UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

TUXUR - UM *FRAMEWORK* PARA DIVISÃO DINÂMICA DE TAREFAS MALEÁVEIS EM GRADE COMPUTACIONAL

DISSERTAÇÃO DE MESTRADO

Roberto Wiest

Santa Maria, RS, Brasil

2014

TUXUR - UM *FRAMEWORK* PARA DIVISÃO DINÂMICA DE TAREFAS MALEÁVEIS EM GRADE COMPUTACIONAL

por

Roberto Wiest

Dissertação apresentada ao Programa de Pós-Graduação em Informática
da Universidade Federal de Santa Maria (UFSM, RS), como requisito
parcial para a obtenção do grau de
Mestre em Computação

Orientador: Prof. Dr. Benhur de Oliveira Stein (UFSM)

Santa Maria, RS, Brasil

2014

Universidade Federal de Santa Maria
Centro de Tecnologia
Programa de Pós-Graduação em Informática

A Comissão Examinadora, abaixo assinada,
aprova a Dissertação de Mestrado

TUXUR - UM *FRAMEWORK* PARA DIVISÃO DINÂMICA DE TAREFAS MALEÁVEIS EM GRADE COMPUTACIONAL

elaborada por
Roberto Wiest

como requisito parcial para obtenção do grau de
Mestre em Computação

COMISSÃO EXAMINADORA:

Prof. Dr. Benhur de Oliveira Stein (UFSM)
(Presidente/Orientador)

Prof. Dr. Lucas Mello Schnorr (UFRGS)

Prof. Dr. Patrícia Pitthan de Araújo Barcelos (UFSM)

Santa Maria, 23 de Maio de 2014.

AGRADECIMENTOS

Quero primeiramente agradecer a Deus, por ter me dado saúde, paciência e sabedoria para conduzir este trabalho até o final. Quero agradecer também ao meu orientador, professor Dr. Benhur Stein, pelas dicas preciosas, pela orientação sempre paciente e atenciosa e também por ter acreditado que eu conseguiria terminar este trabalho.

A minha esposa, Ieda, pela amizade, companheirismo, compreensão e paciência por nunca ter me deixado desistir. A minha mãe e Nelson, pelo apoio incondicional nas decisões que tomei até aqui. Ao Pedro e Maria, pais da Ieda, suas irmãs e cunhados por terem acreditado que eu iria conseguir, pela compreensão e amizade.

Aos meus amigos, colegas professores do Instituto Federal de Educação, Ciência e Tecnologia Sul-rio-grandense, campus Camaquã e Passo Fundo, em especial a Ana Maria Geller, Ânderson dos Santos Ritta, Leonardo Campos Soares, Marcelo Rios Kwecko e Josué Toebe pelas dicas e pelos momentos de descontração.

A todos vocês MUITO OBRIGADO.

“Carrego nas mãos
As marcas de ontem
Já perdi o medo
Pois aprendi
Aprendi a ser valente
Neste meu caminho
Muitas vezes sozinho
Mas cheguei aqui ”

PRIMEIRA ESTROFE DA MÚSICA MANOTAÇOS - CELSO SOUZA

RESUMO

Dissertação de Mestrado
Programa de Pós-Graduação em Informática
Universidade Federal de Santa Maria

TUXUR - UM *FRAMEWORK* PARA DIVISÃO DINÂMICA DE TAREFAS MALEÁVEIS EM GRADE COMPUTACIONAL

Autor: Roberto Wiest
Orientador: Prof. Dr. Benhur de Oliveira Stein (UFSM)
Local e data da defesa: Santa Maria, 23 de Maio de 2014.

Grades de computadores, como outras tecnologias de Tecnologia da Informação, foi criada e desenvolveu-se em meados dos anos 90 no âmbito acadêmico. A ideia desta tecnologia é utilizar os recursos de computadores, sem se preocupar com a aquisição de novos recursos e localização física. O foco das grades computacionais está associado à execução de aplicações que demandam alto poder computacional ou se adaptam a ambientes distribuídos. Isto torna o ambiente dinâmico e heterogêneo, características que fazem a gerência de recursos, escalonamento e tolerância a falhas um grande desafio.

Aplicações de grades são geralmente programadas através de um *framework* de computação em grade. Neste trabalho apresentamos o *TUXUR*, projetado para gerenciar a execução de tarefas em grades computacionais. O *framework* é especializado em tarefas maleáveis, que permitem ser divididas em subtarefas independentes de tamanho conhecido. O *framework* ajusta *dinamicamente* o tamanho das subtarefas de acordo com os recursos disponibilizados pela grade.

O principal objetivo deste ajuste é tentar maximizar a utilização do poder computacional disponibilizado, equilibrando a carga de trabalho entre os componentes da grade, de acordo com a capacidade de cada um. Os resultados encontrados na avaliação evidenciam que esse objetivo foi alcançado, aproveitando ao máximo a utilização dos recursos e diminuindo o tempo final de execução.

Palavras-chave: Computação em grade, framework, tarefas maleáveis.

ABSTRACT

Master's Dissertation
Programa de Pós-Graduação em Informática
Universidade Federal de Santa Maria

TUXUR - A FRAMEWORK FOR DYNAMIC DIVISION OF MALLEABLE TASKS IN GRID COMPUTING

Author: Roberto Wiest
Advisor: Prof. Dr. Benhur de Oliveira Stein (UFSM)

Grid computing, like other technologies of the Information Technology, was created and developed in the mid 90 in the academic scope. The idea of this technology is to use computer resources without worrying about acquisition new resources and physical location. The focus of computational grids is associated with the execution of applications that demand high computing power or adapt to distributed environments. This makes the environment dynamic and heterogeneous, characteristics that make resource management, scheduling and fault tolerance a great challenge.

Grid applications are usually programmed through a framework of grid computing. In this work we present TUXUR, designed to manage the execution of tasks in computing grids. Theis framework is specialized in malleable tasks that allow being divided into independent subtasks of known size. The framework dynamically adjusts the size of the subtasks within the resources made available by grid.

The main objective of this adjustment is to try to maximize the use of available computational power, balancing the workload of the components of the grid, according to the capacity of each one. The results of the evaluation show that this goal was achieved, maximizing resource utilization and decreasing the final execution time.

Keywords: grid computing; framework; malleable tasks.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	Application Program Interface
CF	*Close-to-Files*
CPU	*Central Processing Unit*
GPU	Graphics Processing Unit
(NMET	Instituto Nacional de Meteorologia
IP	*Internet Protocol*
LSC	Laboratório de Sistemas de Computação
MPI	Message Passing Interface
OVs	Organizações Virtuais
SO	Sistema Operacional
UFPEL	Universidade Federal de Pelotas
UFRGS	Universidade Federal do Rio Grande do Sul
UFSM	Universidade Federal de Santa Maria
WQR	*Workqueue with Replication*
WF	*Worst-Fit*

SUMÁRIO

1 INTRODUÇÃO

1.1 Contextualização e Motivação

Os Sistemas Operacionais dos primeiros computadores (*mainframes*) eram capazes de atender simultaneamente diversos usuários, devido ao emprego de técnicas de multiprogramação e compartilhamento de tempo. Embora os programas que esses computadores executavam serem compostos, basicamente, de cálculos numéricos ou editores de texto, eles custavam muito caro e, por este motivo, havia poucas instituições com recursos financeiros para investir nesses computadores (TANENBAUM, 2010).

Com a evolução da eletrônica possibilitou-se o desenvolvimento de uma nova classe de computadores, os microcomputadores. Essa classe possuía preços muito menores que um computador de grande porte e, assim, popularizou o uso dos computadores fora das grandes empresas.

O crescimento da complexidade das aplicações incentivou a prática da divisão de programas para serem executados em máquinas paralelas, com o objetivo de diminuir o tempo final de execução. A utilização dessas máquinas acarretava as desvantagens dos *mainframes* e muitas vezes, dependendo da aplicação, perdiam em desempenho para os microcomputadores.

Assim, a utilização das máquinas paralelas tornou-se rara, pois a união de um conjunto de microcomputadores atingia o mesmo desempenho. Porém, era preciso interligar essas máquinas para que houvesse troca de informações e compartilhamento de recurso entre elas. Com o uso de redes de computadores não demorou muito para que o objetivo fosse atingido (TANENBAUM, 2007).

Com o surgimento e a consolidação da rede mundial de computadores, a *Internet*, foi possível construir sistemas envolvendo máquinas geograficamente distribuídas que permitissem o compartilhamento de recursos entre projetos de colaboração científica, podendo

essas pertencerem a diferentes organizações e domínios. Nasceu assim, o conceito de grades de computadores.

Inicialmente, elas foram definidas como sendo uma infraestrutura de *software* e *hardware* que fornece acesso a uma grande quantidade de recursos computacionais de forma segura, consistente, abrangente e barata (FOSTER; KESSELMAN; TUECKE, 2001; FOSTER; KESSELMAN, 2003). Esses recursos podem ser de diferentes tipos, tais como *hardware* e *software* especializado, processadores e armazenamento.

As grades de computadores são constituídas por entidades provedoras e consumidores de recursos. As relações entre elas estão sujeitas a regras bem definidas, estas regras podem estar relacionadas à segurança, compartilhamento, contabilização de recursos e comunicação. As entidades que compartilham determinadas regras formam Organizações Virtuais (OVs) (FOSTER; KESSELMAN; TUECKE, 2001; JACOB; BROWN; FUKUI, 2005). A figura 1.1 ilustra uma visão geral de uma grade de computadores.

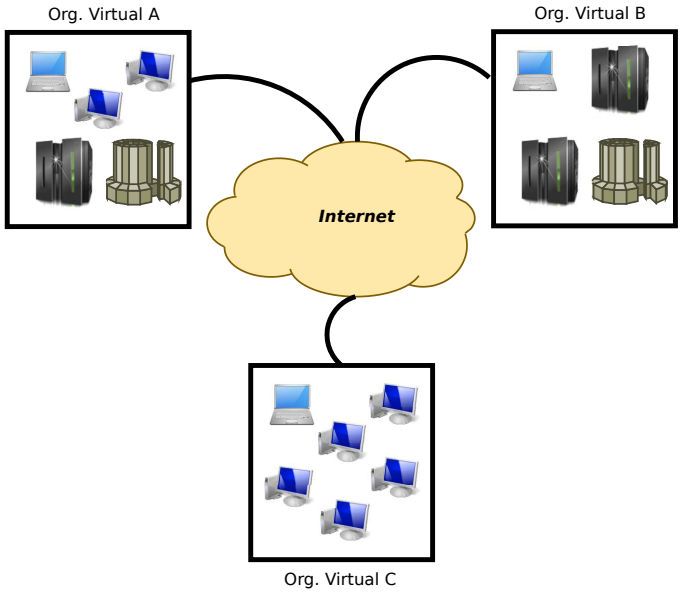

Figura 1.1: Visão geral de uma grade de computadores.

As OVs são constituídas de instituições ou mesmo de ambientes domésticos onde é possível executar aplicações que exigem computação intensiva de dados. Por outro lado, surgem empresas que ofertam processamento a quem estiver interessado em submeter seus trabalhos para serem executados em uma grade de computadores (COMPUTATION, 2014; FIGHTAIDSATHOME, 2014; SETIATHOME, 2014). A vantagem em obter esses serviços é a obtenção de resultados mais rapidamente ou a utilização de equipamentos

sofisticados e de alto custo disponibilizadas pela grade. Nestes dois casos é necessário haver um consenso entre as partes envolvidas, isto é, entre quem está compartilhando o recurso e quem irá acessá-lo para que a utilização dos mesmos não atrapalhe nenhuma das partes.

Ofertar um bom desempenho e confiabilidade em grades de computadores não é tarefa banal e se faz necessário uma ferramenta, *middleware* ou *framework*, capaz de fornecer suporte para o processamento de diferentes tipos e tamanhos de aplicações em ambientes heterogêneos e dinâmicos. Essa ferramenta deve ser capaz de prover alguns serviços indispensáveis para o funcionamento de uma grade, são os serviços de gerência de recursos, monitoramento de recursos, gerenciamento de processos e o serviço de escalonamento de tarefas (JACOB; BROWN; FUKUI, 2005).

Devido ao compartilhamento de recursos computacionais geograficamente distribuídos em um ambiente de grade é difícil obter e manter centralizada as informações sobre as tarefas a serem executadas. Sem poder contar com essas informações o problema de escalonamento de tarefas se torna complexo.

Este problema de escalonamento engloba um ambiente de execução da aplicação e a própria aplicação. Assim, o escalonador é encarregado de conceder a uma aplicação os recursos necessários para seu processamento. A principal finalidade deste é diminuir o tempo final de execução (*makespan*) organizando os recursos de forma a potencializar o paralelismo das tarefas e consequentemente otimizar a utilização dos recursos.

A aplicação, no ambiente de grade, pode ser constituída de tarefas que possuem dependência ou independência de dados. No caso de tarefas independentes, por exemplo, tarefas maleáveis (*malleables tasks*), o custo de comunicação e computação das tarefas podem ser menores que no caso de tarefas dependentes (MOUNIE; RAPINE; TRYSTRAM, 1999; DUTOT; TRYSTRAM, 2001).

A alta dinamicidade dos recursos computacionais que fazem parte de uma grade de computadores, que conta com recursos heterogêneos e que possui gerência descentralizada, inseriu mais problemas ao escalonamento em ambientes distribuídos, necessitando conhecer as técnicas de escalonamento, o tipo de arquitetura e o tipo de tarefa que será escalonada. (DONG; AKL, 2006).

1.2 Objetivo

Este trabalho propõem uma ferramenta, chamada *TUXUR*, capaz de escalonar dinamicamente tarefas maleáveis de uma aplicação baseada em tarefas independentes em uma grade computacional. O sistema possibilita a distribuição de tarefas à grade de maneira a diminuir o tempo total de execução de uma determinada aplicação.

Para isso, a distribuição das tarefas é realizada conforme a disponibilidade dos recursos computacionais. As tarefas suportadas são maleáveis de forma que a ferramenta possui um certo conhecimento sobre o seu tamanho (em termos de quantidade de processamento necessário para sua execução). O tamanho da tarefa é determinado de acordo com um tempo que o sistema estima adequado para que um recurso computacional possa operar de forma autônoma sem sobrecarregar o sistema. O intuito é que este trabalho contribua na consolidação de grades computacionais como uma opção viável para a execução de aplicações que demandam alto processamento.

1.3 Organização do Texto

Esta dissertação está organizada em cinco capítulos, estes obedecem a seguinte estrutura:

- *Capítulo 2 - Revisão Bibliográfica*: Visa contextualizar as grades de computadores, modelos e escalonamento de tarefas. Em trabalhos relacionados são apresentadas algumas soluções existentes para o melhor aproveitamento da arquitetura de um ambiente de grade. Para cada uma destas soluções descreve suas características, funcionalidades, formas de escalonamento e modelos de tarefas utilizados;

- *Capítulo 3 - Framework TUXUR*: Inicialmente descreve suas características, objetivos e contribuição. Em seguida, descreve os componentes que fazem parte da arquitetura e a comunicação entre os componentes através de ilustrações. Concluímos com a descrição da implementação de todos os componentes;

- *Capítulo 4 - Avaliação*: Aborda a forma de implementação da proposta em um ambiente de grade. Para demonstrar o seu comportamento são apresentados resultados de testes efetuados em alguns experimentos;

- *Capítulo 5 - Considerações Finais*: Sumariza as conclusões obtidas pelo progresso

do trabalho e também apresenta quais são os possíveis rumos que serão tomados em trabalhos futuros.

2 REVISÃO BIBLIOGRÁFICA

Os conceitos básicos de computação em grade, modelos de tarefas, escalonamento de tarefas e alguns trabalhos relacionados são apresentados neste capítulo. Iniciamos com a contextualização de computação em grade e suas principais características. A seguir são introduzidos os conceitos dos modelos e escalonamentos de tarefas, temas intrínsecos a este trabalho e se conclui com o relato de alguns trabalhos relacionados.

2.1 Grades Computacionais

Grades de computadores se constituem de recursos computacionais compartilhados organizados por Organizações Virtuais, estas por sua vez, se constituem de indivíduos ou instituições que fazem o compartilhamento de recursos controlado e coordenado. Atualmente, computação em grade é a base de estudos associados às execuções de aplicações paralelas que necessitam de alto poder de processamento (FOSTER; KESSELMAN, 2003; BAKER; BUYYA; LAFORENZA, 2002).

Elas são sistemas distribuídos que apresentam uma estrutura que coordena e controla os recursos, serviços e execuções de tarefas. Por se tratar de um ambiente desuniforme, as informações sobre a disponibilidade e utilização dos recursos devem ser mantidas pela estrutura de gerenciamento. A execução de tarefas deve ser organizada a fim de aprimorar a utilização dos recursos e diminuir o tempo final de execução das aplicações (BUYYA; ABRAMSON; GIDDY, 2001; BAKER; BUYYA; LAFORENZA, 2002).

Algumas características básicas de uma grade são apresentadas a seguir (BAKER; BUYYA; LAFORENZA, 2002):

- coordenação de recursos: manter a gerência dos recursos descentralizada. As organizações virtuais possuem suas próprias regras de administração relacionadas as permissões de usuários e segurança, por exemplo;

- heterogeneidade: os recursos computacionais envolvidos abrangem um grande número de *hardware* especializado e *softwares* de diferentes versões;
- escalabilidade: uma grade pode mudar de tamanho, sendo grande em certos momentos e pequena em outros. O ambiente deve estar preparado para essas mudanças;
- dinamicidade: o ambiente de grade é suscetível a falhas, por exemplo, sobrecarga de um computador que pertence à grade, e ele deve ser capaz de se adaptar a estas situações.

Grades computacionais, de acordo com sua funcionalidade, podem ser classificadas em três categorias conforme ilustrado na figura 2.1.

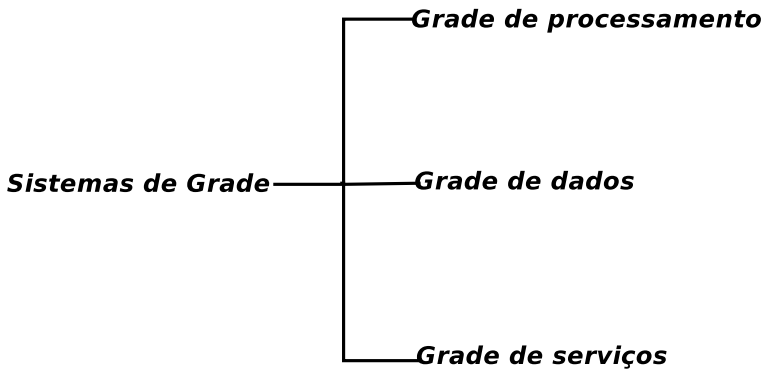

Figura 2.1: Tipo de grades computacionais de acordo com sua funcionalidade (KRAUTER; BUYYA; MAHESWARAN, 2002). Tradução própria.

As grades de processamento (*Computational Grids*) têm por objetivo unificar recursos computacionais geograficamente dispersos com a finalidade de oferecer alta capacidade computacional para execução de tarefas. Neste grupo são incluídas as aplicações que demandam de alto poder de processamento. Esta função ainda pode ser empregada para computação distribuída e computação intensiva de dados. A primeira é utilizada para solucionar problemas inviáveis de serem resolvidos por um único recurso computacional. Na computação intensiva de dados o objetivo é extrair e sumarizar informações baseadas em uma grande quantidade de dados.

A classe grade de dados (*Data Grid*) é formada por uma infraestrutura especializada para armazenamento distribuído de informações. Neste caso, a grade de computadores fica encarregada de juntar as informações dispersas e sintetizá-las para a utilização do usuário.

O último grupo é chamado de grade de serviços (*Service Grid*) devido à quantidade de serviços que oferece. Esse grupo se divide em três classes:

- Colaborativa - Nesta classe, um recurso é compartilhado entre os usuários através de ambientes virtuais compartilhados. Permite que indivíduos fisicamente espalhados possam trabalhar em conjunto em um mesmo projeto;
- Multimídia - Oferece, em um mesmo intervalo de tempo, infraestrutura para aplicações que fazem uso de diversas mídias. O principal objetivo desta classe é garantir a qualidade do serviço para todas as máquinas pertencentes à grade;
- Sob demanda - Aloca dinamicamente recursos de acordo com a necessidade da aplicação. Além do processamento, nesta classe, pode-se compartilhar *software* e dados, mas estes recursos são compartilhados por um determinado tempo.

Devido as características de cada classe é difícil de implementar uma grade de computadores que possua os três serviços citados. Assim, a implementação de grades especializadas é mais comum objetivando a satisfação do usuário. Neste trabalho, considerou-se grade de processamento, uma vez que, o *framework* proposto se relaciona para a melhoria do desempenho de aplicações com determinada quantidade de processamento.

2.1.1 Modelos de tarefas

Os modelos de tarefas paralelas (tarefas que requerem um ou mais processadores para sua execução) têm sido estudados extensivamente nos últimos anos como um método para a programação de aplicações paralelas, especialmente nos casos onde a granularidade de uma aplicação é alta (FEITELSON; RUDOLPH, 1996; DUTOT; MOUNI; TRYSTRAM, 2003; DUTOT; TRYSTRAM, 2005).

Basicamente, uma tarefa paralela é uma tarefa (também chamado de *job*) que reúne operações básicas, tipicamente uma rotina numérica, que contém em si própria paralelismo suficiente para ser executada por mais de um processador.

FEITELSON; RUDOLPH (1996) descrevem quatro modelos de tarefas que podem ser amplamente utilizados em grades computacionais. A figura 2.2 ilustra a organização destes modelos.

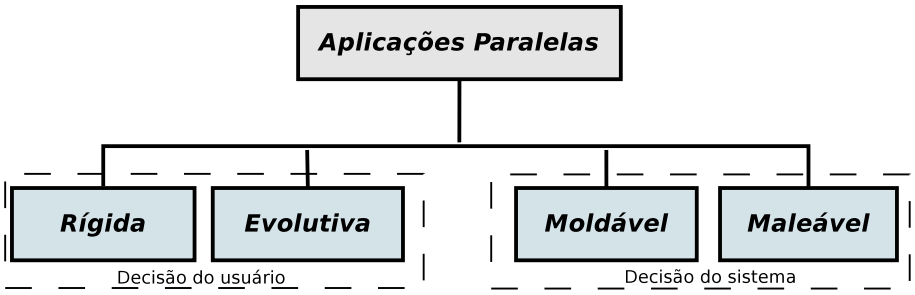

Figura 2.2: Organização dos modelos de tarefas

Rígida (*Rigid*). Para uma aplicação rígida faz-se necessário que o programador especifique um número fixo de unidades de processamento para a execução de uma aplicação paralela. Este número é rígido, isto é, a aplicação não aceita qualquer número menor ou maior de unidades de processamento;

Evolutiva (*Evolving*). Tarefas evolutivas podem alterar o número de unidades de processamento utilizadas. A mudança pode ocorrer durante a execução, ou antes, de uma aplicação ser iniciada, isto é, para cada execução da aplicação um número diferente de unidades de processamento pode ser especificado. No entanto, a modificação é apenas acionada pela aplicação paralela e o ambiente de execução tende a cumprir com a nova definição. Alguns modelos de programação fornecem primitivas para alterar o número de unidades de processamento durante o tempo de execução, por exemplo, OpenMP (OPENMP, 2012);

Moldável (*Moldable*). Tarefas moldáveis, também chamadas de carga divisível, se adaptam a qualquer número de unidades de processamento. O programador não precisa especificar qual o número de unidades de processamento, tal número é especificado pelo próprio ambiente de computação e esse permanece fixo durante sua execução. A maioria das aplicações paralelas que utilizam o modelo de programação MPI (Message Passing Interface) são aplicações moldáveis (MPI, 2012);

Maleável (*Malleable*). Tarefa é maleável se é capaz de se adaptar automaticamente as mudanças de número de unidades de processamento no ambiente de computação. Por muitas vezes, na literatura esse tipo de aplicação é chamado de particionamento dinâmico (*dynamic partitioning*).

