Fabricio Jorge Barboza de Oliveira

ESCALONAMENTO DINÂMICO, DISTRIBUÍDO E COLABORATIVO DE APLICAÇÕES PARALELAS E AUTÔNOMAS COM PRAZOS

Dissertação apresentada ao programa de Pós-Graduação em Computação da Universidade Federal Fluminense, como requisito parcial para obtenção do Grau de Mestre. Área de Concentração:
Redes e Sistemas Distribuídos e Paralelos.

Orientador: Prof. PhD. Eugene Francis Vinod Rebello

Niterói

2016

FABRICIO JORGE BARBOZA DE OLIVEIRA

ESCALONAMENTO DINÂMICO, DISTRIBUÍDO E COLABORATIVO DE APLICAÇÕES PARALELAS E AUTÔNOMAS COM PRAZOS

Dissertação apresentada ao programa de Pós-Graduação em Computação da Universidade Federal Fluminense, como requisito parcial para obtenção do Grau de Mestre. Área de Concentração:
Redes e Sistemas Distribuídos e Paralelos.

Aprovada em 13 de Julho de 2016.

BANCA EXAMINADORA

Prof. PhD. Eugene Francis Vinod Rebello – Orientador
IC-UFF

Profa. PhD. Maria Cristina Silva Boeres
IC-UFF

Prof. D.Sc. José Viterbo Filho
IC-UFF

Profa. D.Sc. Cristiana Barbosa Bentes
FEN-UERJ

Niterói
2016

"Dedico esse trabalho a mim mesmo. Porque só eu sei quanto me esforcei para concluí-lo."

AGRADECIMENTOS

Agradeço à minha família, Taiana Gomes da Silva Barboza, por entender e aceitar os períodos que precisei afastar-me para a conclusão deste trabalho.
Agradeço aos meus familiares por aceitarem minhas ausências e apoiar-me nesse período.
Agradeço ao meu orientador, pela paciência comigo e por exigir sempre o meu melhor.
Agradeço a todos meus amigos, sem exceção, pelo apoio durante esse período, em especial, Bill, Burlamaqui, Fernanda e Tamaki.

"Ninguém baterá tão forte quanto a vida. Porém, não se trata de quão forte pode-se bater, trata-se de quão forte pode ser atingido e continuar seguindo em frente. É assim que a vitória é conquistada."

Rocky Balboa - Sylvester Stallone.

Resumo

Muitas das aplicações desenvolvidas, especialmente no contexto do *e-Science*, nas diversas áreas de conhecimento como engenharia, medicina e biologia, demandam ambientes de execução de larga escala que possibilitem não só trabalhar com um grande conjunto de dados, mas também conseguir seus resultados em tempos viáveis. *Data centers* são as principais fontes para suprir essa demanda, oferecendo supercomputadores compostos de servidores com cada vez mais recursos de processamento, memória e armazenamento. Porém, dado o alto custo para a aquisição, implantação e manutenção dos *data centers*, seus administradores têm procurado mecanismos para tornar sua utilização mais eficientes. As estratégias mais clássicas concentravam-se em mecanismos que realizavam reserva dos recursos para cada *job* individualmente, tornando o *job* detentor dos servidores alocados até o término da sua execução. O grande problema dessa abordagem é a ineficiência na utilização dos servidores e os relativamente longos tempos de respostas (*turnaround times*).

O escalonamento de tarefas é um dos principais problemas da computação, o qual pesquisadores tentam obter métodos eficientes de escalonar grandes conjuntos de tarefas sobre números significativos de recursos. Escalonamentos tradicionalmente concentram-se em otimizar o tempo total de execução (*makespan*), mas recentemente outros objetivos também têm sido buscados para otimização como, por exemplo: a vazão dos *jobs*; consumo energético; e particularmente no contexto de computação em nuvem, o custo de aluguel dos recursos necessários, a quantidade de atendimentos de SLA (*Service Level Agreements*) e prazos de execução.

Este trabalho propõe uma estratégia alternativa para o escalonamento de um conjunto de aplicações paralelas em recursos compartilhados. Ao invés de tentar alcançar o menor tempo de execução, esse trabalho procura permitir que as próprias aplicações se escalonem para conseguir finalizar suas execuções em seus respectivos prazos de tempo (ou metas) determinados pelos seus usuários. A nova política leva em consideração que ambientes devem compartilhar seus recursos a fim de melhorar a utilização, para que as aplicações paralelas não tentem somente atingir seus objetivos, mas permitir que as outras consigam também. Usando o conceito de aplicações autônomas, foi proposta uma heurística de comportamento que controla a demanda da aplicação para recursos de um servidor em função das demandas inferidas das aplicações concorrentes. Experimentos realizados verificam a possibilidade de criar comportamentos altruístas em aplicações paralelas sem ter qualquer comunicação

explícita. Foi observado durante a execução em vários cenários que a implementação proposta, baseada na sociologia de aplicações paralelas, foi capaz de atender a maioria das metas definidas e apresentou resultados promissores para trabalhos futuros nesta abordagem nova de gerenciamento de recursos.

Palavras-chave: Computação distribuída; Computação autônoma; Aplicações paralelas; Escalonamento dinâmica de tarefas; MPI; EasyGrid AMS.

Abstract

In the context of e-Science, many of the applications being developed in the various areas of expertise such as engineering, medicine and biology, for example, require large-scale execution environments that not only handle large data sets, but also provide the results in acceptable run times. Data centres are currently the favoured answer to meet these demands, offering clsuters composed of growing number of servers each with ever increasing processing, memory and storage capacities. However, given their high cost of acquisition, deployment and maintenance, administrators are constantly seeking efficient management mechanisms to make use the data centres more efficient. Most classic scheduling strategies focused on mechanisms that reserve resources in advance for each job individually, making the job the exclusive user of the allocated servers until the end of its execution. The major problem with this approach is the inefficient use of server capacity and the relatively long turnaround times.

Task scheduling is a key performance issue in computing, for which research has focused intensely on trying to obtain efficient solutions to map large sets of tasks on significant numbers of resources. Algorithms have traditionally focused on optimizing the total execution time (*makespan*), but recently other objectives have also become the focus of optimization, for example, throughput of jobs; energy consumption; and particularly in the context of cloud computing, the rental cost of the required resources and satisfying Service Level Agreements (SLAs) and execution deadlines.

This dissertation proposes an alternative strategy for scheduling a set of parallel applications on a collection of shared computing resources. Instead of trying to achieve the lowest execution time, this work seeks to allow the applications to schedule themselves in order to complete their executions within their respective time limits, defined previously by their users. This new strategy takes into account that environments should share their resources to achieve better utilization, and that parallel applications should try to not only achieve their goals, but allow others to succeed as well. Using the concept of autonomic applications, a heuristic has been proposed that controls the application's demand for the resources of a server in light of its needs and the inferred needs of competing applications. Experimental analysis has verified the possibility of creating altruistic behaviour in parallel applications without any explicit communication between them. The concurrent execution of a number of applications, in various scenarios, has shown that proposed implementation, based on the sociology of parallel applications, is able to permit these applications to meet the

majority of their target execution times. These promising results should motivate future work on further research into this new approach to resource management.

Keywords: Distributed computing; Autonomic computing; Parallel applications; Dynamic task scheduling; MPI; EasyGrid AMS.

Lista de Ilustrações

Figura 2.1 Ilustração de um problema de *multiple objetive optimization*.....6
Figura 3.1 Ilustração de um cluster computacional.....15
Figura 3.2 Ilustração de uma grade computacional.....16
Figura 3.3 Ilustração Cloud Computing 18
Figura 4.1 Modelo 1PProc.....25
Figura 4.2 Modelo 1PTask.....26
Figura 4.3 Exemplo gráfico de um DAG.....27
Figura 4.4 Exemplo gráfico de um BoT 28
Figura 4.5 Estrutura hierarquia do EasyGrid AMS 29
Figura 4.6 Estrutura em camadas do EasyGrid AMS 30
Figura 5.1 Aplicação em um ambiente distribuído com um broker central.....33
Figura 5.2 Aplicação EasyGrid no Ambiente distribuído com um broker central.....34
Figura 5.3 Aplicação no ambiente distribuído com middleware EasyGrid SA.....35
Figura 5.4 Linha tempo estado da aplicação em relação a meta soft 41
Figura 5.5 Linha tempo estado da aplicação e relação da meta hard 41
Figura 5.6 Linha do Tempo aplicação que perdeu meta.....42
Figura 6.1 Tempo por número de jobs aplicação.....51
Figura 6.2 Média do tempo de execução sozinho entre as versões 52
Figura 6.3 Média do tempo de execução de 3 jobs sequenciais 54
Figura 6.4 Média do tempo de execução de 3 jobs concorrente.....54
Figura 6.5 Distribuição dos escalonadores das aplicações nos recursos do ambiente de teste no modelo clássico (com reserva de recursos- isolado) 57
Figura 6.6 Distribuição dos escalonadores das aplicações nos recursos do ambiente de teste no modelo proposto (com compartilhamento de recursos- compartilhado).....57
Figura 6.7 Media dos Tempo de execução com compartilhamento (Compartilhado) e sem compartilhamento (Isolado) 57
Figura 6.8 Média dos Tempo de execução com compartilhamento (Compartilhado) e sem compartilhamento (Isolado) para job com I/O 58
Figura 6.9 Tempo de execução política estática 61
Figura 6.10 Meta x Tempo total de execução - teste 1 61
Figura 6.11 Gráfico utilização CPU x Tempo de um dos Host - teste 2.....63
Figura 6.12 Meta x Tempo total de execução - teste 2 64

Figura 6.13 Meta x Tempo total de execução - teste 3 65
Figura 6.14 Utilização de CPU de um dos traces da execução em um dos hosts..... 66
Figura 6.15 Meta x Tempo total de execução - teste 4 67
Figura 6.16 Meta x Tempo total de execução - utilização recursos - teste 1 69
Figura 6.17 Meta x Tempo total de execução - priorização da execução - teste 2 70
Figura 6.18 Trace de execução utilização CPU x Tempo 71
Figura 6.19 Gráfico do tempo médio de execução 73
Figura 6.20 Trace de execução vazão de jobs por tempo corrente 73
Figura 6.21 Resultado de execução teste estático..... 75
Figura 6.22 Execução política SA 76

Lista de tabelas

Tabela 5.1 Resumo de ações por estado/comportamento 48
Tabela 6.1 Tempos da execução por número jobs 50
Tabela 6.2 Média dos tempos de execução sozinho entre as versões 53
Tabela 6.3 Media de execução de 3 jobs sequenciais 54
Tabela 6.4 Média do tempo de execução de 3 jobs concorrente 55
Tabela 6.5 Tempos de execução das estratégias 57
Tabela 6.6 Resultado execução política estática 62
Tabela 6.7 Meta x Tempo total de execução teste 1 63
Tabela 6.8 Meta x Tempo total de execução - teste 2 64
Tabela 6.9 Meta x Tempo total de execução - teste 3 65
Tabela 6.10 Tempo de termino de cada uma dos jobs do trace analisado 66
Tabela 6.11 Meta x Tempo total de execução - teste 4 67
Tabela 6.12 Resultados de execução - teste 4 68
Tabela 6.13 Meta x Tempo total de execução - teste 1 69
Tabela 6.14 Meta x Tempo total de execução - priorização da execução - teste 2 70
Tabela 6.15 Resultado da segunda execução teste 2 - ocioso 71
Tabela 6.16 Avaliação da adaptação-teste1 72
Tabela 6.17 Relação de APP e tempo de inicio 74
Tabela 6.18 Resultados execução política estática 75
Tabela 6.19 Resultado de execução política dinâmica 76

Lista de abreviaturas e siglas

LAN	:	Rede local de dados.
QoS	:	Qualidade de Serviço
GM	:	Escalonador de Máquina.
GS	:	Escalonador de Site.
GG	:	Escalonador Global
NP	:	Nondeterministic Polynomial
HT	:	Hyper-Threading da Intel
I/O	:	*Input/Output*
CFS	:	*Completely Fair Scheduler*
RMS	:	*Resource Management System*
UMS	:	*User Management System*
AMS	:	*Application Management System*
PER	:	Política de Execução Restrita
PEI	:	Política de Execução Irrestrita
UFF	:	Universidade Federal Fluminense
IC	:	Instituto de Computação

Sumário

Capítulo 1 – Introdução 1

1.1 Objetivo 2

1.2 Contribuições 3

1.3 Organização 3

Capítulo 2 – Trabalhos Relacionados 4

2.1 Artigos Relacionados 5

2.2 Comparações 12

2.3 Resumo 13

Capítulo 3 – Contexto 14

3.1 – Sistemas Distribuidos 14

3.2 – Aplicações Paralelas 20

3.3 – MPI (Message Passing Interface) 20

3.3.1 – Open MPI 22

3.3.2 – LAM/MPI 22

3.4 – Sistemas de Gerenciamento 22

3.5 – Resumo 24

Capítulo 4 – Middleware EasyGrid AMS 25

4.1 – Contexto 25

4.2 – Modelo de Representação das Aplicações 27

4.3 – Estrutura do EasyGrid AMS 28

4.4 – Resumo 31

Capítulo 5 – Politica Sociedade Altruísta 32

5.1 – Visão Geral 32

5.2 – Modelo das Aplicações 35

5.3 – Proposta Sociedade Altruísta – Gerenciamento sem gerente 37

5.3.1 – Estimativa previsão termino 38

5.3.2 – Classificação da aplicação em estados 40
5.3.3 – Ajuste da vazão 42
5.3.3.1 – Decisão baseado no estado 42
5.3.3.2 – Decisão baseada em políticas auxiliares 44
5.4 – Algoritmo Sociedade Altruísta 46
5.5 – Resumo 48
Capítulo 6 – Análise Experimental 49
6.1 – Configurações do Ambiente 49
6.2 – Experimentos 50
6.2.1 – Sobrecarga de compartilhamento de recursos 50
6.2.1.1 – Configuração dos jobs 50
6.2.1.2 – Testes 50
6.2.2 – Sobrecarga da implementação da nova versão do EasyGrid 51
6.2.2.1 – Configuração dos jobs 52
6.2.2.2 – Testes 52
6.2.3 – Execução compartilhada (Avaliando o Overhead) 55
6.2.3.1 – Configuração dos jobs 56
6.2.3.2 – Testes 56
6.2.4 – Avaliação da priorização das tarefas 59
6.2.4.1 – Configuração dos jobs 59
6.2.4.2 – Testes 60
6.2.5 – Priorização sobre cenários com com folgas de tempo 68
6.2.5.1 – Configuração dos jobs 68
6.2.5.2 – Testes 68
6.2.6 – Avaliação da adaptação 71
6.2.6.1 – Configuração dos jobs 72
6.2.6.2 – Teste 72

6.2.7 – Simulação de um cenário HPC 74

6.3 – Resumo 76

Capítulo 7 –Conclusão 77

Referências Bibliográficas 80

Capítulo 1– Introdução

Com a meta de facilitar e agilizar a execução de aplicações nos ambientes distribuídos, faz-se necessário ferramentas que permitam a aplicação tirar o máximo do ambiente o qual se encontra, sem que para isso perca-se o controle das aplicações em execução. Como exemplo, podemos imaginar um conjunto de computadores interligados por uma rede de comunicação na qual usuários resolvam iniciar suas aplicações . Este ambiente compartilhado poderia ficar sobrecarregado, demorando um tempo inesperado para gerar os resultados, uma vez que diversos processos poderiam ganhar acesso simultaneamente aos recursos. Esse cenário nos sugere que uma reserva temporal dos recursos (isto é, por exemplo, a alocação de uma aplicação paralela por vez aos recursos) é a solução mais apropriada. Este controle de acesso, que por sua vez enfileiraria o acesso ao sistema pelos usuários, tende a gerar ociosidade de parte dos recursos no sistema com muita frequência. Dado a grande ênfase em melhorar a eficiência de sistemas, a questão pertinente hoje é: Como evitar esta ociosidade?

Segundo [2], o futuro da execução dos ambientes distribuídos se encontra em um novo tipo de *middleware* – software que realiza a mediação entre as aplicações e os ambientes ao qual estão imersos. Foi proposto um middleware descentralizado onde a gerência dos *jobs* da abordagem centralizada é migrada para um nível mais próximo da aplicação. Essa proposta originalmente focou em sistemas distribuídos de larga escala como as grades computacionais [3], por serem ambientes heterogêneos e geralmente compartilhados entre múltiplas aplicações simultaneamente. Hoje, poderíamos facilmente estender essa proposta para os demais ambientes de alto desempenho existentes, como por exemplo, clusters e nuvens.

Nos ambientes computacionais distribuídos é comum que o acesso ao meio seja gerenciado por alguma ferramenta computacional. Tradicionalmente, estas ferramentas, conhecidas como *job schedulers*, utilizam uma abordagem centralizada para alocar aplicações (ou *jobs*) de usuários nos recursos disponíveis. Este escalonador central é responsável pela a gerência de todos os *jobs* até o termino da execução, do aceite da aplicação à fila do escalonador, a definição da ordem de execução, a alocação de recursos, e monitoramento dos *jobs*. Mesmo esta proposta sendo altamente utilizada nos ambientes computacionais distribuídos, ela não parece ser escalável quando considera a quantidade crescente de *jobs* e o crescimento no tamanho dos sistemas computacionais. O que a torna pouco prática em ambientes computacionais modernos e menos ainda nos ambientes do futuro, onde esperamos possuir milhares de aplicações em execução simultaneamente utilizando milhares a milhões

de recursos computacionais. Como um exemplo, o ambiente computacional EC2 (*Elastic Compute Cloud*) da Amazon [4].

Nesse contexto surgem ferramentas com soluções alternativas às abordagens com reserva de recurso. Em linha com a proposta feita em [2], o EasyGrid AMS [5] é um sistema gerenciador de aplicações (AMS) que funciona como um meio entre a aplicação e o sistema operacional, de forma a facilitar o uso eficiente dos recursos considerando características da própria aplicação.

1.1 Objetivos

O objetivo deste trabalho é identificar uma estratégia viável para melhorar o aproveitamento dos ambientes distribuídos de larga escala (tanto *clusters* quanto *Grids* e *Clouds*) para que possam atender mais eficientemente às demandas de processamento feitas por um número significativo de aplicações concorrentes. Ao contrário do *broker* central, a abordagem mais usual no momento, a proposta permite explorar melhor os recursos e todos seus componentes (CPU, GPU, I/O, etc.). Esta estratégia decentralizada permite um controle gerencial das aplicações mais simples, porém efetivo, e pretende maximizar a utilização dos recursos.

Para tal, nesta proposta realizaremos uma abordagem diferente da usual. Para prover a melhor utilização dos recursos, não realizaremos o processo de reserva de recurso de modo restrito. Ou seja, nessa proposta as aplicações que estiverem utilizando alguns ou todos os recursos do ambiente não terão seu uso de modo exclusivo, podendo compartilhar parte ou todo o recurso a qualquer momento da execução. Para contornar o problema de escalabilidade do ambiente que utiliza a proposta clássica, que é um requisito fundamental para os novos ambientes computacionais, propomos descentralizar a gerência do ambiente através da utilização de um *middleware* que fornecerá à aplicação a capacidade de se escalonar e permite que estas aplicações autônomas colaborarem entre si. Assim, o ambiente consegue gerenciar-se e ainda sim trabalhar com um grande número de aplicações. O acesso aos hosts e à ordem de execução dos *jobs* passam a ser controlada pelo próprio, que uma vez munida de sua meta ou tempo de execução desejado para o termino, irá adaptar-se a realidade do ambiente para tentar entregar seu resultado dentro do prazo.

1.2 Contribuição

As contribuições desse trabalho são: (i) Propor um ambiente de execução distribuído que melhor adapte-se à demanda atual de processamento; (ii) Uma heurística capaz de criar o comportamento menos guloso nas aplicações autônomas, pensando no coletivo, de baixa complexidade e de baixo sobrecarga nos ambientes de alto desempenho existentes; (iii) Um conjunto de modificações evolutivas no projeto EasyGrid AMS que é mantido pelo SGCLab do IC-UFF, que facilitam o processo de desenvolvimento com a refatoração de módulos do código e a criação de níveis de execução para depuração.

1.3 Organização

Esta dissertação está organizada na seguinte forma. No Capítulo 2 apresentamos alguns trabalhos relacionados enquanto o Capítulo 3 contextualiza o projeto, apresentando alguns conhecimentos prévios necessários para a compreensão do trabalho. O Capítulo 4 descreve o *middleware* EasyGrid AMS que foi a base para a realização deste projeto. O Capítulo 5 descreve a proposta, uma abordagem baseada no contexto de uma Sociedade Altruísta (SA), e é seguido pelo Capítulo 6 que apresenta uma análise experimental da heurística de comportamento. Finalmente, o Capítulo 7 traz algumas conclusões.

