção de criação de grafo automaticamente pelo sistema, ir para o passo* (6-a).

b) *Se solicitada a opção de criação de grafo manual, disponibilizar interface para criar manualmente um novo grafo, ir para o passo* (6-b).

c) *Se solicitada a opção de carregar um grafo de tarefas já gravado pelo sistema, carregar todas as informações do grafo, a partir do arquivo escolhido pelo usuário.*

d) Se *solicitada a gravação das modificações do grafo, ir para o passo* (6-d).

(6) Montagem de um grafo de tarefas.

a) *Criar grafo arbitrário de tarefas, a partir dos parâmetros informados pelo usuário, através da chamada a classe GraphGenerator, ir para o passo* (1-c).

b) *A partir da interface apresentada para construção do grafo de tarefas, através da chamada a classe Graph, montar cada tarefa e suas respectivas conexões. Solicitar ao usuário os parâmetros de cada tarefas e suas respectivas conexões.*

c) *Encerrar o cadastramento, fechando as janelas de parâmetros, ir para o passo* (1-c).

d) *Gravar as informações do grafo de tarefas e suas respectivas conexões em arquivo, ir para o passo* (1-c).

(7) Geração de Informações para escalonamento.

a) *Se solicitada a opção de gerar a identificação dos parâmetros sobre as máquinas do cluster.*

b) *Chamar a opção identify parameters para definir tempo médio que cada máquina gasta por ciclo de CPU. Apresentar janela e aguardar confirmação das informações, ir para o passo* (1-c).

(8) Geração de escalonamentos.

a) *Se solicitada a opção para gerar novo escalonamento do grafo que está em uso, verificar qual escalonamento foi escolhido pelo usuário.*

b) *Calcular o escalonamento definido e apresentar o resultado em janela com o mapa de Gantt, ir para o passo* (1-c).

(9) Apresentação da documentação técnica da ferramenta.

a) *Se* s*olicitada a opção para visualizar a documentação técnica da ferramenta, chamar browser ativo, passando como parâmetro o arquivo html inicial da documentação.*

b) *Após o fechamento do browser, ir para o passo* (1-c).

(10) Opção para execução automática ou manual da aplicação, representada pelo grafo.

a) *Se* s*olicitada a opção simulate and execute será criado um sistema exemplo, de acordo com o grafo de tarefas e iniciado sua execução em paralelo, efetuado todos os passos automaticamente e com monitoração, ir para o passo* (11-a).

b) *Se* s*olicitada a opção simulate será criado pelo framework um sistema exemplo, vinculado ao grafo de tarefas e simulando o tempo de cada tarefa com suas respectivas dependências, ir para o passo* (11-b).

c) *Se* s*olicitada a opção execute, para que seja iniciada a execução em paralelo do sistema que está vinculado ao grafo de tarefas, ir para o passo* (11-c).

(11) Preparação para execução do sistema.

a) *carregar as informações do grafo e do cluster e efetuar todos os procedimentos para execução do grafo automaticamente, com monitoramento das máquinas do cluster, ir para o passo* (12-a).

b) *carregar as informações do grafo e gerar todos os programas exemplo, os quais são dimensionados para utilizarem o tempo previsto da tarefa e para enviar/receber mensagens de acordo com as dependências descritas no grafo, ir para o passo* (1-c).

c) *carregar as informações do grafo e do cluster, gerar arquivo procgroup com as tarefas em suas respectivas máquinas, de acordo com a ordem gerada pelo escalonamento, ir para o passo* (14-a).