2.1.2 Escalonamento de tarefas

Em grades computacionais, devido à sua heterogeneidade e dinamicidade, é difícil conseguir obter e manter informações sobre as tarefas a serem executadas e sem poder contar com essas informações é difícil encontrar ferramentas capazes de oferecer um bom escalonamento de tarefas. Geralmente o problema de escalonamento de tarefas em grades computacionais é NP-Completo (DONG; AKL, 2006; KLEIN; PÉREZ, 2010).

Nesse ambiente, diversas aplicações são executadas simultaneamente e elas buscarão aplicar as informações atuais da grade para potencializar sua execução. Para atingir um bom desempenho o escalonamento de tarefas é fundamental (SILVA et al., 2003).

DONG; AKL (2006) propõem uma taxonomia para algoritmos de escalonamento em sistemas de computação paralela e distribuída, conforme a figura 2.3.

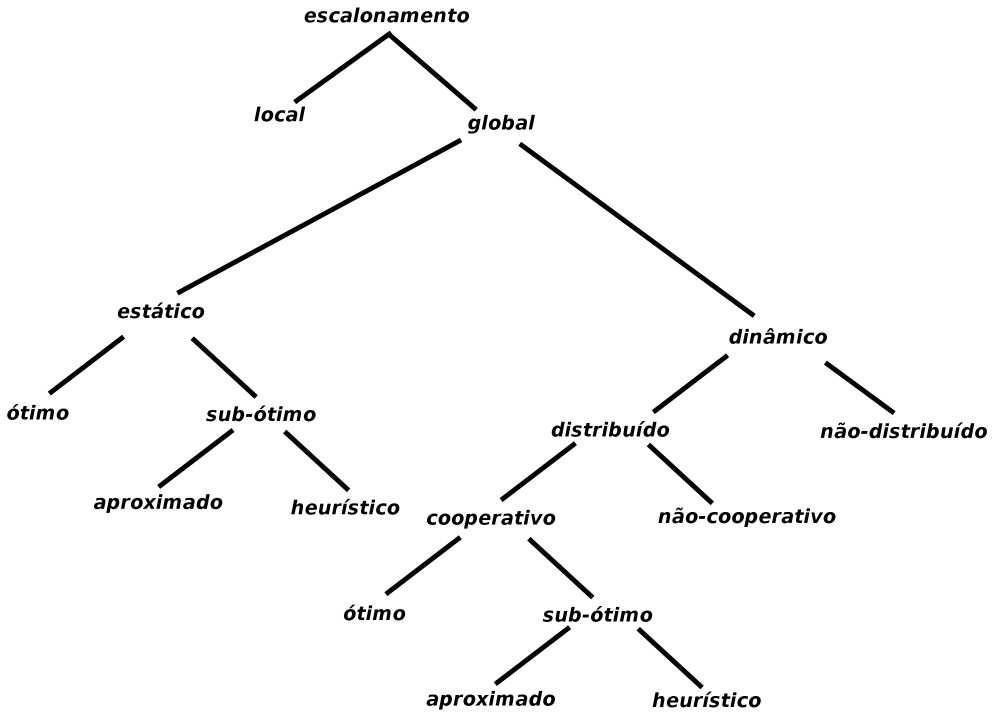

Figura 2.3: Taxonomia de escalonamento (DONG; AKL, 2006). Tradução própria.

A seguir iremos explicar cada nível da hierarquia de classificação exposta pela figura 2.3:

- Local versus Global - O escalonamento local geralmente envolve o escalonamento de processos, realizado pelo Sistema Operacional (SO), visando a utilização de uma única CPU (*Central Processing Unit*). No global, o escalonamento envolve a distribuição de processos a mais de um recurso, por exemplo, conjunto de computadores

formadores de uma grade de computadores ou aglomerado de computadores (*cluster*);

- Estático versus Dinâmico - No estático as decisões de escalonamento são tomadas antes da execução do programa e no dinâmico as decisões são tomadas durante a execução do programa. Quando objetiva-se atingir um bom balanceamento de carga no escalonamento dinâmico, quatro políticas são definidas. A primeira é a política de localização, sua função é encontrar recursos aptos a receber tarefas. A próxima é a política de seleção, que deve selecionar uma tarefa para ser enviada. A terceira é a política de transferência, que tem por finalidade enviar a um recurso uma tarefa. Por fim, a política de informação responsável por manter o estado dos recursos atualizados;

- Ótimo versus Subótimo - Para o escalonamento ser ótimo é necessário conhecimento prévio de todo o sistema e da aplicação. Muitas vezes esta tarefa é inatingível, pois é muito difícil estimar o tempo de execução de uma aplicação e ocorre variação de carga dos recursos escolhidos para tal aplicação. Devido a essas dificuldades, geralmente, os algoritmos de escalonamento são subótimos, estes podem ser alcançados através de heurísticas e aproximações;

- Heurística versus Aproximação - Um algoritmo heurístico leva em consideração alguns parâmetros que afetam o sistema, por exemplo, carga do sistema. A avaliação desse tipo de solução é geralmente baseada em experiências do mundo real. O algoritmo de aproximação segue as mesmas definições do algoritmo ótimo, porém, não é necessário encontrar a solução ótima basta uma solução que atenda a uma aproximação primeiramente definida;

- Distribuído versus Não Distribuído - Quando o processo é escalonado globalmente por um único processador, o escalonamento é não distribuído, caso contrário, se a tarefa de escalonar processos provém de um conjunto de processadores, o escalonamento é chamado de distribuído;

- Cooperativo versus Não cooperativo - A classificação cooperativa é dada, por exemplo, quando cada processador de um sistema determina como seus recursos serão utilizados. Já os nãos cooperativos, eles atuam como entidades independentes, não

se preocupam com a melhora de desempenho.

Em um algoritmo de escalonamento, os parâmetros usados para tomar as decisões de escalonamento podem mudar dinamicamente de acordo com os estados das tarefas e recursos. Na seção 2.1.2.1 iremos discutir este tipo de escalonamento.

2.1.2.1 Escalonamento adaptativo

Este tipo de escalonamento emprega algoritmos e parâmetros de acordo com os recursos do ambiente de grade. O escalonamento adaptativo é centrado na adaptação de recursos (*resource adaptation*), adaptação dinâmica do desempenho (*dynamic performance adaptation*) ou adaptação da aplicação (*application adaptation*), conforme exposto na figura 2.4.

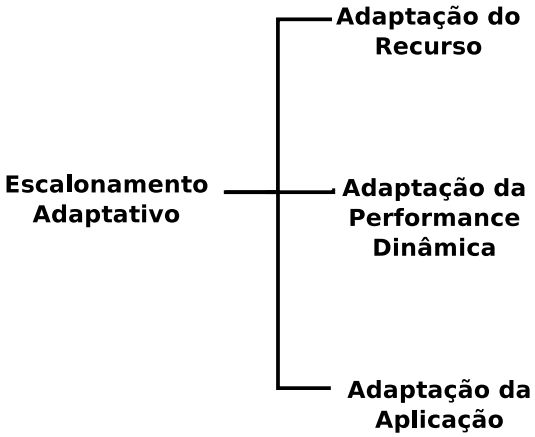

Figura 2.4: Taxonomia para escalonamento adaptativo (DONG; AKL, 2006). Tradução própria.

O principal objetivo da adaptação de recurso é aumentar o desempenho, por exemplo, diminuir o tempo final de execução de uma tarefa. Para almejar este objetivo o algoritmo tenta encontrar e selecionar o melhor conjunto de recursos a partir, por exemplo, do tamanho da memória e poder computacional. A adaptação de performance dinâmica altera as regras de escalonamento, dividindo a carga de trabalho conforme a disponibilidade dos recursos computacionais. Na adaptação da aplicação, o escalonador encontra-se na própria aplicação, sendo assim, esta se auto escalona (DONG; AKL, 2006).

2.1.2.2 Grupos de escalonamento

Conforme DONG; AKL (2006) a computação em grade possui dois grandes grupos, são chamados de consumidores de recursos e provedores de recurso. Este último tem

por objetivo compartilhar recursos entre os grupos que entram no ambiente por alguma motivação. Funções objetivo (*objective functions*) são as atuações desta motivação. Os consumidores de recursos são os usuário responsáveis por submeter as aplicações no ambiente de grade.

A figura 2.5 ilustra que as funções objetivo são classificadas em centrada na aplicação (*application centric*) e centrada no recurso (*resource centric*).

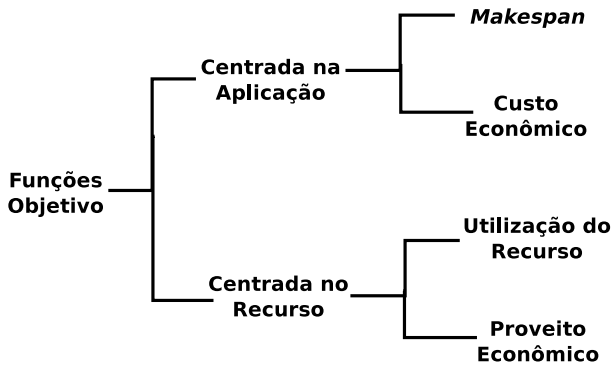

Figura 2.5: Funções Objetivo (DONG; AKL, 2006). Tradução própria.

Os algoritmos centrados na aplicação buscam potencializar o funcionamento da aplicação, mais conhecido como *makespan*. Esta é a medida que quantifica o tempo gasto entre o início da execução da primeira tarefa e o final da execução da última tarefa. Para obter essa medida costuma-se utilizar o *speedup*, razão entre o tempo de execução gasto no programa sequencial (T_{max}) e o tempo de execução gasto no programa paralelo (T). Para encontrar o *speedup* basta calcular $\frac{T_{max}}{T}$. Um *speedup* é chamado de linear ou ideal quando a razão for igual ao número de processadores, se esta razão for maior que o número de processadores é chamado de superlinear e caso for menor o *speedup* é sublinear.

Ao contrário do *makespan*, pode ser utilizado o custo econômico baseado em modelos econômicos. As principais medidas de desempenho utilizadas por essa função é o tempo e o custo financeiro. Ao utilizar esse tipo de função o escalonador deve possuir, como principal característica, a facilidade de se adaptar ao ambiente de grade.

Os algoritmos que aderem ao escalonamento centrado no recurso têm como objetivo potencializar a eficácia dos recursos. Isto está relacionado ao *throughput*, que é a capacidade de processar um número de tarefas em um determinado período de tempo.

2.1.2.3 Dependência de tarefas

As relações entre as tarefas em um ambiente de grade podem ser dependentes e independentes, estas são de fundamental importância para um algoritmo de escalonamento. Tarefa dependente significa que uma tarefa somente pode ser iniciada depois que a tarefa da qual ela depende tenha sido concluída. Ao contrário desta, na execução de tarefas independentes não é necessário que outra tarefa tenha sido finalizada (DONG; AKL, 2006).

2.2 Trabalhos relacionados

O escalonamento de tarefas tem sido alvo de vários estudos nos últimos anos. Com o objetivo de escolher o melhor algoritmo foram propostas diversas políticas e *frameworks* para escalonamento de tarefas. A seguir são descritas algumas dessas políticas e *frameworks* utilizados em grades computacionais.

Em (MAHESWARAN et al., 1999; SILVA; SENGER, 2009) propõem três heurísticas chamadas *Max-Min*, *Min-Min* e *Sufferage* que escalonam tarefas independentes. A primeira objetiva-se atribuir as maiores tarefas para os melhores recursos, estaticamente. A característica desta heurística é que ela consegue alcançar baixos tempos de resposta quanto maior for a quantidade de tarefas menores em relação às maiores. Para que uma tarefa possa ser escalonada são necessárias três informações, o número de instruções da tarefa, a parte do recurso que está sendo utilizado e a velocidade do recurso onde a tarefa será executada. A heurística *Max-Min* não costuma apresentar bons resultados por falta de acesso as informações de recursos e das aplicações que estão sendo executadas. Além disso, por ser estática, é incapaz de se adequar as mudanças no sistema.

A *Min-Min*, ao contrário da primeira, atribui as menores tarefas aos melhores recursos. Nesta heurística, os melhores recursos são liberados mais rapidamente e consequentemente executa novas tarefas. Por fim, a *Sufferage*, essa política exige as mesmas informações que a *Max-Min*, além disso, necessita das informações de CPU e tempo estimado de execução da tarefa. A particularidade está no fato que a tarefa será atribuída ao recurso que oferece o menor tempo para executá-la.

(CIRNE; BERMAN, 2002) e (SILVA et al., 2003) usam o termo moldável para tarefas independentes que podem se adaptar a diferentes processadores. Apresentam o algoritmo *Workqueue with Replication* (WQR), nele as tarefas de uma mesma aplicação são escolhidas para serem replicadas nos computadores ociosos. Para realizar o escalonamento o

algoritmo não precisa obter informações dos recursos. Quando alguma réplica acabar a execução as demais são abortadas para que não haja desperdício de processamento.

A heurística *WQR* perde desempenho causada pela falta de informações das tarefas e devido ao fato de utilizar replicação de tarefas com carga muito pequenas causam muitas réplicas ocasionando maior comunicação (REIS et al., 2009; SILVA; SENGER, 2009).

Em (KALE; KUMAR; DESOUZA, 2002) o autor apresenta o *framework Charm++* que possui suporte para tarefas maleáveis independentes. O autor descreve uma estratégia que tem como princípio básico a definição que cada tarefa que será submetida deverá especificar o número mínimo e máximo de processadores que podem utilizar. Após isto, o escalonador recalcula novamente o número necessário de processadores e só então submete as tarefas.

(SUDARSAN; RIBBENS, 2007) e (SUDARSAN; RIBBENS, 2010) apresentam um *framework* para executar tarefas maleáveis e moldáveis, chamado *ReSHAPE*. O escalonador escolhe, aleatoriamente, o número de processadores onde irá submeter as tarefas, para realizar este procedimento ele aguarda até que o número de processadores solicitados na descrição da tarefa esteja pronto e após as tarefas são submetidas. Este *framework* possui suporte para tarefas independente e dependentes.

O *framework Koala* é apresentado em (MOHAMED; EPEMA, 2008) e (SONMEZ et al., 2009). Este *framework* possui suporte para tarefas moldáveis e maleáveis. O escalonamento das tarefas independentes é baseado em duas políticas: *Worst-Fit* (WF), onde os processadores que estão ociosos por mais tempo irão receber uma grande quantia de tarefas e a segunda *Close-to-Files (CF)*, que utiliza informações de arquivos de entrada para decidir onde escalonar as tarefas.

Mais recentemente, (SILVA; SENGER, 2009) propôs um algoritmo chamado de *Dynamic Cluster* que realiza o escalonamento de tarefas independentes moldáveis. O algoritmo baseia-se no grau de compartilhamentos de arquivos para realizar o escalonamento de uma tarefa. As tarefas são enviadas de acordo com dois parâmetros: o primeiro é a soma em *bytes* de todos os arquivos necessários para executar uma tarefa e o segundo é a velocidade relativa de todos os processadores pertencentes à grade. Ao final deste artigo, compara a escalabilidade do algoritmo com os algoritmos propostos em (MAHESWARAN et al., 1999) e CIRNE; BERMAN (2002); SILVA et al. (2003).

A tabela 2.1 apresenta uma síntese das características referentes aos trabalhos relaci-

onados descritos acima.

	Informações de tarefas	**Informações de recursos**	**Dinamicidade**	**Tarefas**
Max-Min	Sim	Sim	Não	Independentes
WQR	Não	Não	Sim	Independentes
Charm++	Não	Sim	Não	Independentes
ReShape	Não	Sim	Sim	Independentes
Koala	Não	Sim	Não	Independentes
Dynamic Cluster	Sim	Sim	Sim	Independentes
TUXUR	Sim	Sim	Sim	Independentes

Tabela 2.1: Principais características presentes nos trabalhos relacionados.

No ambiente de grade computacional, devido à grande variedade de recursos computacionais e a sua localização geográfica, o escalonamento representa uma tarefa desafiadora. O escalonamento dinâmico, neste tipo de ambiente, tem se mostrado mais adequado por causa da variação de carga dos processadores (REIS et al., 2009)

O próximo capítulo apresentará o *framework TUXUR*, ferramenta capaz de escalonar dinamicamente tarefas maleáveis através da obtenção de informações das tarefas e recursos computacionais. Com isso, objetiva-se diminuir do tempo final de execução de uma aplicação baseada em tarefas independentes utilizando as características presentes em algumas definições descritas neste capítulo.

3 O FRAMEWORK TUXUR

O *TUXUR* é um *framework* de computação em grade que está sendo desenvolvido no Laboratório de Sistemas de Computação (LSC) localizado na Universidade Federal de Santa Maria (UFSM). O trabalho a que se refere esta dissertação contribui para o seu desenvolvimento, implementando o componente de gerenciamento responsável pelo escalonamento de tarefas. Neste capítulo são abordados primeiramente os objetivos, características e a contribuição do *TUXUR* para a área de processamento paralelo e distribuído. A seguir, aborda-se a arquitetura e a descrição detalhada da implementação do *framework*. Para finalizar, são, brevemente, descritas as formas de avaliação do *framework*.

3.1 Objetivos e características

O *TUXUR* foi planejado para possibilitar a utilização de recursos computacionais em uma grade de computadores de forma simples e eficiente. Para obter êxito nestes objetivos se faz necessário atacar dois pontos chaves: qual modelo de tarefa será gerenciado por este *framework* e quais nós irão executá-las.

A fim de aproveitar ao máximo os recursos computacionais que uma grade disponibiliza é fundamental que a diversidade de *hardware*, existente nestes computadores, seja explorada. Assim, o *TUXUR* busca potencializar a utilização da heterogeneidade dos recursos computacionais, por exemplo, utilização de processadores gráficos (GPUs - *Graphics Processing Unit*) (LINCK, 2010), permitindo que as tarefas sejam executadas concomitantemente em CPUs tradicionais e nas GPUs.

Outra característica diz respeito ao gerenciamento do número de computadores. A intenção desta característica é inserir ou remover nós antes ou durante o processamento de uma tarefa. Neste trabalho, abordamos somente a inserção de nós, as características de remoção e utilização de GPUs serão tratadas em trabalhos futuros.

Dentre os tipos de tarefas suportadas pelo *TUXUR* optou-se por incluir suporte às tarefas moldáveis, como *Bag-of-Tasks* (CIRNE; BERMAN, 2002; CIRNE et al., 2006), e maleáveis. Esta pode ser considerada a mais importante para o *framework*.

Neste trabalho usamos o termo maleável para tarefas em que o número de processadores pode ser escolhido pelo sistema durante sua execução, mas restrito ao caso em que as tarefas são constituídas de subtarefas independentes que potencialmente podem ser criadas mais subtarefas ou escalonadas em um processador.

Assim como a política de escalonamento *Dynamic Cluster*, o *TUXUR* utiliza informações das tarefas para distribuí-las, mas difere-se na forma de obter as informações dos recursos computacionais. Além disto, o *framework TUXUR* se propõe a dividir automaticamente e escalonar dinamicamente as tarefas, características que se diferem das políticas adotadas pelos *frameworks KOALA* e *Charm++* e do algoritmo *Max-Min*.

O objetivo da característica de dinamicidade é assegurar que tarefas que demandem elevada quantidade de processamento não sobrecarreguem computadores com baixo poder computacional. Este objetivo é satisfeito quando se criam subtarefas a partir de uma tarefa sabidamente demorada para ser executada. Uma subtarefa é criada a partir do tempo de execução necessário para sua conclusão, e esse tempo é compatível com o poder computacional do nó que a executará.

Nas seções seguintes são discutidos a arquitetura do *framework*, suas funcionalidades e detalhes da implementação. Finalizamos com uma breve descrição das formas de avaliação.

3.2 Arquitetura

Uma grade é geralmente constituída de várias organizações virtuais, cada uma delas disponibilizando certo número de recursos computacionais. Tipicamente tem-se uma homogeneidade em termos administrativos e em capacidade de comunicação maior no interior de uma organização virtual do que entre essas organizações. Isso leva a uma visão organizacional hierárquica no acesso aos recursos disponibilizados pela grade. Uma grade é bem mais explorada por um *framework* quando essa hierarquia natural é considerada no escalonamento.

Para tentar explorar os recursos disponibilizados por essa grade o *TUXUR* reflete essa organização internamente. Sendo assim, a descoberta de recursos e lançamentos dos nós

é realizada pelo *Lançador* que é responsável pela formação hierárquica apresentada na figura 3.1. Basicamente, ele estabelece a hierarquia de nós que hospedam o componente *Trabalhador* e nós que hospedam o componente *Gerente*. Chamaremos o processo de criação desta hierarquia de *lançamento* no decorrer deste trabalho.

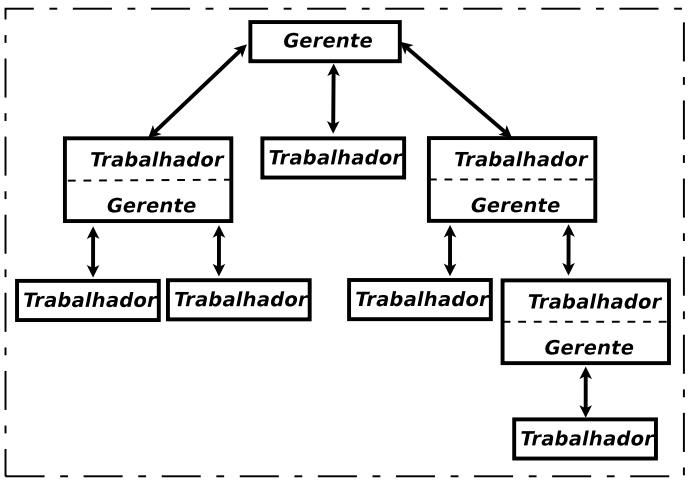

Figura 3.1: Arquitetura do *TUXUR*.

O componente básico do *TUXUR* é o *Trabalhador*, que representa um *nó* de computação. Sua função se resume em resolver tarefas enviadas pelo seu *Gerente* e retornar o resultado encontrado. Da mesma forma como ele processa tarefas e envia resultados, também é responsável por manter o seu gerente atualizado sobre seu estado.

O outro componente é o gerente, este possui basicamente três atribuições. A primeira é gerenciar um conjunto de trabalhadores, garantindo que eles tenham trabalho em quantidade suficiente. Controlar o estado dos trabalhadores é a sua segunda atribuição e a última é compor o resultado final da tarefa.

Cada gerente é por sua vez gerenciado por outro gerente, também chamado de gerente do gerente. Para facilitar a implementação da hierarquia, um gerente possui a mesma interface de um trabalhador, para a recepção de mais tarefas. É indistinguível para um gerente se seus subordinados são trabalhadores ou se são gerentes.

O estado é composto por dois valores. O primeiro é a capacidade, medida que quantifica quanto trabalho o trabalhador consegue realizar por unidade de tempo, ela é obtida através da razão entre o tamanho de uma tarefa e o tempo em que o trabalhador conseguiu executar tal tarefa. Já a capacidade de trabalho de um gerente é a união das capacidades de trabalho de seus subordinados. O último valor é a carga de trabalho, que indica a quantidade de trabalho que o trabalhador já recebeu e ainda tem por realizar. No gerente, o

volume de tarefas a executar é a soma das tarefas em sua fila e nas de seus trabalhadores.

Sabendo o estado dos seus trabalhadores cada gerente sabe quando e qual carga de trabalho ele deve enviar para um determinado subordinado, para que este não fique sobrecarregado ou ocioso. No momento que o gerente recebe uma mensagem de estado de um determinado trabalhador ele atualiza o estado deste localmente. Essas mensagens são enviadas após alguns eventos. Um evento que origina uma mensagem é a recepção de novos trabalhadores por um gerente. Outra mensagem ocorre quando um trabalhador conclui a execução de uma tarefa.