Capítulo 2– Trabalhos Relacionados

O problema de escalonamento de tarefas, também conhecido como *job shop scheduling*, é um dos grandes problemas da computação distribuída. Sua resolução resume-se a achar uma ordenação de *n jobs*, aplicações de usuários submetidas para execução, de tamanhos (carga de processamento) variados sobre *m* recursos idênticos que minimize o *makespan[6]*. Embora esse problema seja de simples formulação, sua solução é extremamente difícil quando o número de *jobs* e de recursos é razoavelmente grande.

O *job shop scheduling* é um problema que pertencente ao grupo dos NP-COMPLETO, que é o conjunto de problemas no qual não se conhece uma solução polinomial em função do tamanho da entrada, no qual a obtenção de uma solução polinomial de um dos problemas do grupo implica na solução de todos os problemas do grupo. Devido à grande dificuldade de se obter o escalonamento ideal sem que para isso tenha-se que testar todo o espaço das soluções, que consequentemente poderia ser mais demorado do que a própria execução dos *jobs*. Pesquisadores buscam heurísticas capazes de gerar bons escalonamentos, sem ter que pagar o alto custo de tempo da busca exaustiva.

A proposta clássica, que é aplicada em ambientes computacionais como *clusters* e *grids*, baseia-se na ideia de uma aplicação central (*job scheduler*) responsável por gerenciar todos os *jobs,* da submissão à conclusão. Alocando, enfileirando e controlando acesso aos recursos (CPU, disco, memória, LAN). Essas propostas geralmente são limitadas na obtenção de bons resultados, justamente devido ao fato que a complexidade de tempo cresce exponencialmente em relação a tamanho dos *jobs* e a quantidade dos recursos.

O trabalho reformula o problema como sendo um problema de escalonamento distribuída que permite concorrência de *jobs* e, ao invés de minimizar o *makespan*, utiliza uma métrica nova (*uma meta de tempo de execução*) para manter um priorização na execução das tarefas. Em outras palavras, o problemas é de escalonar *n jobs* de tamanhos distintos sobre *m* recursos, não necessariamente idênticos, que reduz o *makespan* total e tenta entregar os *jobs* dentro de suas respetivas metas.

Algumas das abordagens mais modernas tentam contornar a limitação da alta complexidade na obtenção do escalonamento ótimo com heurísticas bem elaboradas. Neste capítulo, iremos apresentar alguns projetos relacionados de pesquisadores que trabalham com problemas similares ao trabalho apresentado.

2.1 Artigos relacionados

Em [7], é apresentado um escalonador de tarefas multiobjetivo que através de conceitos de teoria dos jogos, teoria que se popularizou recentemente no problema de escalonamento, busca otimizar a execução dos *job* através redução do consumo de energia ao mesmo tempo que o *makespan.* O foco deste trabalho são os problemas computacionais CPU intensivo onde cada *job* possui seu *deadline* de tempo.

A solução proposta pelo autor, para resolução do problema multiobjetivo dado, utiliza-se da teoria dos jogos formulando um problema do tipo jogo cooperativo. Neste jogo, o autor garante baseado na formulação do problema: a existência do ponto de barganha, conceito na teoria dos jogos no qual pelo menos dois jogadores, que aqui são representados pelas aplicações, podem aumentar seu estado de satisfação caso cheguem a um acordo, o que é condição suficiente para garantir que a solução irá convergir para um resultado (escalonamento válido). Um detalhe bem importante na abordagem utilizada é que, diferentemente da técnica usual de escalonamentos embasados na teoria dos jogos que usam axioma de Nash, a proposta do trabalho é de transformar o problema de escalonamento em um max-max-min, problema de otimização com solução em tempo polinomial, para tentar gerar soluções rápidas.

Para alcançar a redução da energia proposta, o artigo utiliza-se das técnicas de redução do cache-miss no host e a redução da frequência de operação do processador para ajustar o consumo do processamento (DVS – *Dynamic Voltage Scaling*), o que reduz consequentemente a energia dissipada. Em suma, esse trabalho destina-se a problemas de escalonamento/mapeamento de tarefas heterogêneas sobre arquitetura de processadores *multicores* com ênfase no baixo consumo energético.

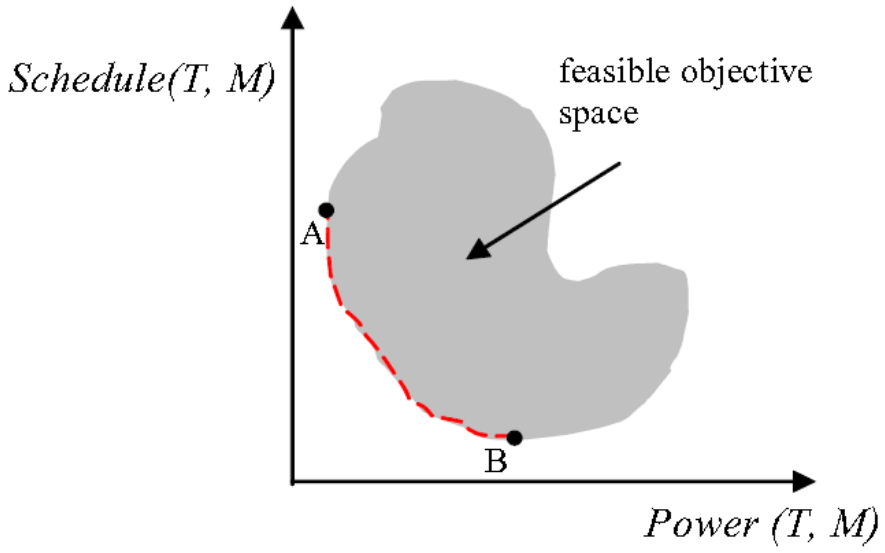

Figura 2.1 Ilustração de um problema multiple objetive optimization

A resolução do problema de escalonamento aqui proposto pode ser interpretada como um problema de *multiple objetive optimization* (MMO), que é uma classe de problemas de otimização matemática que envolve mais de um objetivo. Nesta classe de problemas, é comum que existam mais de uma solução para um problema e com objetivos conflitantes, neste caso *makespan* e economia energia, necessitando assim achar um meio termo entre os objetivos definidos. A classe de soluções dos problemas MOO pode ser interpretada como a fronteira do espaço das soluções de um gráfico, condição suficiente e existente nos problemas min-max, que representa os objetivos a serem maximizados (Figura 2.1). Existem diversos métodos na literatura que, segundo o autor, resolveriam o problema proposto. Entretanto a complexidade associada ao método demandaria muito tempo na execução do escalonamento. Assim, a proposta para contornar essa complexidade e simplificar a resolução do problema é através da criação de um conjunto de estratégias ditas como simples. Para tal, três estratégias foram definidas para representar o meio termo entre o *makespan* e economia de energia.

1. Assuma que saibamos o escalonamento ótimo. Reduza o consumo de energia piorando o escalonamento através da adição de folgas sobre o tempo de execução.
2. Minimizar o tempo requerido para um provisionamento de energia: Para cada aplicação paralela, identificar o requerimento normal de energia e tenta reduzir esse requerimento para um número abaixo de um fator (ex.:80%). Buscando descobrir o menor tempo que satisfaça essa quantidade de energia.
3. Encontre as tarefas e restrições de energia que a "função penalidade" seja minimizada. Dado um provisionamento energético e requerimentos de

execução, caso ambos requisitos não sejam satisfeitos, a meta é minimizar a função penalidade (função que irá calcular a penalidade por não ter satisfeito as restrições definidas).

Para facilitar a interpretação do problema, o autor define a unidade de processamento HeMP, que consiste de um número de núcleos heterogêneos cada um equipado com um módulo DVS. Também é definido baseado nas premissas apresentadas o problema EATA (*Energy-Aware Task Allocation*), que trata-se do problema o qual deseja-se minimizar o consumo de energia de um *job* mas que ainda seja possível que suas tarefas terminem dentro de seu *deadline*. Podendo ser descrito sobre as condições:

1. Os núcleos das máquinas utilizadas possuem processamento variável entre um valor mínimo e máximo de frequência, ambos maiores que zero.
2. Existem especificações nas arquiteturas do *host* são elas: tipo de *host*, velocidade em GHz, I/O, tamanho do *cache* local e da memória em *bytes*.
3. Uma aplicação paralela é definida como $T=\{t_1,t_2,t_3,..t_n\}$, onde cada tarefa é caracterizada por: número de ciclos de CPU necessários e conhecidos pelo escalonador, deadline e tipo de arquitetura (as arquiteturas especificadas são hipotética A, B, C).

A heurística proposta para a solução inicia-se definindo que todas as tarefas devem ser mapeadas em um *core*, e satisfazer suas as restrições. Uma vez que as tarefas foram mapeadas, as iterações são realizadas através do conceito na teoria dos jogos NBS (*Nash Bargain Solution).* Uma solução NBS é uma solução onde os jogadores barganham porções de alguma entidade (algum recurso que no caso é processamento) até que todos os jogadores alcancem suas demandas. A partir do algoritmo NBS e do problema de EATA, o autor cria o algoritmo definido como algoritmo NBS-EATA. Algoritmo que combina técnicas clássicas de teoria dos jogos com condições Kuhn-Tucker e a derivada de Lagrange para identificar o ponto de barganha e consequentemente a solução do problema, que é finalizado quando a iterações dos jogos convergem para tal ponto.

Neste projeto temos as características do problema tratado bem semelhantes a nossa abordagem (Capítulo 5). O atendimento de multiplos-objetivos, que no artigo mencionado é a procura do meio termo entre *makespan* e o consumo de energia, e em nossa abordagem é a meta e *makespan* dos *jobs*.

Em [8], é apresentado um algoritmo multiobjetivo híbrido que tem como diferencial integrar as duas principais etapas da execução de uma aplicação. O processo de planejar e o de escalonar as tarefas visando melhorar o *makespan* da execução, diferente do processo de

escalonamento clássico, cujo as etapa do planejamento e escalonamento são realizadas separadamente.

Tradicionalmente, o processo de planejamento consiste em resolver todas as pendências e decisões para executar o job. Incluindo identificar os servidores, ferramentas, as restrições dos recursos, resolução de precedências dos *jobs*, etc. O processo de escalonamento pode ser visto como a ponte entre o design do produto e sua produção. No qual colocamos o processo em execução, que tradicionalmente é realizado após a etapa de planejamento. O grande problema que pode acontecer nesse processo são as condições do ambiente de execução mudarem durante as etapas, consequentemente levando a execução desbalanceada dos *job*, o que por conseguinte piora seu tempo de execução.

Neste contexto, o autor apresenta o problema de IPPS (*integrated process planning and scheduling*) que pode ser definido como: Dado um conjunto de *n* partes a serem processadas em máquinas com alternativas de recursos, selecione o conjunto de recursos e a sequências de operações de modo que defina um escalonamento que atenda às restrições e os objetivos possam ser alcançados.

Na proposta apresentada no artigo, o de um escalonamento multi objetivo com as etapas de planejamento e escalonamento integrado, o escalonamento assume dois papéis diferentes, o de *Job shop scheduling* e o modelo matemático do IPPS (modelo de programação inteira) criando assim um escalonamento que mistura os dois conceitos. Para o escalonamento multi-objetivo apresentado são definidos três objetivos a serem otimizados. O *makespan* (tempo total para realizar todos os *jobs*), minimizar o máximo tempo processamento gasto em uma máquina (MMW - carga máxima por máquina) e o de minimizar máximo tempo de trabalho gasto em todas as máquinas (TWM- Total de carga na máquinas).

Para a execução da heurística apresentada no artigo, algumas premissas são definidas para o modelo proposto, são elas: (i) *Jobs* são independentes, sem preempção e cada *host* executa somente um *job* por vez; (ii) as diferentes operações de um *job* não podem ser executadas simultaneamente; (iii)Todos os *jobs* e máquinas estão disponíveis no tempo zero simultaneamente. Após o término de um *job* é iniciado imediatamente o próximo *job* do processo, desconsiderando o tempo de comunicação. As sequências de operação são independentes, inclusive no tempo de processamento.

O modelo "*multi-objective optimization problem*" (MOO) é o modelo que representa o problema dos objetivos o qual queremos atender. Esse problema pode ser descrito como:

Max/Min $y=f(x)=\{f_1(x),f_2(x),...,fk(x)\}$;

$e(x)= \{e_1(x),e_2(x),..,e_m(x)\} \leq 0,$
$x = (x_1,x_2,...,x_n) \in X$ $y = (y_1,y_2,...,y_k) \in Y.$

Onde x são variáveis, y os objetivos, X espaço das variáveis, Y espaço do objetivos e $e(x) \leq 0$ as restrições.

Mapeando o MOO na teoria dos jogos para um jogo não cooperativo temos: $G=\{S,u_1,u_2,u_3\}$, onde cada objetivo foi mapeado para um jogador e o espaço das variáveis foram mapeadas em S.

A solução proposta baseia-se na teoria dos jogos sobre um jogo não cooperativo para atender os múltiplos objetivos. Para tal, foi criado um algoritmo híbrido IPPS baseado em GA[9] e TS[10] para atender os múltiplos objetivos em um vasto espaço de busca. Onde alcançar o equilíbrio de Nash é considerado como resultado ótimo.

A proposta centralizada apresentada nesse trabalho busca metas bem semelhantes à nossa. O de unir as duas etapas do processo de escalonamento clássico, com intuito de otimizar três objetivos no processo de escalonamento dos *jobs* (*Makespan*, MMW, TWM). Embora não seja explícito, na nossa proposta (Capítulo 5) buscamos minimizar o MMW em nossos trabalhos, pois buscamos que os recursos estejam sempre balanceados, e o TWM pois ajustamos a carga da aplicação ao *host*. Entretanto, nossa proposta se prontifica a realizar uma abordagem muito mais simples.

Em [11], temos um problema de escalonamento que se propõe a refletir a realidade dos sistemas distribuídos atuais. O escalonamento de aplicações de larga escala composto de milhares de tarefas homogêneas independentes, *BoT (bag of task*), onde têm-se dois objetivos: ter bom tempo de execução, e um baixo custo econômico. Esta proposta tem uma característica bem especial, o fato de considerar comportamentos que são comuns nos ambientes atuais, como: múltiplos recursos e as restrições dos ambientes. O que aumenta ainda mais dificuldade do problema.

Neste artigo, os *jobs* são modelados por grafos acíclicos direcionados(GAD), onde Y= (BS,DD), BS= $\bigcup_{k=1}^{K} T_k$ é o conjunto das K tarefas, *bag of tasks* (BoTs), and DD = $(T_s < T_d \mid \{T_s, T_d\} \subset BS)$ são as arestas que representam o fluxo de dependências de dados. O modelo abstrato definido no trabalho tem por base o ambiente de computação distribuída [12]. Que consiste de uma nuvem híbrida, com o mínimo de uma nuvem privada e uma pública. As tarefas ou *jobs* que chegam em cada nuvem podem pertencer a múltiplas aplicações. A execução de cada aplicação é controlada por um "*application manager*" que pode executar concorrentemente a outro job com seu próprio *application manager* por recurso. Para

trabalhar com este modelo mais realista, o gerenciador de aplicação mantém uma fila local para cada site na nuvem, para ordenar e limitar o número de *jobs* baseado na taxa de processamento do site.

A meta do artigo é: dado um número de *n* aplicações consistindo de tarefas que podem categorizar k diferentes BoTs, e o ambiente composto de M sites. O objetivo da proposta é achar uma solução que minimize o *makespan*, o custo financeiro e o consumo de banda utilizada por toda a aplicação. A importância da redução de banda é crucial para aplicações que rodam com essa característica de ambiente, pois sobrecarga de comunicação é um fator crucial.

A formulação do problema na teoria dos jogos requer que sejam definidos três parâmetros: os jogadores, as estratégias e o pagamento. Para este trabalho é considerado o jogo como sendo um jogo cooperativo, onde cada gerente de aplicação, que são os jogadores, tenta minimizar durante um certo instante t_k do $BoT\ T$ baseado no número de tarefas e na taxa de processamento sobre cada site. Os pagamentos são as distribuições de tarefa e alocação de banda. Assim, cada gerente de aplicação manuseia um BoT e tem como objetivo minimizar o *makespan* e o custo econômico satisfazendo as restrições de armazenamento e banda.

Na teoria dos jogos, as partidas são coordenadas pelo grupo de jogadores que são representadas por uma função, no qual as funções mais importantes fazem com que o jogo tenha uma transferência de habilidade. Com isso, um jogador que tenha mais vantagens tem a habilidade de compensar outro jogador que não tenha. Fornecendo meios justos de compartilhar recursos.

Segundo o autor, para aplicações de larga-escala com BoTs o máximo *makespan* é próximo ao *makespan* agregado dividido pelo número de processadores. O que é suficiente para definir como meta da aproximação o *makespan* mínimo agregado. A solução usual utiliza-se os multiplicadores de Lagrange, com um custo extremamente alto para a obtenção da solução ótima devido à alta complexidade do problema, contudo, a solução apresentada no artigo promete conseguir bons resultados com boa taxa de convergência em poucos estágios. Sendo sua complexidade de tempo O(n.N).

A proposta possui características bem interessantes por abordar ambientes computacionais mais modernos, adicionar o middleware no papel da gerência das tarefas de cada BoT no site, reduzindo o custo gerencial para obter o escalonamento que antes era de gerenciar todo o sistema distribuído, para somente seu universo de execução. Isso em um modelo de ambiente complexo com mais características aos recursos. Entretanto, a proposta

ainda é de uma abordagem estática, realizado previamente a execução na etapa de planejamento e não se adaptando a modificações no ambiente.

Em [12], temos a ferramenta de escalonamento de aplicações soft real-time GRACE-OS, cuja sua principal meta é escalonar aplicações multimídias atendendo a qualidade de serviço e meta de economia de energia. Aplicações soft real-time podem ser descritas como aplicações onde a perda do deadline de algumas tarefas é aceitável para a execução do job. Entretanto, as perdas influenciam na qualidade do resultado final, sendo comumente definido um limiar de perdas.

A principal característica é o fato de ser um problema de escalonamento multi objetivo que ao mesmo tempo que precisa executar as tarefas antes de seus deadlines, deve também ajustar a velocidade de processamento para economizar energia. Assim, a frequência de operação da CPU precisa ser ajustada para baixo, afim de o consumo da energia possa ser minimizado, entretanto, rápido o suficiente para que as tarefas possam ser escalonadas e executadas dentro de suas metas. Ou pelo menos, o número mínimo de tarefas possa ser executado dentro do limiar de qualidade definido.

Nesse trabalho, as tarefas são decodificadas em quadros de vídeos de 30ms cada. Os jobs são as unidades de computação básica com restrições e compostas de 3 informações para execução: tempo de inicio, tempo de fim e soft deadline.

O GRACE-OS, algoritmo proposto, possui sua estrutura dividida em três componentes: um profiler, um escalonador SRT(soft real-time), e um adaptador de velocidade. O profiler é responsável por monitorar o ciclo de CPU utilizados individualmente por cada tarefa. O escalonador SRT é responsável por alocar ciclos para as tarefas e alocá-las garantindo a desempenho. O adaptador de velocidade ajusta a velocidade de CPU dinamicamente com o intuito de economizar energia. Ele adapta o tempo de execução de cada tarefa baseado no tempo de alocação fornecido pelo escalonador SRT e a distribuição da demanda fornecido pelo profiler. Sua proposta prevê alcançar a eficiência energética através da estimativa online da distribuição da demanda. Assim, o primeiro passo é estimar a distribuição probabilística de cada tarefa demandada. Segundo passo é alocar os ciclos baseados na distribuição das tarefas providos estatisticamente. A informação sobre o número de ciclos gasto por cada tarefa pode ser realizado graças ao contador de ciclo que cada tarefa possui incluído em seu código, que contabiliza quantos ciclos foram necessários para finalizar a tarefa.

Nos testes apresentados no artigo, o resultado do escalonamento estocástico mostrou-se depende da probabilidade da distribuição das tarefas. Se uma tarefa possui uma demanda

estável, o escalonador consegue estimar com uma janela pequena de tempo necessário para a tarefa e fazer uma boa alocação. Por outro lado, se a aplicação for instável. A janela de tempo pode ser enorme.

A proposta desse trabalho possui jobs com características de soft real-time semelhantes a nossas. Sendo o "trade-off" das metas, a execução dos n jobs dentro dos deadline contra a liberação de processamento para os demais jobs.