(12) Identificação do algoritmo de escalonamento.

a) *Calcular o escalonamento do grafo utilizando todos os algoritmos disponíveis no sistema e salvar o custo de execução (makespan) e o tempo para executar esta tarefa por escalonamento.*

b) *Escolher o algoritmo com menor custo de execução.*

(13) Preparação para execução automática do sistema.

a) *Carregar as informações do grafo e gerar programas que executam um loop durante o tempo previsto da tarefa e enviar/receber mensagens de acordo com as dependências descritas no grafo.*

b) *Carregar as informações do cluster, gerar arquivo procgroup com as tarefas em suas respectivas máquinas, de acordo com a ordem gerada pelo escalonamento. São incluídas neste arquivo somente as tarefas que ainda não foram marcadas como executadas.*

(14) Execução em paralelo do sistema.

a) *Chamar programa MPI para execução das tarefas em paralelo, de acordo com o arquivo procgroup criado. O sistema será executado nas máquinas escravas e passará para a máquina front end os tempos de cada tarefa. Via porta paralela são sinalizados ao front end, o início e término de cada tarefa.*

b) *Se a execução for com monitoração, iniciar em paralelo, se ainda não iniciado, thread com o monitoramento da disponibilidade das máquinas do cluster, senão, ir para o passo* (14-h).

c) *Manter o monitoramento enquanto o cluster estiver executando as tarefas.*

d) *Re-escalonar o grafo se houver alteração na disponibilidade das máquinas, de acordo com os passos* (12-a) *e* (12-b).

e) *Em caso de perda de máquinas, re-escalonar somente se a máquina que saiu ainda possuía tarefas programadas para execução.*

f) *Em caso de ganho de máquinas, re-escalonar somente se com nova máquina houver diminuição no tempo de execução do sistema.*

g) *Se houve re-escalonamento do grafo (com ganho ou perda de máquinas), parar a atual execução do sistema, ignorar a perda das trocas de mensagens, ir para o item* (13-b), *senão, ir para o próximo item.*

h) *Após o encerramento da execução do sistema, apresentar mapa de Gantt detalhando o resultado da execução das tarefas por máquina.*

i) *Se a opção de execução foi com monitoração, encerrar o processo de monitoração e apresentar o tempo gasto com a tarefa de escolha do menor escalonamento e eventuais re-escalonamentos.*

(15) Encerramento do processo.

a) *Fechar as janelas com os resultados do escalonamento.*

b) *Ir para o passo (1-c).*

(16) Encerramento da ferramenta.