A carga de trabalho de um trabalhador é calculada de acordo com um *tempo alvo*, o tempo que o sistema estima como adequado para que um trabalhador possa operar de forma independente. Um gerente distribui mais trabalho para um trabalhador quando estimar que o tempo que ele levará para calcular o trabalho que ainda tem é inferior a certo percentual (p.ex. 50%) do tempo alvo e envia trabalho de forma que ele não fique com uma carga de trabalho que exija mais tempo que outro percentual do tempo alvo (p.ex. 150%).

A escolha de um tempo alvo correto é crucial ao funcionamento do *TUXUR*. Um tempo alvo muito pequeno eleva a frequência de comunicações entre gerentes e trabalhadores e um tempo muito grande desequilibra o volume total de trabalho entre os trabalhadores aumentando o *makespan*. Para diminuir a chance de ter esse problema, a quantidade de trabalho com os trabalhadores deve ser menor mais próximo ao final da execução.

Após definida a carga de trabalho, uma tarefa com uma determinada quantidade de trabalho é criada e enviada para o trabalhador. A tarefa é a representação da aplicação que será executada e as rotinas que a computam. Dependendo da carga de trabalho que será atribuída a um trabalhador uma tarefa pode originar outras tarefas, também chamadas de subtarefas neste trabalho. A criação das subtarefas ocorre logo antes do escalonamento, cujo intuito é retardar ao máximo essa divisão a fim de manter os computadores com poucas tarefas.

As tarefas enviadas aos trabalhadores são gerenciadas pelo gerente a fim de quando recebidos os resultados eles possam ser atribuídos à qual tarefa ele pertence. Ao receber um resultado o gerente localiza a tarefa a qual a resposta pertence e atribui o seu resultado. Caso todos os resultados da tarefa já tenha sido encontrado o gerente informa o resultado final ao seu gerente. Na próxima seção iremos descrever detalhes sobre a implementação

do *TUXUR*.

3.3 Implementação

A seguir iremos descrever as funcionalidades dos componentes e demais elementos pertencentes a implementação do *TUXUR*. Primeiramente comentam-se sobre o gerente suas características e funcionalidades. Em seguida abordam-se os atributos e atividades desempenhadas pelo trabalhador. O processo de lançamento de um nó, desempenhada pelo lançador, e as características e funções da tarefa são descritas a seguir. A interface do usuário e de comunicação encerra esta seção.

3.3.1 Gerente (*Manager*)

Para simplificar o gerenciamento interno e para facilitar a implementação optamos em criar três *subgerentes*. Isto simplifica, porque cada subgerente fica encarregado de executar um trabalho em específico e as comunicações entre os componentes ficam mais fáceis de serem definidas. Sendo assim, esses subgerentes foram denominados:

- *Gerente de estados*: calcula quando um trabalhador precisa receber mais tarefas. Para isto, ele precisa saber o estado de todos os trabalhadores. Para cada trabalhador que estiver com pouca tarefa este gerente envia uma mensagem ao *gerente de tarefas* requisitando mais trabalho;

- *Gerente de tarefas*: encarregado de atender as requisições de trabalho. O gerente de tarefa possui um método denominado *newJob* que é invocado toda vez que o gerente de estados requisita uma nova tarefa;

- *Gerente de respostas*: responsável por receber as respostas das tarefas de todos os trabalhadores e formar a resposta final.

A comunicação entre eles é realizada através de chamadas de métodos. Por outro lado, a comunicação deles com os trabalhadores ocorre através da troca de mensagens conforme descrito na seção 3.3.6. Nas seguintes subseções descrevemos as funcionalidades de cada subgerente.

3.3.1.1 Atualização de estado

A incumbência de atualizar o estado dos trabalhadores e requisitar novo trabalho é do gerente de estado. Para que ele possa distribuir trabalho de forma adequada ele precisa saber o estado dos seus trabalhadores. Para que a informação de estado seja propagada através da grade se utiliza uma mensagem (*NODE_STATUS*) contendo a identificação do trabalhador, a sua capacidade e a carga de trabalho.

Esta mensagem é enviada por todo trabalhador para o gerente de estados regularmente. As mensagens são enviadas após três eventos. O primeiro evento que origina uma mensagem de estado é a recepção de novos trabalhadores por um gerente. Outra mensagem de estado ocorre quando um trabalhador conclui a execução de uma tarefa. E por final, o sistema define um tempo de atualização, a cada 50% do tempo alvo uma nova mensagem de estado é enviada. Caso esse tempo for menor que a porcentagem o sistema define o próprio tempo alvo como o tempo de atualização. A figura 3.2 ilustra a atualização de estado.

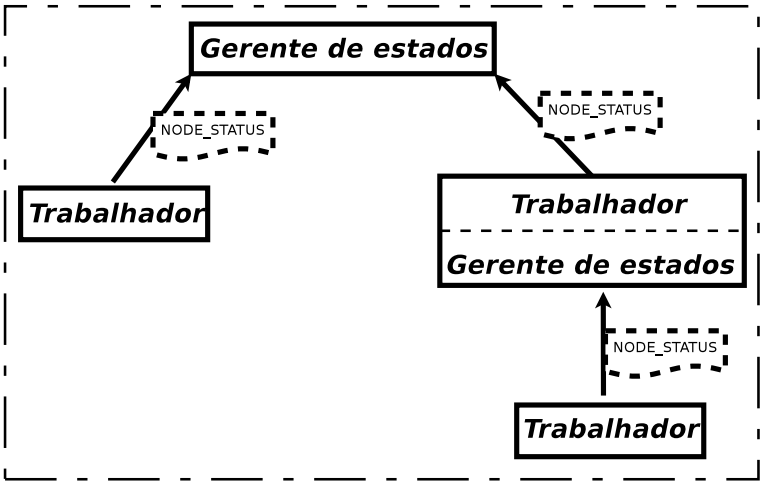

Figura 3.2: Atualização de estado.

Possuindo as informações de estado o gerente é capaz de atualizar o estado de cada subalterno no momento em que ele recebe cada mensagem. A fórmula 3.1 representa a atualização da quantidade de trabalho de um subalterno corrigida para o tempo atual.

$$Wc = ((W - (C * (D2 - D1))) + We) \tag{3.1}$$

Wc representa a quantidade de trabalho corrigida para o tempo atual do trabalhador, *W* representa a quantidade de trabalho que o trabalhador possui, *C* é a capacidade de

trabalho, estes dois últimos valores são obtidos nessa mensagem de estado, *D2* é a data atual, *D1* data em que a mensagem de estado foi recebida e *We* representa a quantidade de trabalho enviada ao trabalhador desde a recepção dessa mensagem de estado.

Isso permite calcular por quanto tempo aproximadamente o trabalhador pode continuar trabalhando com a carga de trabalho que possui. Esse estado está sempre mudando, porque a carga de trabalho que está sendo executada e a carga por executar diminui. Por isso, o estado está associado a uma data.

A fórmula 3.2 representa o tempo restante de trabalho, assim se define se o trabalhador deve, ou não, receber mais trabalho.

$$T = \frac{Wc}{C} \tag{3.2}$$

Para evitar a ociosidade de um trabalhador, devem-se enviar novas tarefas antes que ele fique sem ter o que fazer. A fim de impedir uma frequência alta de envio de pequenas tarefas, para complementar a fila de um trabalhador e também para evitar que o gerente tenha que dividir tarefas com tamanhos precisos definiu-se como aceitável que a fila de um trabalhador contenha tarefas de tamanho a que correspondam a um tempo entre 50% e 150% do tempo alvo. Caso o tempo restante calculado seja inferior a 50% do tempo alvo o gerente gera uma nova requisição (para ele próprio) de novos trabalhos para o trabalhador. O tamanho das tarefas pode levar a carga total de trabalho desse trabalhador a até 150% do tempo alvo. O cálculo para definição do tempo alvo é descrito na próxima seção.

3.3.1.2 Definição do tempo alvo

Possuindo as informações da quantidade de trabalho a realizar e a capacidade total de seus trabalhadores é possível estimar qual será o tempo total de execução. Assim, o tempo alvo será proporcional ao tempo total de execução. Este tempo não é constante, pois conforme a execução evolui o tamanho da fila de trabalhos também diminui e consequentemente o tempo também diminui.

Portanto, ao final da execução teremos uma maior comunicação entre os recursos que pertencem à grade de computadores, isto é preferível, pois gastando comunicação ao final se reduz a probabilidade de nós ficarem ociosos e consequentemente a grade ficar desequilibrada. A fórmula 3.3 ilustra o cálculo para definir o tempo alvo.

$$DT = ((\frac{W}{C}) * P) \tag{3.3}$$

DT representa o tempo alvo de execução, *W* representa a quantidade de trabalho, *C* a capacidade de um trabalhador e *P* um percentual do tempo restante total esperado como tempo alvo. *W* e *C* são obtidos a partir da última mensagem de estado enviada por cada trabalhador.

O primeiro gerente da hierarquia, também chamado de nó raiz, é responsável por realizar o cálculo do tempo alvo e através de uma mensagem (*DESIRED_TIME*, ver na subseção 3.3.1.4 a figura 3.4) informa os demais trabalhadores. Logo, é mantido o mesmo tempo alvo para toda hierarquia. Somente o nó raiz tem noção completa do trabalho total a realizar e da capacidade total de trabalho da grade. Como na implementação atual o tempo alvo é calculado com base nesses valores, ele deve ser calculado pelo nó raiz. Na subseção 3.3.1.3 descrevemos o cálculo inicial da capacidade de um nó.

3.3.1.3 Cálculo inicial da capacidade de um nó

Antes de enviar a primeira tarefa a um trabalhador é necessário definir a capacidade do nó, mas esta não pode ser prevista com exatidão e isto representa um sério problema, pois não se sabe qual a quantidade exata de trabalho enviar. Para resolver este problema, definimos a execução de uma pequena tarefa no momento da inicialização de um nó a fim de obter uma melhor precisão sobre sua capacidade. Esta pequena tarefa deve ser implementada pelo usuário. A fórmula 3.4 ilustra o cálculo para definir a capacidade de um trabalhador.

$$C = \frac{TT}{TE} \tag{3.4}$$

C define a capacidade de um trabalhador, *TT* tamanho de uma tarefa e *TE* o tempo em que o trabalhador conseguiu executar tal tarefa. Descrevemos na próxima seção como as tarefas são divididas e delegadas a um trabalhador.

3.3.1.4 Divisão e delegação de tarefas

O gerente de tarefas é encarregado de atender as requisições de trabalho, cabe a ele dividir e delegar tarefas aos seus trabalhadores. Dispondo do tempo alvo e a capacidade do trabalhador o gerente obtém uma determinada tarefa da fila de trabalhos não realizados e,

se necessário, a divide dando origem a subtarefas cuja quantidade de trabalho é compatível com esta estimativa. Na figura 3.3 ilustramos o código fonte para selecionar um conjunto de tarefas.

```
1   public List<IJob> jobsSelect(int min, int max) {
2       // TODO Auto-generated method stub
3       try {
4           List<IJob> tempJList = new ArrayList<IJob>();
5           int tempJListSize = 0;
6
7           while (!this.jobsList.isEmpty() && tempJListSize < min) {
8               IJob tempJ = this.getJobList();
9               if (tempJListSize + tempJ.getWorkload() > max) {
10                  List<IJob> list = tempJ.jobBreak();
11                  Iterator<IJob> itr = list.iterator();
12                  while (itr.hasNext()) {
13                          IJob jj = itr.next();
14                          this.setJobList(jj);
15                          itr.remove();
16                  }
17                  this.setParentJobList(tempJ.getID(), tempJ);
18              } else {
19                  tempJListSize += tempJ.getWorkload();
20                  tempJList.add(tempJ);
21              }
22          }
23          return tempJList;
24      } catch (Exception e) {
25          e.printStackTrace();
26          return null;
27      }
28  }
```

Figura 3.3: Código fonte do método para realizar a seleção de um conjunto de tarefas.

O método *jobsSelect* recebe como parâmetro dois valores (*min* e *max*) (linha 1) que dizem respeito ao tamanho da tarefa que será enviada ao trabalhador. A carga de trabalho será baseada no tempo alvo que poderá ser superior (*max*) ou inferior (*min*) a 50% desse tempo, conforme descrito na seção 3.3.1.1. Após, o método irá selecionar as tarefas menores até ter o suficiente (linha 7), caso o tamanho desta tarefa ultrapasse o valor máximo (*max*) estipulado, ela é então dividida (linhas 9-17) retornando em uma estrutura um conjunto de tarefas (linha 23).

A divisão de tarefas não é obrigatória, pois existe a possibilidade de existirem tarefas de um tamanho adequado. Embora seja realizada pelo gerente de tarefas, cabe ao usuário que criou a tarefa programar a rotina pré-definida que efetua a divisão de uma tarefa em subtarefas. As rotinas que devem ser implementadas pelo usuário estão descritas na seção 3.3.4.1.

Após, uma tarefa ou um conjunto de tarefas são enviadas a esse trabalhador através de uma mensagem (*NEW_JOB*), esta contém a identificação e descrição da tarefa. As tarefas enviadas ao trabalhador são armazenadas em uma fila pelo gerente, isto é necessário para

saber o que fazer quando uma resposta for recebida (ver seção 3.3.1.5) e caso as subtarefas não são enviadas elas são mantidas na fila de tarefas não resolvidas e podem ser remetidas a outros trabalhadores. Como as mensagens de nova tarefa e do tempo alvo trafegam pela grade são exibidas pela figura 3.4.

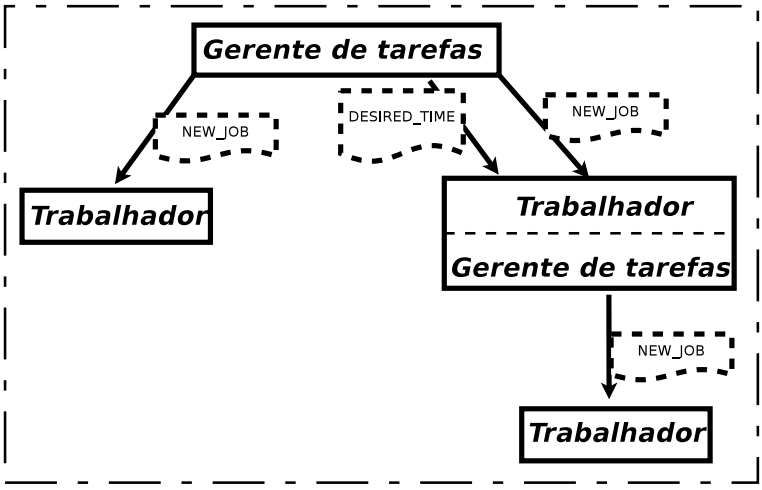

Figura 3.4: Mensagem de nova tarefa e tempo alvo.

Conforme são criadas e enviadas subtarefas, também são criados e recebidos sub-resultados. Estes devem então ser recebidos por um gerente e a partir deles compor o resultado final. Esta é a responsabilidade do gerente de respostas, a qual é abordada a seguir.

3.3.1.5 Recepção e formação da resposta final

Quando ocorre a quebra de uma tarefa em tarefas menores estas são resolvidas pelos trabalhadores. Assim que terminarem seu trabalho, o resultado encontrado é transmitido para seu gerente. A figura 3.5 ilustra como esse procedimento ocorre.

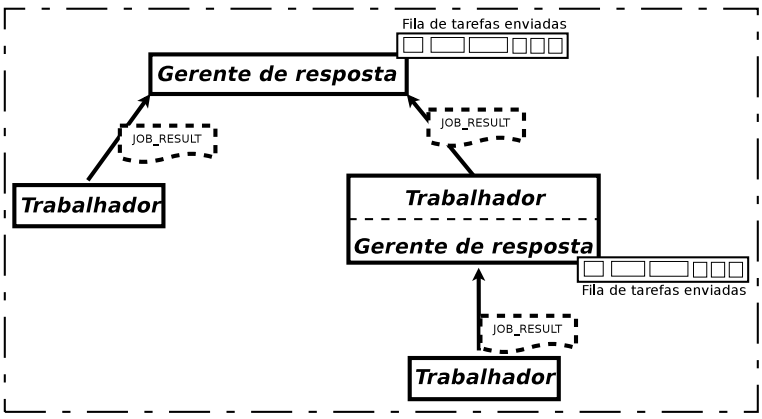

Figura 3.5: Mensagem referente a resposta de uma tarefa.

No momento em que um trabalhador terminar de processar uma tarefa ele envia uma mensagem (*JOB_RESULT*) contendo a identificação da tarefa e a descrição do resultado para o gerente de respostas. O gerente deve então, a partir de todos os sub-resultados encontrados, formar a resposta final da tarefa que originou as subtarefas. A figura 3.6 exibe o código fonte utilizado para gerar a resposta final.

```
70                  ....
71                  this.job = this.jobManager.getSendJobList(this.j.getID());
72                  this.job.setResultDescription(this.j.getResultDescription());
73                  if (this.job.isSolved()) {
74                      if (this.job.getParentJob() != null) {
75                          this.parentJob = this.jobManager.getParentJobList(this.job.getParentJob());
76                          this.parentJob.setSubResult(this.job.getResultDescription());
77                          this.parentJob.reduceSubJobCount(1);
78                          if (this.parentJob.isSolved()) {
79                              this.jobManager.setSendJobList(this.parentJob.getID(), this.parentJob);
80                              this.answerList.put(this.parentJob);
81                          } else {
82                              this.jobManager.setParentJobList(this.job.getParentJob(), this.parentJob);
83                          }
84                      } else {
85                          this.managerProxy.resultSend(this.getID(), this.job.getID(),
86                                  this.job.getResultDescription());
87                      }
88                  } else {
89                          this.jobManager.setSendJobList(this.job.getID(), this.job);
90                  }
91                  ....
```

Figura 3.6: Código fonte utilizado para gerar a resposta final

Para lograr êxito na obtenção da resposta final o gerente deve saber a qual subtarefa pertence um determinado resultado (linha 71). Ao receber uma resposta, o gerente localiza na fila de tarefas enviadas a tarefa a qual resposta pertence e atribui-se o seu resultado (linha 72). Caso a resposta de uma determinada tarefa não esteja completa armazena-se a tarefa novamente na fila de tarefas enviadas, caso contrário, forma-se a resposta final e esta são enviadas ao gerente do gerente (linhas 73-90).

Quando não for mais possível enviar uma mensagem de resposta, ou seja, não existir um gerente do gerente implica que se chegou ao resultado final da execução de uma tarefa que havia sofrido sucessivas divisões ou não.

3.3.2 Trabalhador (*Worker*)

Conforme descrito na seção da arquitetura do *TUXUR* o trabalhador é responsável por processar e devolver os resultados das tarefas processadas a seu gerente. Além disto, ele é também responsável por mantê-lo informado sobre seu estado. A fim de permitir que as atividades internas desempenhadas por um trabalhador sejam independentes definimos três funções **recepção, processamento de uma tarefa** e **atualização de estado**. Estas

características são descritas nas próximas duas subseções (3.3.2.1 e 3.3.2.2).

3.3.2.1 Recepção e Processamento de uma tarefa

Ao receber uma tarefa a função recepção adiciona esta tarefa em uma fila de tarefas inacabadas. Após, a função processamento retira, quando houver, uma tarefa da fila e a executa. Caso não houver tarefas nesta fila a função fica aguardando a espera de uma tarefa. Ao terminá-la, uma mensagem de resposta é formada e enviada ao seu gerente. Este processo se repete sucessivamente até que esta fila fique vazia.

3.3.2.2 Atualização de estado

Ao concluir o processamento de uma tarefa a função de atualização de estado envia uma mensagem de estado para o gerente. Conforme descrito anteriormente esta mensagem possui a identificação do trabalhador, quantidade de trabalho e sua capacidade.

Conforme o trabalhador vai completando a execução das tarefas ele atualiza o número que representa sua capacidade com os dados da última execução. Ele mantém uma média ponderada da capacidade das últimas execuções, para isso ele aplica a fórmula 3.5.

$$CN = CV * A + (\frac{W}{T}) * (1 - A) \tag{3.5}$$

Onde *CN* é a nova capacidade, *CV* representa a capacidade anterior, *W/T* é a carga sobre o tempo de execução da última tarefa (a capacidade medida para a última tarefa) e *A* é um fator de peso da última medida em relação às anteriores, esse valor deverá ser entre 0 e 1.

3.3.3 Lançador (*Launcher*)

O lançador é responsável pela descoberta e lançamento de novos nós e o estabelecimento de um canal de comunicação entre eles. Na próxima subseção descrevemos como o processo de lançamento ocorre.

3.3.3.1 Lançamento de um nó

Atualmente, a implementação do lançador é centralizada e simplificada que isoladamente realiza o processo de criação da topologia dando origem à grade de computadores. A criação da topologia baseia-se em um arquivo de descrição que contém todos os computadores que pertencem à grade. Dentre os dados deste arquivo constam, por exemplo,

a forma de comunicação com o lançador (*Internet Protocol* (IP) + PORTA) e o programa a ser lançado. Futuramente existirá um programa distribuído ocupando seu lugar.

A figura 3.7 ilustra o procedimento de adição de um novo nó à grade.

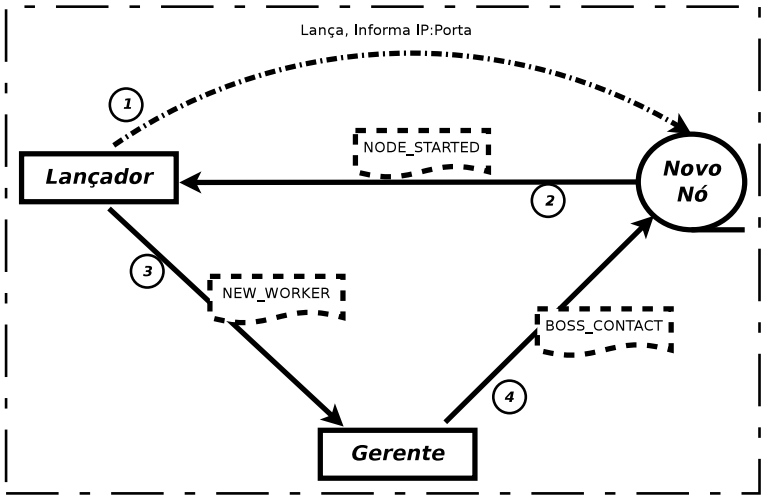

Figura 3.7: Lançamento de um novo nó à grade.

No primeiro momento o lançador executa o programa que deve ser lançado, passando como argumento um identificador para o novo nó e uma descrição de como ele deve se comunicar com o lançador (**1**). Após, o novo nó envia uma mensagem ((**2**) *NODE_STARTED*) para o lançador contendo a descrição do canal de comunicação que o gerente deverá usar para se comunicar com ele. Então, o lançador envia uma mensagem ((**3**) *NEW_WORKER*) para o gerente contendo o identificador e a descrição do canal de comunicação do novo trabalhador. Por fim, o gerente estabelece uma conexão com o trabalhador, através do canal recebido, e envia a ele uma mensagem ((**4**) *BOSS_CONTACT*) contendo a descrição do seu canal de comunicação e com isso a comunicação é dada como estabelecida.

3.3.4 Tarefa (*Job*)

Uma tarefa é essencialmente a representação da aplicação a ser resolvida. Basicamente, ela é composta pelas rotinas que a computam e pelos dados que são processados nestas rotinas. A programação das rotinas computacionais relacionadas à resolução de uma tarefa é de responsabilidade do usuário. Estas rotinas serão descritas a seguir.

3.3.4.1 Métodos a serem implementados pelo usuário do TUXUR

O usuário deve programar algumas rotinas computacionais pré-definidas que permitam uma tarefa ser: inicializada, descrita, calculada, dividida em tarefas menores e que verifique se uma tarefa já está completamente resolvida. Além disto, rotinas que descrevem um resultado e componham o resultado final também devem ser implementadas. A seguir iremos descrever cada uma destas rotinas.

Execução. Calcula uma tarefa. Deve conter a programação necessária para encontrar a resposta desta tarefa. A implementação utiliza um método (*solve()*).