2.2 Comparações

Nosso trabalho, descrito no capítulo 5, possui características semelhantes as quatro propostas aqui apresentadas. Entretanto, mesmo ele compartilhe de muitas características disponíveis nas propostas mencionadas anteriormente, nenhuma dela trata exatamente do mesmo problema que atacamos. Contudo, podemos citar algumas características marcantes.

Como em [12] , temos características de soft real-time o qual desejamos atender. Com a diferença que o nosso soft real-time são jobs independentes , não compondo um todo que compõem um serviço. A perda do deadline não implica que deixamos de querer os resultados, apenas estamos dispostos a esperar um pouco mais do que planejávamos.

Como em [11], que é bastante semelhante em estrutura à nossa proposta. Trabalhamos com aplicações BoTs que possuem características de grande quantidades de tarefas homogêneas (embora não seja requisito). Possuímos uma implementação o qual utiliza-se de um *middleware* atuando no nível do site (mais detalhes seção 4.3), entretanto com a adição também da monitoração no nível de *host*. Nossos *jobs* disputam recursos quando em execução, porém uma disputa mais sociável onde as aplicações são mais suscetíveis a ceder do que a solicitar.

Assim como realizado em [8], nosso escalonamento integra as etapas de planejamento e escalonamento, mas também integramos a execução. Isso porque nosso escalamento é dinâmico, sempre se ajustando e replanejando mediante a realidade do ambiente.

Semelhante ao trabalho [7], temos metas que se contrapõem nosso "*trade-off*" gira em torno do egoísmo/sociável, temos que ceder processamento para sermos sociáveis e ajudar as demais aplicações mas não queremos perder nossa meta de execução.

Assim como todos os trabalhos mencionados temos múltiplos objetivos, incluindo terminar cada aplicação dentro da sua meta; eliminar ociosidade dos recursos no sistema e priorizar aplicações. Tudo isso mantendo uma boa utilização do ambiente através de ajustes na vazão dos processos, de acordo com as demandas atuais do ambiente. Nossa proposta prevê adicionar comportamentos na aplicação até então não mencionadas nos trabalhos

anteriores e remover qualquer controle gerencial central entre as aplicações. Consideramos que as aplicações fazem parte de uma Sociedade de Aplicações (SA) e terão acesso compartilhado aos recursos do ambiente. Como numa sociedade tradicional, cada aplicação deve comportar-se de uma maneira aceitável perante os demais participantes da sociedade, mas adaptando-se seu comportamento suficiente para alcançar sua meta.

Em todas as propostas apresentadas como trabalhos relacionados, podemos notar uma característica comum: os *jobs* estão disponíveis para a execução simultaneamente. Este tipo de escalonamento é considerado escalonamento estático. Propostas para escalonar ambientes dinâmicos, onde os *jobs* são alocados para escalonar todo o tempo são menos comuns que os demais. Isso porque sua complexidade de execução é muito maior que o estático, precisando reescalonar o sistema a cada nova entrada de *job*. O que torna esse problema extremamente difícil, principalmente para as abordagens centralizadas.

Nossa proposta prevê atender um ambiente computacional bem mais complexo que os trabalhos propostos, porém com uma solução bem mais simples. Pretendemos atender ambientes dinâmicos, onde os *jobs* estão ficando disponíveis a qualquer momento durante a execução fornecendo um ambiente altamente adaptativo.

2.3 Resumo

Neste capítulo apresentamos quatro proposta na literatura de escalonadores de tarefas. Nenhuma das apresentadas, ou pesquisadas, apresentou completamente características idênticas da abordagem que utilizamos, ou mesmo, o problema que propormos tratar. Criação de um ambiente descentralizado que execute jobs de com justiça no escalonamento e priorização na execução. Entretanto, todas possuem pelo menos uma das características apresentadas em nosso problema de escalonamento de tarefas: objetivos conflitantes, abordagem descentralizada, aplicações soft real-time, redução do makespan e priorização das jobs. Com isso, acreditamos que essa proposta pode ser uma inicio de uma nova abordagem de como escalonar tarefas em ambientes distribuídos.

Capítulo 3– Contexto

O escalonamento de tarefas é um tópico extremamente importante na ciência da computação. A definição da ordem que os *jobs* serão executados dentro de um sistema computacional é fundamental para que os hosts sejam bem utilizados. Um escalonamento ruim tende a gerar ociosidade nos hosts o que só aumenta o *makespan*[13]. O desenvolvimento de heurísticas e algoritmos que consigam obter um escalonamento eficiente depende de diversos conhecimentos prévios. Neste capítulo, apresentaremos tecnologias e estudos relacionados que ajudarão a melhor compreensão do trabalho.

3.1 - Sistemas Distribuídos

A execução de aplicações complexas, principalmente as científicas, exige um custo computacional muito alto para que um computador convencional consiga executá-lo em um tempo hábil. Para executar tais aplicações comumente utiliza-se de sistemas de grande poder computacional, conhecidos como supercomputadores.

Os supercomputadores da atualidade são essencialmente sistemas distribuídos, que segundo [14], podem ser definidos como "uma coleção de computadores independentes entre si que se apresenta ao usuário como um sistema único e coerente". Nestes sistemas os usuários submetem suas aplicação para execução, que geralmente entram em uma fila, criando o que definimos como job, uma instancia em memória de uma aplicação que ira ou está em execução. Podemos citar três estruturas de sistemas distribuídos fundamentais na atualidade, são elas *cloud computing*, *grid computing* e *cluster computing*.

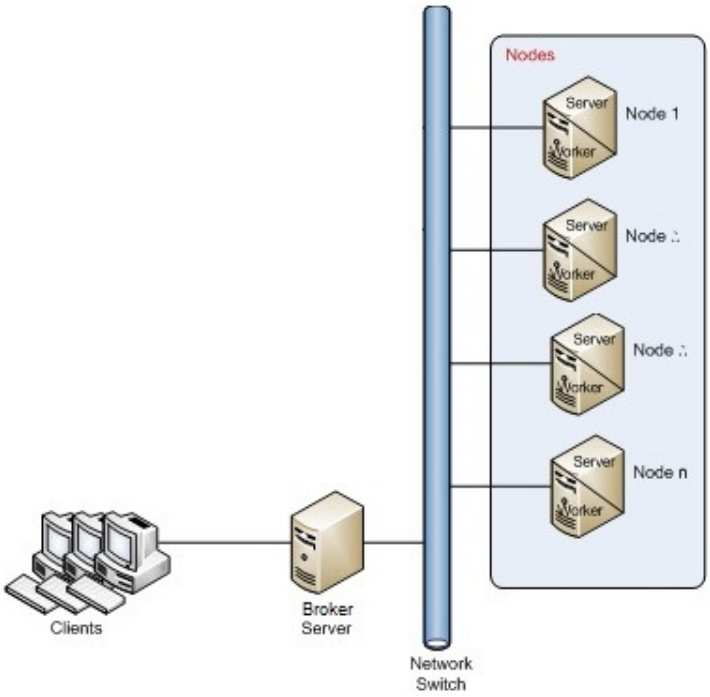

Figura 3.1 Ilustração de um cluster computacional.

Fonte: https://en.wikipedia.org/wiki/HPCC#/media/File:Fig3_Roxie_Cluster.jpg

Cluster computing consiste de um conjunto de computadores homogêneos interligados por uma rede de alto desempenho o qual pode ser visto como um único sistema. Isso porque os hosts funcionam de forma colaborativa para a resolução de um problema. Os clusters em sua concepção inicial eram o fornecimento de poder computacional sob demanda proveniente do CPU. Ao longo da evolução dos computadores, novos recursos computacionais se tornaram importantes para o compartilhamento. Com isso, na atualidade podemos dividir os clusters computacionais em três tipos: *High Performance Computing, High Availability Computing, Load Balance Computing*.[15]

- ***High Performance Computing*****:** O tipo mais comum de cluster visa fornecer um alto poder de processamento. Esse tipo de cluster que em sua concepção fornecia essencialmente processamento oriundo de CPU's, nas versões atuais são capazes de utilizar outros componentes dos PC's modernos para fornecer processamento. Como GPU's (*Graphics Processing Unit*) e PPU's (*Physics Processing Unit*). Ex.: *Tianhe-2 (MilkyWay-2) Cluster* – número 1 na lista do top 500 de novembro de 2015. [35]
- ***High Availability Computing:*** São clusters que foram concebidos para reduzir a chance de que falhas tornem o cluster inutilizável. Ou seja, um cluster deve tentar sempre ter parte de seus recursos online, mantendo assim as aplicações em execução sempre progredindo. Esses clusters são amplamente utilizados em aplicações consideradas criticas,

que envolvem risco a vida ou a transações financeiras. Como aplicações monetárias, sistemas de voos e grandes *e-commerces*.

- ***Load-balance computing:*** São clusters que têm como responsabilidade distribuir de maneira equilibrada a utilização dos recursos computacional de um sistema a fim de obter melhores tempos de resposta (geralmente processamento ou acesso aos dados em discos ou bancos de dados). Normalmente aplicado na execução de *aplicações* de rápida execução, esse tipos de clusters são usualmente empregados em servidores de e-commerce onde a distribuição das requisições é fundamental para otimizar os tempos de resposta.

Os clusters são vistos como o sistema distribuído de mais fácil execução de aplicações dentre os modelos de sistemas distribuídos mencionados. Isso se deve principalmente à homogeneidade dos seus hosts e à rede local de alto de alto desempenho. O que facilita para seu utilizador o desenvolvimento das aplicações sem se preocupar (ou pelo menos se preocupando menos) com a variação de desempenho e lentidões nos diferentes hosts.

Tradicionalmente, os clusters computacionais tiveram sua utilização como um sistema distribuído único do seu detentor, cada empresa/universidade possui o seu próprio cluster.

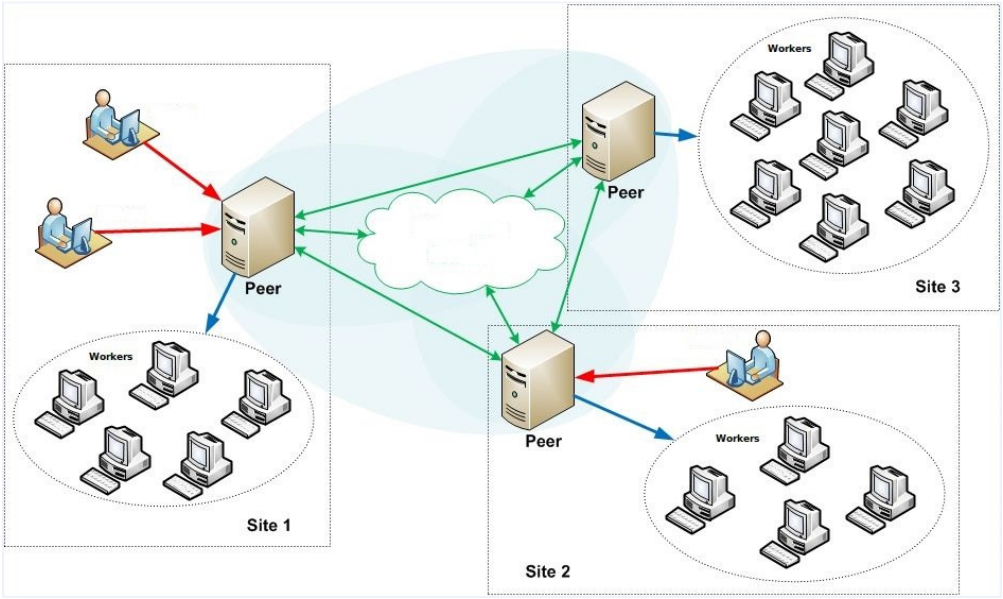

Figura 3.2 Ilustração de uma grade Computacional

Fonte: https://sciencenode.org/feature/grid-cooperative.php

Grid computing (computação em grades) trata-se de um conjunto de computadores heterogêneos, geograficamente distribuídos, e interligados por uma rede que não necessariamente é uma rede de alto desempenho. Sua constituição pode ser vista como uma

evolução do cluster, onde algumas das restrições do cluster são afrouxadas, como a necessidade de ter uma rede de alto desempenho e homogeneidade dos recursos, conseguindo assim agregar um número muito maior de recursos (computadores)[15].

Uma característica comum das grades é a sua distribuição geográfica. Normalmente, os hosts que compõem um *grid* não possuem todos seus recursos localizados fisicamente próximos, sendo distribuídos em pequenos grupos, em alguns casos de apenas um host. A rede de comunicação que interliga os nós normalmente é a Internet, o que não é uma rede com altas taxas de velocidade na comunicação, passando a exigir ainda mais da aplicação, necessitando que sejam maleáveis aos atrasos na comunicação.

Com o objetivo de fornecer poder computacional *on-demand*, não importando a origem do fornecimento, a proposta das grades é ampliar significativamente o número de hosts, explorando quaisquer recursos com processamento ocioso dentro do seu alcance e, consequentemente conseguindo um poder computacional maior que oferecido pelos os clusters.[16,17]

As grades computacionais exigem que o desenvolvedor das aplicações tenha um conhecimento muito maior da infraestrutura, sobretudo suas limitações. O que acarreta um esforço maior para o desenvolvimento da aplicação.

Segundo [18], as grades computacionais podem ser classificadas em três gerações. A primeira geração é a do compartilhamento de recursos e dados (como o European DataGrid [19]). A segunda geração é caracterizada pelo uso de middleware para a utilização de diferentes tecnologias de redes (como o Globus Toolkit [20]). A terceira é resultado da combinação dos recursos web com a segunda (como o Boinc [21], um projeto de middleware *open source* de computação voluntária), capaz de alcançar uma escala de uma grade computacional de dimensão mundial.

Um grande impasse do grid é seu modelo de negócio, ou a falta dele. Como a proposta do grid nunca foi efetivamente aplicada como modelo de negócio, somente como um meio colaborativo para aumento de recursos no grid. Seu modelo de computação distribuída teve pouca aderência no mundo comercial, e tendo uma maior aceitação em empresas de grande porte como multinacionais e no meio acadêmico entre as universidades e centros de pesquisa.

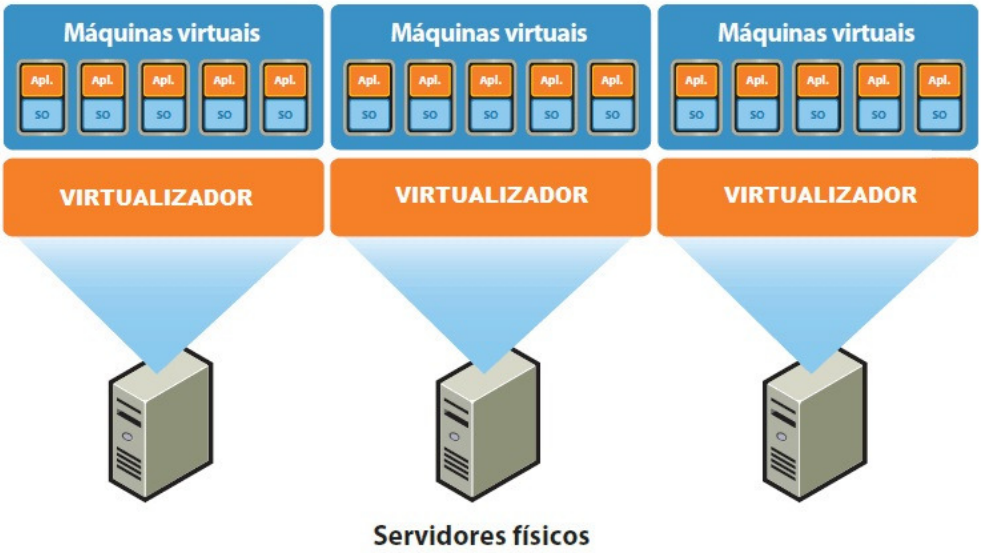

Figura 3.3 Ilustração Cloud Computing

Fonte: http://cronnussolucoes.com.br/solucoes/virtualizacao

Cloud computing é um paradigma de fornecimento de poder computacional recente e amplamente difundido. O seu funcionamento propõe-se a aplicar uma camada de virtualização sob o *data center* de tal maneira que o processamento existente nos servidores possa ser melhor fracionado, e passe a ser utilizado entre múltiplas aplicações ou mesmo seja vendido como computação excedente. Seu surgimento acompanhado da popularização das tecnologias de multicores e de virtualização embarcado nos processadores (Intel VT, AMD-V) que otimizaram os processo de virtualização, com instruções própria para minimizar o custo de virtualização do ambiente.[22].

O paradigma do cloud computing trouxe junto ao seu surgimento um novo modelo de negócio. Uma forma de monetizar o poder computacional. Esse é o grande diferencial que fez com que a cloud computing popularizasse tão rapidamente.

Segundo [1], o paradigma cloud computing define três tipos básicos de serviços:

- ***Infraestructure as a Service*** **(IaaS)** é a estrutura de Cloud computing mais básica dentre as existentes. Esse serviço prevê o fornecimento de estações de trabalho (inclusive no formato de maquinas virtuais) limpas, com o mínimo para o consumidor do serviço possa montar seu ambiente e utilizar para executar suas demandas. As maquinas virtuais podem ser configuradas com capacidades diferentes de recursos básicos como CPUs, quantidades de memória e armazenamento. Adicionalmente, costuma-se escolher a um sistema

operacional, pelo menos uma das maquinas virtuais com acesso a internet, uma VLAN para conectar as maquinas virtuais (VMs) e um sistema de comunicação remoto (normalmente SSH). Ex.: Amazon EC2[4].

- ***Platform as a Service* (PaaS)** pode ser vista como um passo a mais sobre a estrutura IaaS. Trata-se de um ambiente montado e configurando com um conjunto de ferramentas (DB, Framework, Cache, Compiladores, Webservers), geralmente definido pelo cliente na hora da contratação do serviço, para que possa rapidamente provisionar sua aplicação. Nesse modelo, o cliente alvo é o desenvolvedor de software. Ex.: Heroku[23].
- **Software as a Service (SaaS)** é o modelo mais focado no usuário final. Nesse serviço, o usuário passa a acessar uma aplicação remotamente (Software CAD, Microsoft Office 365, etc.) que ira processar o resultado na infraestrutura da nuvem e o terminal do usuário será responsável somente pela tarefa de renderizar as telas com os resultados finais. Ex.: Google Cloud[24].

Independente do serviço, a computação em nuvem traz inúmeras vantagens para o usuário utilizador. Podemos destacar entre elas[1]:

- **Isolamento do Ambiente:** Uma vez alocada uma VM, o usuário tem o ambiente de maneira exclusiva, diminuindo assim a possibilidade de que outras aplicações interferiram em seu funcionamento.
- **Abstração da infraestrutura:** Como contratante de um serviço terceirizado, o funcionamento e manutenção do servidor passa ser uma preocupação a menos para o contratante que agora não precisa adquirir seus próprios recursos.
- **Elasticidade:** A infraestrutura pode prover mais/menos recursos baseado na demanda existente. O que possibilita não só suportar melhor os picos de utilização, mas também economizar nos vales da utilização da infraestrutura, reduzindo o custo com a locação.
- **Viabilidade econômica:** A utilização de uma infraestrutura de alto poder computacional (mesmo que por algumas horas) passa ser economicamente viável para muitos usuários finais. Pois agora, somente é necessário arcar com os custos do aluguel por algumas horas, e não a de aquisição e manutenção de uma grande infraestrutura.

Contudo, mesmo os sistemas distribuídos têm suas limitações. Para a execução de aplicações de alta demanda de processamento, uma boa infraestrutura computacional seguindo por um modelo sistema distribuído eficiente pode não ser o suficiente. Necessita-se de mais do que isso, como técnicas que permitam que as aplicações executem partes de sua demanda em diversos computadores: sejam para viabilizar a execução técnica da aplicação; viabilizar a obtenção dos resultados em um tempo viável; ou o rápido término da aplicação que por consequência poderia ter um menor custo de utilização. Podemos dizer que somente uma boa infraestrutura não basta, necessitamos de aplicações eficientes, o que nesse caso pode ser visto como aplicações paralelizáveis, e com um bom escalonamento.

3.2 - Aplicações Paralelas

Aplicações paralelas são aplicações compostas por vários processos ou threads, mas que executam como se fosse um único processo. São muito utilizadas em ambientes distribuídos quando se faz necessário um melhor tempo de execução, ou um tempo hábil que seria improvável de se conseguir em um único computador. Normalmente, quando se precisa utilizar de um supercomputador para executar sua aplicação, é necessário que a aplicação seja desenvolvida em uma linguagem de programação paralela, ou uma linguagem com extensões que permitam explorar paralelismo[25]. As ferramentas existentes para paralelização de aplicações são voltadas para um dos dois tipos específicos de aplicações: as que executam em ambientes de memória compartilhadas, ou as que executam em ambientes de memória distribuída. As extensões mais conhecidas para desenvolvimento de aplicações paralelas são: OpenMP[26] para memória compartilhada e MPI[27] para sistemas de memória distribuída.