a) *Fechar todas as janelas e finalizar o framework.*

*** Fim do Processo ***

ANEXO II – PROGRAMA SCHEDULING.JAVA COM OS ALGORITMOS DE ESCALONAMENTO

```
package project.control;
/*
 * Classname: ThreadClusterController
 *
 * Version information: 1.0
 *
 * Date: 2007/07/21
 *
 */
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;
import java.io.*;
import project.model.*;
import project.GanttChart.*;

/**
 * This class implements a thread controller to support machines available
 *
 * @version  1.0 23 Jul 2007
 */

public class ThreadClusterController extends Thread {
                /** object to the clusterController of tasks */
                ClusterController clusterController;

                /** Index that determinate the smaller scheduler of the current graph */
                static int indexSched;

                /** Directory where auxiliary executable files are stored */
```

```
static String execFilesDirectory;

/**
 * Constructor method
 */
public ThreadClusterController(String sExecFilesDirectory, ClusterController oClusterController, int iIndexSched) {

        try {
            execFilesDirectory = sExecFilesDirectory;
            clusterController = oClusterController;
            indexSched = iIndexSched;
        }
        catch (Exception e) {
            JOptionPane.showMessageDialog(null, e);
        }
}

/**
 * @return null
 */
public void run() {

  Graph graph = Status.getGraphController().getGraph();
  Cluster cluster = Status.getCluster();

  while(true){
        int numberOfMachines = cluster.getNumberOfMachines();
       // do something if returns false otherwise do nothing
       int resultTest;
       try {
           resultTest = testMachines(cluster, graph);
           if (resultTest != 0) {          // Lost or gained machines
```

```
                        clusterController.setChanged(resultTest);
                    }
                    try { Thread.sleep(10000); }
                      catch (InterruptedException e) {
                            JOptionPane.showMessageDialog(null, "Thread stop while
sleeping: " + e);
                     break;
                     }
                }
                    catch (Exception e) {
                        JOptionPane.showMessageDialog(null, "Error while testing
machines: " + e);
                    }
                }
                }

                /**
                * Tests which machines of the cluster are available and usable
                * Tests all machines, even those which are not selected
                * @param cluster
                * @throws Exception
                * @return array machines
                */
                static public int testMachines(Cluster cluster, Graph graph) throws Exception {

//                int numberOfMachines = cluster.getNumberOfMachines();
                 ArrayList machines = cluster.getAllMachines();
                 int numberOfMachines = machines.size();
                 Machine machine;

                 Runtime r=Runtime.getRuntime();
                 String masterProgram = execFilesDirectory + "frontendTest";
                 String slaveProgram = execFilesDirectory + "machineTest";
```

```
Process masterProcess, slaveProcess;
BufferedReader in;
int pin;
int changed = 0;
double makespan;

for (int i = 0; i < numberOfMachines; i++) {

pin = -1;
machine = cluster.getMachineOfIndex(i);

        // Ex:    \home\maya\project\execFiles\frontend
        masterProcess = r.exec(masterProgram);
        System.out.println("Test: Executing... masterProgram");

        // Ex:    rsh clusterX \home\maya\project\execFiles\slavePinTest
        slaveProcess = r.exec("rsh " + machine.getId() + " " + slaveProgram);

        //slaveProcess = r.exec(slaveProgram);
        in = new BufferedReader(new
InputStreamReader(slaveProcess.getInputStream()));
        System.out.println("Test: Executing... rsh " +
                machine.getId() + " " + slaveProgram);
        System.out.println("Slave output: " + in.readLine());

        masterProcess.waitFor();
        slaveProcess.destroy();

        in = new BufferedReader(new
InputStreamReader(masterProcess.getInputStream()));
        pin = Integer.parseInt(in.readLine());

// if the machine answer - verify if in use and test the scheduler
```

```
                    if (pin != -1 ) {
                  // if not selected/active, verify and put the new machine to execute if is
necessary
                  if ( !machine.isSelected() || !machine.isActive()) {

                        int indexShedTemp;
                        makespan = graph.getMakespan();
                        machine.setActive(true);
                        machine.setSelected(true);
                        // make the test
                        indexShedTemp = MPIController.makeScheduling(-1, graph, cluster,
false);

                        // if we gain some time with new machine
                        if (makespan > graph.getMakespan()) {
                                machine.setPinNumber(pin);
                                        changed = 1;
                              indexSched = indexShedTemp;
                              MPIController.makeScheduling(indexSched,  graph,  cluster,
true);
                              System.out.println("pin number " + pin);
                        }
                        else {     // undo the process and dont use
                              machine.setActive(false);
                              machine.setSelected(false);
                              MPIController.makeScheduling(indexSched,  graph,  cluster,
false);
                        }
                  }
                    }

              // machine no answer, verify if selected and the impact
              else if ( pin < 1 && (machine.isActive() || machine.isSelected()) ) {
```

```
                    makespan = graph.getMakespan();
                    machine.setActive(false);
                    machine.setSelected(false);
                    // make the test
                    MPIController.makeScheduling(indexSched, graph, cluster, false);

                    // if we lost some time with the break machine redo the scheduling
                    if (makespan < graph.getMakespan()) {
                                machine.setActive(false);
                        indexSched = MPIController.makeScheduling(-1, graph, cluster,
true);
                        changed = -1;
                    }

                }
                    in.close();
                }
                return changed;
            }
        }
```