Dimensão. Define a quantidade de trabalho de uma tarefa. *int workload()* é o método que esta rotina implementa retornando o tamanho da tarefa.

Divisão. Uma tarefa cria uma subtarefa a partir de um tamanho predeterminado. A implementação desta rotina é desempenhada pelo método denominado *jobBreak()*;

Inicialização. Inicializa uma tarefa cujo número a ser executado é passado como argumento. Atualmente, a inicialização é implementada através do método construtor da tarefa.

Descrição. Descrever uma tarefa significa fornecer todos os dados necessários para instanciar essa tarefa, possivelmente em outro nó. Na implementação atual, a obtenção da descrição de uma tarefa é representada pelo método (*getDescription()*).

Resolução. Esta rotina gera de forma incremental o resultado final de uma tarefa. A implementação descreve um método (*setSubResult()*) que recebe a descrição do resultado. A rotina que fornece a descrição do resultado é implementada pelo método (*getResultDescription()*).

***Solução*.** Rotina que define se uma tarefa está resolvida ou não. Implementa um método (*boolean isSolved()*) que retorna um valor *booleano*. Retorna *true* caso a tarefa já esteja resolvido e *false* do contrário. A tarefa está resolvida quando não possuir mais subtarefas associadas à ela.

3.3.5 Interface do usuário (*Interface*)

Este componente é responsável pelo relacionamento com os usuários do *framework*. Permite que o usuário realize consultas e envie ordens. Dentre as consultas listamos: in-

formações sobre os andamentos das tarefas, tarefas terminadas, respostas já encontradas, o tempo total de processamento, quantidade de nó que integram a grade, etc. Quanto às ordens, podemos destacar: submissões de tarefas, inclusão de novos computadores à grade.

3.3.6 Interface de comunicação (*Communicator*)

A interface de comunicação não é verdadeiramente um componente, os métodos fundamentais para o envio e recebimento de mensagens (*send/receive*) são implementados por esta interface. Toda a comunicação é realizada através de vetores de bytes.

Optou-se por manter a comunicação a mais simples possível baseada na troca de mensagens pré-definidas entre os componentes. Para efetuar a troca de mensagens foram identificados cenários nos quais é necessária a comunicação. A seguir serão descritos os tipos de mensagens bem como a definição de cada uma:

- **NODE_STARTED**: mensagem enviada de um trabalhador ou gerente recém criado na estrutura hierárquica para o lançador, contendo a identificação do nó e a descrição do seu canal de comunicação;
- **NEW_WORKER**: mensagem enviada para um gerente informando a existência de um novo trabalhador. A mensagem possui a identificação e a descrição do canal de comunicação do trabalhador recém criado;
- **BOSS_CONTACT**: mensagem enviada pelo gerente a um novo trabalhador para estabelecer a comunicação, contendo a descrição do canal de comunicação do gerente;
- **NEW_JOB**: mensagem enviada pelo gerente a um trabalhador. A informação contida nesta mensagem é a identificação da tarefa e sua descrição;
- **NODE_STATUS**: mensagem enviada por um trabalhador para seu gerente, possui a identificação do trabalhador, a capacidade e sua carga de trabalho;
- **JOB_RESULT**: mensagem enviada por um trabalhador para seu gerente. A identificação da tarefa e a descrição do resultado fazem parte desta mensagem;
- **DESIRED_TIME**: mensagem enviada pelo gerente aos seus trabalhadores contendo o tempo alvo de execução.

Finalizada a descrição das funcionalidades e componentes do *TUXUR*, o próximo capítulo aborda a avaliação do *framework* aqui proposto. Para isso, iremos executar uma aplicação que simula o crescimento populacional de insetos em uma lavoura de trigo. A seguir iremos descrever e discutir os resultados obtidos.

4 AVALIAÇÃO

Este capítulo aborda assuntos relacionados a avaliação do *TUXUR*. Antes de executar a aplicação que serviu para avaliar e validar o *framework*, realizamos testes com uma aplicação que nos permitisse criar uma tarefa simples e de fácil divisão. Para tanto, no apêndice B apresentamos os resultados obtidos bem como a descrição da aplicação utilizada.

Na próxima seção discorre-se sobre a metodologia empregada para a realização da avaliação, onde são definidos como e quais objetivos espera-se alcançar. A discussão sobre os resultados obtidos finaliza este capítulo.

4.1 Metodologia

Deve-se encontrar uma forma de avaliar o quão bem o gerente desenvolve sua função. De um modo geral, a função deste é delegar tarefas a um conjunto de trabalhadores garantindo que eles tenham trabalho suficiente a fim de que não fiquem ociosos por um período de tempo muito longo e assim desequilibrando o volume de trabalho. Entende-se por bem, nesse caso, se ele está sendo eficiente na delegação de tarefas utilizando como requisito o tempo que o sistema estima como adequado para que um trabalhador possa operar de forma independente (*tempo alvo*).

Desta forma, optamos por coletar durante o processo de resolução das tarefas as seguintes informações: a capacidade de processamento, a quantidade de trabalho a resolver, o tempo de resolução, o tempo em que um trabalhador ficou ocioso a espera de novas tarefas e a evolução do tempo alvo. Todas estas informações foram coletadas nos gerentes e trabalhadores.

De posse destes dados, é possível fazer uma representação gráfica das informações obtidas na resolução das tarefas. Espera-se que tal representação ilustre a eficiência na

delegação das tarefas pelo gerente, com isto diminuindo o tempo final de execução.

Planejamos três cenários para realização da avaliação. Definimos comparar o desempenho entre configurações diferentes, verificando o funcionamento e comportamento em diferentes alterações de ambientes. Sendo assim, o primeiro cenário envolve a escolha, pelo gerente, de um tempo alvo fixo para a computação de uma tarefa. Já no segundo a utilização do tempo alvo dinâmico, onde além do tempo alvo fixo o gerente define limites para os tempos de execuções. Por fim, no terceiro cenário avaliamos a adaptabilidade do ambiente, incluindo novos nodos e alterando a capacidade dos nós.

Para a avaliação envolvendo estes cenários, decidimos resolver um problema que envolva a simulação computacional do crescimento populacional de insetos pragas agrícolas. Optamos por um problema que envolva a execução das instruções sobre uma grande quantidade de variáveis. A seguir iremos descrever mais detalhes sobre este problema.

4.1.1 O problema a ser representado

Para a comunidade mundial o trigo destaca-se como um alimento de grande importância do ponto de vista econômico, além disto, sua produtividade é variada podendo ser afetada por alguns elementos como as condições do clima, pragas, entre outras. (TOEBE; PAVAN; FERNANDES, 2014) apresentam um modelo baseado em agentes e avaliam a hipótese da simulação de patossitemas do nanismo amarelo da cevada em trigo (Triticum aestivum L.) e o afídeo Rhopalosiphum padi L. (R. padi) (Hemiptera: Aphididae). Para este afídeo o modelo proposto utiliza como parametrização diferentes aspectos e especificidades do ciclo de vida dos insetos, como a reprodução, a movimentação, a mortalidade, a alimentação e o desenvolvimento.

A utilização de modelos de simulação baseada em agentes, na área agrícola, busca prever a ocorrência de fenômenos bióticos ou abióticos, que possam impactar a produtividade nesta área. Esses modelos são utilizados na simulação de ações e interações através de sistemas dinâmicos. O modelo apresentado oferece suporte para a utilização com diferentes espécies de insetos e flexibilizando a possibilidade de parametrização de dados iniciais, tornando o modelo apto a lidar com um problema específico. Contudo, os modelos baseados em agentes apresentam alguns problemas. Dentre esses problemas podemos citar o grande consumo de memória e o tempo de processamento para sua execução, em especial quando o número de agentes é elevado.

O R.padi é uma das espécies mais a infestar os cereais no sul do Brasil. Seu ciclo de vida depende fortemente das condições climáticas. Em regiões de clima muito frio, regiões polares, sua reprodução pode ser através da produção de ovos e em regiões de climas mais amenos, como no sul do Brasil, sua reprodução é através de larvas. Outro fator que favorece a reprodução deste tipo de inseto é o alto período de estiagem, quanto maior este período maior será a quantidade de insetos e quanto maior o nível de chuva menor será sua reprodução.

Para executar o modelo utilizaram-se dados climáticos das estações climatológicas de Passo Fundo, Santa Rosa, Caçapava do Sul e Lagoa Vermelha, todas localizadas no Estado do Rio Grande do Sul e integradas à rede do Instituto Nacional de Meteorologia (INMET). Além disso, o Laboratório de Entomologia da Embrapa Trigo, Passo Fundo, disponibilizou dados obtidos por seus experimentos e avaliações com trigo, no período de 2008 a 2013.

4.1.1.1 Descrição da tarefa

O problema a ser resolvido consiste em determinar qual é o mês mais favorável para uma espécie de inseto se reproduzir e com isto indicar quantas plantas foram infestadas no período de 2008 à 2013, em uma área de $10m^2$, nas quatro estações climatológicas. Basicamente, a aplicação necessita de alguns parâmetros para ser executada, por exemplo, quantidade de insetos iniciais, índice de chuva, temperatura (mínima, média e máxima), velocidade e direção do vento. Neste trabalho utilizamos um inseto no início da execução e os parâmetros foram obtidos das estações e laboratório citados na subseção anterior.

A descrição da tarefa consiste em um vetor de 288 posições, contendo em cada posição os dados mencionados no parágrafo anterior referente a cada mês de cada estação climatológica. A quantidade de trabalho da tarefa é representada pelo tempo obtido na execução sequencial de cada mês. Este número é uma precisão dada pela própria tarefa.

Para realizar a divisão de uma tarefa em subtarefas é necessário programar o método que realiza esta subdivisão. O código para realizar a divisão de uma tarefa pode ser visualizado na figura 4.1.

```
List<IJob> listTemp = new ArrayList<IJob>();
List<DadosParaCalculos> listDadosCalculosTemp =
        new ArrayList<DadosParaCalculos>();

int div = (this.listDadosCalculos.size() / 2);
if (div == 0) {
    listDadosCalculosTemp.add(this.listDadosCalculos.get(div));
    IJob j1 = new CampoTrigo(listDadosCalculosTemp);
    j1.setID(getID() + "A");
    j1.setParentJob(getID());
    listTemp.add(j1);
} else {
    for (int i = 0; i < div; i++) {
        listDadosCalculosTemp.add(this.listDadosCalculos.get(i));
    }
    IJob j1 = new CampoTrigo(listDadosCalculosTemp);
    j1.setID(getID() + "A");
    j1.setParentJob(getID());
    listTemp.add(j1);
}
listDadosCalculosTemp = new ArrayList<DadosParaCalculos>();
for (int i = div; i < this.listDadosCalculos.size(); i++) {
    listDadosCalculosTemp.add(this.listDadosCalculos.get(i));
}
IJob j2 = new CampoTrigo(listDadosCalculosTemp);
j2.setID(getID() + "B");
j2.setParentJob(getID());
listTemp.add(j2);
```

Figura 4.1: Código do método para realizar a divisão de tarefas.

Neste trabalho uma tarefa origina outras duas tarefas a partir da divisão do vetor pela metade. Após a divisão as tarefas de maior tamanho são inseridas no início do vetor. As demais rotinas que devem ser implementadas pelo usuário são apresentadas no apêndice A. Os resultados obtidos, conforme a metodologia planejada e em seus respectivos cenários, são apresentados na seção a seguir.

4.2 Resultados Obtidos

Os testes foram realizados sobre a grade regional de alto desempenho (GRIDRS) (GRIDRS, 2014). Esta plataforma se caracteriza por possuir recursos computacionais geograficamente distribuídos e ser heterogênea. Assim, foram empregados ao total seis computadores cujas características de hardware são apresentadas na tabela B.1.

ID - Localização	CPU	Memória	Sist. Oper.	Num. Cores
Node1-2 - UFPEL	Intel Quad-core i5 2.9Ghz	16GB	Debian 6	2
Node3 - UFSM	Intel Xeon 1.9Ghz	8GB	Debian 7	4
Node4 - PUC-RS	Intel Xeon 3Ghz	2GB	Debian 6	2
Node5-6 - UFRGS	Intel Pentium III 1.1Ghz	1GB	Debian 6	2

Tabela 4.1: Descrição do *hardware* dos computadores utilizados na avaliação.

Devido a quantidade de máquinas que o ambiente possui e verificar a capacidade do gerenciamento hierárquico do *TUXUR*, organizamos uma estrutura hierárquica com dois gerentes. Futuramente o *TUXUR* poderá ser avaliado com outras configurações de hierarquia.

O *Gerente1* (G1) e *Gerente2* (G2) gerenciam seis trabalhadores, totalizando doze trabalhadores (T1-T12). Os gerentes localizam-se, respectivamente, nos Node1 e Node3. A figura 4.2 ilustra a organização hierárquica com dois gerentes.

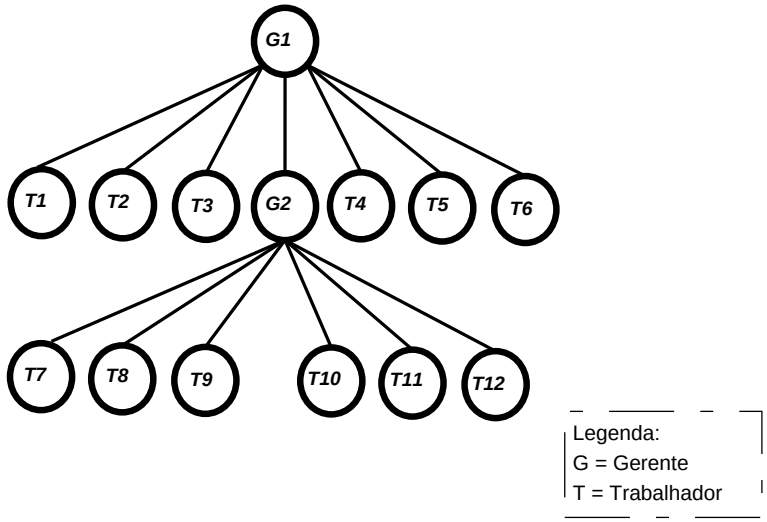

Figura 4.2: Organização hierárquica dos gerentes.

Antes de aplicar a metodologia apresentada, realizamos três execuções sequenciais do programa em um nó de cada site. A tabela 4.2 apresenta o tempo final de cada execução.

Site	Execução 01	Execução 02	Execução 03
UFRGS	8015	7862	7845
UFPEL	7526	7520	7528
UFSM	8076	8110	8082
PUCRS	8118	8123	8105
Média Total			**7909**
Desvio Padrão Total			**289**

Tabela 4.2: Tempos, em segundos, das execuções sequenciais em um nó de cada site.

Para comparar com os demais tempos obtidos nos três cenários iremos utilizar a média entre os tempos sequenciais obtidos em cada site. Conforme mencionado anteriormente, a avaliação foi realizada em três cenários. Uma descrição sobre eles e os resultados encontrados são discutidos a seguir.

4.2.1 Cenário 1: tempo alvo fixo

Para o cenário um utilizamos o tempo alvo fixado em 90 segundos. Aproximadamente 10% do tempo total estimado para o cálculo, para que cada nó recebesse aproximadamente 10 tarefas. A figura 4.3 ilustra a execução da aplicação onde cada linha representa um trabalhador (T1-T12), com cada retângulo correspondendo a uma tarefa executada por esse trabalhador. Por simplicidade, iremos discutir os resultados do primeiro e o último trabalhador a terminar suas tarefas.

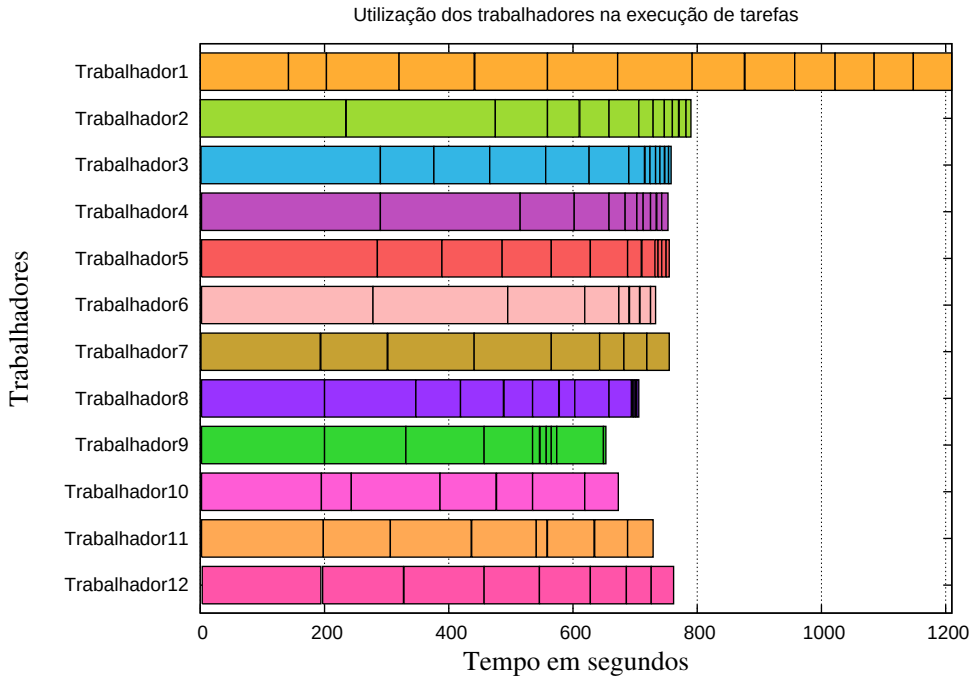

Figura 4.3: Utilização dos recursos com tempo alvo fixo.

Pode-se ver que alguns trabalhadores não estão sendo utilizados na totalidade durante o período em que estiveram disponíveis para a grade. O tempo total de execução do *Trabalhador1*, por exemplo, foi superior ao tempo total de execução do *Trabalhador9*, nesta situação observamos que não está sendo tirado proveito total da grade, por termos uma má distribuição de tarefas. O tempo final total de execução foi 1235 segundos, redução de 84% em relação ao tempo de execução sequencial.

O problema de má distribuição ocorreu porque quando aconteceu o primeiro caso de um trabalhador ficar sem tarefa e não existirem mais tarefas a distribuir (e consequentemente esse trabalhador teve que parar de trabalhar), ainda existia outros trabalhadores que tinham tarefas para resolver. Uma forma de resolver este problema seria definir limites para os tamanhos das tarefas a enviar.

A figura 4.4 exibe o gráfico da quantidade de trabalho recebida e a capacidade do *Trabalhador1*. Ressaltamos que neste cenário o tempo alvo é fixo (90 segundos).

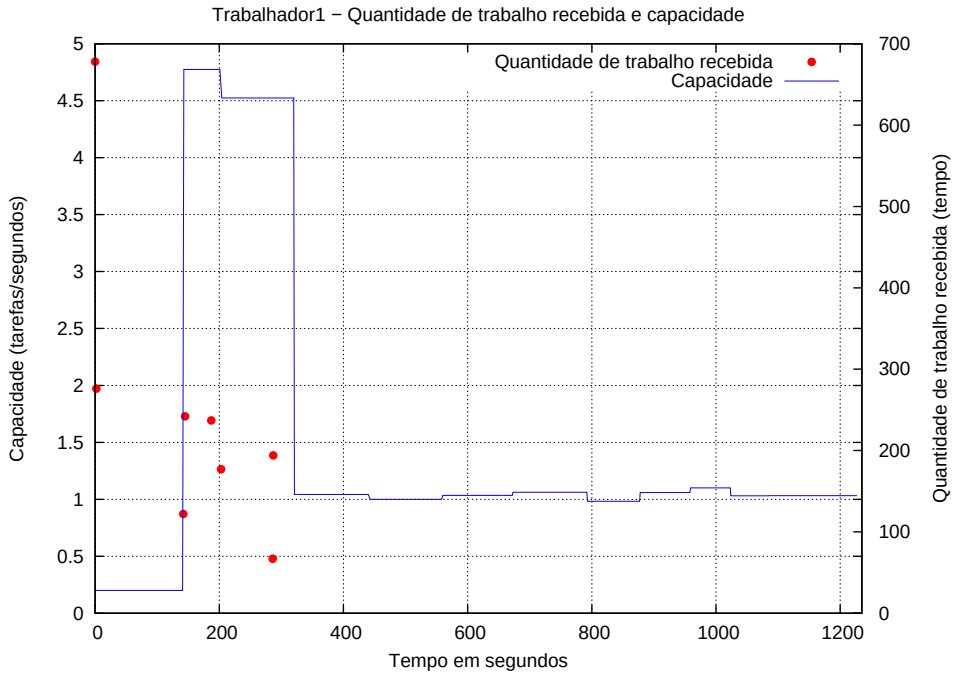

Figura 4.4: *Trabalhador1* - Quantidade de trabalho recebida e capacidade.

No início da execução o trabalhador recebeu trabalho bem acima de sua capacidade, isto se deve ao fato de que todas as tarefas de maior tamanho e que não puderam ser mais divididas foram designadas a serem executadas por primeiro. Após esta execução as demais tarefas ficaram de acordo com os limites estipulados, ou seja, aproximadamente no tempo 200 o *Trabalhador1* apresentava capacidade em torno de 4 e 90 de tempo alvo, como resultado a quantidade de trabalho recebida foi aproximadamente 120. Ressaltamos que a capacidade é recalculada conforme descrito na seção 3.3.2.2. A figura 4.5 ilustra os valores apresentados pelo *Trabalhador9*.

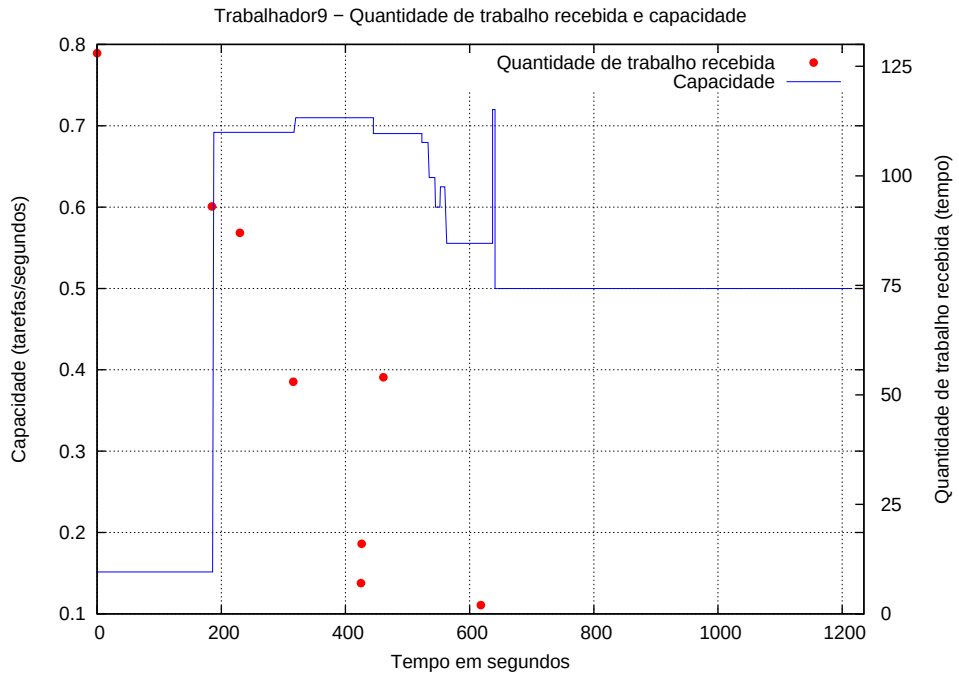

Figura 4.5: *Trabalhador9* - Quantidade de trabalho recebida e capacidade.

O *Trabalhador9* no início de sua execução também recebeu trabalho acima da sua capacidade, mas após concluída a execução das primeiras tarefas e descobrindo sua real capacidade, as tarefas posteriormente recebidas foram se adaptando conforme sua capacidade de processamento.