Nesse trabalho, focamos especificamente nas aplicações que executam em ambientes de **memórias distribuídas**, por serem os ambientes predominantes e uma característica comum nos supercomputadores da atualidade, como os clusters reais (*data centers*) e clusters virtuais (fornecido por uma infraestrutura de "cloud computing"). Em especial usando o MPI, que é o padrão na execução de aplicações por troca de mensagens, e é amplamente utilizado nos supercomputadores, muitos do top500, quando se deseja desenvolver aplicação distribuídas.

3.3 – MPI (Message Passing Interface)

O MPI (Message Passing Interface) é uma padronização para implementações de bibliotecas de programação distribuída por troca de mensagens. Neste padrão de programação distribuída,

utiliza-se a troca dos dados através de mensagens entre processos, abstraindo sua localização na infraestrutura.

A primeira versão do padrão MPI foi definida em 1994, ao longo do tempo o padrão foi evoluindo até chegar a versão atual, a versão 3.1, em junho de 2015. Com o surgimento dos sistemas distribuídos, ficou evidente a necessidade de um padrão para a realização de comunicação por troca de mensagens entre sistemas diferentes. A principal motivação era a definição de um padrão coeso e portável, que não se prende o sistema à arquitetura.

O desenvolvimento do padrão MPI foi realizado por um esforço coletivo que reunia cerca de 80 pessoas de 40 organizações diferentes, localizadas principalmente nos Estados Unidos e Europa. A maioria das organizações envolvidas diretamente era de vendedores de computadores distribuídos, mas também existia a presença de representantes de universidades, órgãos governamentais e indústrias [27]

O padrão MPI destacou-se dentre as ferramentas de programação paralela pela sua proposta ser robusta e facilmente portável. Contudo, mesmo fornecendo métodos para a paralelização de aplicações, o esforço para decidir como paralelizar a aplicação e onde alocar as tarefas mos recursos disponíveis ainda são de responsabilidade do usuário.

O MPI destacou-se entre as ferramentas de programação paralela pela sua robustez e portabilidade, tendo as seguintes características como marcantes:

- Portável para diversas arquiteturas de computadores;
- Possui mais de 125 funções para programação e analise de desempenho;
- Possui versões para diversas linguagens de programação, dentre elas: C, C++, Fortran, JAVA, Python, Ruby, etc.;
- Possui um modelo de paralelismo explícito - o usuário define explicitamente no código da aplicação onde o processo ira executar, quantidades de processos, o tamanho da carga, etc.

A principal característica da interface MPI é prover uma topologia virtual que permita a comunicação entre os processos em um conjunto de computadores, independente do meio computacional existente.

A interface MPI possui diversas implementações, MPI-F[28], MPICH[29], INTEL[30], Open MPI[31], FT-MPI[32], LA/MPI[33] e LAM/MPI[34]. Daremos um destaque especial para as duas ultimas implementações referenciadas: a primeira por ser a

versão que mais ganha adeptos nos últimos tempos e a segunda por ser a versão que foi utilizada no trabalho.

3.3.1 Open MPI

Open MPI é hoje a distribuição que mais cresce dentre as distribuições de MPI. Esse crescimento se dá principalmente ao seu constante desenvolvimento e manutenção por um consorcio de parceiros de pesquisa, academia e indústria. Sua versão é resultado da união de três implementações anteriores: FT-MPI, LA-MPI e LAM-MPI. Utilizado por alguns dos computadores da TOP500[35], essa distribuição é considerada a mais importante distribuição open source do MPI na atualidade.

3.3.2 LAM/MPI

O LAM/MPI é a distribuição da biblioteca MPI Ohio Supercomputer Center, sua principal característica se comparado às demais distribuições do MPI é o suporte à criação dinâmica de processos. Essa funcionalidade nos permite que em tempo de execução processos MPI possam criar processos filhos, não necessariamente sobre a mesma aplicação, que podem se comunicar com os demais processos do job.

Esse modelo é fundamental para o modelo de execução adotado no EasyGrid AMS para aproveitar ambientes dinâmicos e heterogêneos. Por isso, essa versão do MPI foi a escolhida para desenvolvimento do EasyGrid AMS.

3.4 Sistemas de Gerenciamento

Desenvolver para um ambiente computacional de alto desempenho composto por recursos de infraestrutura, adaptando-se as características do ambiente é uma tarefa difícil. O desenvolvimento da aplicação torna-se uma tarefa complexa, pois o desenvolvedor precisa preocupar-se em adaptar a sua implementação, em alguns casos, a própria solução ao ambiente.

Além disso, nos ambientes computacionais distribuídos e compartilhados, geralmente a aplicação sofre algum tipo de monitoração/controle de acesso aos recursos. Tanto controle espacial (hosts utilizados) quanto temporal (total de tempo por qual utilizara o recurso), parte ou todo o recurso é alocado para uma aplicação durante um determinado tempo. Consequentemente, se faz necessário o gerenciamento das aplicações no ambiente.

Um Sistema Gerenciador é uma ferramenta que forneça serviços à aplicação do usuário, transparecendo a complexidade para manipular um determinado componente do

ambiente. Além de gerenciar a aplicação no uso do sistema distribuído, tais sistemas podem ser vistos como uma camada, situada entre a aplicação e o sistema operacional, facilitando a execução da aplicação e seu gerenciamento.

Dentre os serviços usuais que esses sistemas costumam oferecer podemos citar: escalonamento de tarefas, tolerância a falhas e medições de desempenho. No processo de gerenciar a aplicação, o acesso ao recurso é comumente controlado por um componente chamado de *broker* central. Este componente fica responsável por fazer a gestão do acesso, gerenciando diversos *jobs* existentes. Sendo as mais comuns:

- **Controle de aceite de aplicações:** Define se a grade tem recursos disponíveis o suficiente para aceitar aquela tarefa.
- **Alocação de Recursos:** Define quais recursos serão alocados para quais *jobs*, através do enfileiramento dos *jobs*.

Os sistemas de Gerenciamento para Grades podem ser classificados, segundo [3], em três tipos: Sistema Gerenciadores de Recursos (RMS), Sistema de Gerenciadores de Usuário (UMS) e Sistemas Gerenciadores de Aplicação (AMS).

Sistemas Gerenciadores de Recursos (RMS) têm o objetivo de maximizar a utilização do sistema, independente dos requisitos dos *jobs*. Tipicamente são sistemas centralizados e possuem uma visualização global dos recursos. O Broker central é responsável por gerenciar os processos de todas as aplicações que estão ou vão entrar em execução. Nesse sistema, a gerência é mais fácil devido ao conhecimento de todos os processos em execução por parte do escalonador. Entretanto, as informações obtidas sobre a execução dos processos são mais limitadas se comparadas aos demais sistemas gerenciadores (visto que essa ferramenta se encontra em um nível fora da aplicação, e precisa gerenciar vários processos), o que prejudica algumas tomadas de decisões do gerenciador, tais como decisões do escalonamento de tarefas. Esses sistemas representam a maior parte dos sistemas gerenciadores existentes no mercado

Sistemas Gerenciadores de Usuário (UMS) são sistemas de gerenciamento descentralizado sendo responsável por gerenciar somente uma aplicação de usuário por vez. Possui como pontos negativos a necessidade de estar instalado em todas as máquinas do ambiente. Ferramentas dessa categoria requerem que o usuário especifique as necessidades da aplicação para seu funcionamento.

Sistemas Gerenciadores de Aplicação (AMS) são sistemas onde a gerência é descentralizada, sendo um único gerenciador por *job*. As ações executadas pelos gerenciadores são realizadas mediante as necessidades particulares do *job*. Esses sistemas gerenciadores são acoplados à aplicação na forma de uma biblioteca e atuam como um middleware que pode inclusive ajustar a aplicação durante sua execução com o intuito de melhorar a utilização dos recursos disponíveis. .

3.5 Resumo

Nesse capítulo apresentamos os três principais modelos de sistema distribuído conhecidos, que são os sistemas computacionais no qual nosso trabalho tem como possíveis ambientes de execução. Apresentamos as duas principais ferramentas de programação distribuída existentes, uma para memória compartilhada (OpenMP) e outra para programação distribuída (MPI). Dando um foco maior na ferramenta de programação distribuída (MPI) que é a ferramenta mais comum nos sistemas distribuídos atuais e a utilizada neste projeto. Mencionamos a principal característica que é o diferencial da distribuição LAM/MPI[34], a criação dinâmica de tarefas, que é a versão utilizada no projeto.

Por fim apresentamos e classificamos os sistemas de gerenciamentos existentes. Da abordagem do clássico RMS que representa a maioria das ferramentas de gerenciamento de sistemas distribuídos ao AMS que é o sistema de gerenciamento o qual nossa proposta se baseia e utilizamos para propor como estratégia mais viável para criação do ambientes distribuído de execução.

Capítulo 4– Middleware EasyGrid AMS

Desenvolvido pelo SGCLab da UFF, middleware *EasyGrid Application Management System* é responsável por controlar a execução de uma aplicação MPI em ambientes distribuídos e compartilhados com recursos heterogêneos[36]. Uma vez que o EasyGrid AMS foi compilado e executado junto à aplicação alvo (aplicação que tenha sido paralelizada com MPI), ele prove um conjunto de serviços para otimizar e facilitar a execução, transformando-a em uma aplicação autônoma.

Os principais serviços oferecido pelo EasyGrid AMS são: alocação estática de tarefas, realocação dinâmica de tarefas, tolerância a falhas e monitoramento da aplicação. Cada um desses serviços funciona de modo transparente na aplicação do usuário, reduzindo assim a complexidade do código para o desenvolvedor e permitindo que o utilizador se preocupe exclusivamente com o problema.

4.1 Contexto

O projeto EasyGrid AMS foi iniciado em [5], desde então já sofreu varias melhorias e atualizações que lhe permitiu sua rápida evolução no provisionamento de serviços a aplicações dos usuários. A cada recurso adicionado ou experimento realizado com a ferramenta, novos artigos acadêmicos foram produzidos, [37] e [38].

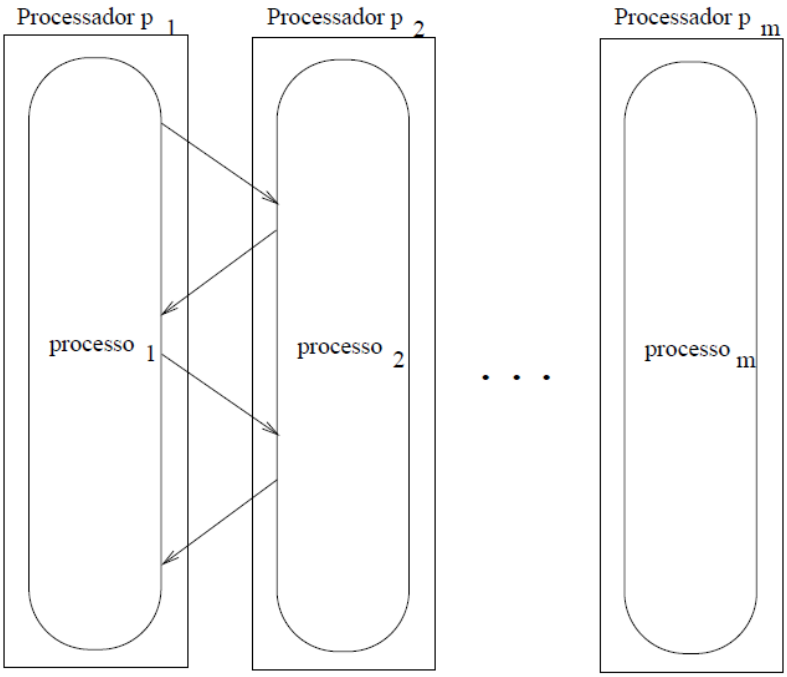

Figura 4.1: Modelo 1PProc

Fonte: [37], p.24

Normalmente, uma aplicação MPI é paralelizada de modo estático. Criando-se um número de processos equivalente ao número de cores disponíveis no ambiente. Este modelo é referenciado em [39] como "Um processo por processador – (1PProc)" e ilustrado na Figura 4.1. Embora essa prática seja de simples aplicação e bastante utilizada, existem alguns pontos negativos nessa abordagem: os processos da aplicação do usuário comumente não são CPU intensivo, isso porque praticamente toda a aplicação do mundo real precisa que alguma entrada de dado via I/O. O que significa que não usam o processador por todo o tempo que foi alocado (normalmente realizando operações de I/O) e a divisão igualitária da massa de dados de entrada não garante que o custo computacional será dividido igualmente. Esses fatores contribuem negativamente para a execução dos processos, gerando desperdício no uso do CPU e o aumento no tempo total da aplicação.

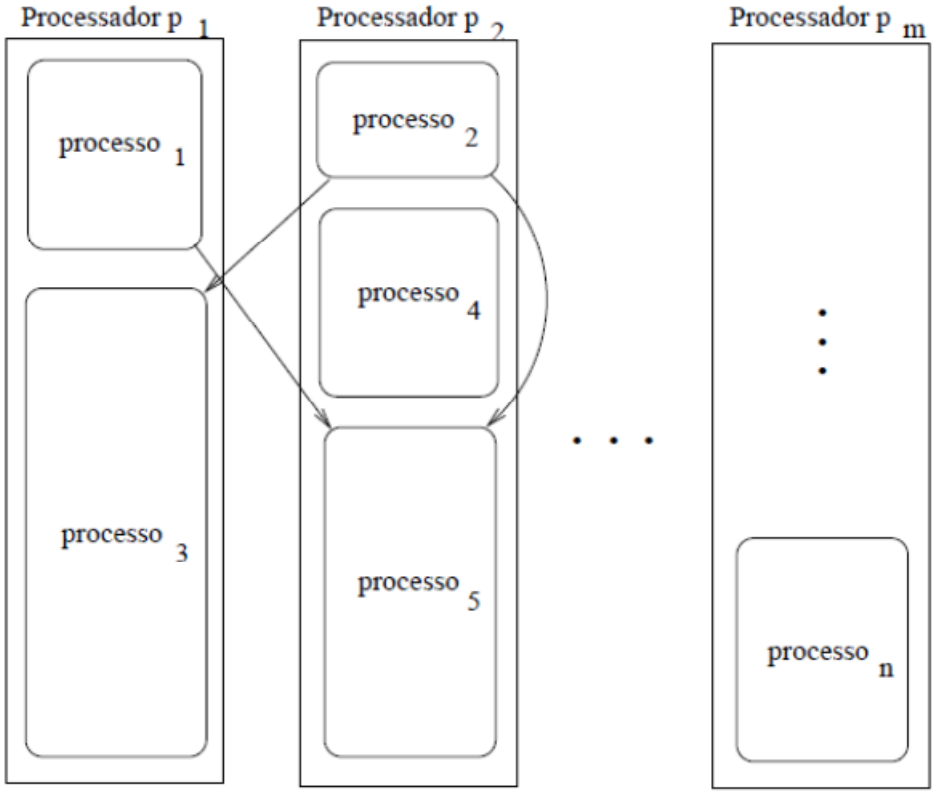

Figura 4.2: Modelo 1PTask Fonte: [40], p.28

O EasyGrid, por sua vez, utiliza de uma abordagem diferente da mencionada anteriormente: a alocação das tarefas é realizada por um modelo nomeado de "um processo de aplicação por tarefa - (1PTask)", que pode ser visto na Figura 4.2. Nessa abordagem a paralelização das tarefas é realizada baseada na granularidade da aplicação, gerando um número de tarefas independentes, geralmente muito maior, do que o número de CPU ou núcleos existentes. Isso permite que as tarefas da aplicação sejam melhor distribuídas entre os recursos, o que proporciona uma melhor utilização dos mesmos.

Uma aplicação paralelizada e executada no EasyGrid, com a realocação dinâmica ativada, é composta de diversas tarefas de granularidade pequena. Cada tarefa pode ser realocada em qualquer um dos recursos, ao longo da execução, de forma que as cargas da aplicação mantenham-se sempre balanceadas entre os recursos dentro de um limiar de erro (com visibilidade somente para as tarefas da aplicação). Cada tarefa quando iniciada já possui suas dependências satisfeitas, não dependendo de mais nenhuma outra tarefa para concluir sua execução, e após a iniciar a execução não será interrompida pela middleware até sua finalização, salvo caso de falha no sistema e cancelamento do *job*.

4.2 Modelo de representação das aplicações

A representação das aplicações paralelas executadas no EasyGrid é realizada através do modelo de GAD (Grafo Acíclico Direcionado). Que pode ser definido como, um GAD G é representado formalmente como $G= (V,E,\varepsilon,\omega)$, onde V representa um conjunto de vértices, E um conjunto de arestas direcionadas, tal que $\forall e \in E$, $e=(v_1,v_2)$ onde $v_1,v_2 \in V$, $v_1 \neq v_2$, $(v_1,v_2) \neq (v_2,v_1)$. ε é a função custo dos vértices, assim $v1 \in V$, $\varepsilon(v_1)$ é igual ao custo associado ao vértice v_1, por último, mas não menos importantes, ω é a função custos associados a uma aresta. Onde, $e \in E$, temos que $\omega(e)$ é o custo associado à aresta e. Na Figura 4.3 temos o modelo gráfico de um GAD.

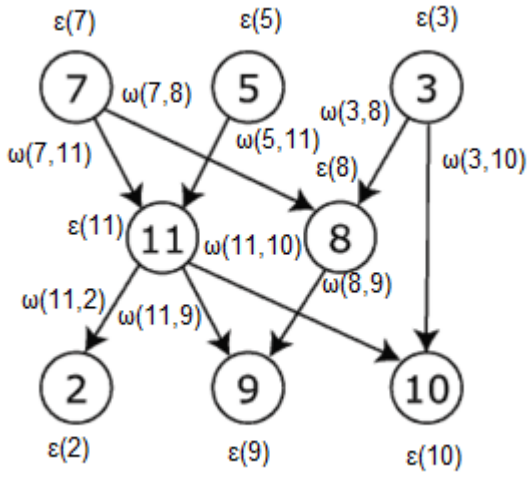

Figura 4.3 Exemplo gráfico de um DAG

Nesse modelo, as tarefas são associadas aos vértices e a comunicação às arestas. As funções peso do vértice ε e peso da aresta ω representam respectivamente a quantidade de trabalho de cada tarefa e volume de dados de cada comunicação. Nas aplicações modeladas com o GAD é comum que as tarefas possuam dependências em relação a outras tarefas para a

execução. O que torna a execução mais complexa, pois essa dependência de dados é um fator a mais a se preocupar no processo de escalonamento.

Neste trabalho, iremos lidar exclusivamente com uma classe de GADs onde as tarefas não possuem dependências entre si. Esse subconjunto de GADs representa uma classe de aplicações conhecida como Bag-of-task (BoT).

Aplicações BoT, também conhecidas como aplicações mestre-trabalhador, são aplicações no qual os processos não possuem dependências de dados entre si, como ilustrada na Figura 4.4. Sendo assim podem ser executados na ordem e agrupamento que for mais conveniente. Nessas aplicações, cada trabalhador recebe um conjunto de dados de entrada, processa e em seguida devolve para a tarefa mestre, para novamente iniciar o ciclo. Esse ciclo é repetido até que não se tenham mais tarefas a serem executadas.

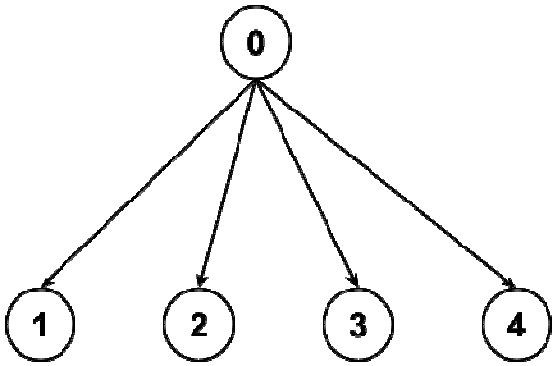

Figura 4.4 Exemplo gráfico de um BoT

4.3 Estrutura do EasyGrid AMS

O EasyGrid AMS possui sua estrutura organizada em três níveis hierárquicos. Na parte mais inferior da hierarquia encontra-se o gerenciador de máquina (GM) que é responsável por gerenciar a listas de processos prontos para execução, disparando os processos alocados e controlando o número de processos concorrentes em execução. No nível imediatamente acima, encontra-se o gerenciador de sites (GS) que é o responsável por gerenciar um conjunto de hosts locais. Nesse nível, o gerenciador pode balancear os processos entre os hosts baseado nas informações provenientes do nível inferior. No topo dessa hierarquia se encontra o gerenciador global (GG) que é o responsável por gerenciar toda a execução do sistema e que pode fazer o balanceamento das cargas e a tolerância a falha entre os diversos sites. Esse nível se comunica única e exclusivamente com o nível imediatamente inferior, sendo ainda o nível mais alto da hierarquia como descrito na Figura 4.5.