ANEXO III – PROGRAMA *THREADCLUSTERCONTROL.JAVA* COM AS ROTINAS DE MONITORAÇÃO DO CLUSTER

```
package project.scheduling;

/*
 * Classname: Scheduler
 *
 * Version information: 2.0
 *
 * Date: 2007/12/08
 *
 */

import java.util.*;
import java.io.*;
import project.model.*;
import project.GanttChart.*;

 /**
  * This class implements scheduling functions that schedules a graph
  * into a set of processors using different algorithms
  *
  * @version   2.0 15 Jul 2007
 */

public class Scheduler {

        /** Indicates whether all machines or only the active ones
         * are used for the scheduling
         */

        static boolean allMachines = true;

      //* Front end Machine's cpu clock rate given in Hertz (homogeneous cluster) */
      public static double cpuClock = 349000000;

        /**
         * Sets whether all machines should be used in the scheduling
         * or whether only the active ones should be used.
         * @param state
         */
        public static void setAllMachines(boolean state) {

                allMachines = state;
        }

        /**
         * @return whether all machines should be used in the scheduling
```

```
 * or whether only the active ones should be used.
 */
public static boolean isAllMachines() {

	return allMachines;
}

/**
* Updates the deadline task by the t-level calc of every task in a graph
* It's using the bLevel variable to store the values.
* @param graph
*/
private static void calculateTlevel(Graph graph) throws Exception {
	ArrayList edges = graph.getLinks();
	int numberOfEdges = graph.getNumberOfLinks();
	int i = 0;

	SystemParameters sp = SystemParameters.instance();
	double to = sp.getTo();
	double tr = sp.getTr();
	if ((to == 0) || (tr == 0)) {
		throw new Exception("System Parameters must be updated.");
	}

	// Updates the transmission time of each edge
	double tt;
	int dataType;
	Link edge;
	for (i = 0; i < numberOfEdges; i++ ) {
		edge = ((Link)edges.get(i));
		dataType = edge.getDataType();
		tt = edge.getNumberOfDataItems()*sp.sizeOf(dataType);
		tt = to + (tt-1)*tr;
		edge.setTransmissionTime(tt);
	}

	Task[] tasks = graph.getTopologicalOrder();
	int index = tasks.length;

	double max, com;
	int j;
	int numberOfParents;
	Task parent;

	clearStartTimes(graph);

	for ( i = 0; i < index; i++) {
		max = 0;
		numberOfParents = tasks[i].getNumberOfEntryLinks();
		for (j = 0; j < numberOfParents; j++) {
```

```
                        parent = tasks[i].getEntryLinkOfIndex(j).getSource();
                        if (parent.getBlevel() > max) {
                                max = parent.getBlevel();
                        }
                }
                tasks[i].setBlevel(tasks[i].getExecutionTime() + max);

                if (tasks[i].getDeadlineTime() <= 0) {
                        tasks[i].setDeadlineTime(tasks[i].getExecutionTime() + max);
                }
        }
    graph.setMakespan(tasks[(index-1)].getBlevel() + tasks[index -
1].getExecutionTime() );
    System.out.println("Makespan :" + graph.getMakespan());
}

/**
* Updates the static b-level of every task in a graph
* @param graph
*/
private static void calculatebLevel(Graph graph) throws Exception {

        ArrayList edges = graph.getLinks();
        int numberOfEdges = graph.getNumberOfLinks();

        SystemParameters sp = SystemParameters.instance();
        double to = sp.getTo();
        double tr = sp.getTr();
        if ((to == 0) || (tr == 0)) {
                throw new Exception("System Parameters must be updated.");
        }

        // Updates the transmission time of each edge
        double tt;
        int dataType;
        Link edge;
        for (int i = 0; i < numberOfEdges; i++ ) {
                edge = ((Link)edges.get(i));
                dataType = edge.getDataType();
                tt = edge.getNumberOfDataItems()*sp.sizeOf(dataType);
                tt = to + (tt-1)*tr;
                edge.setTransmissionTime(tt);
        }

        Task[] tasks = graph.getTopologicalOrder();
        int index = tasks.length;
        double max, com;
        int j;
        int numberOfChildren;
        Task child;
```

```
                clearStartTimes(graph);
                while (index != 0) {
                        index--;
                        max = 0;
                        numberOfChildren = tasks[index].getNumberOfExitLinks();
                        for (j = 0; j < numberOfChildren; j++) {
                                child = tasks[index].getExitLinkOfIndex(j).getDestination();
                                if (child.getBlevel() > max) {
                                        max = child.getBlevel();
                                }
                        }
                        tasks[index].setBlevel(tasks[index].getExecutionTime() + max);
                        if (tasks[index].getDeadlineTime() <= 0) {
                                tasks[index].setDeadlineTime(tasks[index].getExecutionTime()
+ max);
                        }
                }

            graph.setMakespan(tasks[0].getBlevel() + tasks[0].getExecutionTime());
            System.out.println("Makespan :" + graph.getMakespan());
        }

        public static void EDFMschedule(Graph graph, Cluster cluster, boolean show) throws
Exception {