Definindo um tempo alvo fixo temos o problema de má distribuição das tarefas levando alguns trabalhadores a ficarem ociosos no final da execução, conforme discutido no início desta seção e ilustrado na figura 4.3. Uma forma de resolver o problema de má distribuição de tarefas seria definir limites para o tamanho da tarefa a enviar. Na próxima seção apresentamos esta proposta juntamente com os resultados obtidos.

4.2.2 Cenário 2: tempo alvo dinâmico

A diferença deste cenário para o cenário um é a utilização do tempo alvo dinâmico. O problema de se distribuir trabalho antes de se ter uma ideia melhor da capacidade total de trabalho é de se ter uma forma de distribuição de trabalho que acabe enviando um trabalho exageradamente grande para um trabalhador quando se tem poucos trabalhadores, diminuindo a capacidade de distribuição do gerente mais tarde. Esse problema pode ser evitado com um limite máximo de tamanho de trabalho a enviar.

Sendo assim, definimos utilizar o tempo alvo dinâmico utilizando limites para esse

tempo. Para testarmos o comportamento do sistema com tempo alvo variável, definimos o tempo alvo como sendo $\frac{1}{10}$ do tempo restante estimado até o final da execução, limitado a um valor mínimo e máximo de 5 e 90 segundos, respectivamente. Caso o tempo alvo calculado ultrapasse um destes limites o tempo alvo assume o valor do tempo limite. Futuramente poderá haver outros ambientes de testes com outras estimativas de tempo.

A figura 4.6 ilustra a utilização dos recursos na execução da aplicação para determinar qual é o mês mais favorável para uma espécie de inseto se reproduzir.

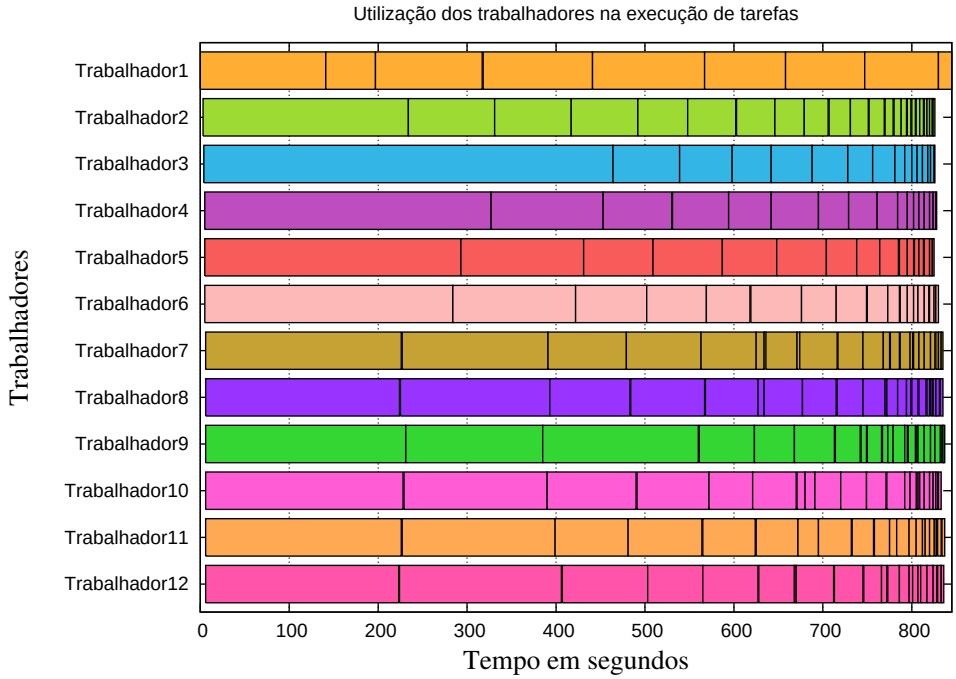

Figura 4.6: Utilização dos recursos com tempo alvo dinâmico.

A utilização dos trabalhadores nesta forma de atribuição de tarefas se mostrou mais equilibrada, ao final da execução, do que a do cenário um. A execução de cada tarefa seguiu as regras conforme definimos, ou seja, cada trabalhador recebeu uma tarefa razoavelmente grande o suficiente para que a grade ficasse equilibrada e conforme a execução evoluiu o tamanho das tarefas foi reduzindo, este tamanho diminuiu pelo motivo de que o tempo total de execução também reduziu. Sendo assim, o tempo total de execução deste cenário foi de 869 segundos, aproximadamente 30% menor que o tempo apresentado no cenário um que foi se 1235 segundos.

Por simplicidade iremos apresentar os resultados obtidos do primeiro e último trabalhador a terminarem suas tarefas. As figuras 4.7 e 4.8 ilustram a quantidade de trabalho recebida e a capacidade referente ao *Trabalhador1* e *Trabalhador5*.

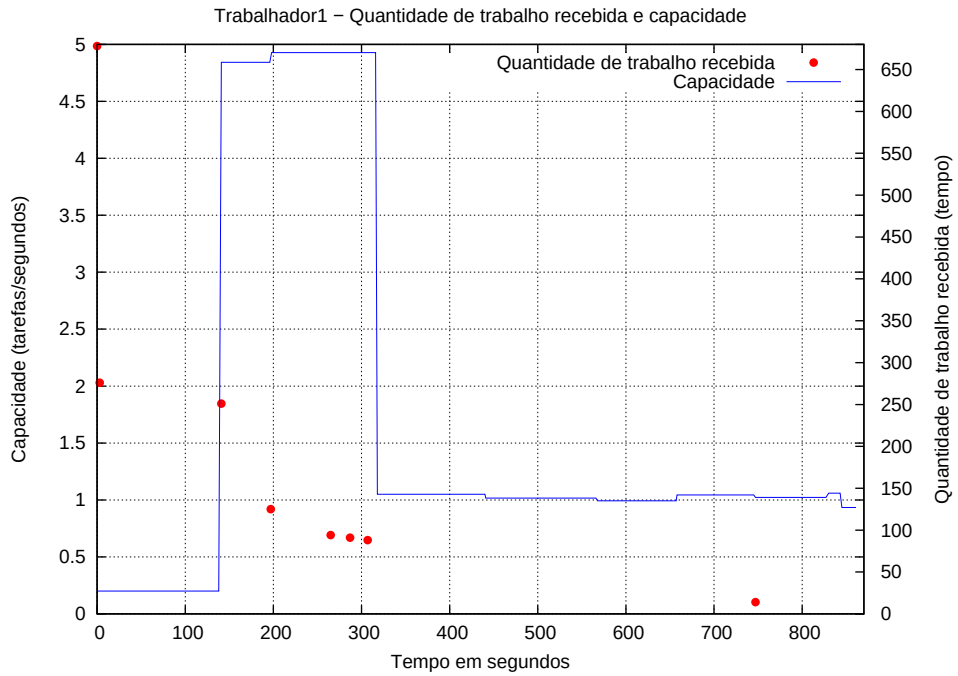

Figura 4.7: *Trabalhador1* - Quantidade de trabalho recebida e capacidade.

Assim como no primeiro cenário, no início da execução os *Trabalhador1* e *Trabalhador5* receberam trabalho bem acima de suas capacidades, isto se deve ao fato de que todas as tarefas de maior tamanho e que não puderam ser mais divididas foram designadas a serem executadas por primeiro. Depois de concluída a execução da primeira tarefa, a capacidade do *Trababalhador1* teve um aumento considerável em relação a sua capacidade inicial, consequentemente a quantidade de trabalho recebida foi proporcional a ela.

Depois de concluído o processamento da terceira tarefa, verificar figura 4.6, a capacidade do *Trabalhador1* teve uma redução de aproximadamente 80%, estabilizando-se em torno de 1 até o final da execução. Reduzindo esta capacidade seu gerente constatou que não era necessário receber mais trabalho por um período aproximado de 450 segundos, pois a quantidade de trabalho que continha era suficiente para este tempo.

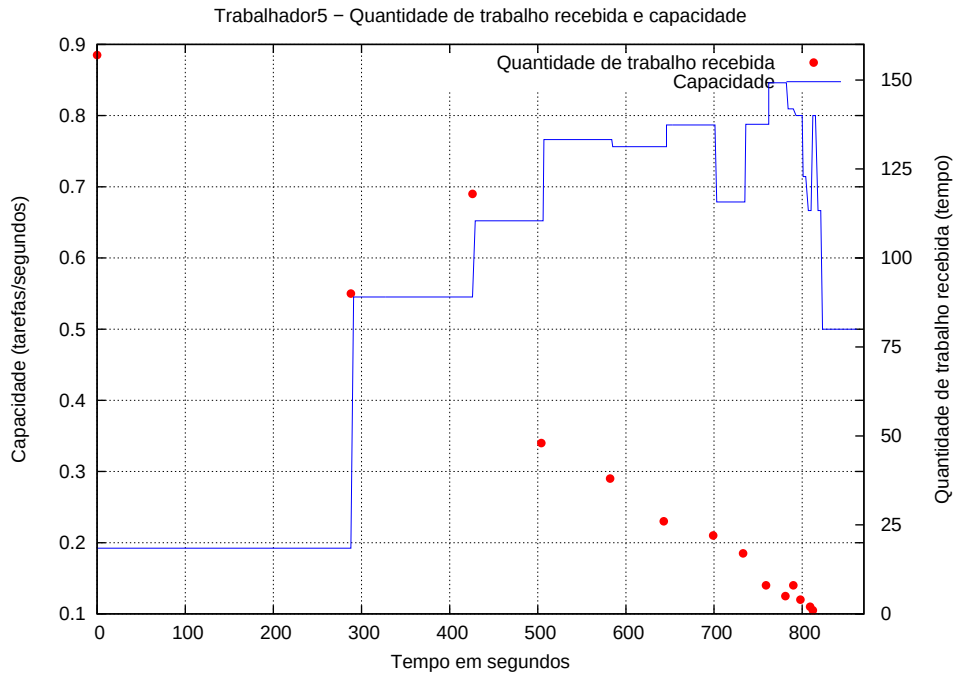

Figura 4.8: *Trabalhador5* - Quantidade de trabalho recebida e capacidade.

A quantidade de trabalho a receber, por cada trabalhador, deve ser proporcional ao tempo alvo e sua capacidade. Observa-se que aproximadamente no tempo 700 o *Trabalhador5* possuía capacidade em torno de 0.8 e o tempo alvo era 6, consequentemente a quantidade de tarefa recebida ficou próxima a 8. Conforme a execução evoluiu o tamanho das tarefas foram diminuindo, este tamanho diminuiu pelo motivo de que o tempo total de execução também reduziu como consequência o tempo alvo também reduziu. A figura B.12 exibe a evolução do tempo alvo.

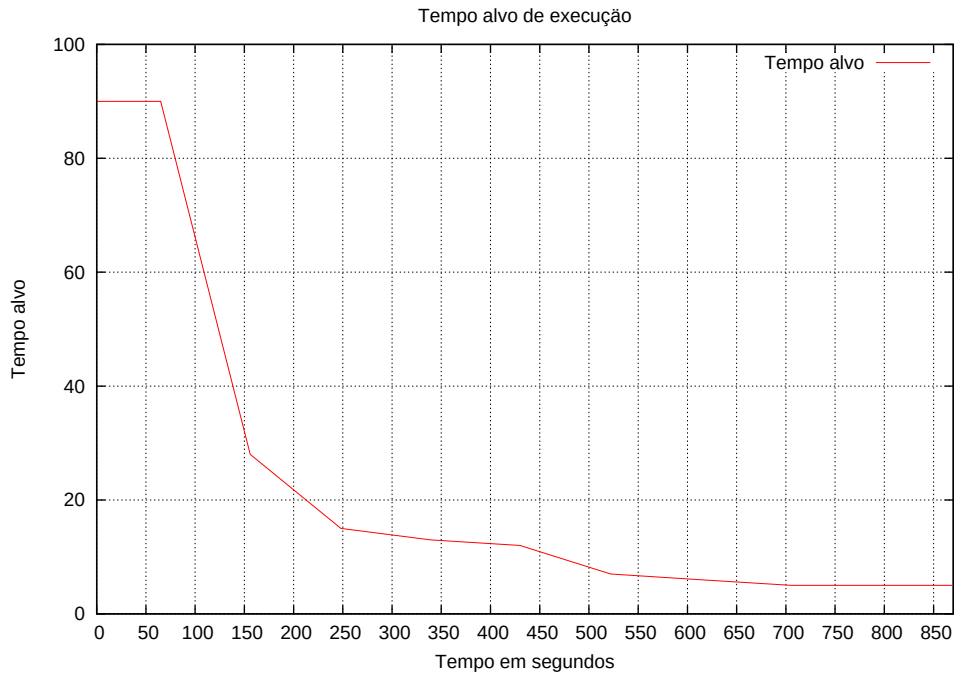

Figura 4.9: Evolução do tempo alvo.

O cálculo de tempo alvo é realizado pelo primeiro gerente da hierarquia, pelo motivo de ele possuir as informações da quantidade de trabalho a realizar e a capacidade total de seus trabalhadores. De posse destas informações é possível estimar qual será o tempo total de execução. Maiores detalhes sobre o cálculo do tempo alvo é descrito na seção 3.3.1.2.

4.2.3 Cenário 3: inclusão de novos recursos

O cenário três avalia o comportamento do *TUXUR* quando a quantidade ou a capacidade dos nós de trabalho são alterados durante a execução de uma tarefa. O suporte a remoção de nós será implementado futuramente. A tarefa executada e a configuração da grade são as mesmas dos cenários anteriores. Foram realizados dois tipos de testes. No primeiro, adicionamos quatro novos nós no meio da execução; no segundo, diminuiu-se a capacidade de trabalho de alguns dos nós existentes, sobrecarregando-os com a execução de tarefas externas ao *TUXUR*. Esses dois experimentos são descritos a seguir.

4.2.3.1 Agregação de nós

Neste cenário avaliamos como o ambiente se comporta adicionando mais trabalhadores após um determinado tempo de execução. Definimos este tempo em 5 minutos, sendo assim, utilizando as mesmas definições do cenário dois e após 5 minutos de execução

foram adicionados a grade quatro trabalhadores subordinados ao *Gerente2*. Estes trabalhadores possuem as configurações de *hardware* apresentadas pelos *node5-6*. Portanto, conforme a organização hierárquica representada na figura 4.2 a estrutura da grade de computadores passou a ser formada por 16 trabalhadores.

Na figura 4.10 exibimos a utilização dos recursos da grade desde o momento em que estes recursos foram disponibilizados até o momentos em que eles ainda estavam disponíveis, mas não foram mais utilizados.

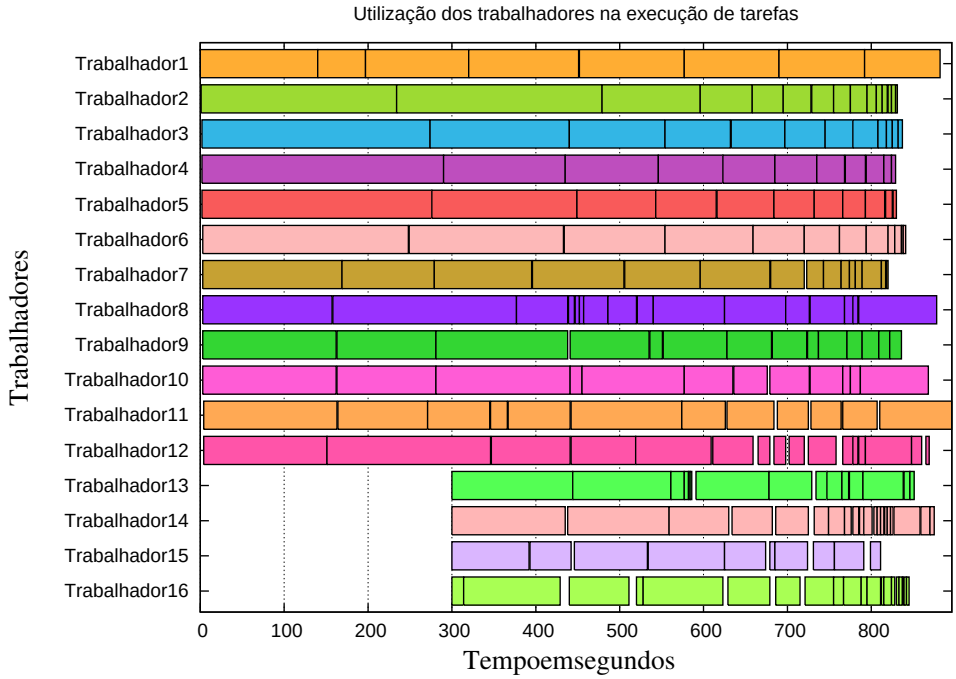

Figura 4.10: Utilização dos recursos com tempo alvo dinâmico e agregação de nós.

Os trabalhadores 13 até 16 começaram a processar as tarefas e utilizar a grade no tempo em que foi estipulado. Comparando o gráfico da figura 4.10 com o da figura 4.6 se percebe que alguns trabalhadores ficaram ociosos por um período de tempo e além disto a execução deste cenário não foi tão eficiente quanto a apresentada no cenário dois. Neste cenário o tempo final de execução foi de 920 segundos enquanto no cenário anterior foi de 869 segundos.

No gráfico da figura 4.11 exibimos a quantidade de trabalho recebida e a capacidade de processamento do *Gerente2* desde o início da execução até o momento em que não existiam mais tarefas para eles distribuírem.

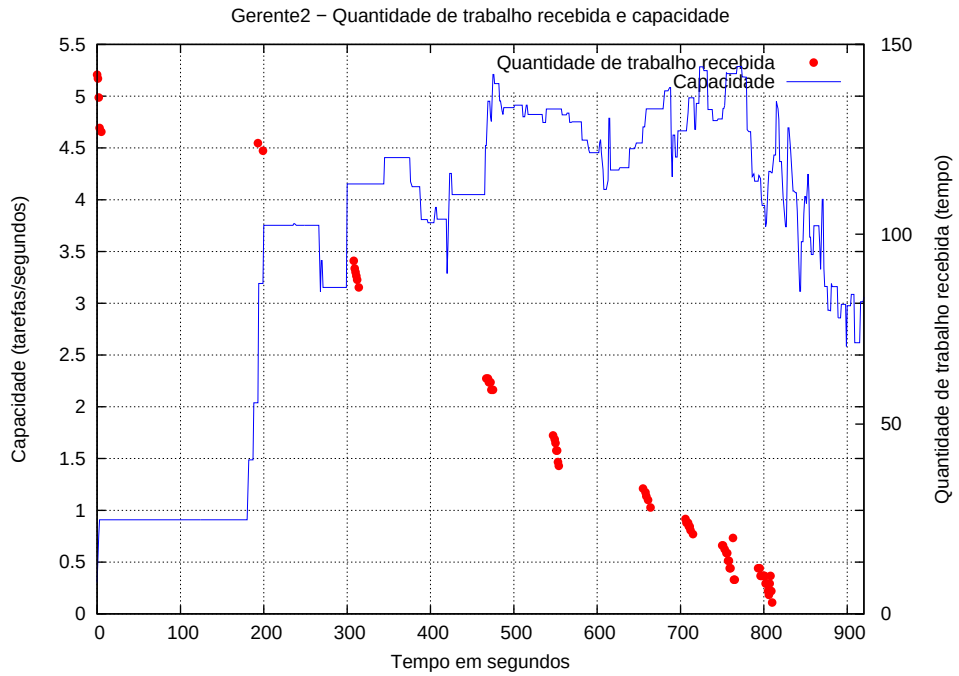

Figura 4.11: *Gerente2* - Quantidade de trabalho recebida e capacidade.

A capacidade do *Gerente2* teve um pequeno aumento aproximadamente no tempo 300. Este aumento deve refletir na redução do tempo alvo que é apresentado na figura 4.12. Observa-se que aproximadamente no tempo 350 houve uma pequena redução deste tempo.

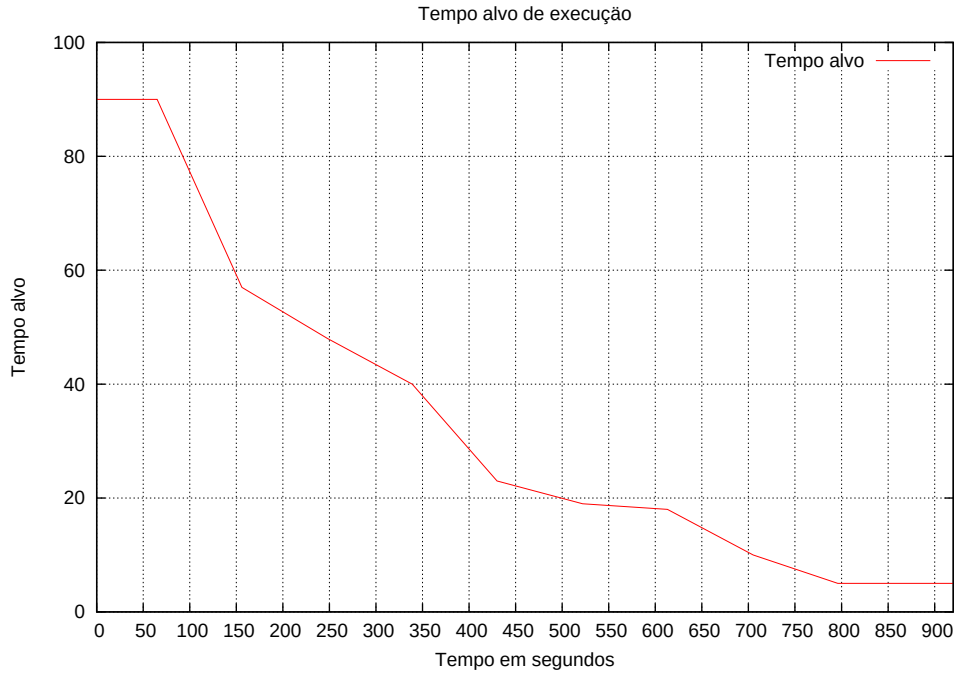

Figura 4.12: Evolução do tempo alvo.

4.2.3.2 Carga Extra

A fim de testar a dinamicidade do *TUXUR* foi gerada uma carga extra, externa ao *framework*, em um nó dos sites da UFPEL (Node2) e UFSM (Node3). Esses nós foram carregados por programas externos após 500 segundos de utilização da grade, durante o período de 300 segundos. Neste cenário utilizamos as mesmas configurações apresentadas no cenário dois.

Por simplicidade iremos exibir os gráficos de um trabalhador do Node2 e Node3. Estes trabalhadores foram os que acabaram primeiro a execução das tarefas. As figuras exibem a capacidade e a quantidade de trabalho recebida referente ao *Trabalhador6* e *Trabalhador8*.