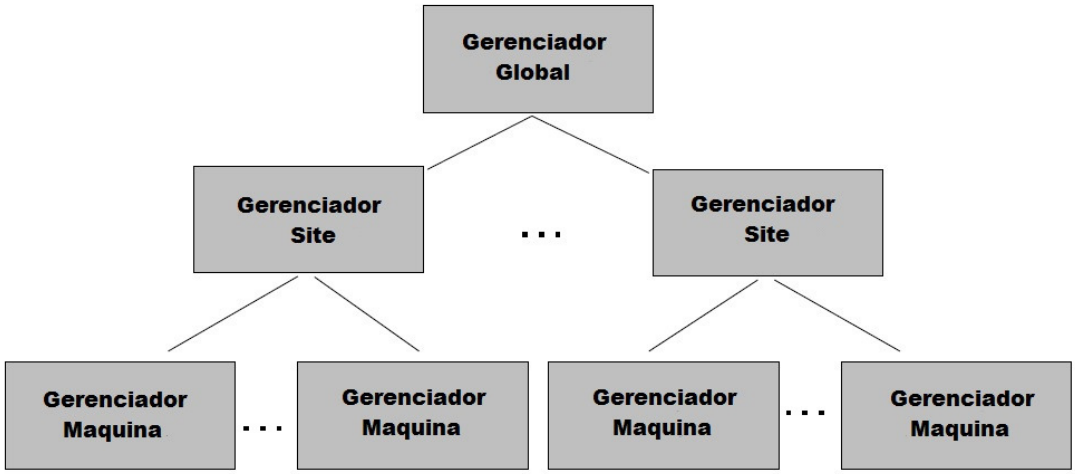

Figura 4.5 Estrutura hierarquia do EasyGrid AMS

Fonte: [41*], p.20*

O escalonador dinâmico sobre a ótica da estrutura hierárquica pode ser descrito: possui seu funcionamento iterativamente, monitorando todos os gerenciadores no nível de maquina(GM) e calculando a previsão de termino das sua fila de tarefas. O gerenciador do nível acima(GS) com base na previsão de término de cada host redistribui as tarefas do GM para que as previsões de término fiquem próximas, com tempos GM balanceados. O mesmo processo é realizado no nível acima entre GG que redistribui as cargas entre os sites(GS) para que sua previsão de termino fiquem balanceadas.

Para melhor compreensão do funcionamento dos componentes do EasyGrid AMS e a comunicação entre os diversos mecanismos, descreveremos seu funcionamento sob a perspectiva de camadas (Figura 4.6). O fluxo inicia-se pela camada de monitoramento da aplicação que é o componente responsável por coletar informações sobre os processos MPI que foram executados e que estão em execução. Essas informações são utilizadas pelas outras camadas do sistema para realizar a tomada de decisões apropriadas para a efetividade da aplicação autônoma: tal como a camada de escalonamento dinâmico, que uma vez habilitada, pode tomar decisões gerenciais de realocação das tarefas nos recursos para conseguir uma melhor performance, ou a camada de tolerância a falhas que verifica se um processo está executando corretamente, sem a existência de problema na sua execução, reiniciando o processo caso necessário sem a necessidade de parar a execução da aplicação. A camada de gerenciamento de processo é responsável por criar/redirecionar de mensagens entre os processos MPI da aplicação e pelo escalonamento estático das tarefas.

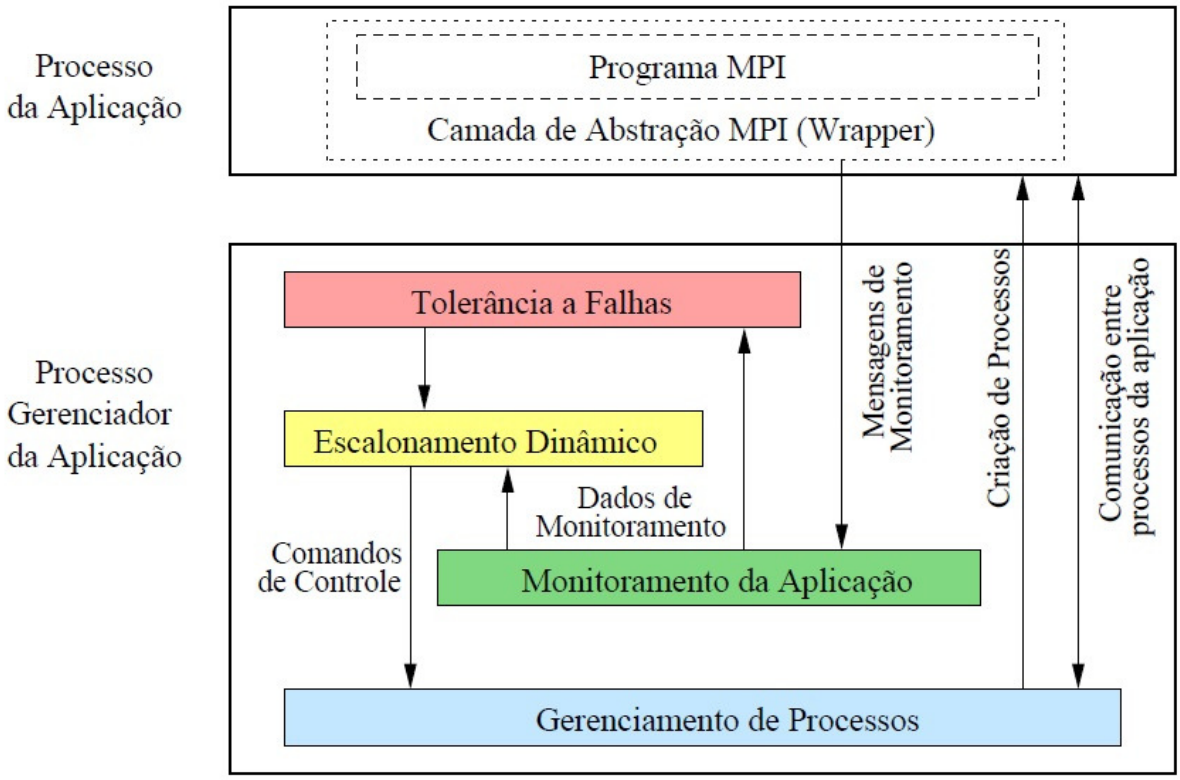

Figura 4.6 Estrutura em camadas do EasyGrid AMS

Fonte:[41],p.21

A versão atual do EasyGrid possui três configurações fundamentais no controle de execução das tarefas. São elas:

- Política de escalonamento de tarefas: existem duas possibilidades de políticas de escalonamento de tarefas. O escalonamento estático, escalonamento no qual o usuário define previamente a execução onde as tarefas serão alocadas e a ordem de execução. Escalonamento dinâmico, escalonamento no qual o "monitoramento de aplicações" coleta dados sobre a execução das tarefas para prever o término das filas de cada um dos host e realoca as tarefas para que o término das filas sejam em tempos próximos, reduzindo a ociosidade dos recursos.
- Política de utilização do Host: Essa política define qual deve ser o critério do middleware na hora de disparar a criação das tarefas. Existem duas possibilidades de políticas: PER onde o EM dispara a criação das tarefas em um determinado host somente se o mesmo não estiver sendo utilizado por nenhuma outro job. Utilizada quando desejamos realizar uma reserva do host para ser o único a possuir o recurso e útil para evitar possíveis interferências na execução do job, entretanto sua utilização fica sujeito à criação de ociosidade no recurso. PEI política onde o EM dispara a criação das tarefas independente se

existe outra aplicação utilizando o host. Essa abordagem é mais apropriada quando desejamos utilizar melhor os recursos dos hosts, entretanto requer uma gerência maior na criação dos jobs, para evitar a criação de sobrecargas nos hosts.

- Vazão do Host: Um dos principais parâmetros na execução com o EasyGrid é definir a vazão das tarefas ou grau de concorrência no host realizado pelo EM. Esse parâmetro é definido estaticamente no middleware e geralmente é associado ao número de núcleos no host.

4.4 Resumo

Neste capítulo, apresentamos o EasyGrid AMS, que é a ferramenta utilizada na tentativa de criar o ambiente distribuído para a execuções aplicações distribuídas almejado. Um ambiente onde os jobs executem compartilhando os recursos paralelamente sem a existência de um limitador central que impeça que as aplicações consigam tirar o máximo do ambiente.

O middleware EasyGrid AMS nos fornece um conjunto de serviços que tornam a aplicação autônoma. Sendo capaz de adaptar-se ao ambiente para maximizar a utilização dos servidores na execução minimizando o makespan. Entretanto seu comportamento natural é guloso e possui poucos mecanismos na execução concorrente de jobs.

Na proposta apresentada nesse projeto nos preocuparemos com a parte mais baixa do fluxo hierárquico da aplicação, o Gerenciador de Maquina (GM). Uma aplicação EasyGrid MPI possui nesse nível um controle meramente estático da vazão das tarefas. Uma vazão previamente estabelecida pelo usuário que será priorizada/executada unicamente pelo escalonador do SO. Que embora nas versões atuais dos sistemas operacionais mais robustos, como Linux, sejam bem eficientes. O conhecimento sobre priorização apropriada da tarefa distribuída por parte do S.O é completamente desconhecida. Nossa proposta é adicionar inteligência nas decisões nesse ponto da aplicação para que ela possa passar a controlar a vazão de modo dinâmico e apresentar ganhos em termos de desempenho e utilização, ajustando assim à vazão mais apropriada para tirar o máximo dos recursos, manter o controle sob a execução dos jobs e permitir que os demais jobs também possam executar suas tarefas.

Nesse novo ambiente de execução ao qual almejamos, queremos que os jobs tenha um comportamento mais sociável e menos guloso. Buscamos um ambiente onde as aplicações tenham comportamentos altruístas.

Capítulo 5 – Política Sociedade Altruísta

O objetivo desse capítulo é descrever o modelo de funcionamento da nossa heurística, juntamente com sua implementação que utiliza como base o EasyGrid. Ao longo do capítulo apresentaremos uma comparação entre a versão clássica do EasyGrid e as modificações realizadas nesse projeto. O modelo da aplicação proposta, os fundamentos da nossa proposta, os mecanismos que compõem nossa camada de controle de vazão, e os diversos cenários que queremos cobrir.

5.1 Visão Geral

Um *job* que esteja executando em uma grade com *broker central* possuirá um modelo de execução semelhante o representado na Figura 5.1. Neste modelo de execução, o qual se aplica reserva de recursos, no momento da submissão do *job*, o *broker central* irá definir se poderá ou não atender àquela solicitação, e em caso positivo, enfileirará o *job* até que chegue a sua vez de executá-lo. Alocando para si os recursos solicitados no momento da sua submissão.

Entretanto, sabemos que a reserva de recurso não é a melhor opção quando desejamos maximizar a utilização do ambiente computacional, pois nessa abordagem é comum a criação de ociosidade, isso porque os jobs não usam todos os recursos por todo o tempo que foi alocado para si.

Neste cenário, o *broker central* faz o papel de alocar os recursos de maneira exclusiva reservando parte do recurso, ou todo, para o *job* baseado nos parâmetros de *host* informados na submissão. Esses parâmetros informados pelo usuário geralmente refletem a execução do *job* no ápice da execução, reservando um número de *hosts* que serão utilizados somente no pico da execução e sob o pior cenário possível. Este acréscimo no número de *hosts*, que será utilizado na média da execução, resultará em ociosidade. Pois neste modelo de execução, os recursos são reservados exclusivamente para si, cabendo ao *job* definir qual é a melhor forma de usá-los. Deixando a execução do *job* dependente da eficiência da alocação das tarefas nos recursos reservados, o que geralmente é uma tarefa difícil (NP-Completo), uma vez que a própria aplicação terá que fazer a implementação que será responsável por realizar essa tarefa. Consequentemente, a própria execução da fila do broker central no ambiente fica prejudicada, pois uma lentidão na execução dos *jobs* contribuirá negativamente para a liberação do recursos utilizados e, consequentemente, a execução dos *jobs* subsequentes na fila.

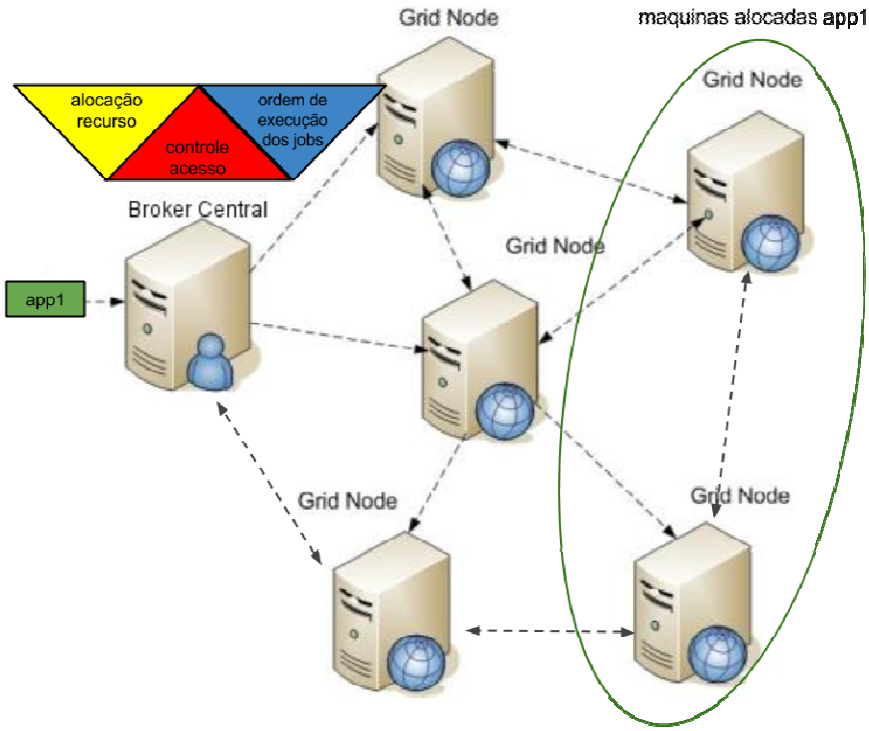

Figura 5.1 Aplicação em um ambiente distribuído com um broker central

O cenário citado poderia ser facilmente melhorado com a simples adição do EasyGrid AMS à aplicação, que dentre os muitos serviços que fornece a execução do job, um deles é otimizar o balanceamento das tarefas entre os recursos alocados, acelerando o término da execução, e consequentemente, a liberação dos recursos alocados para demais *jobs* na fila (Figura 5.2).

Entretanto, mesmo com um bom escalonamento é inevitável ter alguma ociosidade quando levamos em consideração as características da aplicação. Isso porque aplicações diferentes tendem a usar recursos diferentes em quantidades diferentes. Assim, mesmo um bom balanceamento não conseguiria utilizar todo o recurso alocado para ele durante todo o tempo o qual foi alocado. Um exemplo é uma aplicação *I/O Bound*. Este é um caso onde o desperdício de processamento durante sua execução é inevitável quando executado com alocação de recursos de modo restrito ou exclusivo em um sistema distribuído.

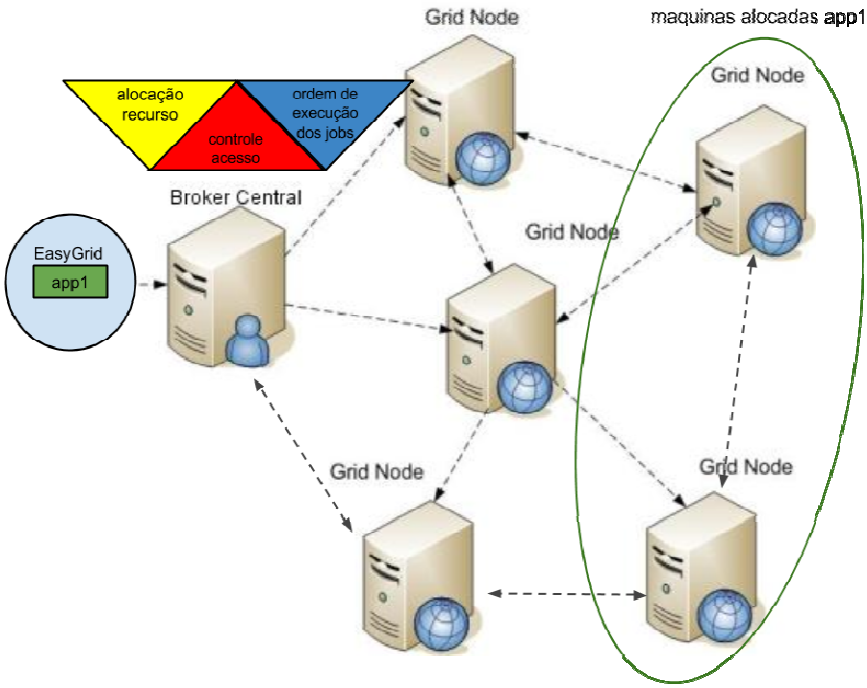

Figura 5.2 Aplicação EasyGrid no Ambiente distribuído com um broker central

Uma possível solução para o problema citado seria desabilitar a execução restrita realizada pelo *broker central*. Permitindo que o mesmo seja capaz de alocar múltiplos *jobs* em recursos comuns do sistema distribuído. Entretanto, esta abordagem requereria muito mais do gerenciamento dos *jobs* que um *broker centra*l consegue oferecer. Isso porque, a execução compartilhada requer que os *jobs* sofram um gerenciamento muito maior da utilização de recursos por parte do escalonador para que um job não atrapalhe drasticamente a execução de outro. Como por exemplo, monopolizando os recursos. Esse nível de controle e detecção está além das informações que são obtidas por um RMS. No geral, essa classe de escalonadores é limitada em informações sobre os *jobs*, bem como possibilidades de ajustes que podem realizar na execução dos *jobs*. Para tal controle, seria necessário um escalonador que executasse em um nível muito mais próximo dos *jobs*, conseguindo não só obter informações mais especificas da execução mas também ajustar sua execução ao ambiente para que consiga tirar o melhor proveito dos sistemas distribuídos disponíveis. Nossa proposta é aumentar o poder gerencial da aplicação. Assim como o EasyGrid AMS já permite que as aplicações gerenciem-se, permitindo conseguir um bom balanceamento dos seus processos, vamos estender suas funcionalidades, permitindo que possam também:

- Decidir-se pelo compartilhamento ou não hosts entre si;
- Definir quanto de cada recurso devem utilizar;
- A prioridade ou ordem parcial de execução entre eles.

Tudo isso de maneira intrínseca e reduzindo o papel do *broker central* a somente o aceite ou não da submissão do *job* ao sistema distribuído (Figura 5.3).

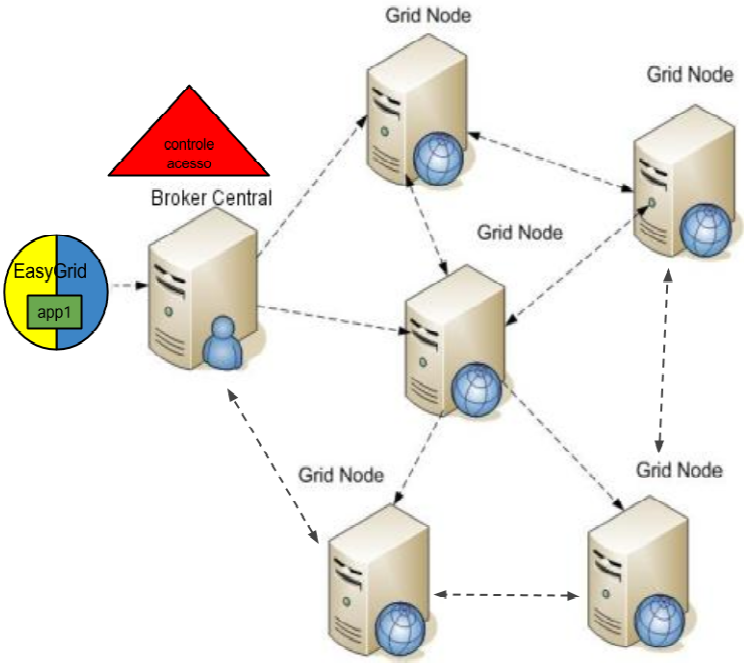

Figura 5.3 Aplicação no ambiente distribuído com middleware EasyGrid SA

5.2 Modelo das Aplicações

Nesse trabalho seguimos o mesmo modelo de execução proposto em [41], onde as aplicações MPI são executadas com o *middleware* EasyGrid AMS em um ambiente distribuído compartilhado. Consideramos que este ambiente de execução é composto de um conjunto de recursos (que podem ser usados para formar um *cluster*, *grids* ou *clouds*) que é compartilhado simultaneamente por um conjunto de *jobs.*

Nosso problema pode ser descrito como: executar um conjunto de jobs que têm uma meta de execução associada a eles no momento da submissão. As metas podem ter duas conotações diferentes e são especificadas no *job*. Metas do tipo *hard* são idênticas a *deadlines*, enquanto metas *soft* considera o tempo como desejável, sendo aceito possíveis atrasos. Ambos tipos de metas são apresentados como duração de tempo, incluindo o tempo de espera na fila e independente dos recursos disponíveis. O escalonamento desse problema é dinâmico, podendo novos *jobs* serem submetidos a todo tempo. Os objetivos a serem otimizados são: finalizar os jobs dentro da meta, minimizar a ociosidade do recurso e minimizar o *makespan*.