                Task task, parent, child;
                Link link;
                Machine machine;

            System.out.println("EDFM tempos: /n");
            long time1 = getTime();

                int numberOfChildren, numberOfParents;
                int numberOfMachines = cluster.getNumberOfMachines();
                int numberOfTasks = graph.getNumberOfTasks();
                // auxiliary variable to indicate if a node is ready to enter the ready list
                boolean ready;
                ArrayList readyList = graph.getEntryTasks();
                ArrayList nextReadyList = new ArrayList();
                double[] machineTimes = new double[numberOfMachines];
                double[] startTimes = new double[numberOfMachines];
                int earliestMachine = 0, earliestTask = 0;
                double earliestTime = Double.POSITIVE_INFINITY;
                double time, newTime;

                int index = 0;
                int i, j, k;

                // Compute the static t-level of each node
                calculateTlevel(graph);
            Task taskTemp = graph.getTask(1);
```

```
                for (i = 0; i < numberOfMachines; i++) {
                        machineTimes[i] = 0;
                }

                while (readyList.size() != 0) {
                        earliestMachine = 0;
                        earliestTime = Double.POSITIVE_INFINITY;
                        earliestTask = 0;

                        // Calculate the earliest start-time on each processor for each
                        // node in the ready pool.
                        for (i = 0; i < readyList.size(); i++) {

                                task = (Task)readyList.get(i);
                                numberOfParents = task.getNumberOfEntryLinks();
                                for (j = 0; j < numberOfMachines; j++) {
                                        machine = cluster.getMachineOfIndex(j);
                                        time = 0;
                                        // Selects the longest time dependence to parents
                                        // Before this time is not possible to start the task
                                        for (k = 0; k < numberOfParents; k++) {
                                                link = task.getEntryLinkOfIndex(k);
                                                parent = link.getSource();
                                                newTime = parent.getStartTime() +
parent.getExecutionTime();
                                                if (parent.getMachine() != machine)
                                                        newTime += link.getTransmissionTime();
                                                time = Math.max(time, newTime);
                                        }

                                        // Pick the node-processor pair that gives the earliest
                                        // time using the non-insertion approach.
                                        if (Math.max(time, machineTimes[j]) < earliestTime) {
                                                earliestTime = Math.max(time,
machineTimes[j]);
                                                earliestMachine = j;
                                                earliestTask= i;
                                        }
                                        // Ties are broken by selecting the node with a lower
                                        // static t-level
                                        if (Math.max(time, machineTimes[j]) == earliestTime) {
                                                if (task.getBlevel() <

        ((Task)readyList.get(earliestTask)).getBlevel()) {
                                                        earliestTime = Math.max(time,
machineTimes[j]);
                                                        earliestMachine = j;
                                                        earliestTask= i;
                                                }
                                        }
```

```
                                        // Check the deadline parameter and choose the earliest
deadline
                                        if(                task.getDeadlineTime()                <
((Task)readyList.get(earliestTask)).getBlevel() ) {
                                                earliestTask = i;
                                        }
                                }
                        }

                        task = (Task)readyList.get(earliestTask);

                        // Schedule earliest task
                        task.setMachine(cluster.getMachineOfIndex(earliestMachine));
                        task.setStartTime(earliestTime);
                        task.mark(true);
                        machineTimes[earliestMachine]          =          earliestTime          +
task.getExecutionTime();
                        // Remove scheduled task from ready pool
                        readyList.remove(earliestTask);

                        numberOfChildren = task.getNumberOfExitLinks();

                        // Add the newly ready nodes to the ready node pool.
                        for (j = 0; j < numberOfChildren; j++) {