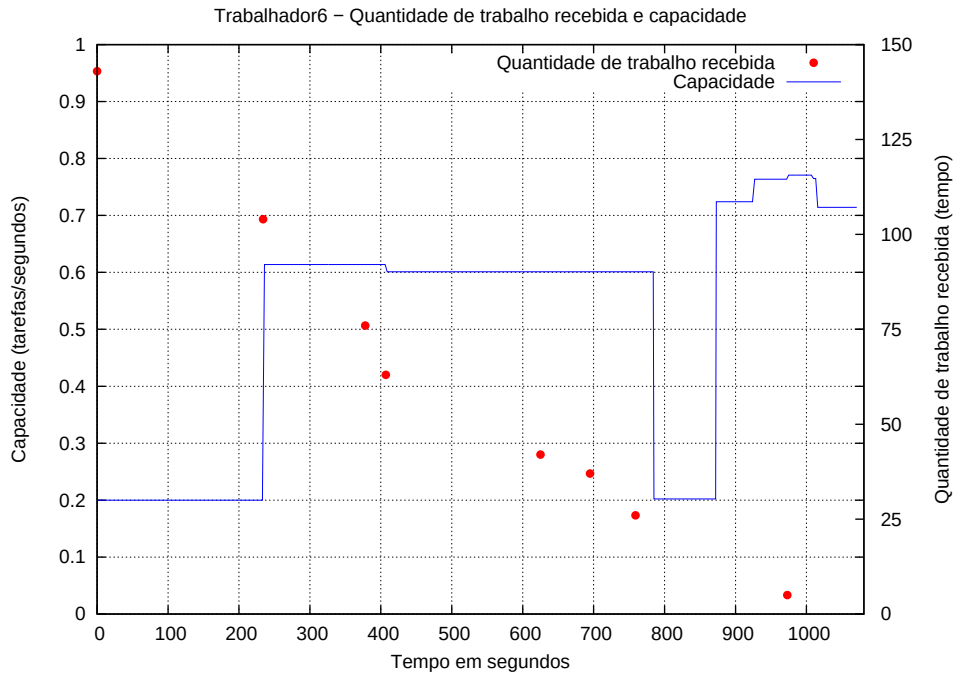

Figura 4.13: *Trabalhador6* - Quantidade de trabalho recebida e capacidade

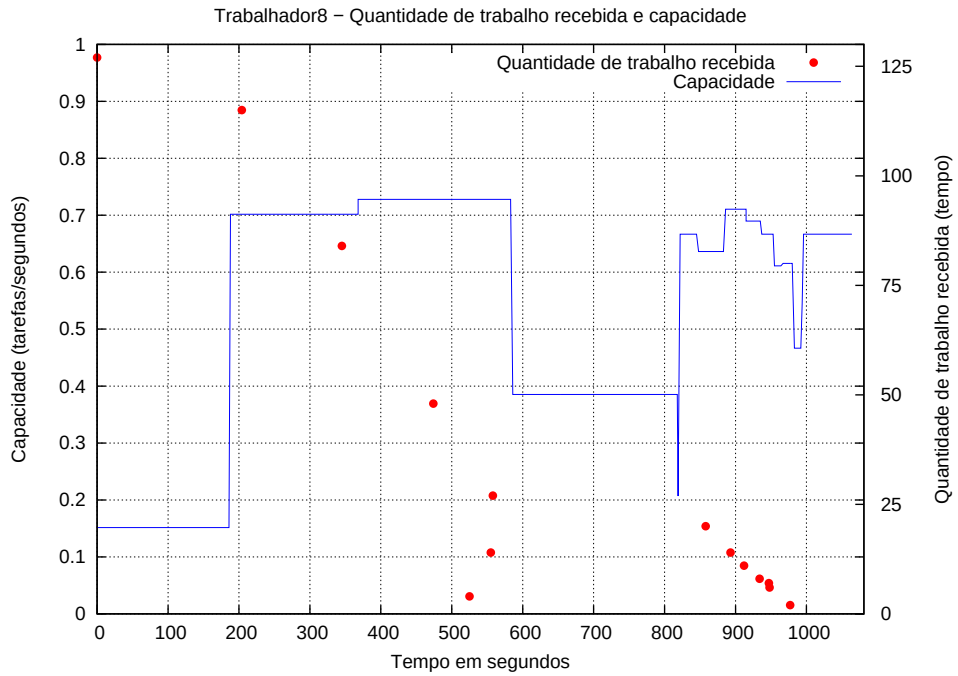

Figura 4.14: *Trabalhador8* - Quantidade de trabalho recebida e capacidade

Constata-se que entre os tempos 500 e 800 houve redução na capacidade dos trabalhadores. Consequentemente, houve redução na quantidade de trabalho recebida onde o *Trabalhador8* não recebeu nenhuma tarefa entre os tempos 600 e 800.

5 CONSIDERAÇÕES FINAIS

Esta dissertação apresentou a importância do emprego de grades de computadores na busca de solução para diversos tipos de problema. A revisão bibliográfica apresentada no capítulo 2 apresenta conceitos e definições que fazem das grades de computadores uma crescente área de pesquisa no meio computacional. Nos trabalhos relacionados foram apresentados diversos *frameworks* e algoritmos para escalonamento de tarefas que buscam solucionar um dos desafios apresentados por essa área.

O presente trabalho buscou contribuir nesse sentido, apresentando o *framework TUXUR*, descrito no capítulo 3, e desenvolvendo o seu componente de gerenciamento que permite, automaticamente e dinamicamente, dividir e delegar tarefas maleáveis, baseada em um tempo alvo, conforme a disponibilidade de recursos e poder computacional dos computadores que compõem a grade.

O *framework* teve o seu funcionamento validado no capítulo 4. Esta avaliação foi dividida em três cenários. No cenário um, optamos em utilizar um tempo alvo fixo, cerca de 10% do tempo total estimado. Os resultados apresentados não foram satisfatórios, pelo motivo de que os recursos não foram ocupados durante todo o tempo em que estiveram disponíveis. Mas, houve redução no tempo final de execução, cerca de 84% em relação ao tempo sequencial.

No cenário dois, optamos em utilizar o tempo alvo variado definindo limites para esse tempo. Os resultados obtidos nesta configuração de ambiente se mostraram aceitáveis em relação ao resultados obtidos no cenário um, obtendo um tempo final de execução 30% menor que o obtido no cenário um. Sendo assim, ressaltamos a importância da definição de uma tempo alvo variado onde alcançamos o principal objetivo do escalonamento, a baixa existência de nós ociosos e a grade praticamente equilibrada.

Por fim, no cenário três objetivamos testar o comportamento do *TUXUR* com a agre-

gação de novos nós e a capacidade de se adaptar sobrecarregando alguns nós com carga extra. No primeiro caso, agregação de novos nós, os resultados foram parcialmente satisfatórios pois alguns trabalhadores ficaram ociosos e a execução não ficou totalmente equilibrada, mas o tempo final de execução foi menor que o obtido no cenário um.

Os trabalhos futuros estão relacionados as melhorias que serão futuramente implementadas no *framework TUXUR* e no componente *Gerente*. Dentre elas, pretende-se:

- Estudar e implementar novas políticas de escalonamento para tarefas maleáveis;
- Realizar estudos comparativos entre a política de escalonamento apresentada com as demais apresentadas nos trabalhos relacionados (ver seção 2.2;
- Realizar estudos comparativos entre outras configurações de hierarquia e estimativas de tempo alvo;
- Implementar a remoção de gerentes e trabalhadores a grade;
- Implementar outro mecanismo de lançamento de nós;
- Gerenciamento de inserção e remoção de outros recursos computacionais. Por exemplo, unidades de processamento gráfico;
- Validar o *TUXUR* em outros tipos de ambientes e com outras aplicações reais.

REFERÊNCIAS

BAKER, M.; BUYYA, R.; LAFORENZA, D. Grids and grid technologies for wide-area distributed computing. **Softw. Pract. Exper.**, New York, NY, USA, v.32, n.15, p.1437–1466, Dec. 2002.

BUYYA, R.; ABRAMSON, D.; GIDDY, J. A Case for Economy Grid Architecture for Service Oriented Grid Computing. In: HETEROGENEOUS COMPUTING WORKSHOP; HCW 2001 (WORKSHOP 1) - VOLUME 2, 10., 2001, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2001. p.20083.1–. (IPDPS '01).

CIRNE, W.; BERMAN, F. Using moldability to improve the performance of supercomputer jobs. **J. Parallel Distrib. Comput.**, Orlando, FL, USA, v.62, n.10, p.1571–1601, Oct. 2002.

CIRNE, W.; BRASILEIRO, F.; ANDRADE, N.; COSTA, L.; ANDRADE, A.; NOVAES, R.; MOWBRAY, M. Labs of the World, Unite!!! **Journal of Grid Computing**, [S.l.], v.4, p.225–246, 2006.

COMPUTATION, P. **Project Pioneer**. Disponível em `http://www.parabon.com`. Acessado em Janeiro de 2014.

DONG, F.; AKL, S. G. **Technical Report No. 2006-504 Scheduling Algorithms for Grid Computing**: state of the art and open problems. 2006.

DUTOT, P.-F.; MOUNI, G.; TRYSTRAM, D. Scheduling Parallel Tasks: approximation algorithms. **Handbook of Scheduling: Algorithms, Models, and Performance Analysis**, [S.l.], 2003. Disponível em: http://hal.archives-ouvertes.fr/hal-00003126/en/. Acessado em Janeiro de 2014.

DUTOT, P.-F.; TRYSTRAM, D. Scheduling on hierarchical clusters using malleable tasks. In: ACM SYMPOSIUM ON PARALLEL ALGORITHMS AND ARCHITECTURES, 2001, New York, NY, USA. **Proceedings...** ACM, 2001. p.199–208. (SPAA '01).

DUTOT, P.-F.; TRYSTRAM, D. A BEST-COMPROMISE BICRITERIA SCHEDULING ALGORITHM FOR PARALLEL TASKS. **Proceedings of WEA'05 (4th International Workshop on Efficient and Experimental Algorithms)**, [S.l.], 2005. Disponível em: hal.archives-ouvertes.fr/docs/00/05/74/16/PDF/dt_wea05.pdf. Acessado em Janeiro de 2014.

FEITELSON, D. G.; RUDOLPH, L. Towards Convergence in Job Schedulers for Parallel Supercomputers. In: WORKSHOP ON JOB SCHEDULING STRATEGIES FOR PARALLEL PROCESSING, 1996, London, UK, UK. **Proceedings...** Springer-Verlag, 1996. p.1–26. (IPPS '96).

FIGHTAIDSATHOME. **Project FightAIDSAtHome**. Disponível em `http://fightaidsathome.scripps.edu`. Acessado em Janeiro de 2014.

FOSTER, I.; KESSELMAN, C. (Ed.). **The Grid 2**: blueprint for a new computing infrastructure. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.

FOSTER, I.; KESSELMAN, C.; TUECKE, S. The Anatomy of the Grid: enabling scalable virtual organizations. **Int. J. High Perform. Comput. Appl.**, Thousand Oaks, CA, USA, v.15, n.3, p.200–222, Aug. 2001.

GRIDRS. **The GridRS Platform**. Disponível em `http://gridrs.lad.pucrs.br`. Acessado em Janeiro de 2014.

JACOB, B.; BROWN, M.; FUKUI, K. Introduction to Grid Computing. **IBM Redbooks**, [S.l.], 2005. Disponível em: http://ibm.com/redbooks. Acessado em Janeiro de 2014.

KALE, L. V.; KUMAR, S.; DESOUZA, J. A Malleable-Job System for Timeshared Parallel Machines. In: IEEE/ACM INTERNATIONAL SYMPOSIUM ON CLUSTER COMPUTING AND THE GRID, 2., 2002, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2002. p.230–230. (CCGRID '02).

KLEIN, C.; PÉREZ, C. **Scheduling Rigid, Evolving Applications on Homogeneous Resources**. [S.l.]: INRIA, 2010. Research Report. (RR-7205).

KRAUTER, K.; BUYYA, R.; MAHESWARAN, M. A Taxonomy and Survey of Grid Resource Management Systems for Distributed Computing. **Softw. Pract. Exper.**, New York, NY, USA, v.32, n.2, p.135–164, Feb. 2002.

LINCK, G. **Um Componente para exploração da capacidade de processamento de GPUS em grades computacionais**. Disponível em `http://cascavel.cpd.ufsm.br/tede/tde_busca/arquivo.php?codArquivo=3521`. Acessado em Maio de 2013.

MAHESWARAN, M.; ALI, S.; SIEGEL, H. J.; HENSGEN, D.; FREUND, R. F. Dynamic Matching and Scheduling of a Class of Independent Tasks Onto Heterogeneous Computing Systems. In: EIGHTH HETEROGENEOUS COMPUTING WORKSHOP, 1999, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 1999. p.30–. (HCW '99).

MOHAMED, H.; EPEMA, D. KOALA: a co-allocating grid scheduler. **Concurr. Comput. : Pract. Exper.**, Chichester, UK, v.20, n.16, p.1851–1876, Nov. 2008.

MOUNIE, G.; RAPINE, C.; TRYSTRAM, D. Efficient approximation algorithms for scheduling malleable tasks. In: ACM SYMPOSIUM ON PARALLEL ALGORITHMS AND ARCHITECTURES, 1999, New York, NY, USA. **Proceedings...** ACM, 1999. p.23–32. (SPAA '99).

MPI. **Extensions to the Message-passing Interface**. Disponível em `http://www.mpich.org`. Acessado em Janeiro de 2014.

OPENMP. **OpenMP Application Program Interface**. Disponível em `http://www.openmp.org`. Acessado em Janeiro de 2014.

REIS, V. Q.; SANTANA, M. J.; MELLO, R. R. F.; SANTANA, R. H. C. Uma Política de Escalonamento Orientada à Redução do Tempo de Resposta de Aplicações Paralelas. **WSCAD**, [S.l.], 2009.

SETIATHOME. **Project Search for Extraterrestrial Intelligence**. Disponível em `http://setiathome.berkeley.edu`. Acessado em Janeiro de 2014.

SILVA, D. P. D.; CIRNE, W.; BRASILEIRO, F. V.; GRANDE, C. Trading Cycles for Information: using replication to schedule bag-of-tasks applications on computational

grids. In: APPLICATIONS ON COMPUTATIONAL GRIDS, IN PROC OF EURO-PAR 2003, 2003. **Anais...** [S.l.: s.n.], 2003. p.169–180.

SILVA, F. A. B. da; SENGER, H. Improving Scalability of Bag-of-Tasks Applications Running on Master-slave Platforms. **Parallel Comput.**, Amsterdam, The Netherlands, The Netherlands, v.35, n.2, p.57–71, Feb. 2009.

SONMEZ, O.; GRUNDEKEN, B.; MOHAMED, H.; IOSUP, A.; EPEMA, D. Scheduling Strategies for Cycle Scavenging in Multicluster Grid Systems. In: IEEE/ACM INTERNATIONAL SYMPOSIUM ON CLUSTER COMPUTING AND THE GRID, 2009., 2009, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2009. p.12–19. (CCGRID '09).

SUDARSAN, R.; RIBBENS, C. J. ReSHAPE: a framework for dynamic resizing and scheduling of homogeneous applications in a parallel environment. **2012 41st International Conference on Parallel Processing**, Los Alamitos, CA, USA, v.0, p.44, 2007.

SUDARSAN, R.; RIBBENS, C. J. Design and performance of a scheduling framework for resizable parallel applications. **Parallel Comput.**, Amsterdam, The Netherlands, The Netherlands, v.36, n.1, p.48–64, Jan. 2010.

TANENBAUM, A. S. **Sistemas Distribuídos**: princípios e paradigmas. 2rd.ed. [S.l.]: Pearson Education do Brasil, 2007.

TANENBAUM, A. S. **Sistemas Operacionais Modernos**. 3rd.ed. [S.l.]: Pearson Education do Brasil, 2010.

TOEBE, J.; PAVAN, W.; FERNANDES, J. M. C. **Um Modelo Baseado em Agentes para o Ciclo de Vida de Insetos**: aplicação na interação afídeo-planta-vírus. Tese apresentada ao Programa de Pós-Graduação em Agronomia da Faculdade de Agronomia e Medicina Veterinária da UPF, para obtenção do título de Doutor em Agronomia. Área de concentração em Fitopatologia. Disponibilizada digitalmente pelo autor.

APÊNDICE A MÉTODOS IMPLEMENTADOS

Este apêndice faz a descrição das funções pré-definidas implementadas neste trabalho. As seguintes subseções abordam as funções implementadas pela tarefa.

A.1 Inicialização

Para que uma tarefa possa ser inicializada é necessário instanciar o objeto CampoTrigo. A figura A.1 ilustra o método construtor da classe CampoTrigo.

```
56      public CampoTrigo() {
57          try {
58              this.dadosDoTempoDeExec.Initialize(new File(TEMPOS_DESC));
59              this.carregaDadosCalculos();
60              this.setWorkload(this.workloadSum());
61              this.result = new ArrayList<Resultado>();
62              this.subJobCount = 0;
63              this.idParentJob = null;
64          } catch (Exception e) {
65              e.printStackTrace();
66          }
67      }
```

Figura A.1: Código do método para inicializar uma tarefa.

Os códigos apresentados nas linhas 58 e 60 representam a quantidade de trabalho da tarefa descrito na seção 4.1.1.1. A linha 59 carrega os dados das estações climatológicas, já o atributo *subJobCount* define que ela não possui nenhuma subtarefa e *idParentJob* define a inexistência da tarefa "pai".

A.2 Descrição

O código representado na figura A.2 e A.3 retorna a descrição de uma tarefa através de um vetor de *bytes*.

```
122     @Override
123     public byte[] getDescription() {
124         byte[] bytes = null;
125         try {
126             bytes = serialize(this.listDadosCalculos);
127         } catch (Exception ex) {
128             ex.printStackTrace();
129         }
130         return bytes;
131     }
```

Figura A.2: Código do método para obter a descrição de uma tarefa.

```
2168    public byte[] serialize(Object obj) throws IOException {
2169        ByteArrayOutputStream out = new ByteArrayOutputStream();
2170        ObjectOutputStream os = new ObjectOutputStream(out);
2171        os.writeObject(obj);
2172        return out.toByteArray();
2173    }
```

Figura A.3: Código do método para converter um vetor de objetos em bytes.

A.3 Resolução

Para formar a resposta final da tarefa é necessário implementar a rotina responsável por definir os resultados. A figura A.4 ilustra o método responsável por definir estes resultados.

```
249     @Override
250     public void setSubResult(byte[] subResultDescription) {
251         List<Resultado> resultTemp = null;
252         try {
253             resultTemp = (List<Resultado>) deserialize(subResultDescription);
254             this.result.addAll(resultTemp);
255         } catch (Exception ex) {
256             ex.printStackTrace();
257         }
258     }
```

Figura A.4: Código do método para definir o resultado.

Este método recebe como argumento um vetor de *bytes* contendo a descrição do resultado de uma tarefa, após este vetor é convertido para o tipo de dado correspondente, ver figura A.5, ao atributo resultado e então o resultado desta conversão é incrementado a lista *result* inicializada no método construtor da classe. O função que fornece a descrição do resultado de uma tarefa é implementada pelo método ilustrado na figura A.6.

```
2175    public Object deserialize(byte[] data) throws IOException,
2176                                          ClassNotFoundException {
2177        ByteArrayInputStream in = new ByteArrayInputStream(data);
2178        ObjectInputStream is = new ObjectInputStream(in);
2179        return is.readObject();
2180    }
```

Figura A.5: Código do método para converter um vetor de bytes em um objeto.

```
133     @Override
134     public byte[] getResultDescription() {
135         byte[] bytes = null;
136         try {
137             bytes = serialize(this.result);
138         } catch (Exception ex) {
139             ex.printStackTrace();
140         }
141         return bytes;
142     }
```

Figura A.6: Código do método para retornar a descrição do resultado.

Como retorno este método fornece a descrição, através de um vetor de *bytes*, do resultado de uma tarefa. Para que isto aconteça é necessário converter um vetor, ver figura A.3, de tamanho igual ao tipo de dado que corresponde o atributo *result*.

A.4 Solução

O método que implementa a rotina que define se uma tarefa está resolvida ou não é ilustrado na figura A.7.

```
171     @Override
172     public boolean isSolved() {
173         if (subJobCount == 0) {
174             return true;
175         }
176         return false;
177     }
```

Figura A.7: Código do método para retornar o estado de solução de uma tarefa.

Caso uma tarefa encontra-se resolvida é retornado o valor *true* caso contrário o retorno é *false*.

APÊNDICE B AVALIAÇÃO DE TESTE

Este capítulo aborda assuntos relacionados a avaliação de teste do componente de gerenciamento do *TUXUR*. Inicialmente apresentamos a metodologia utilizada. A discussão sobre os resultados obtidos finaliza este capítulo.

B.1 Metodologia

Para realizar a avaliação de teste optamos por um tipo de tarefa simples e de fácil divisão em subtarefas e de fácil cálculo de complexidade. Para realizar os testes planejamos três cenários. Com isso, objetivamos verificar o funcionamento e o comportamento em diferentes configurações de ambientes. O primeiro cenário envolve a escolha, pelo gerente, de um tempo alvo fixo para a computação de uma tarefa. Já no segundo há utilização do tempo alvo variado, onde além do tempo alvo fixo o gerente define limites para os tempos de execuções. Por fim, no terceiro cenário avaliamos a adaptabilidade do ambiente, incluindo novos nodos e alterando a capacidade dos nós.

B.2 Aplicação utilizada

A tarefa escolhida para a avaliação envolvendo os cenários é representada pelo cálculo do o n-ésimo número da sequência de *fibonacci* implementada de forma recursiva. A sequência de *fibonacci* é uma sequência numérica onde o primeiro número é 0, o segundo é 1 e cada subsequente é igual a soma dos dois números anteriores. Na figura B.1 apresentamos o código utilizado para realizar o cálculo do *fibonacci*.

```
public long fibo(int n) {
  if (n <= 1) {
    return n;
  } else {
    return fibo(n - 1) + fibo(n - 2);
  }
}
```

Figura B.1: Código do método para realizar o cálculo de *fibonacci*.

A tarefa a ser implementada consiste em calcular a sequência de *fibonacci* do número 65 (fibo(65)). Inicialmente faz-se necessário fornecer o número que representa a quantidade de trabalho da tarefa. Este número é uma previsão dada pela própria tarefa e neste trabalho representado através da fórmula B.1.

$$w = 1,5^n \tag{B.1}$$

Onde *w* representa a quantidade de trabalho da tarefa e *n* é o próprio número a ser calculado. Esta fórmula é aplicada a cada inicialização de uma tarefa. Já o 1,5 é a evolução do tempo obtido na execução sequencial.

Para realizar a divisão de uma tarefa em subtarefas é necessário implementar o método que realiza esta subdivisão. Conforme descrito no início desta seção, a sequência numérica de *fibonacci* de um dado número (números subsequentes de 1) é a soma de seus dois antecessores. Sendo assim, neste trabalho uma tarefa origina outras duas tarefas a partir destes dois números. O código para realizar a divisão de uma tarefa pode ser visualizado na figura B.2.

```
public List<IJob> jobBreak() {
    try {
        List<IJob> jobs = new ArrayList<IJob>();

        IJob j1 = new Fibo(numero - 1);
        j1.setID(getID()+"A");
        j1.setParentJob(getID());
        jobs.add(j1);

        IJob j2 = new Fibo(numero - 2);
        j2.setID(getID()+"B");
        j2.setParentJob(getID());
        jobs.add(j2);

        this.result = 0;
        this.subJobCount = 2;
    return jobs;
    } catch (Exception e) {
        e.printStackTrace();
    }
    return null;
}
```

Figura B.2: Código do método para realizar a divisão de tarefas.

O método *jobBreak* retorna uma lista de tarefas contendo os dois números que antecedem o valor que está sendo calculado. Além disto, para cada nova tarefa criada o identificador da tarefa principal é setado. Os resultados obtidos, conforme a metodologia planejada e em seus respectivos cenários, são apresentados na seção a seguir.

B.3 Resultados obtidos

O teste foi realizado sobre um conjunto de máquinas configuradas como dois pequenos *clusters*, em dois prédios da UFSM. Para isso, foram empregados ao total oito computadores, interligados por uma rede de 1Gbps, cujas características de hardware são apresentadas na tabela B.1.

ID - Localização	CPU	Memória	Sistema Operacional
Node1 - cluster1	Virtualizada em KVM 8 CPUs	4GB	ArchLinux
Node2 - cluster1	Virtualizada em KVM 8 CPUs	4GB	ArchLinux
Node3 - cluster1	Virtualizada em KVM 8 CPUs	4GB	ArchLinux
Node4-8 - cluster2	AMD Phenom(tm) II X2 2.9Ghz	3GB	ArchLinux

Tabela B.1: Descrição do *hardware* dos computadores utilizados na avaliação.

Os computadores *Node1*, *Node2* e *Node3* são máquinas virtualizadas executadas em um computador *SGI AltixXE210 Xeon E5335 quad core de 2GHz*.