Em nosso modelo de execução, cada usuário poderia submeter sua aplicação acoplada ao *EasyGrid AMS* juntamente com uma meta de tempo no qual pretende conseguir o

resultado, e o *job* guiaria-se através dessas metas para definir a estratégia de escalonamento dinâmico. Entretanto, pensando diretamente em aplicações que têm ou não características de *real time*, criamos um modificador para o que definimos como as duas classes de meta: *soft* e *hard*. Aplicações com meta *hard* precisam terminar sua execução dentro do tempo definido e devem ter prioridade durante a execução. Enquanto aplicações com metas *soft* desejam terminar naquele tempo, sendo que atrasos serão toleráveis na obtenção do resultado. O principal motivo para existência da meta é sua representatividade para os modelos de ambiente distribuídos que prevalecem atualmente. Por exemplo, se consideramos as nuvens, que são ambientes onde é possível encontrar cobraças *on-demand* pelo tempo de alocação dos recursos e também contratos prévios de alocação[4], poderíamos modelar nossas restrições associando a meta a um pacote de tempo previamente acordado pelo usuário que irá executar sua aplicação. Como nem sempre é conhecido o tempo de execução exato de sua aplicação (além das possíveis oscilações na execução), o contrato teria que ser realizado com um tempo acima o provável tempo gasto. Assim, temos o primeiro dilema, quanto maior for o tempo, mais caro sairá o custo da execução, mesmo que a aplicação termine antes do contratado. Porém, se o tempo for pequeno, pode não ser o suficiente para executar toda a aplicação. O que fazer nesse caso? Embora não exista um padrão e o comportamento possa variar de prestadora para prestadora, o serviço mais restrito poderia simplesmente cancelar a execução da sua aplicação (modelo meta *hard*), o que pode acarretar na reexecução da aplicação e mais gastos. Já o prestador mais maleável poderia permitir a execução após o período previamente contratado, mas com a adição de custos extras. O que pode possuir um custo de hora superior à hora do contrato inicial.

Este modelo de execução apresentado, que não é um modelo distante da realidade dos provedores de computação em nuvem [4], [24], [42] (em especial o modelo utilizado pela Amazon EC2 na modalidade instancia reservada), poderia ser facilmente modelado em nossa proposta. Através da definição de uma meta e uma restrição a aplicação definindo se poderá continuar ou não após perder seu prazo.

5.3 Proposta Sociedade Altruísta – Gerenciamento sem gerente

Nessa seção, descreveremos o funcionamento das heurísticas que compõem a sociedade altruísta e sua integração no EasyGrid AMS. O compartilhamento de recursos de forma apropriada é a principal proposta deste trabalho. Nosso objetivo é permitir que aplicações possam tirar o máximo proveito dos recursos de uma infraestrutura através do compartilhamento de recursos e ações altruístas entre os *jobs*. Isso obviamente, sem prejuízos

às metas e nenhum aumento exagerado na gerência das aplicações. Podemos até especular que esta proposta reduz a complexidade da questão de gêrencia e escalonamento das aplicações e torna a solução mais escalável do atual modelo de gerência.

Nossa proposta tem como alvo de execução os grandes ambientes computacionais, como *grids*, *clusters* e *cloud* onde existam diversas aplicações sendo submetidas para execução. Tradicionalmente, nesse contexto, existe um *broker central* que irá fracionar a infraestrutura e delegar partes dos recursos por períodos de tempos aos *jobs*. Embora a proposta clássica seja considerada, por muitos, funcional e atenda a praticamente a todos os ambientes computacionais existentes, ao considerarmos a performance da execução, essa proposta contém algumas limitações, como por exemplo: (i) Alocar as aplicações nos recursos é uma tarefa difícil, por isso raramente a melhor alocação será alcançada pelo escalonador; (ii) Aplicações podem não usar toda capacidade dos recursos alocados por todo o tempo, mas mesmo assim as requisições ao *broker* são realizadas sempre levando em consideração o pico da execução (o que pode ser muito superior à média da utilização); (iii) Aplicações possuem características diferentes, o que implica em um possível uso de componentes diferentes de um mesmo recurso compartilhado. Sendo assim, duas ou mais aplicações com características distintas (por exemplo, uma *I/O bound* e outra *CPU bound*) poderiam coexistir sem causar prejuízos em termos de desempenho a ambas.

Levando em consideração essas características, vamos definir nosso modelo de execução. O *Ambiente Computacional Compartilhado* utilizado é um ambiente onde existam várias aplicações em execução sem reserva de recursos. O esquema de gerenciamento utilizado neste modelo possui o comportamento diferente do usual: primeiramente, o *broker* central não tem a mesma inteligência e nem a complexidade de um broker clássico, sua função se limita a aceitação de um *job* para execução no ambiente, baseado em seus requisitos de tempo e recursos. Uma vez aceita, a aplicação poderá ganhar todos ou parte dos recursos solicitados mesmo que outras aplicações já estejam utilizando os recursos. A aplicação por sua vez, passa a se gerenciar, graças a adição de um middleware (o EasyGrid AMS com a inclusão da nova política Sociedade Altruísta SA) que terá informações de execução da aplicação em tempo real e modificará seu comportamento para utilizar a quantidade de recursos necessários para terminar sua execução dentro de sua meta.

A proposta do algoritmo Sociedade Altruísta é que a vazão dos processos que cada GM (Gerenciador Máquina) de uma aplicações em execução com o EasyGrid AMS possa ser alterada em tempo de execução, para um valor que seja mais apropriado para sua execução

naquele momento. Porém, a grande dificuldade é definir a tomada de decisão de acordo com o estado que a aplicação se encontra.

Relembrando que em nosso problema temos um conjunto de servidores (que podem ser virtuais ou físicos) que são utilizados de forma concorrente por um conjunto de aplicações. Onde cada aplicação é uma aplicação paralela, sendo assim podendo (e geralmente irá) utilizar vários recursos simultaneamente. Logo, queremos definir as cargas que cada aplicação deve executar em cada nó. Esse ajuste na vazão é realizado alterando-se os números máximos de processos que os GMs (Gerenciador Máquina) do EasyGrid AMS podem submeter para execução simultaneamente em cada host (o que consequentemente representará a fatia de processamento que cada aplicação ganha daquele host) tal que a aplicação consiga principalmente atingir sua meta, apesar da concorrência, mas ainda permitir que as outras aplicações, caso existam, possam terminar também dentro de suas metas.

Um importante detalhe do funcionamento da nossa heurística é que ela aproveita-se da eficiência da redistribuição dinâmica de cargas, existente entre os diversos níveis do EasyGrid, para reduzir ou aumentar a vazão da aplicação baseado na vazão de cada um dos GM. Onde cada um deles concentra-se no seu escopo local para verificar qual é a sua previsão de conclusão das tarefas. Como o reescalonador dinâmico encontra-se ativo, ele utiliza-se dessa previsão para realocar tarefas entre os demais GM do ambiente, fazendo com que todos os GMs tenham previsões de término próximas. Esse comportamento nos permite concentrarmos especificamente no nível mais baixo, pois os resultados ali obtidos serão automaticamente distribuídos entre os demais hosts e transmitidos para os demais níveis da aplicação.

Para a realização desse objetivo dividimos o processo "ajustar a vazão da aplicação no host" em duas etapas, o processo de "*estima previsão término*" e de "*ajuste da vazão baseado na previsão término*".

5.3.1 Estimativa previsão término

Ao longo da execução de um *job* encapsulado com o EasyGrid, a camada de monitoramento obtém inúmeros dados sobre a execução. Esse conjunto de dados não só contem informações sobre o estado da execução do *job*, como também obtém dados dos hosts utilizados. Nosso monitoramento no projeto concentra-se essencialmente na camada mais baixa da hierarquia. Onde a cada um dos EM do *job* executado com EasyGrid adicionamos um conjunto de

monitores que obtêm dados das tarefas e do *host* o qual está executando. Estimando a previsão de término daquele conjunto de tarefas sobre *host*.

Como dependemos diretamente da precisão das heurísticas no EasyGrid para conseguir realizar uma previsão acurada de término, nesse projeto alteramos e criamos algumas funções internas de medições para melhorar a precisão os dados obtidos sobre o ambiente de execução. Sendo que as principais informações que queremos obter do host durante a execução são:

- *Percentual total de carga do host em um determinado instante*: Através do *System Activity Reporter*[43] (SAR), o EasyGrid AMS obtém o consumo de CPU instantâneo do *Host*, calculando a média dos consumos dos cores.
- *Percentual que a aplicação ganha do host em um determinado instante*. Este percentual é obtido de duas maneiras diferentes: *a primeira forma* é obtida ao termino de uma das tarefas da aplicação. Dados da execução daquele processo são obtidos pelo EasyGrid AMS. Dentre eles, o percentual de processamento que a tarefa ganhou durante sua execução. *A segunda forma* é uma alternativa para o caso no qual a aplicação fica um tempo longo sem que nenhuma tarefa termine. Neste caso, é realizada uma consulta no status de execução da tarefa, localizado no sistema de arquivos em *"/proc/[pid]/stat"* que trata-se de um pseudo file system do sistema que interfaceia a comunicação com o kernel do S.O. Essa forma de consultar ao status do processo é utilizada restritamente nessa condição, isso devido ao custo de se realizar a leitura e a imprecisão dos dados que é maior do que da *primeira forma.*
- *Percentual de ociosidade do host em um determinado instante*. O percentual de ociosidade do *host* em um determinado instante é realizado através do calculo "um menos *Percentual total de carga do host em um determinado instante"*.
- *Sou a única aplicação a utilizar esse recurso?* Verificamos se a aplicação é a única a obter o recurso comparando se (*Percentual de CPU que o job ganha do host em um determinado instante é igual ao Percentual de CPU do host no mesmo instante)* dentro de uma margem de erro.
- *Meta/Previsão* trata-se de uma estimativa para prever quão maior é a meta em relação ao tempo previsto de execução. Seu cálculo é realizado dividindo o tempo restante de execução pela previsão de término das tarefas no EM.
- *Pilot Task.* Realizado uma única vez no inicio da execução do *Job*, período no qual a aplicação não tem informação suficiente ainda para determinar o grau de

disponibilidade do *host*. Nesse procedimento executa-se uma única tarefa em cada host trabalhador (tentando minimizar a influencia de cargas externas) e utilizamos a função Meta/Previsão para estimar o tempo ótimo de execução do *job*.

As medições realizadas nessa etapa juntamente com a meta definida para a aplicação são utilizadas para avaliar como o GM (Gerenciador Máquina) irá se comportar para alterar sua vazão do *job*.

Uma das mais importantes métricas para as tomadas de decisão a serem realizadas é o cálculo da estimativa de término. Essa estimativa é realizada através da função *previsão de término* que estima o tempo restante mais provável para o término do *job* naquele instante baseada na última tarefa do mesmo a terminar sua execução. Este cálculo é executado pelo somatório do tempo atual, acrescido do tempo da última tarefa executada normalizada (dividido pelo seu peso) vezes o somatório dos pesos das tarefas restantes vezes o teto do número de tarefas restantes dividido pelo número de cores. Ele é executado individualmente por cada EM e pode ser dado pela função:

$$pt = ta + (t_{lt} / p_{lt}) * \sum_{1}^{n_{rt}} p * \lceil n_{rt} / n_c \rceil \quad (1)$$

Onde pt = Previsão de término; ta = tempo atual; t_{lt} = tempo de execução da última tarefa do *Job* neste *host*; p_{lt} = peso da última tarefa; n_{rt} = número de tarefas restantes do *Job* nesta máquina; p = peso da tarefa; n_c = número de cores. Um detalhe crucial da função previsão de término é que a variável t_{lt}, que representa o tempo de execução da última tarefa, terá um valor indefinido até que a primeira tarefa escalonada neste *host* finalize.

5.3.2 Classificação da aplicação em estados

Uma vez que o EM possui a previsão de término das tarefas que lhe estão alocadas, juntamente com alguns dados adicionais, ele irá identificar o estado mais condizente com sua execução baseado no seu provável término naquele cenário de execução. São eles:

- *Indefinido: Estado transitório até que um primeiro processo do job termine.*
- *Muito adiantado*: se a previsão de término for ***pt < y*meta, onde 0 < y < 1***.
- *Adiantado:* se a previsão de término for ***y*meta < pt < meta, para 0 < y < 1.***
- *No Prazo:* se a previsão de término for coincidir com a meta, dentro de uma margem de erro. ***meta -ε < pt < meta + ε***
- *Atrasado com chances:* se a previsão de término for ***meta < pt < x*meta, para x > 1***

- *Atrasado com poucas chances:* se a previsão de término for ***x*meta < pt. , para x > 1***
- *Atrasado sem chances:* se tempo atual for maior que a meta, ***ta > meta.***

  ∴ Onde, na implementação atual, usamos y = 0.25, x = 2 e $\varepsilon = 0.05$*. Valores obtidos empiricamente através dos testes e experimentos realizados.*

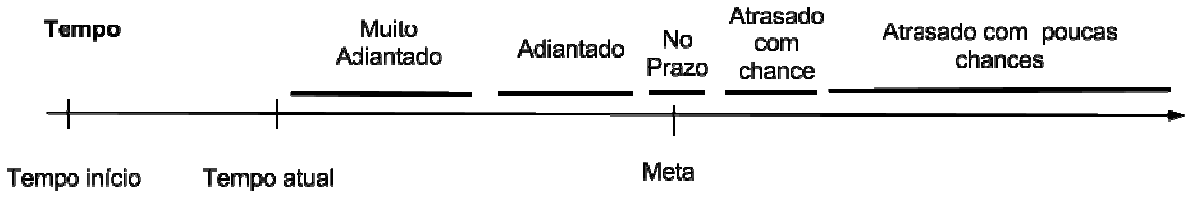

Figura 5.4 Linha tempo estado da aplicação em relação a uma meta soft

A linha do tempo é o principal atributo no auxílio da tomadas de decisão. O mesmo controle que é realizado para processos com meta *soft* é realizado para processos com meta *hard*, entretanto com o controle de estado mais rigoroso na tentativa de atender a meta restritamente. A principal diferença para classificação da meta no tipo *hard* é que eliminamos os estados adiantados para evitarmos que os *jobs* com essa classificação cedam um possível ciclo de processamento que possa contribuir para a perda de sua meta. Sendo assim uma meta está no prazo se ***pt < meta***.

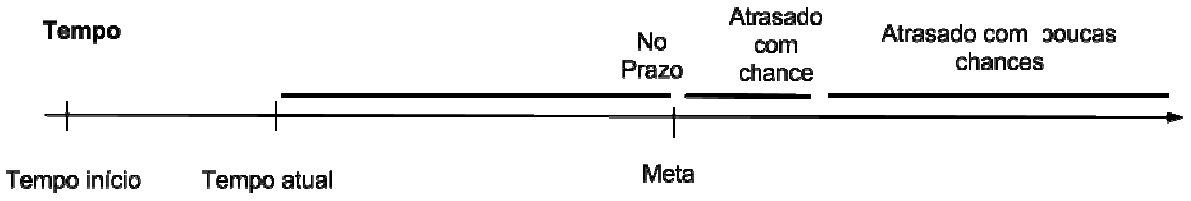

Figura 5.5 Linha tempo estado da aplicação e relação de uma meta hard

Porém, mesmo com esse controle, inevitavelmente algumas aplicações ainda estarão sujeitas a perder suas metas. Mesmo nesses casos, precisamos identificar as perdas das metas para tomar alguma decisão condizente, por exemplo, decidir se a aplicação permanecerá executando até seu término ou terá um *deadline* onde cancelaremos sua execução.

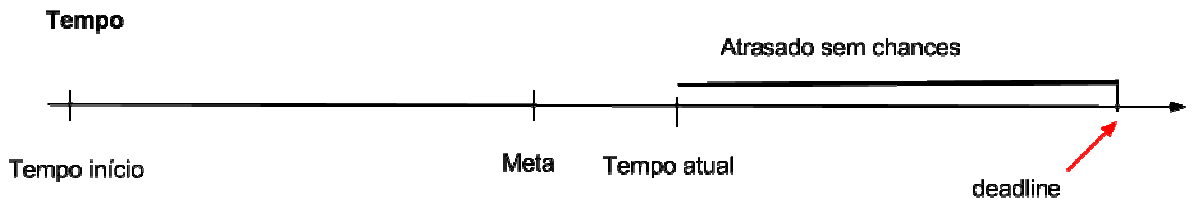

Figura 5.6 Linha do Tempo aplicação que perdeu meta

Com esse conjunto de informações, estamos munidos com os dados necessários para modificar o comportamento da aplicação para conseguirmos finalizar dentro de sua meta.

5.3.3 Ajuste da Vazão

Uma vez que o monitor esteja com informações sobre a execução do *job* obtidas na seção 5.2.2, precisamos agora tomar medidas que permitam que a aplicação execute dentro de sua meta. Nossa implementação do EasyGrid AMS foi modificada para trabalhar com até 6 possíveis estados e três componentes à parte para detectar comportamentos complementares da aplicação que o detector de estado não é capaz de refletir.

Para cada estado proposto será apresentado uma vazão associada ao estado, que representa uma fração da *carga máxima* que será executada no host. Na versão atual da implementação do EasyGrid AMS, a carga máxima é definida como sendo 1,5 vezes o número de cores no host. esta proporção é uma melhoria nos resultados obtidos em [44], pois nos experimentos realizados, no capítulo 6, conseguimos continuar mantendo uma baixa ociosidade do recurso e a redução da proporção, comparado à versão anterior, aumentou o grau de flexibilidade para realocar as tarefas caso seja necessário.

Para cada um dos componentes definidos será explicado sua aplicabilidade e seu escopo que pode ser complementar ao resultado de uma execução ou sobrepor uma medida definida pelo controle de estados.

5.3.3.1 Decisão baseado no estado

Para cada um dos estados já mencionados iremos apresentar ações tomadas sempre que existam execuções concorrentes de *Jobs* nos recursos. As medidas tomadas aqui são para que possamos alterar a vazão dos processos que sempre buscam atingir os objetivos de executar dentro da meta e evitar ociosidade de recursos.

- **Estado Adiantado**: Quando é detectado que o estado momentâneo do *Job* é adiantado, a vazão é alterada para que o *EM* passe a trabalhar em uma carga reduzida. O objetivo dessa redução é de exibir um comportamento altruísta no qual a aplicação reduz seu processamento para uma vazão de ***30%*** da carga máxima inicialmente, podendo reduzir até um mínimo de ***20%*** da carga máxima baseado no uso do CPU do *host*. Este mecanismo é condicionado a: Enquanto o *host* estiver no estado *Adiantado e* o uso de cpu do *host* estiver alto, acima de ***97%*** *de uso,* será reduzido *1%* da fração da carga máxima a cada verificação. Esse mecanismo foi criado para permitir que outros *jobs* mais necessitados possam usar a capacidade de processamento disponível para agilizar sua conclusão. Este comportamento só é realizado com a meta do tipo *soft*, não se manifestando para aplicações com meta *hard.*
- **Estado Muito Adiantado:** Esse estado representa aplicações que encontram-se com previsão de término muito antes de sua meta. Para esse estado, trabalhamos com uma vazão ainda mais reduzida que o estado *adiantado, utilizando* ***20%*** *carga máxima*. Com o adendo comportamental de que uma aplicação que permanece nesse estado por um longo tempo (*Z* ciclos de medição) será colocada em estado de hibernação no nó. Esse comportamento é o comportamento mais altruísta que nossa aplicação executa. Ela interrompe completamente sua utilização do nó, liberando a máquina para os demais *Jobs*. Durante o estado de hibernação o EM trabalha com custo de processamento bem baixo, próximo do zero, podem realocar suas tarefas para os demais nós ativos. Esse comportamento só é realizado com a meta do tipo *soft*, não se manifestando para aplicações com meta *hard*.