                                child = task.getExitLinkOfIndex(j).getDestination();
                                numberOfParents = child.getNumberOfEntryLinks();
                                k = 0;
                                ready = true;
                                while ((k < numberOfParents) && (ready == true)) {
                                        parent = child.getEntryLinkOfIndex(k).getSource();
                                        ready = parent.isMarked();
                                        k++;
                                }
                                if (ready == true) {
                                        readyList.add(child);
                                        child.setAdded(true);
                                }
                        }
                }
                long time2 = getTime();
                printTime(time1, time2);
                if (show) {
                    createPreviewGanttChart(graph, cluster);
            }

        }
```

```
        public static void LLFMschedule(Graph graph, Cluster cluster, boolean show) throws
Exception {

                Task task, parent, child;
                Link link;
                Machine machine;
            System.out.println("LLFM tempos: /n");
            long time1 = getTime();
                int numberOfChildren, numberOfParents;
                int numberOfMachines = cluster.getNumberOfMachines();
                int numberOfTasks = graph.getNumberOfTasks();
                // auxiliary variable to indicate if a node is ready to enter the ready list
                boolean ready;
                ArrayList readyList = graph.getEntryTasks();
                ArrayList nextReadyList = new ArrayList();
                double[] machineTimes = new double[numberOfMachines];
                int earliestMachine = 0;
                double earliestTime;
                double time, newTime;

                int index = 0;
                int i, j, k;

                // set the deadline default if not set by user
                calculateTlevel(graph);

                // Calculate the static b-level of each node
                calculatebLevel(graph);

                for (i = 0; i < numberOfMachines; i++) {
                        machineTimes[i] = 0;
                }

                while (readyList.size() != 0) {
                // Ordering by start
                Collections.sort(readyList, new Comparator() {
                                                        public int compare(Object a, Object b) {
                                                          double blevel = ((Task)a).getStartTime()
- ((Task)b).getStartTime();
                                                            if (blevel < 0) return -1;
                                                            if (blevel > 0) return 1;
                                                                return 0;
                                                          }
                                                });
                        // Make a ready list in a descending order of static b-level.
                        Collections.sort(readyList, new Comparator() {
                                                        public int compare(Object a, Object b) {
                                                          double  blevel  =  ((Task)a).getBlevel()  -
((Task)b).getBlevel();
                                                            if (blevel < 0) return 1;
```

```
                                        if (blevel > 0) return -1;
                                            return 0;
                                        }
                                    });

                task = (Task)readyList.get(0);
                numberOfParents = task.getNumberOfEntryLinks();
                earliestMachine = 0;
                earliestTime = Double.POSITIVE_INFINITY;

                // Schedule the first node in the ready list to a processor that
                // allows the earliest execution, usingthe non-linear approach.
                for (j = 0; j < numberOfMachines; j++) {
                        machine = cluster.getMachineOfIndex(j);
                        time = 0;

               if (machine.isActive()) {
                        // Selects the longest time dependence to parents
                        for (k = 0; k < numberOfParents; k++) {
                                link = task.getEntryLinkOfIndex(k);
                                parent = link.getSource();
                                newTime = parent.getStartTime() +
parent.getExecutionTime();
                                if (parent.getMachine() != machine)
                                        newTime += link.getTransmissionTime();
                                    time = Math.max(time, newTime);
                        }
                        if (Math.max(time, machineTimes[j]) < earliestTime) {
                                earliestTime = Math.max(time, machineTimes[j]);
                                earliestMachine = j;
                            }
                        }
                }

                // take off the task
                task.setMachine(cluster.getMachineOfIndex(earliestMachine));