Com o objetivo de utilizar todo o processamento disponibilizado pelo hardware, organizamos uma estrutura hierárquica com três gerentes, o *Gerente1* e *Gerente2* gerenciam dezesseis trabalhadores e oito o Gerente3, totalizando quarenta trabalhadores (24 cores no

cluster1 e 16 cores no *cluster2*). Os gerentes localizam-se, respectivamente, nos Node1, Node2 e Node3. A figura B.3 ilustra a organização hierárquica com três gerentes.

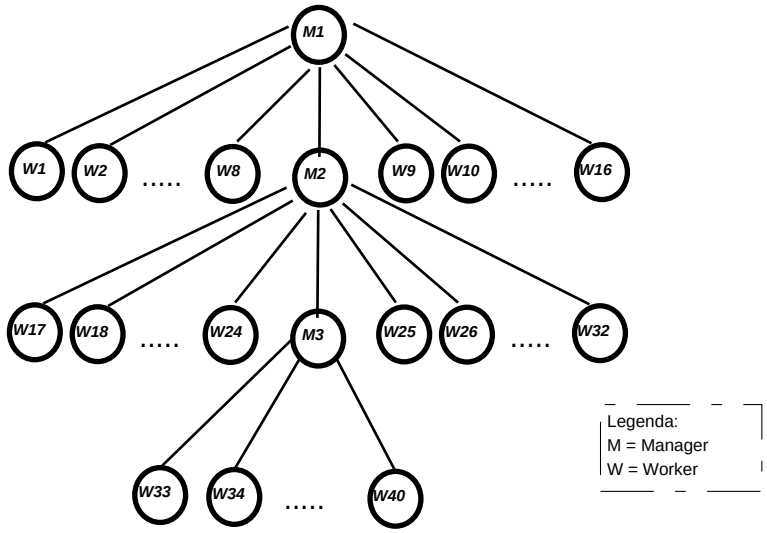

Figura B.3: Organização hierárquica dos gerentes.

Antes de aplicar a metodologia apresentada, realizamos quatro execuções sequenciais do programa em um nó do *cluster1* e do *cluster2*. A tabela B.2 apresenta o tempo final de cada execução e o número de *fibonacci* calculado.

Cluster	Número calculado	Execução 01	Execução 02	Execução 03	Execução 04
Cluster1	65	207260	207372	207252	207249
Cluster2	65	129926	129961	130516	129953

Tabela B.2: Tempos das execuções sequenciais em um nó de cada *cluster*.

Podemos observar que as execuções realizadas no *cluster2* obtiveram melhor desempenho que no *cluster1*. Sendo assim, iremos utilizar o maior tempo (130516) para comparar com os demais tempos obtidos nos três cenários. Conforme mencionado anteriormente, a avaliação foi realizada em três cenários. Uma descrição sobre eles e os resultados encontrados são discutidos a seguir.

B.3.1 Cenário 1

Para o cenário 1 utilizamos o tempo alvo fixado em 300 segundos. Aproximadamente 10% do tempo total estimado para o cálculo, para que cada nó recebesse aproximadamente 10 tarefas. A figura B.4 ilustra a execução do cálculo do número 65 da sequência de *fibonacci*. Cada linha representa um trabalhador, com cada retângulo correspondendo a uma tarefa executada por esse trabalhador. É mostrado o tempo total da execução, mas,

por simplicidade, somente dois trabalhadores de cada gerente são mostrados, o primeiro e o último a terminarem suas tarefas.

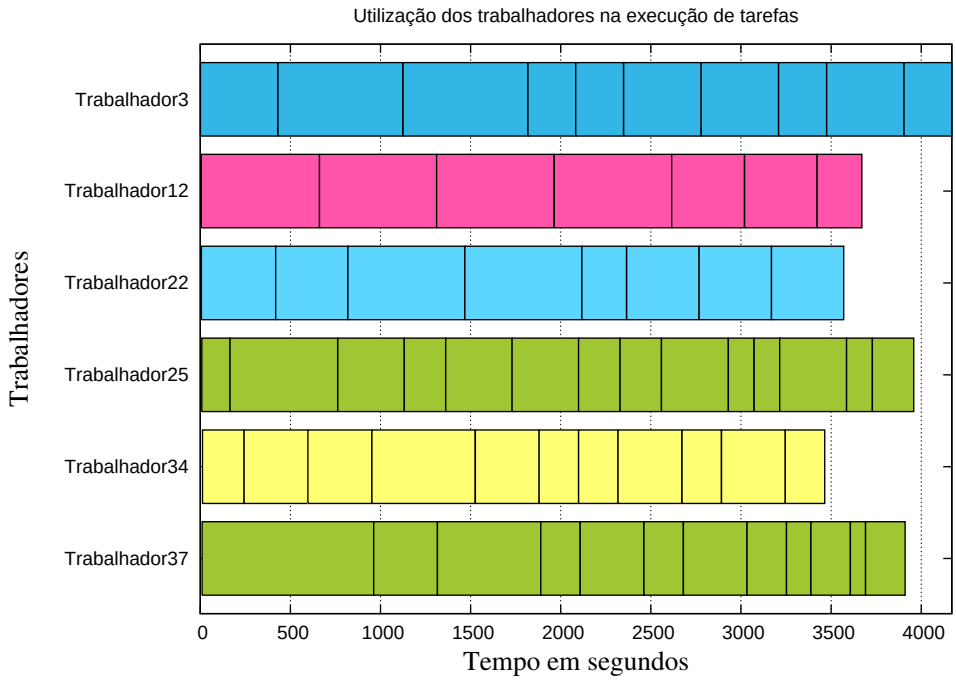

Figura B.4: Utilização dos recursos com tempo alvo variado.

Pode-se ver que alguns trabalhadores não estão sendo utilizados na totalidade durante o período em que estiveram disponíveis para a grade. O tempo total de execução do *Worker3*, por exemplo, foi superior ao tempo total de execução do *Worker34*, nesta situação observamos que não está sendo tirado proveito total da grade, por termos uma má distribuição de tarefas. O tempo final total de execução foi 4175 segundos, redução de 96% em relação ao tempo de execução sequencial.

O problema de má distribuição ocorreu porque quando aconteceu o primeiro caso de um trabalhador ficar sem tarefa e não existirem mais tarefas a distribuir (e consequentemente esse trabalhador teve que parar de trabalhar), ainda existiam outros trabalhadores que tinham tarefas para resolver.

A figura B.5 exibe o gráfico da quantidade de trabalho recebida e a capacidade do *Worker3*. Ressaltamos que neste cenário o tempo alvo é fixo (300).

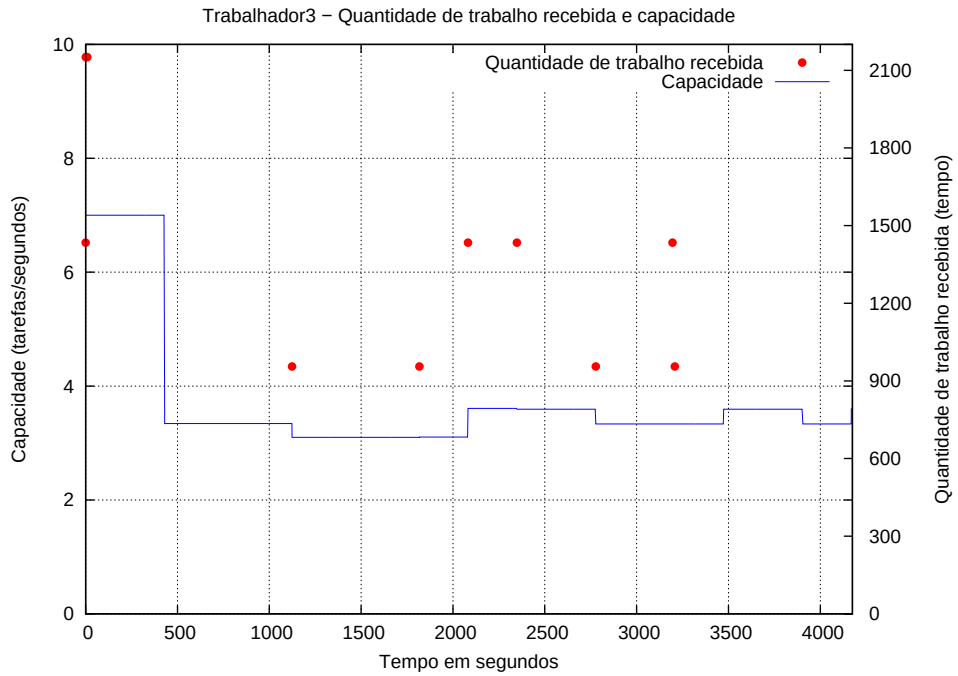

Figura B.5: *Worker3* - Quantidade de trabalho recebida e capacidade.

Baseando-se na regra de distribuição de tarefas as primeiras tarefas recebidas pelo trabalhador ficaram de acordo com os limites estipulados, ou seja, quando iniciado o *Worker3* apresentava capacidade igual a 7 e 300 de tempo alvo, como resultado a quantidade de trabalho recebida foi igual a 1434. Ressaltamos que a capacidade é recalculada conforme descrito na seção 3.3.2.2. A figura B.6 ilustra os valores apresentados pelo *Worker12*.

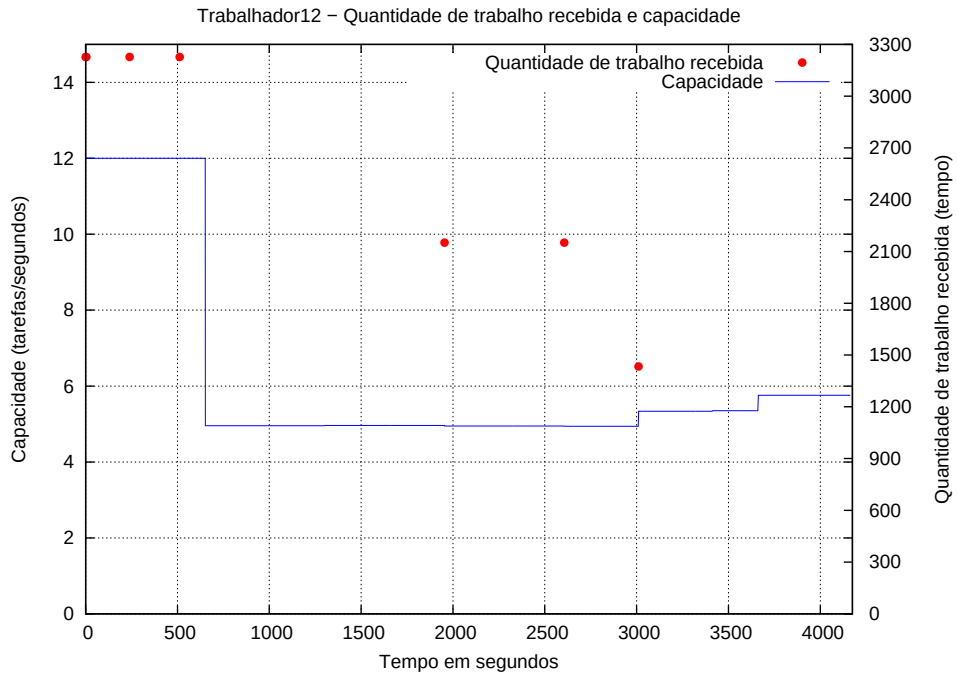

Figura B.6: *Worker12* - Quantidade de trabalho recebida e capacidade.

Já o *Worker12* apresentou no início da sua execução o recebimento de três tarefas de mesmo tamanho, justifica-se pelo fato da sua capacidade, no caso 12, não sofreu nenhuma alteração até aproximadamente o tempo 500. Após isto, se observa que houve diminuição na capacidade, consequentemente, a quantidade de trabalho recebida também foi inferior ao início da execução. Nas figuras B.7 e B.8 apresentamos, respectivamente, os resultados obtidos pelos trabalhadores *Worker25* e *Worker34*.

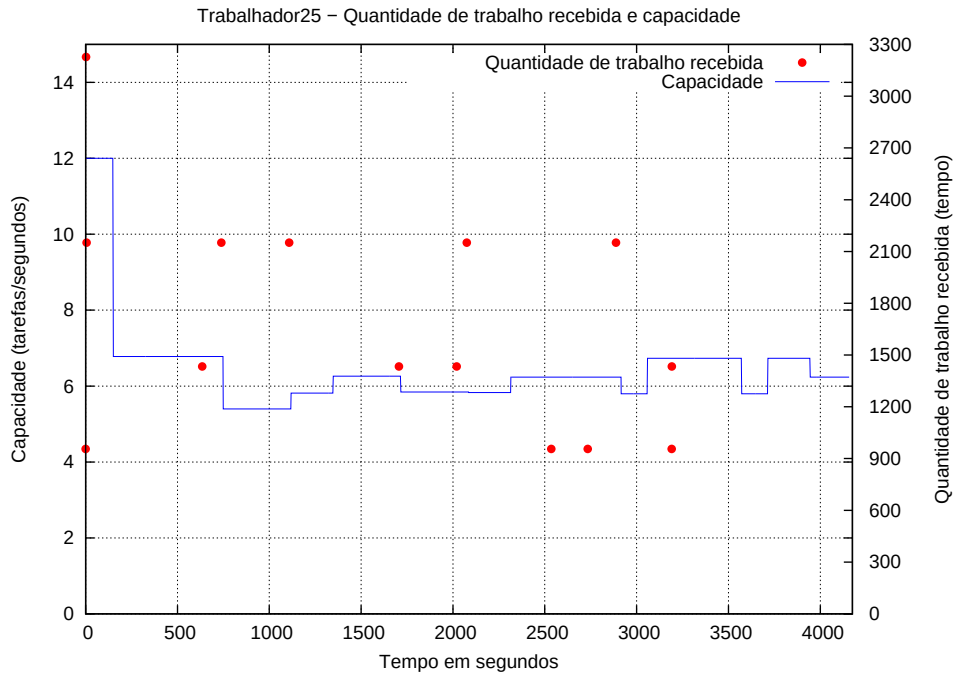

Figura B.7: *Worker25* - Quantidade de trabalho recebida e capacidade.

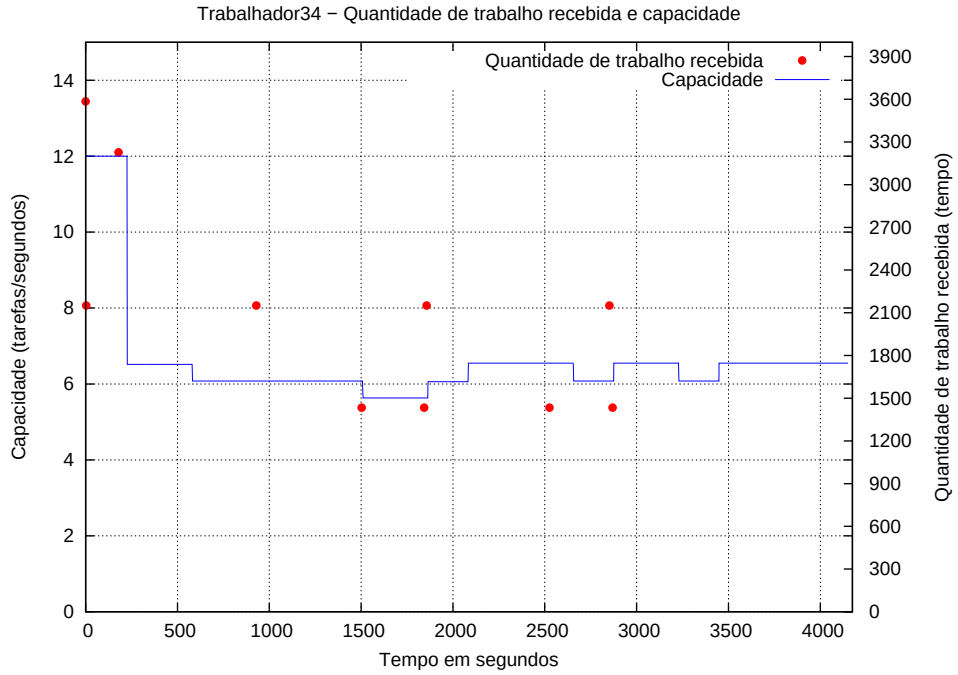

Figura B.8: *Worker34* - Quantidade de trabalho recebida e capacidade.

A capacidade inicial de ambos trabalhadores é igual a 12, como resultado, a quantidade de trabalho recebida concentra-se entre 1800 e 5400. Verifica-se que ambos trabalhadores receberam um conjunto de tarefas a fim de atender as regras estipuladas pelo *framework*. A quantidade de trabalho inicial recebida pelo *Worker25* é igual a 4183 e

3585 pelo *Worker34*.

B.3.2 Cenário 2

A diferença deste cenário para o cenário 1 é a utilização do tempo alvo variado. Definindo um tempo alvo fixo temos o problema de má distribuição das tarefas levando alguns trabalhadores a ficarem ociosos no final da execução, conforme ilustrado na figura B.4.

Um forma de resolver o problema de má distribuição de tarefas seria o trabalhador que ficou sem o que fazer escolher, aleatoriamente, um outro trabalhador e roubar trabalho dele. Outra forma é dar menos trabalho para o trabalhador, para que se tenha pelo menos algum trabalho para dar ao trabalhador que ficaria ocioso.

O problema de se distribuir trabalho antes de se ter uma ideia melhor da capacidade total de trabalho é de se ter uma forma de distribuição de trabalho que acabe enviando um trabalho exageradamente grande para um trabalhador quando se tem poucos trabalhadores, diminuindo a capacidade de distribuição do gerente mais tarde. Esse problema pode ser evitado com um limite máximo de tamanho de trabalho a enviar.

Sendo assim, definimos utilizar o tempo alvo variado utilizando limites para esse tempo. Para testarmos o comportamento do sistema com tempo alvo variável, definimos o tempo alvo como sendo $\frac{1}{10}$ do tempo restante estimado até o final da execução, limitado a um valor mínimo e máximo de 5 segundos e 5 minutos, respectivamente. Caso o tempo alvo calculado ultrapasse um destes limites o tempo alvo assume o valor do tempo limite.

A figura B.9 ilustra a execução do cálculo do número 65 da sequência de *Fibonacci*.

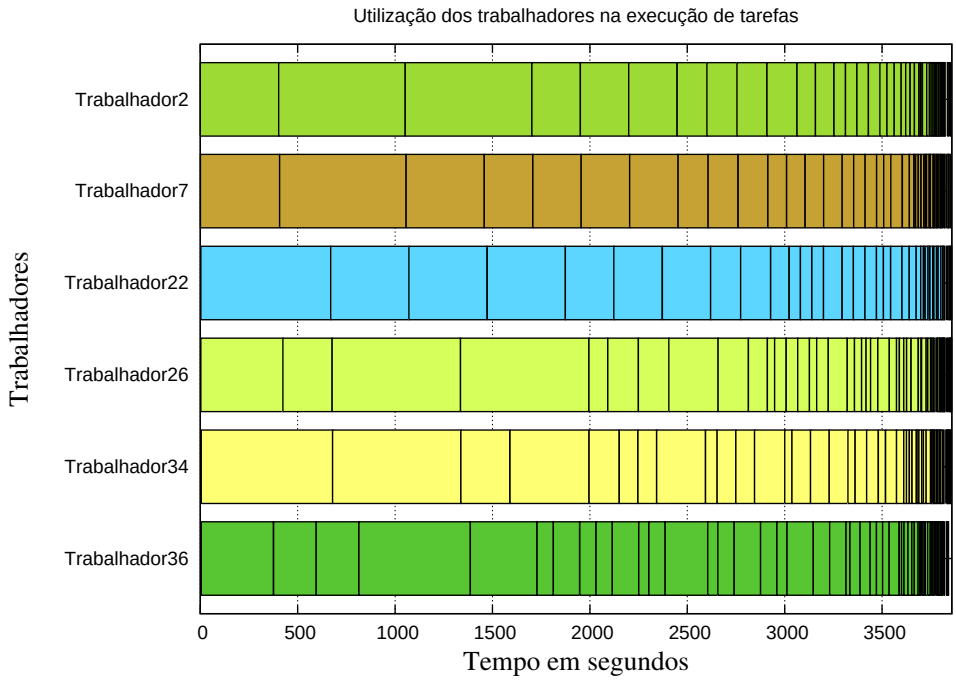

Figura B.9: Utilização dos recursos com tempo alvo variado.

A utilização dos trabalhadores neste método de escalonamento se mostrou mais eficiente do que a do cenário 1. A execução de cada tarefa seguiu as regras conforme definimos, ou seja, cada trabalhador recebeu uma tarefa razoavelmente grande o suficiente para que a grade ficasse equilibrada e conforme a execução evoluiu o tamanho das tarefas foi reduzindo, este tamanho diminuiu pelo motivo de que o tempo total de execução também reduziu. Sendo assim, o tempo total de execução deste cenário foi de 3867 segundos, menor que o tempo apresentado no cenário 1.

As figuras B.10 e B.11 ilustram a quantidade de trabalho recebida e a capacidade referente aos trabalhadores *Worker2* e *Worker34*.

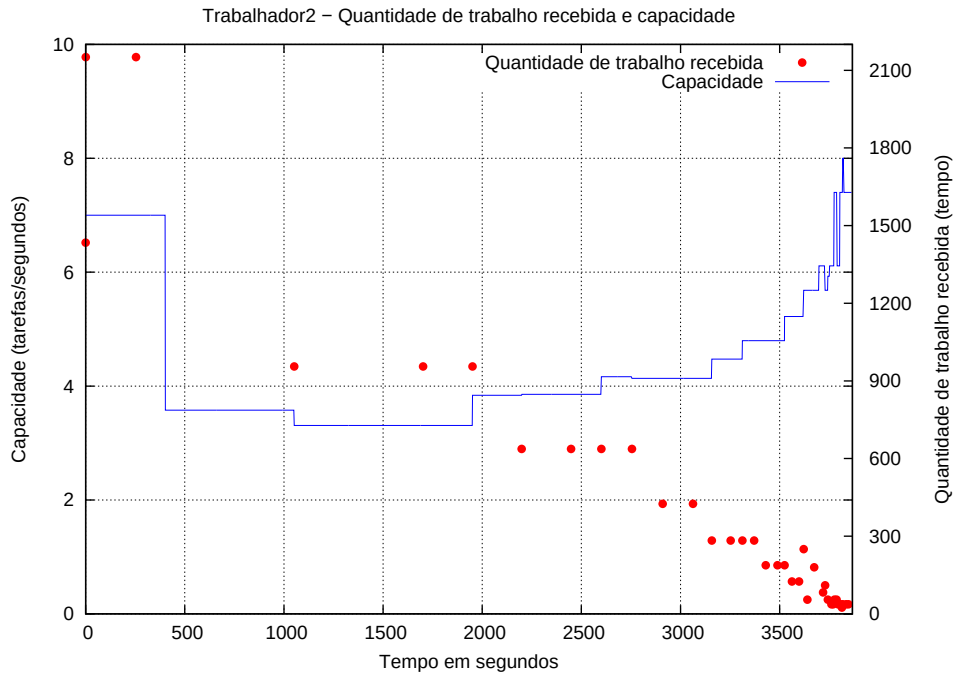

Figura B.10: *Worker2* - Quantidade de trabalho recebida e capacidade.