Obs.: Durante a hibernação caso seja detectado que a *host* ficou sem outros *jobs* ou ocioso o EM irá acorda o *job* para voltar a utilizar o *host*.

- **Estado no prazo:** Quando a aplicação encontra-se dentro do prazo esperado mantemos uma vazão de **40% a 60%** da carga máxima para a aplicação com meta *soft* e **50% a 70%** para meta *hard*. Ambos intervalos de valores foram obtidos através de testes empíricos nos quais esses valores foram os que melhores refletiram o comportamento altruísmo almejado, o percentual ligeiramente maior da meta *hard* (10%) reflete a prioridade que damos para os *jobs* nesse estado. O valor exato da vazão dentro do intervalo mencionado é obtido através da função Meta/Previsão. No qual distribuímos proporcionalmente o percentual da vazão, resultados mais crítico (valores menores que 1) a vazão maiores, e retorno menos críticos (valores maiores que 1) a vazão menores.

- **Estado atrasado com chances:** Quando uma aplicação se encontra nesse estado é o primeiro indício que a execução do job não está indo bem. Esse estado representa que se continuamos por esse caminho iremos perder a meta. Assim, para esse estado a primeira reação é aumentar a vazão. Logo, quando executando uma aplicações soft iniciamos esse estado com uma vazão de **75%** da carga máxima, caso o *job* permaneça por um tempo longo nesse estado (*Z* ciclos de medição) a aplicação irá aumentar para a carga máxima no nó **(100%)**. Para uma aplicação *hard* a carga máxima é configurada imediatamente na entrada desse estado.

  Neste estado podemos observar um comportamento muito mais egoísta por parte do *job*, ele irá aumentar sua carga contando que outros *jobs* que estejam no ambiente em estados melhores do que ele cederão processamento permitindo assim que consiga finalizar dentro de sua meta.
- **Estado atrasado com poucas chances:** Esse é o estado mais *crítico* no qual o job pode está sem ainda ter perdido sua meta. Esse estado indica que a previsão atual de término é muito acima da meta desejada. Assim temos que tomar alguma atitude para que a aplicação consiga acabar dentro da meta. Nesse estado a vazão é configurada para seu valor máximo. Além disso, o *EM* sinaliza para o escalonador acima que precisa de mais nós, que por sua vez, irá tentar acordar algum recurso que esteja dormindo. Esse procedimento funciona em conjunto com o escalonador dinâmico, ao acorda algum recurso que esteja hibernando, o escalonador dinâmico irá redistribuir as tarefas entre os nós (inclusive o novo nó que acabou de acordar) o que irá contribuir para que a aplicação atinja sua meta.
- **Estado atrasado sem chances:** Esse é o estado o qual nunca queremos alcançar. Quando a aplicação encontra-se nesse estado pouca coisa existe para se fazer. Nessa implementação ao chegar nesse estado com a meta *hard,* a execução é cancelada. Para aplicações com meta *soft* a execução será mantida por um período extra até atingir o *deadline*. Esse *deadline* representa apenas uma trava para situações onde aplicação possa ter apresentado algum problema de execução.

5.3.3.2 Decisão baseada em políticas auxiliares

Nessa seção apresentamos alguns mecanismos auxiliares que são executados independentemente da reposta do classificador de estado, e em alguns casos podendo inclusive sobrepor as decisões do estado definido.

- **Política Auxiliar I**: Uma aplicação que encontra-se executando **sozinha** no host, quando detectada pelo escalonador EM do *Job*, será alterado a vazão para *carga-máxima* para tentar utilizar o máximo do *host*, independentemente do estado que se encontra o *job*.
- **Política Auxiliar II:** Mesmo quando temos mais de um *job* executando no host é possível que ocorra períodos longos de ociosidade. Esse comportamento pode ocorrer em diversos cenários como, por exemplo, *jobs* no qual os processos tenham baixo consumo de CPU ou mesmo um cenário com múltiplas aplicações adiantadas. Para tratar esses cenários, no qual existe processamento ocioso no nó, foi criado um algoritmo nomeado de ***"Completa carga"*** que manipula a vazão da aplicação. Seu funcionamento consiste em detectar *X* ciclos de monitoração consecutivos com ociosidade, com a utilização da função interna *Percentual de ociosidade do host em um determinado instante*. Caso isso ocorra, será gerado um número randômico dentro de um pequeno intervalo pré-definido no qual a aplicação irá esperar para fazer uma nova monitoração e confirma esse estado, os *jobs* com metas mais atrasadas são priorizados na decisão, estados mais atraso possuem range menores que estados mais adiantados. Caso isso ocorra, o EM irá aumentar a vazão 1 processo/Host/*job* a cada monitoração para utilizar essa capacidade ociosa.
- **Política Auxiliar III:** Observamos que, mesmo com o nosso classificador de estado funcionando, ainda sim não conseguimos refletir alguns dos comportamentos altruístas almejados. Isso porque, embora possamos alterar a nossa vazão para adaptar-se às realidades da execução, o estado de um *job* altera-se com muita facilidade ao simples término de outro *job. Como* não possuíamos informações suficiente como a previsão de término dos *jobs* concorrentes nosso comportamento altruísta se limitava nesses casos. Sendo assim, para contornamos esses comportamentos, criamos um componente para nossa aplicação que nomeamos de **"Golpe de Sorte"** que tem como objetivo complementar o controle de status mencionado anteriormente. Seu funcionamento consiste em alterar a vazão da aplicação baseado na *função Meta/Previsão(seção 5.2.1) , que representa a proporção do tempo estimado no cenário ótimo sobre a meta*. Para tal, foram criadas três faixas de valores no qual classificaremos o *job* ao longo de sua execução baseado no valor da função *Meta/Previsão*. Para cada faixa, que representa qual é a proporção de tempo restante para executar o job, associamos um percentual que irá alterar a vazão retornada pelo controle de status: A primeira faixa vai de *0 a 1,5*, a segunda faixa de *1,5 a 2,5*, a

terceira faixa para valores maiores que *2,5*. Para cada uma dessas faixas, associamos, aos respectivos valores 100% da vazão original (retornado pelo controle de status), 20% da vazão original e 1 *job*/processo. O funcionamento desse componente funciona complementando o retorno da saída de estado e restringe-se a algumas condições, sendo assim, seu funcionamento não é aplicável à: Meta Hard, aplicações que esteja executando sozinho ou em situações na qual o EM esteja *hibernando*.

5.4 Algoritmo Sociedade Altruísta

Nesse pseudocódigo apresentamos a tomada de decisão para cada estado mencionado anteriormente bem como o funcionamento dos mecanismos auxiliares criados para detectar execução sozinha, ociosidade do recurso e cálculo e sua meta em relação ao melhor tem*po.*

```
00 BEGIN
01    SE (sozinho)
02          VAZAO = 100%
03    SENÃO SE(Estado==NO_PRAZO)
04          SE soft
05                VAZAO = 40%~50%
06          SENÃO  //hard
07                VAZAO = 50%~60%
08    SENÃO SE(Estado==ADIANTADO)
09          VAZAO = 10% ~20%
10    SENÃO SE(Estado==MUITO_ADIANTADO)
11          VAZAO = 10%
12          SE PermanecerMuitoTempoMesmoEstado
13                entraEstadoHibernacao
14    SENÃO SE(Estado==ATRASADO_COM_CHANCE)
15          SE Meta == soft
16                SE (NÃO estaMuitoTempoNoAtrasadoComChances)
17                      VAZAO = 75%
18                SENÃO
19                      VAZAO = 100%
20          SENÃO //hard
21                VAZAO = 100%
22    SENÃO SE (estado==ATRASADO_COM_POUCA_CHANCE)
23          VAZAO = 100%
24          //solicitação para acordar recursos para
25    SENÃO  ATRASADO_SEM_CHANCE
26          SE soft
27                SE (dentroDoLimiteTolerancia)
28                      VAZAO = 100%
29                SENÃO
30                      SolicitacaoDeCancelamento()
31          SENÃO  //hard
32                SolicitacaoDeCancelamento()
33    SE Meta == hard || sozinho || dormindo
34          VAZAO = 100%*VAZAO // Vazão inalterada
35    SENÃO SE (RazaoProporcao()  > 2,5)
36          VAZAO = MIN //Percentual equivalente a 1 processo
37    SENÃO SE (RazaoProporcao() > 1,5)
38          VAZAO = 30%*VAZÃO
39    SENÃO
40          VAZÃO = 100%*VAZAO
41
42    SE ExisteOciosidade
43          CompletaCarga()
44 END
```

5.5 Resumo

Esse capítulo apresentou a nossa proposta de política de ambiente distribuído e o conjunto de algoritmos que utilizamos para alcançar nosso objetivo. Algoritmos que foram elaborados ao longo do mestrado pensando em refletir os diversos cenários possíveis no compartilhamento do ambiente computacional colaborativo e as repostas que tentam manter o altruísmo do sistema.

A tabela a seguir (Tabela 5.1) resume as ações para cada um dos estados/comportamentos detectado pelo EM do *job* num dado *host*. Embora a aplicação guie-se principalmente pelo estado detectado, os mecanismos auxiliares ajudam e complementam a heurística com a execução de medidas auxiliares.

Tabela 5.1 Resumo de ações por estado/comportamento

Estado/Política Auxiliar	**Ação Soft**	**Ação Hard**
Sozinho (Política Auxiliar I)	Aplicação executa em carga máxima.	Aplicação executa em carga máxima.
Processamento ocioso (Política Auxiliar II)	Complemento da carga usando o algoritmo ***"Completa carga"***	Complemento da carga usando o algoritmo ***"Completa carga"***
Adiantado	Vazão entre 20% e 30%	(*Não Aplicável)*
Muito Adiantado	Vazão em 20% podendo fazer com que alguns EM durmam	(*Não Aplicável*)
No Prazo	Vazão em 40% a 60%	*vazão em 50%* a 70%
Atrasado com chance	Vazão em 75% podendo chegar a 100%	Vazão em 100%
Atrasado com poucas chances	Vazão em 100% e solicita mais recursos ao GS.	Vazão em 100% e *solicita* mais recursos ao GS.
Atrasado sem chances	Carga em 100% até atingir o deadline. O qual a aplicação é cancelada.	Aplicação cancelada.
Golpe de Sorte (Política Auxiliar III)	Alterar a vazão do controle de baseado na função *Meta/Previsão:* *Faixa de 0 a 1,5 => 100%* *Faixa de 1,5 a 2,5 => 20%* *Faixa de 2,5 a Inf => 1 proc/job/EM*	(*Não Aplicável*)

Capítulo 6– Análise experimental

Ao longo do trabalho, contextualizamos o problema de escalonamento de tarefas (*Job Shop Scheduling*) juntamente com as desvantagens da proposta tradicional. Apresentamos a nossa proposta de ambiente de execução que se propõe a contornar as limitações das propostas clássicas na utilização de recursos, criando um ambiente de execução computacional com menos ociosidade e mais altruísmo. Por fim, apresentamos o nosso conjunto de heurísticas implementadas em um middleware que é a nossa aposta para conseguir alcançar o ambiente de execução que propomos na Capítulo 5.

Este capítulo apresenta diversos experimentos que foram realizados com o EasyGrid SA ao longo de suas melhorias, avaliando o comportamento altruísta que desejamos para as aplicações. O objetivo é verificar o quão eficiente a heurística proposta demonstra-se na resolução do problema de escalonamento de aplicações com as metas de tempo e a redução do *makespan total* dos jobs. Os experimentos apresentados neste capítulo foram realizados avaliando a heurística sobre o conjunto de métricas que definimos: (i) o sobrecarga de compartilhamento dos recursos; (ii) o sobrecarga da implementação; (iii) a priorização dos jobs, (iv) a redução da subutilização dos hosts e (v) a justiça na priorização dos *jobs*. Estas métricas foram definidas como necessárias para alcançar o ambiente computacional compartilhado desejado.

6.1 Configurações do ambiente

Para a realização dos experimentos, foram utilizados servidores com dois processadores Intel Xeon X5650 2.67GHz Hexa-Core, 24GB DDR3, interligados por uma rede gigabit e munido do Sistema Operacional CentOS 6 com o kernel 2.6.32-71.el6.x86_64. O número de servidores (hosts) utilizados varia de experimento para experimento podendo chegar ao máximo de quatro hosts simultâneos, sendo sempre um host para executar o GG (Gerenciador Global) e o GS (Gerenciador Site) e um número variável de hosts trabalhadores.

Para a execução dos experimentos, utilizamos uma aplicação *bag-of-tasks* nomeada de "*random-float*", que trata-se de uma aplicação MPI paralela que foi calibrada para criar *x* tarefas cada uma com carga computacional de *y* segundos de processamento em um dos hosts. A carga computacional consumida por cada tarefa trata-se de operações de multiplicação de números pontos flutuantes gerados aleatoriamente. Em todos os testes executados, realizamos uma bateria de cinco repetições. Sempre calculando a media das execuções e o desvio padrão.

6.2 Experimentos

6.2.1 Sobrecarga de compartilhamento de recursos

Neste experimento desejamos avaliar como a versão original do EasyGrid AMS (versão sem as modificações realizada neste trabalho) comporta-se com o aumento do número de jobs simultâneos em um mesmo servidor, computando quanto será o *overhead* da concorrência.

6.2.1.1 Configuração dos jobs

Cada aplicação paralela *"random-float"* é composta por 128 tarefas, onde cada tarefa possui uma carga equivalente a 20 segundos de processamento. Para realizar este teste, a aplicação EasyGrid AMS foi configurada com seu escalonador dinâmico ativado, a Política de Execução Irrestrita (PEI) e uma vazão estática de, no máximo, 12 tarefas simultâneos. Ou seja, cada job irá executar concorrentemente 12 tarefas por servidor até que seu conjunto de tarefas termine. Foram utilizados para execução do teste um servidor para execução dos processos GG e GS, e um outro host trabalhador para o GM.

6.2.1.2 Testes

Tabela 6.1 Tempos da execução por número jobs

Quantidade de *jobs* simultâneos	Tempo em segundos	Tempo normalizado
1	288,03	24,0025
2	577,64	24,0683
3	882,94	24,5261
4	1.185,17	24,6910
5	1.487,81	24,7968
6	1.788,60	24,8416
7	2.089,44	24,8742
8	2.386,35	24,8578
9	2.683,64	24,8485
10	3.015,42	25,1285

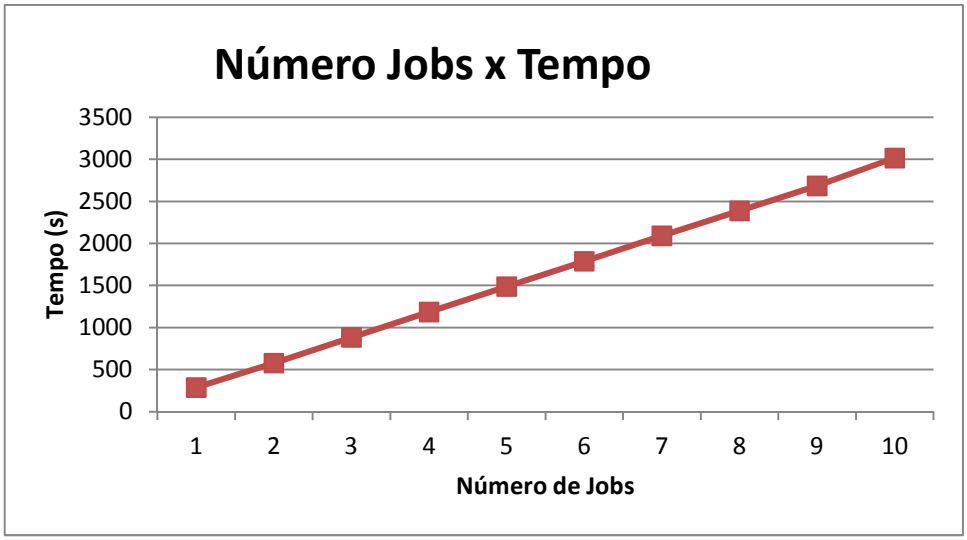

Figura 6.1 Tempo de execução por número de jobs executando concorrentemente

Neste experimento, podemos observar um comportamento praticamente linear dos jobs no ambiente. Note que cada um dos *jobs* está sempre executando 12 tarefas simultaneamente (exceto próximo ao término do job quando a quantidade de processos restantes é inferior a esse número) chegando a 120 tarefas concorrentes no host quando executamos 10 jobs simultâneos. Ainda sim, podemos observar na que a razão entre a execução de 1 e 10 jobs concorrente (o que equivale 10 tarefas por núcleo) é de 1,047. Demonstrando assim que com 10 *jobs CPU Bound* tivemos um overhead de aproximadamente 5% para esta versão do EasyGrid AMS. Boa parte do baixo overhead obtido deve-se ao CFS (*Completely Fair Scheduler*) presente no *kernel* do Linux a partir da versão 2.6 que possui excelentes resultados no escalonamento de tarefas [45][46].

A obtenção deste baixo overhead sobre a concorrência dos *jobs* foi fundamental para continuarmos nosso projeto. Pois ele satisfaz um das premissa para criação do ambiente compartilhado, manter um sobrecarga de concorrência baixa.

6.2.2 Sobrecarga da implementação da nova versão do EasyGrid

Através deste experimento iremos comparar as modificações realizadas no projeto EasyGrid ao longo do nosso trabalho. Realizando um conjunto de testes entre a antiga implementação e a atual, verificando como as modificações afetaram o desempenho do EasyGrid AMS.

6.2.2.1 Configuração dos jobs

Para este experimento, cada aplicação paralela "random-float" é composta por 128 tarefas onde cada tarefa possui uma carga equivalente a 20 segundos de processamento. Para realizar este teste, a aplicação em ambas versões do EasyGrid AMS foram configuradas com seu escalonador dinâmico ativado e a Política de Execução Irrestrita (PEI). Foram utilizados para execução deste teste um servidor para execução dos processos GG (Gerenciador Global) e GS (Gerenciador de Site), e três outros hosts trabalhadores cada um com GM (Gerenciador de Máquina).

6.2.2.2 Testes

Teste 1: Para este teste executamos um único job do SGA por vez, para validar como ele se comporta quando executado sozinho. Realizamos o teste com as seguintes configurações: EasyGrid AMS original (SGA clássico) com uma vazão estática de 12 processos concorrentes, EasyGrid SA (SGA modificado) com uma vazão estática de 12 processos concorrentes e o EasyGrid SA com política SA ativa com meta soft de 120s (SGA modificado com política SA).

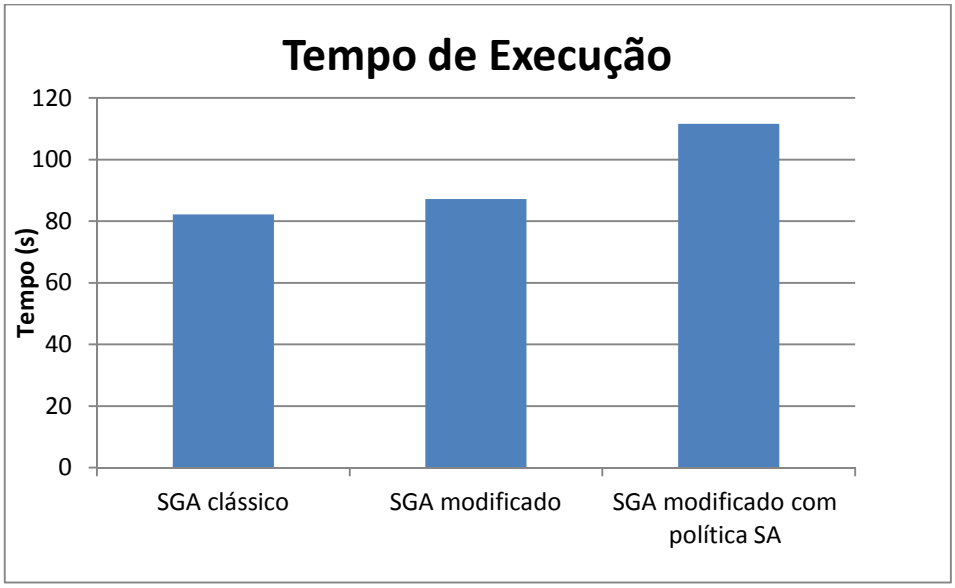

Figura 6.2 Média do tempo de execução sozinho entre as versões

Tabela 6.2 Média dos tempos de execução sozinho entre as versões

Versão	Tempo médio de execução em segundos	Desvio Padrão
SGA clássico	82,19	0,30
SGA modificado	87,18	0,55
SGA modificado com política SA	111,65	1,82

Neste teste, comparamos como as modificações realizadas no EasyGrid afetaram o seu tempo total de execução quando executado um único job. Podemos constatar que nossas modificações acresceram o tempo total de execução do job quando executado sozinho. Esta sobrecarga é resultado de dois mecanismos em especial: (i) A detecção dinâmico da infraestrutura que detecta quantos cores o servidor tem, e; (ii) *Pilot tasks*, no qual a aplicação inicializa-se com somente uma tarefa do *job* para verifica a capacidade computacional sendo obtida pela aplicação em cada host. Embora essas modificações prejudiquem a execução em media 6% em relação ao mecanismo (i) e em media 36% em relação ao mecanismo (ii), estes mecanismos são necessários para tornar o EasyGrid AMS menos dependente num conhecimento previo do capacidade do ambiente quando ocioso. Uma pesquisa já identificou o gargalo do mecanismo (ii) e esta sendo planejando a implementação de uma correção. Ainda assim, vamos continuar a avaliação de execução compartilhada com esta versão do EasyGrid SA.