                task.setStartTime(earliestTime);
                task.mark(true);
                machineTimes[earliestMachine] = earliestTime +
task.getExecutionTime();

                numberOfChildren = task.getNumberOfExitLinks();

                // Update the ready list by inserting the nodes that are now ready
                for (j = 0; j < numberOfChildren; j++) {

                        child = task.getExitLinkOfIndex(j).getDestination();
                        numberOfParents = child.getNumberOfEntryLinks();
                        k = 0;
                        ready = true;
```

```
                        while ((k < numberOfParents) && (ready == true)) {
                                parent = child.getEntryLinkOfIndex(k).getSource();
                                ready = parent.isMarked();
                                k++;
                        }
                        if (ready == true)
                                readyList.add(child);
                }
                readyList.remove(0);
        }

    long time2 = getTime();
    printTime(time1, time2);

    if (show) {
            createPreviewGanttChart(graph, cluster);
    }
}

    public static void LLFPschedule(Graph graph, Cluster cluster, boolean show) throws
Exception {

        Task task, parent, child;
        Link link;
        Machine machine;

    System.out.println("LLFP tempos: /n");

    long time1 = getTime();

        int numberOfChildren, numberOfParents;
        int numberOfMachines = cluster.getNumberOfMachines();
        int numberOfTasks = graph.getNumberOfTasks();
        // auxiliary variable to indicate if a node is ready to enter the ready list
        boolean ready;
        ArrayList readyList = graph.getEntryTasks();
        ArrayList nextReadyList = new ArrayList();
        double[] machineTimes = new double[numberOfMachines];
        int earliestMachine = 0;
        double earliestTime;
        double time, newTime;

        int index = 0;
        int i, j, k;

        // set the deadline default if not set by user
        calculateTlevel(graph);

        // Calculate the static b-level of each node
        calculatebLevel(graph);
```

```
                for (i = 0; i < numberOfMachines; i++) {
                        machineTimes[i] = 0;
                }

                while (readyList.size() != 0) {

                        // Make a ready list in a descending order of static b-level.
                        Collections.sort(readyList, new Comparator() {
                                                        public int compare(Object a, Object b) {
                                                          double  blevel  =  ((Task)a).getBlevel()  -
((Task)b).getBlevel();
                                                            if (blevel < 0) return 1;
                                                            if (blevel > 0) return -1;
                                                                return 0;
                                                          }
                                                });

                        // If the priority task is set make another ready list in a ascending order
of Priority.
                        Collections.sort(readyList, new Comparator() {
                                                        public int compare(Object a, Object b) {
                                                          double                blevel                =
((Task)b).getPriorityTask() - ((Task)a).getPriorityTask();
                                                            if (blevel < 0) return -1;
                                                            if (blevel > 0) return 1;
                                                                return 0;
                                                          }
                                                });

                        task = (Task)readyList.get(0);
                        numberOfParents = task.getNumberOfEntryLinks();
                        earliestMachine = 0;
                        earliestTime = Double.POSITIVE_INFINITY;

                        // Schedule the first node in the ready list to a processor that
                        // allows the earliest execution, usingthe non-linear approach.
                        for (j = 0; j < numberOfMachines; j++) {

                                machine = cluster.getMachineOfIndex(j);
                                time = 0;
                      if (machine.isActive()) {

                                // Selects the longest time dependence to parents
                                for (k = 0; k < numberOfParents; k++) {
                                        link = task.getEntryLinkOfIndex(k);
                                        parent = link.getSource();
                                        newTime          =          parent.getStartTime()          +
parent.getExecutionTime();
                                              if (parent.getMachine() != machine)
```

```
                                        newTime += link.getTransmissionTime();
                                    time = Math.max(time, newTime);
                                }
                                if (Math.max(time, machineTimes[j]) < earliestTime) {