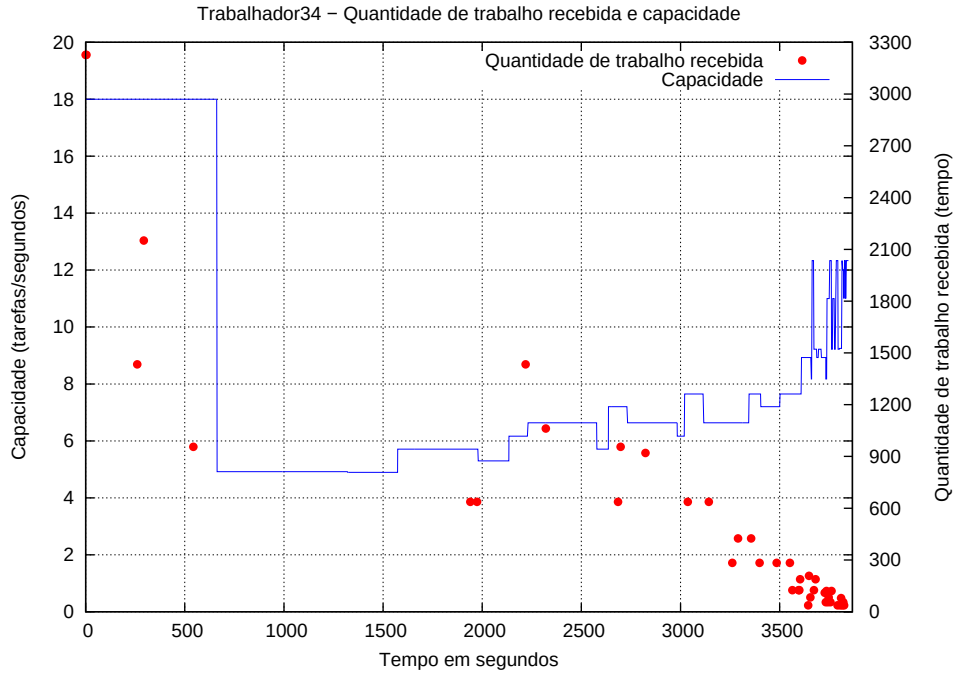

Figura B.11: *Worker34* - Quantidade de trabalho recebida e capacidade.

A quantidade inicial de trabalho recebida pelo *Worker2* é diferente da quantidade recebida pelo *Worker34*. Isso se dá pelo fato de que a capacidade de trabalho é diferente quando ambos trabalhadores iniciam suas execuções. Como a quantidade de trabalho a receber, por cada trabalhador, deve ser proporcional ao tempo alvo e sua capacidade. No

início da execução o *Worker34* com capacidade 18 recebeu tarefa de tamanho igual a 3227 enquanto o *Worker2* recebeu 1437.

Observa-se também que aproximadamente no tempo 600 o *Worker34* teve uma redução significativa em sua capacidade, consequentemente, o mesmo recebe nova tarefa em torno de 1500 segundos depois. Isto se deve ao fato de que até este tempo o trabalhador continha em sua fila trabalhado suficiente. A mesma situação ocorre com o *Worker2*, aproximadamente no tempo 400 também teve uma redução na capacidade e cerca de 800 segundos depois recebe novo trabalho. A evolução do tempo alvo é exibida na figura **??**.

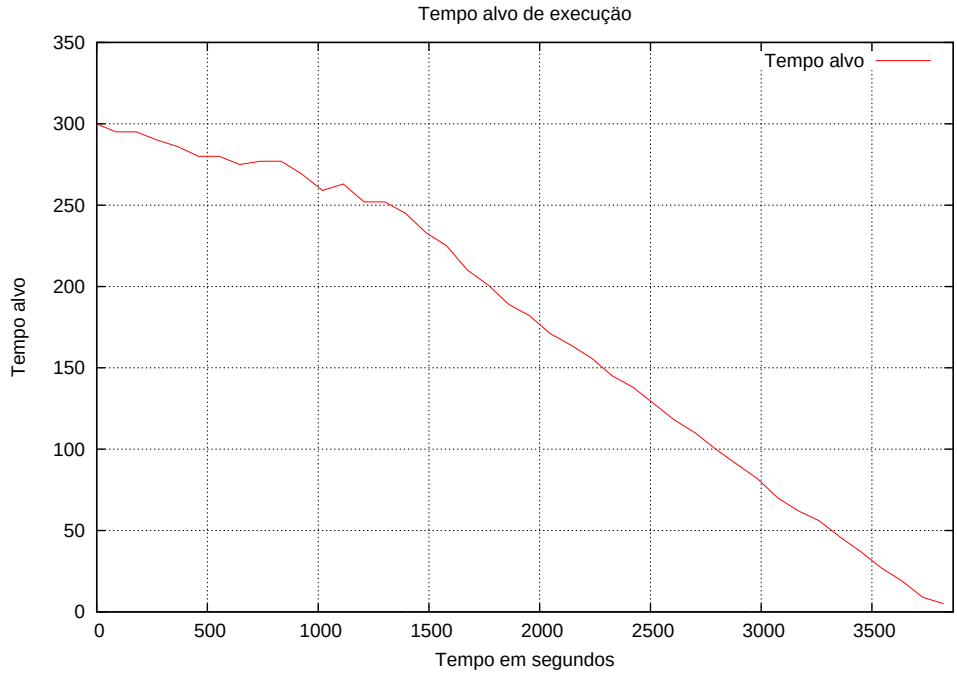

Figura B.12: Evolução do tempo alvo.

A capacidade inicial do *Worker36*, conforme figura B.13, é igual a 18 e o mesmo recebeu trabalho total suficiente que corresponde a esta capacidade, mas, ao concluir a primeira tarefa, aproximadamente no tempo 400, o referido trabalhador não recebeu mais trabalho até aproximadamente o tempo 1300. Ele não recebeu mais trabalhos desde o início da sua execução até este tempo pelo motivo de que a sua capacidade de trabalho era suficiente.

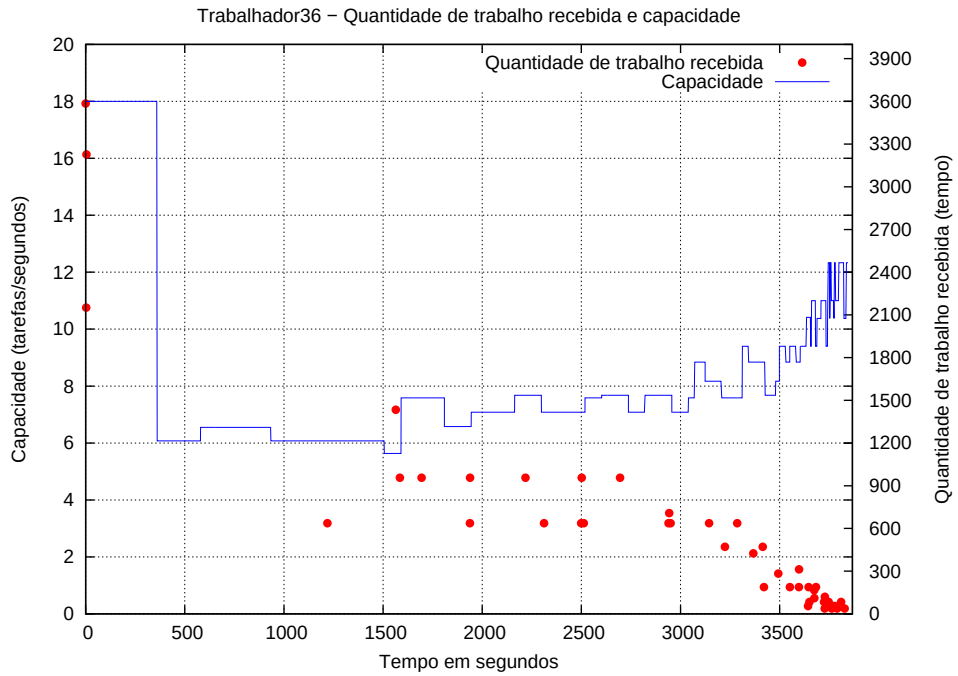

Figura B.13: *Worker36* - Quantidade de trabalho recebida e capacidade.

B.3.3 Cenário 3

O cenário 3 avalia o comportamento do *TUXUR* quando a quantidade ou a capacidade dos nós de trabalho são alterados durante a execução de uma tarefa. O suporte a remoção de nós ainda não está implementado. A tarefa executada e a configuração da grade são as mesmas dos cenários anteriores. Foram realizados dois tipos de testes. No primeiro, foram acrescentado doze novos nós no meio da execução; no segundo, diminuiu-se a capacidade de trabalho de alguns dos nós existentes, sobrecarregando-os com a execução de tarefas externas ao *TUXUR*. Esses dois experimentos são descritos a seguir.

B.3.3.1 Agregação de nós

Neste cenário avaliamos como o ambiente se comporta adicionando mais trabalhadores após um determinado tempo de execução. Definimos este tempo em 15 minutos, sendo assim, utilizando as mesmas definições do cenário 2 e após 15 minutos de execução foram adicionados a grade 16 trabalhadores subordinados ao *Gerente3*. Estes trabalhadores possuem as mesmas configurações de *hardware* utilizadas no *cluster2*. Portanto, conforme a organização hierárquica representada na figura B.3 a estrutura da grade de computadores passou a ser formada por 56 trabalhadores.

Na figura B.14 exibimos a utilização dos recursos da grade desde o momento em que

estes recursos foram disponibilizados até o momento em que eles ainda estavam disponíveis, mas não foram mais utilizados. A figura mostra a execução da tarefa.

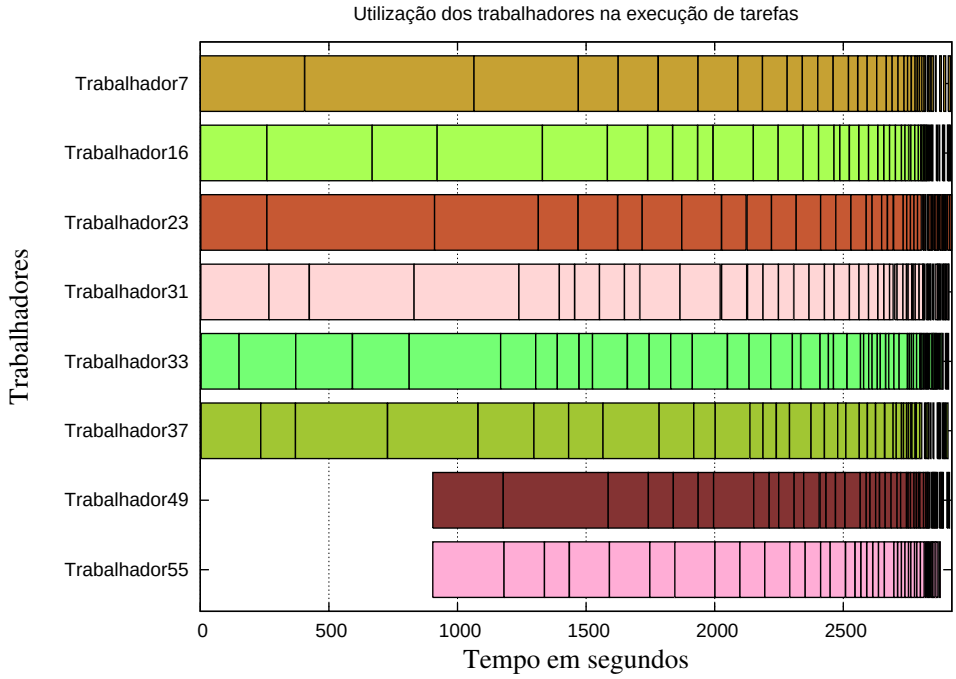

Figura B.14: Utilização dos recursos com tempo alvo variado e agregação de nós.

Conforme definido no início da seção dos resultado referente ao cenário 1 exibimos somente os trabalhadores *Worker49* e *Worker55*. Estes começaram a processar tarefas e utilizar a grade no tempo em que foi estipulado. Comparando o gráfico anterior de utilização da grade com o gráfico da figura B.9 se percebe que apesar de haver uma maior quantidade de subtarefas mais ao final da execução e de haver alguns trabalhadores ociosos por um período de tempo, a execução deste cenário foi eficiente obtendo tempo final de execução igual a 2922 segundos.

Nos gráficos das figuras B.15 e B.16 exibimos, respectivamente, a quantidade de trabalho recebida e a capacidade de processamento do *Gerente2* e *Gerente3* desde o início da execução até o momento em que não existiam mais tarefas para eles distribuírem. Após, exibimos a evolução do tempo alvo no gráfico da figura B.17.

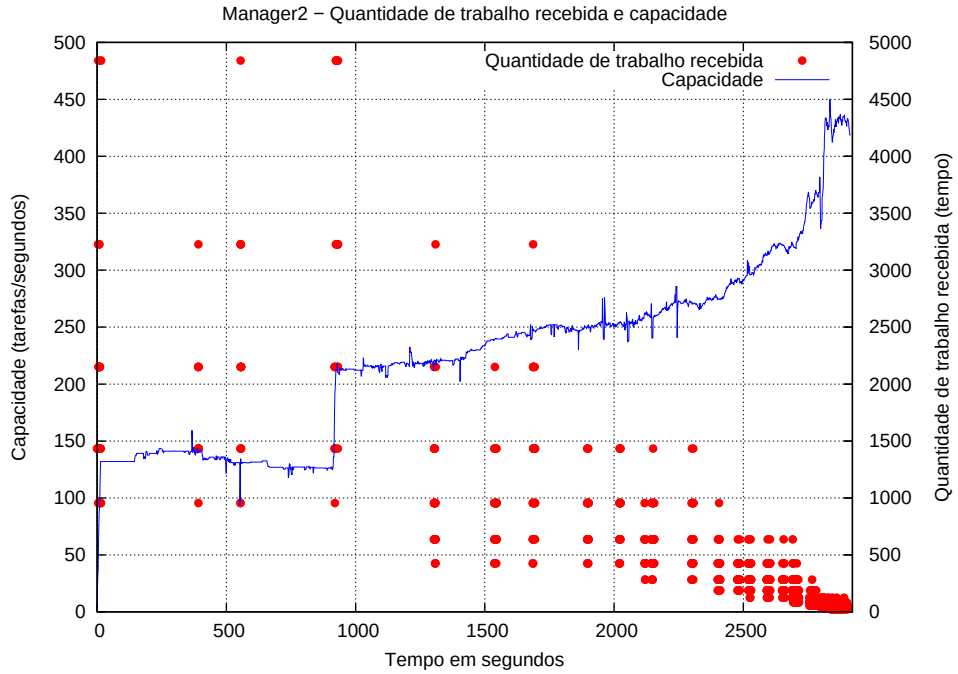

Figura B.15: *Manager2* - Quantidade de trabalho recebida e capacidade.

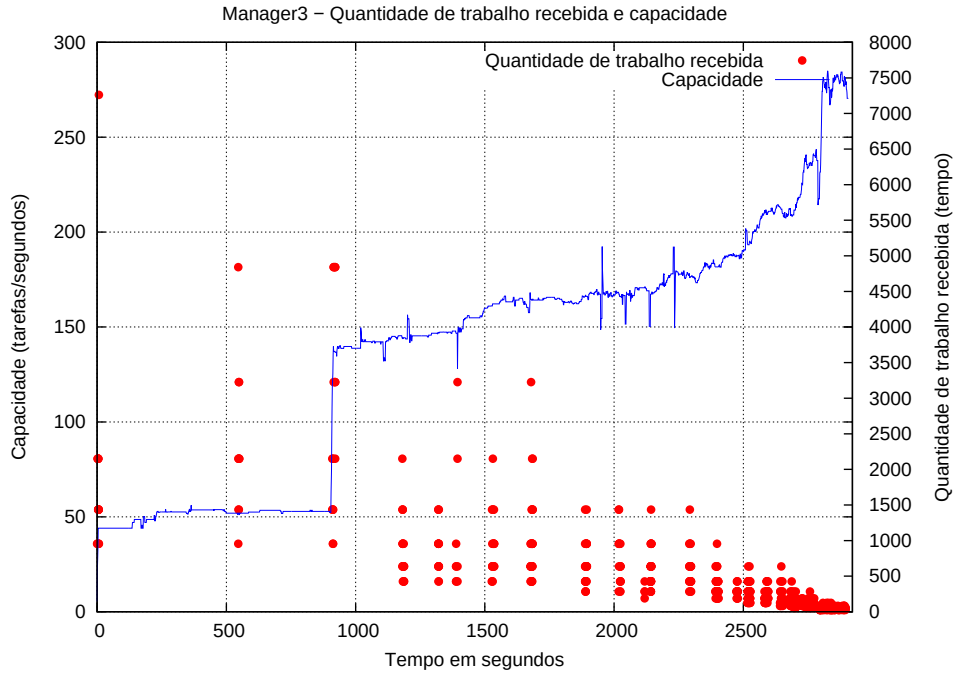

Figura B.16: *Manager3* - Quantidade de trabalho recebida e capacidade.

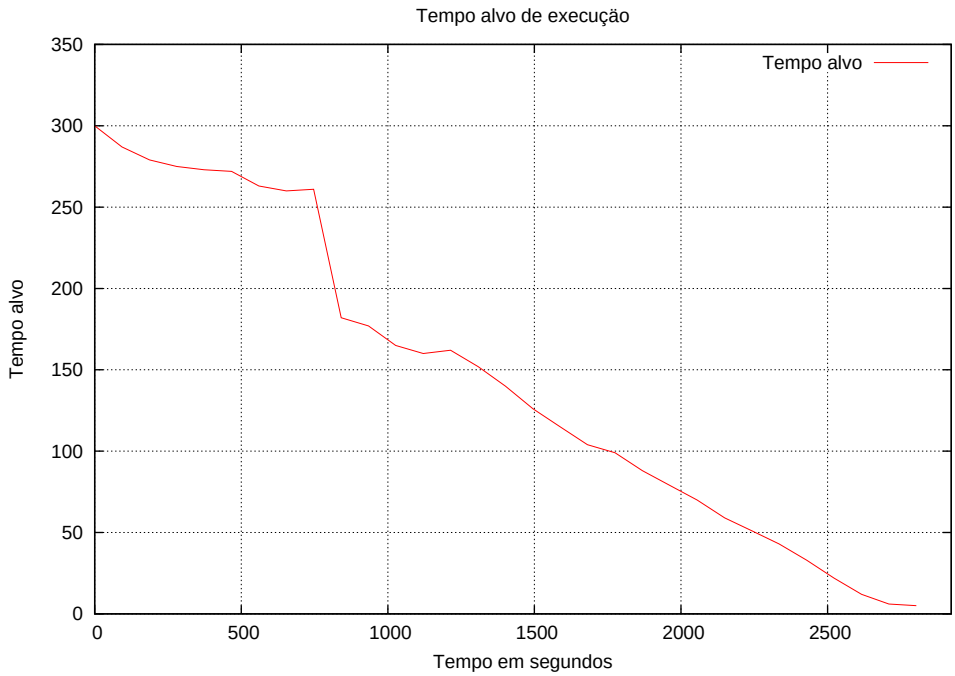

Figura B.17: Evolução do tempo alvo.

A capacidade, de ambos gerentes, aumentou aproximadamente no tempo 900 pelo motivo da inserção dos novos trabalhadores, consequentemente a quantidade de trabalho recebida por eles também aumentou. Além do mais, podemos verificar que a capacidade do *Manager3* está sendo agregada ao seu gerente, ou seja, a capacidade de ambos gerentes aumentou aproximadamente em 100 vezes.

B.3.3.2 Carga extra

A fim de testar a dinamicidade do *TUXUR* foi gerada uma carga extra, externa ao *framework*, em todos os nós do *cluster2*. Esses nós foram carregados por programas externos após 600 segundos de utilização da grade, durante o período de 1800 segundos.

Nos gráficos das figuras B.18 e B.19 exibimos a capacidade e a quantidade de trabalho recebida referente aos trabalhadores *Worker27* e *Worker32*. Após exibimos o gráfico da evolução da tempo alvo de execução.

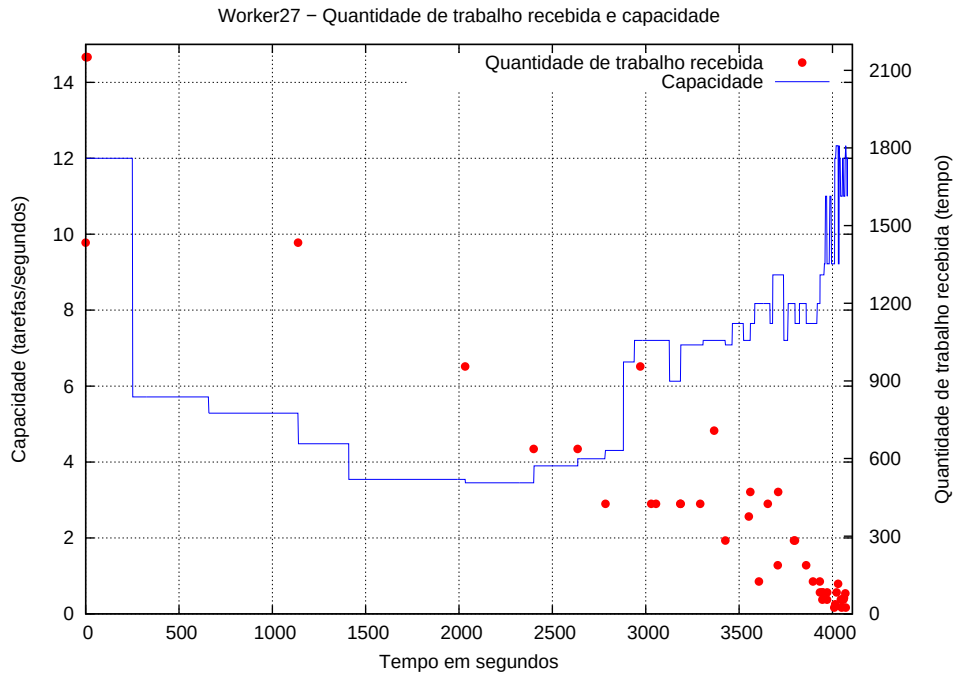

Figura B.18: *Worker27* - Quantidade de trabalho recebida e capacidade.

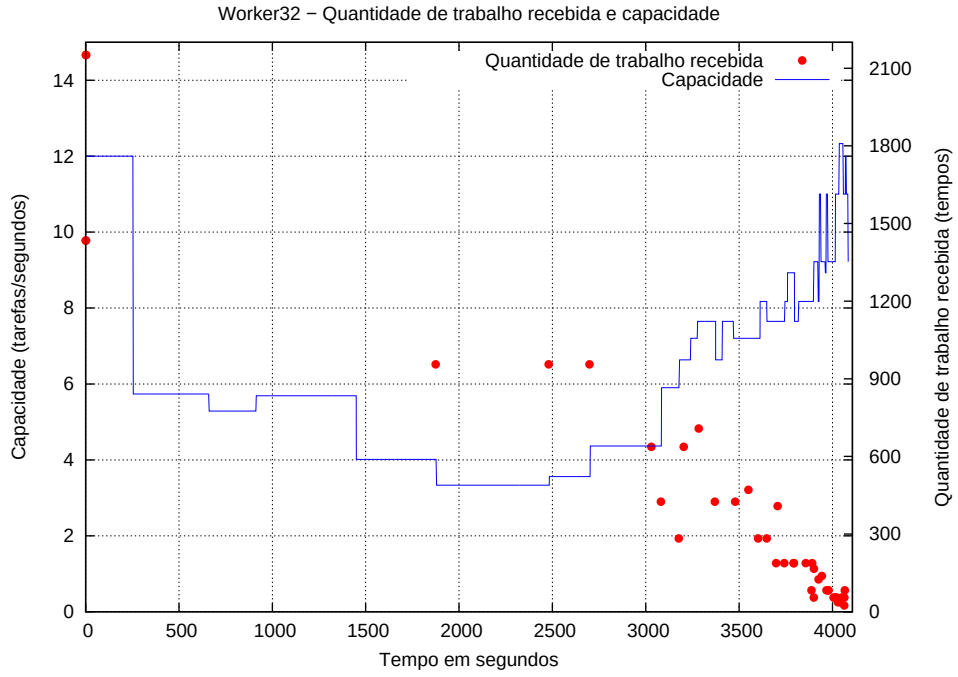

Figura B.19: *Worker32* - Quantidade de trabalho recebida e capacidade.

Constata-se que a capacidade de trabalho dos trabalhadores teve uma diminuição durante o período que originou os testes de carga extra de trabalho (entre 600 e 2700 segundos). Consequentemente, a quantidade de trabalho recebida também teve diminuição.