                                        earliestTime = Math.max(time, machineTimes[j]);
                                        earliestMachine = j;
                                    }
                                }

                            }

                            // take off the task
                            task.setMachine(cluster.getMachineOfIndex(earliestMachine));
                            task.setStartTime(earliestTime);
                            task.mark(true);
                            machineTimes[earliestMachine] = earliestTime +
task.getExecutionTime();

                            numberOfChildren = task.getNumberOfExitLinks();

                            // Update the ready list by inserting the nodes that are now ready
                            for (j = 0; j < numberOfChildren; j++) {
                                    child = task.getExitLinkOfIndex(j).getDestination();
                                    numberOfParents = child.getNumberOfEntryLinks();
                                    k = 0;
                                    ready = true;
                                    while ((k < numberOfParents) && (ready == true)) {
                                            parent = child.getEntryLinkOfIndex(k).getSource();
                                            ready = parent.isMarked();
                                            k++;
                                    }
                                    if (ready == true)
                                            //readyList.add(child);
                                nextReadyList.add(child);
                            }
                            readyList.remove(0);
                        if ( readyList.size() == 0) {
                            while (nextReadyList.size() != 0) {
                                readyList.add(nextReadyList.get(0));
                                nextReadyList.remove(0);
                            }
                        }
                        }

                long time2 = getTime();
                printTime(time1, time2);
                    if (show) {
                   createPreviewGanttChart(graph, cluster);
                }
            }
```

```
/**
 * Creates a gantt chart of the current schedule
 * of the application
 */
static public void createPreviewGanttChart(Graph graph, Cluster cluster) {

        GanttChart ganttChart = new GanttChart();
        GanttLine ganttLine;
        GanttTask ganttTask;
        Task task;
        int numberOfMachines, numberOfTasks, index;
        ArrayList machines;
        ArrayList lines = new ArrayList();
        if (allMachines) {
                machines = cluster.getAllMachines();
        }
        else {
                machines = cluster.getActiveMachines();
        }
        numberOfMachines = machines.size();
        for (int i = 0; i < numberOfMachines; i++) {
                ganttLine = new GanttLine(cluster.getMachineOfIndex(i).getId());
                lines.add(ganttLine);
        }

        numberOfTasks = graph.getNumberOfTasks();
        for (int i = 0; i < numberOfTasks; i++) {
                task = graph.getTaskOfIndex(i);
                ganttTask = new GanttTask("t" + task.getId(),
                                                                task.getStartTime());
                ganttTask.setFinalTime(task.getStartTime()+task.getExecutionTime());
                index = cluster.getMachineIndex(task.getMachine());
                ((GanttLine)lines.get(index)).addTask(ganttTask);
        }

        for (int i = 0; i < numberOfMachines; i++) {
                ganttChart.addGanttLine((GanttLine)lines.get(i));
        }

        ganttChart.update();
        ganttChart.fitToWidth();
        ganttChart.show();

}

static public void clearStartTimes(Graph graph) {

         int numberOfTasks = graph.getNumberOfTasks();
```

```
                Task task;
                for (int i = 0; i < numberOfTasks; i++) {
                        task = graph.getTaskOfIndex(i);
                        task.mark(false);
                        task.setStartTime(0);
                        task.setAdded(false);
                        task.setBlevel(0);
                }
        }

        /**
         * Show the time by the cpuclok values
         */
        static public void printTime(long time1, long time2) throws Exception {

                long diff;
                double timeMs;
                diff = time2 - time1;
                timeMs = (diff/cpuClock) * 1000;
                System.out.println("Tempo em Ms = " + timeMs);
        }

        /**
         * Show the time by the cpuclok values
         */
        static public long getTime() throws Exception {
                Process process;
                BufferedReader in;
                String command1;
                Runtime r;
                long time;

                command1 = "/home/jorge/laico/execFiles/getTime";
                r = Runtime.getRuntime();
                process = r.exec(command1);
                in = new BufferedReader(new InputStreamReader(process.getInputStream()));
                process.waitFor();
                time = Long.parseLong(in.readLine());

           return time;
        }
}
```