Teste 2: Este teste executa três *job*s sequencialmente do EasyGrid AMS por vez, com o objetivo de obter os tempos totais de execução para ser utilizado no Teste 3 desta seção. Realizamos o teste com as seguintes configurações: EasyGrid AMS original (SGA clássico) com uma vazão estática de 12 processos concorrentes, EasyGrid SA (SGA modificado) com uma vazão estática de 12 processos concorrentes e o EasyGrid SA com política SA ativa com meta soft de 120s (SGA modificado com política SA).

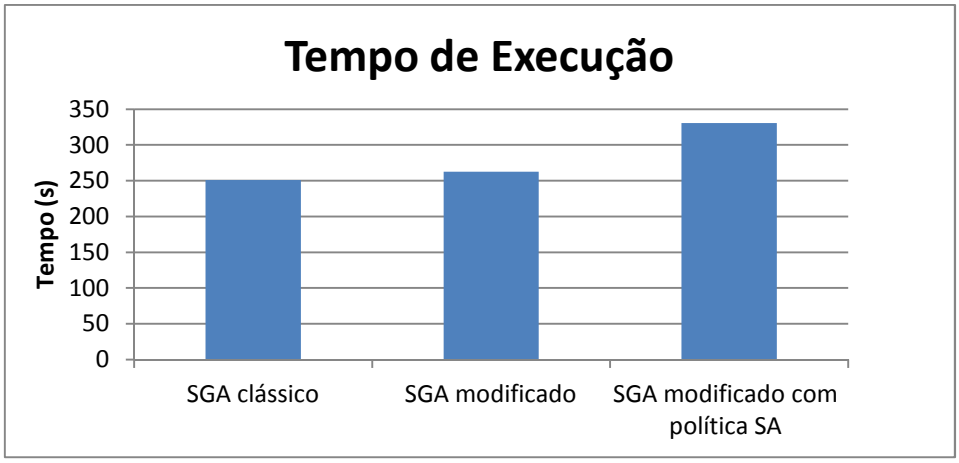

Figura 6.3 Média do tempo de execução de 3 jobs sequenciais

Tabela 6.3 Media de execução de 3 jobs sequenciais

Versão	**Tempo médio de execução em segundos**	**Desvio Padrão**
SGA clássico	250,73	8,50
SGA modificado	262,43	1,66
SGA modificado com política SA	330,61	2,04

Neste teste comparamos o tempo de execução de três jobs sequencialmente executando com o EasyGrid. Assim como esperado, os tempos obtidos são as somas das execuções obtidos em Teste 1. Entretanto será que este comportamento se mantém quando as três *jobs* são executados concorrentemente.

Teste 3: Para este teste executamos os três *jobs* concorrentemente usando as três versões do EasyGrid. O objetivo deste teste é comparar os tempos totais de execução dos *jobs* com o Teste anterior e verificar como as modificações para execução concorrentes irão influenciar nos resultados. Realizamos o teste com as seguintes configurações: EasyGrid AMS original (SGA clássico) com uma vazão estática de 12 processos concorrentes, EasyGrid SA (SGA modificado) com uma vazão estática de 12 processos concorrentes e o EasyGrid SA com política SA ativa com meta soft de 260s (SGA modificado com política SA).

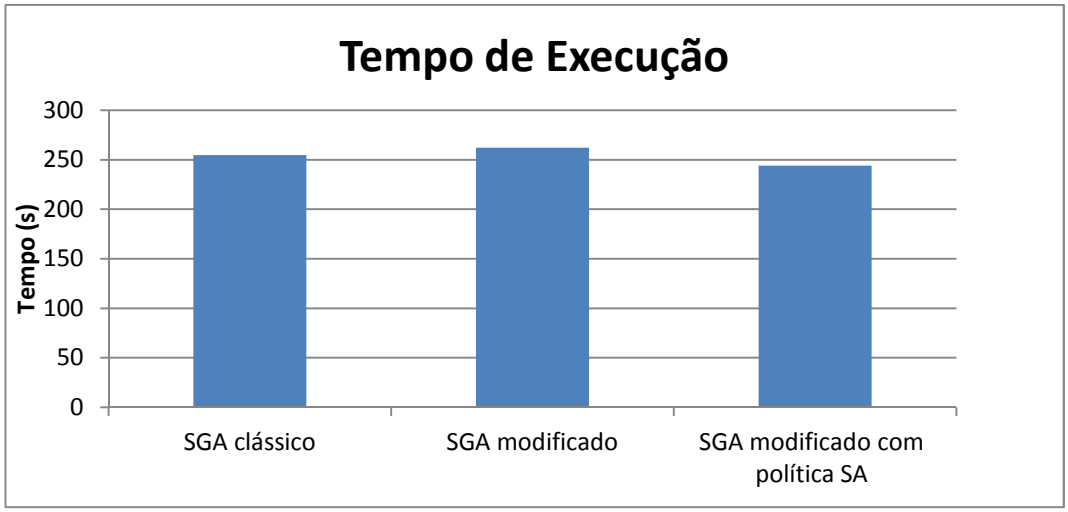

Figura 6.4Média do tempo de execução de 3 jobs concorrente

Tabela 6.4Média do tempo de execução de 3 jobs concorrente

Versão	**Tempo médio de execução em segundos**	**Desvio Padrão**
SGA clássico	254,61	10,87
SGA modificado	262,10	0,66
SGA modificado com política SA	244,16	1,33

Este teste apresenta o ponto forte da nossa implementação. Nele podemos observar que embora nossa implementação tenha apresentado resultados inferiores a versão original quando executamos *jobs* sozinhos. Quando executado concorrentemente a outros *jobs*, EasyGrid SA conseguiu gerenciar melhor a criação dos processos que resultou um *makespan* significantemente melhor do que ao executar com a versão original.

6.2.3 Avaliando o sobrecarga de execução compartilhada

Toda a proposta deste projeto é voltada para aproveitar o compartilhamento de recursos, ou seja, podemos executar aplicações com prioridades diferentes em ambientes compartilhados e, pelo aproveitamento de possíveis momentos de ociosidade nos recursos, melhorar o desempenho? Neste experimento iremos avaliar a estratégia de execução compartilhado quanto à utilização dos recursos, feitas por uma aplicação EasyGrid que adota a política de comportamento ***Sociedade Altruísta***. O teste será realizado através da comparação de duas estratégias de execução dos *jobs* sobre os recursos: na primeira estratégia, cada aplicação

executará seguindo uma reserva temporal e espacial dos recursos alocado para cada *job*. Esta estratégia de execução nomeamos de *execução isolada* (sem compartilhamento de recursos). O segundo estratégia será através da execução compartilhada de todos os *jobs* sobre todos os recursos ao mesmo tempo. Nomeamos este modelo de *execução compartilhada* (com compartilhamento de recursos).

6.2.3.1 Configuração dos *jobs*

Para este experimento cada aplicação paralela *"random-float"* é composta por 128 tarefas onde cada tarefa possui uma carga equivalente a 20 segundos de processamento. Para realizar este teste, a aplicação EasyGrid AMS foi configurada com seu escalonador dinâmico ativado, a Política de Execução Irrestrita (PEI) e uma vazão estática de, no máximo, 12 tarefas simultâneos. Ou seja, cada *job* irá executar concorrentemente 12 tarefas por servidor até que seu conjunto de tarefas termine. Foram utilizados para execução deste teste: um servidor para execução dos processos GG (Gerenciador Global) e GS (Gerenciador de Site); e três outros hosts trabalhadores cada um executando o GM (Gerenciador de Máquina).

6.2.3.2 Testes

Teste 1: Inicialmente utilizamos aplicação *random-float,* uma aplicação *CPU bound*, com 128 tarefas de 20 segundos de duração cada. Cada recurso tem sua configuração descrita como no Capítulo 6.1.

Para a execução da estratégia isolada (com reserva de recursos), reservaremos uma maquina trabalhadora distinta para cada job (GSN10, GSN11, GSN12). Os processos gerenciadores, GG e GS, de cada *job* foram agrupados em outro servidor (GSN13) a qual não está sendo executado nenhuma tarefa dos *jobs.* Para a execução sob a estratégia compartilhada (sem reserva de recurso), todas as aplicações acessarão todos os recursos, como ilustrado na Figura 6.6. O objetivo deste teste é validar que o compartilhamento não é prejudicial à execução.

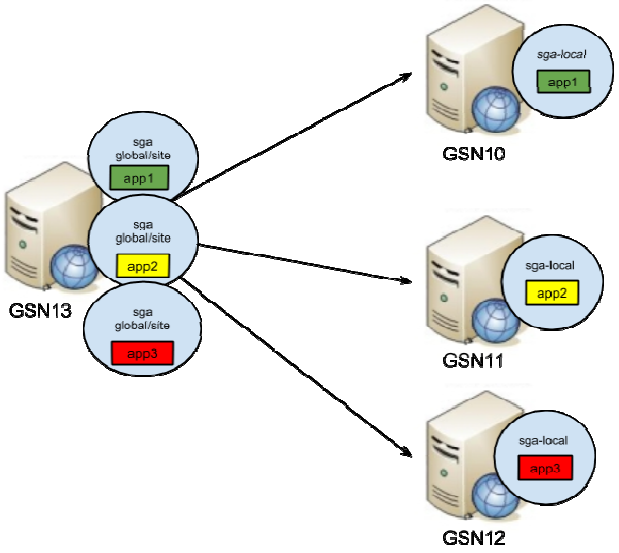

Figura 6.5 Distribuição dos escalonadores das aplicações nos recursos do ambiente de teste de acordo com o modelo clássico de escalonamento (isto é isolado com reserva de recursos exclusivos).

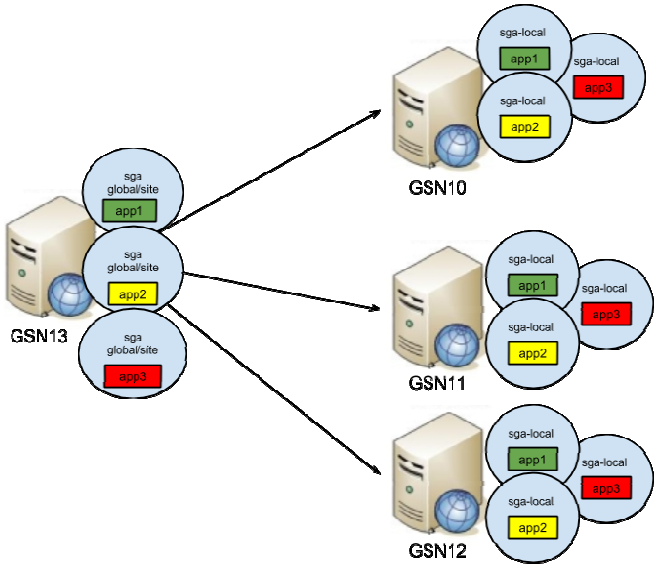

Figura 6.6 Distribuição dos escalonadores das aplicações nos recursos do ambiente de teste de acordo com a estratégia de execução compartilhada.

Tabela 6.5 Tempos de execução das estratégias

Estratégia	Tempo médio de execução em segundos	Desvio padrão
Isolado	228,35	0,38
Compartilhado	221,59	1,68

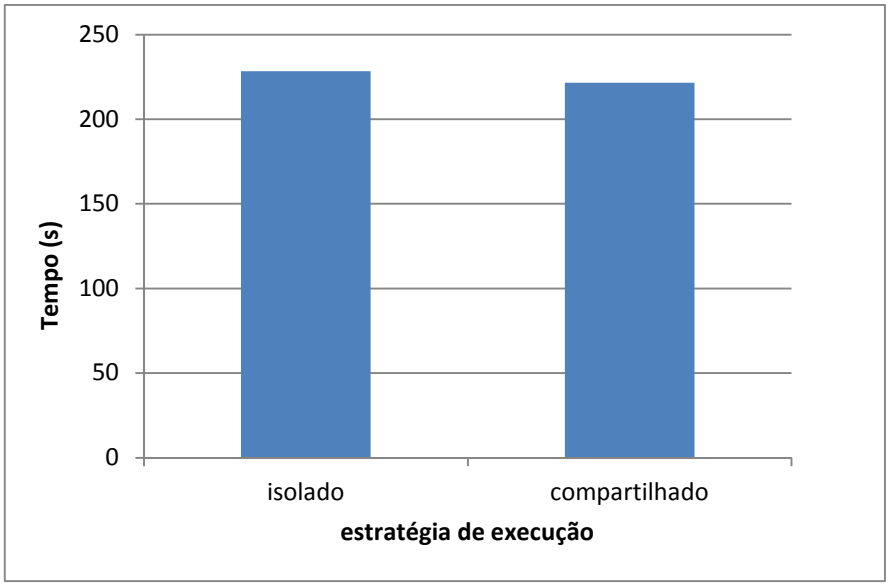

Figura 6.7 Média dos Tempo de execução com compartilhamento (Compartilhado) e sem compartilhamento (Isolado)

Neste experimento, conseguimos validar nossa hipótese. O compartilhamento de recursos, até um determinado ponto – neste caso, limitado a escala deste experimento, consegue manter o desempenho da execução das aplicações com baixo overhead. Em especial no teste realizado, devido ao modelo de execução do EasyGrid AMS, conseguimos uma melhor utilização dos recurso, preenchendo as lacunas de ociosidades entre a criação e termino de cada job. Esta melhora foi refletida na redução do tempo na execução quando compartilhamos os recursos. E demonstra que mesmo em aplicações com características *CPU bound* é possível executar jobs, até certo ponto, com concorrência de recursos.

Agora, o que aconteceria se realizasse o mesmo teste sobre aplicações que não fossem *CPU bound*, como no caso de aplicações que fazem uso de algum outro componente, como acesso ao disco, além do CPU. Assim, para o próximo teste, executaremos uma aplicação que simula I/O.

Teste 2: Para este teste, utilizamos uma variação da aplicação *random-float* que simula I/O*, cada aplicação* com 128 tarefas de 20 segundos cada. Sendo que cada tarefa irá dormir 6 segundos (30% do seu tempo total) durante a execução e irá processar durantes os demais 14 segundos (70% do seu tempo total). Cada recurso tem sua configuração descrita como na Seção 6.1. Os percentuais definidos de processamento e simulação I/O foram escolhidos ao acaso. Nesse teste queremos apresentar um cenário comum onde pode aproveitar o compartilhamento de recursos.

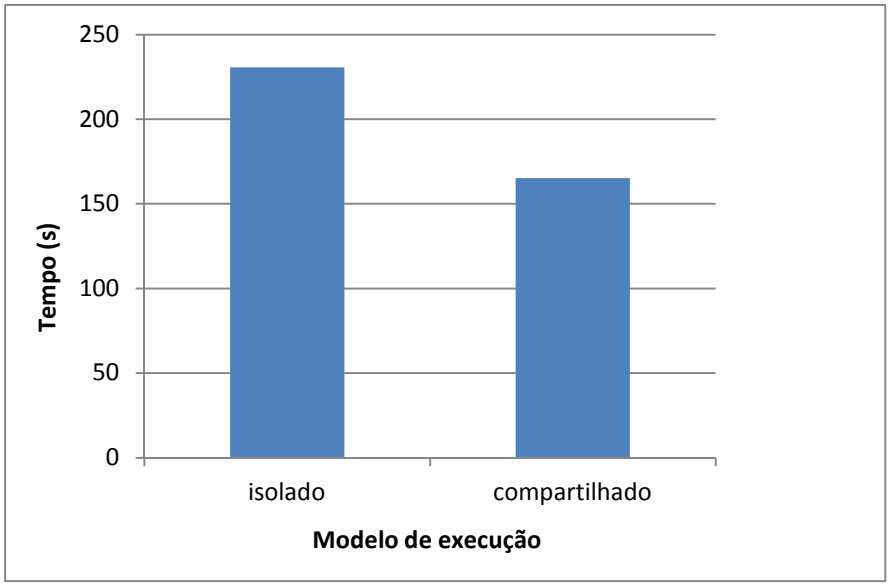

Figura 6.8 Média dos Tempo de execução com compartilhamento (Compartilhado) e sem compartilhamento (Isolado) para job com I/O

Como podemos ver através do Figura 6.8, o compartilhamento foi vantajoso para a execução das aplicações. O compartilhamento permitiu que os recursos (GSN10, GSN11, GSN12) fossem melhor utilizados, pois conseguimos para o mesmo conjunto de jobs (APP1, APP2 e APP3) um melhor tempo de execução com a estratégia compartilhada (Figura 6.6). Esse teste apresentou-se o primeiro ponto positivo da proposta, indicou viabilidade de compartilhar recursos para que alcancemos o comportamento desejado para um ambiente de execução compartilhada sem *broker central* de reserva de recurso.

6.2.4 Avaliação da priorização das tarefas

Uma das principais modificações dessa proposta é remover o controle de ordenação/alocação do *broker central*. Propõe que aplicações passam a autogerenciar e definir suas próprias políticas de execução por meio da meta definida e de processos intrínsecos, características do ambiente que indicam os estado da aplicação e os jobs concorrentes. A inclusão da meta de tempo e o sinalizador que define o tipo de meta (hard ou soft), ver Seção 5.1, são os principais critérios para definir a prioridade das aplicações no ambiente.

6.2.4.1 Configuração dos jobs

Neste experimento, **para todos os testes que executamos,** trabalhamos com três aplicações *random-float*, cada uma sendo gerenciada pelo seu próprio EasyGrid AMS, mas com metas de

término diferentes. Cada aplicação tem 512 tarefas que requerem 5s de processamento cada, além de uma tarefa mestre responsável pelo recebimento dos resultados. Foram utilizados quatro hosts para a execução dos testes, um para executar GG e GS de cada aplicação, e os três recursos restantes como trabalhadores. Os *jobs* executaram com o compartilhamento do ambiente e com uma distribuição inicial igual das tarefas em cada *worker*.

Enquanto o número de tarefas foi aumentado em relação os testes de Seção 6.2.3, o pesos de cada tarefa foi reduzido para manter somatório dos pesos dos *job* iguais. Esta variação foi realizada para melhor refletir a característica dos *jobs* que pretendemos executar, isto é, aplicações de larga escala com muitas tarefas.

6.2.4.2 Testes

Antes de iniciarmos os testes com a execução dos *jobs* concorrentemente, iremos apresentar os tempos da execução sozinho dos *jobs*. Iniciando pelo calculo do tempo teórico, que é de 75,00s, e pode ser determinado pela equação a seguir:

$$TEMPO_TOTAL = TEMPO_EXE_TAREFA * \lceil NUMERO_TOTAL_TAREFAS) / \sum NUMERO_NUCLEO_HOSTS \rceil \qquad (1)$$

Onde: TEMPO_EXE_TAREFA: tempo para execução de uma tarefa
NUMERO_TOTAL_TAREFA: o número total de tarefas de um job,
NUMERO_NUCLEO_HOST: Número total de Núcleos dos hosts utilizados.

Quando o este *job* é realmente executado de forma estática pelo sistema operacional (SO) que dispara as tarefas apenas por um script, o tempo de execução aumenta para uma média de *81,17s*. Ao adicionarmos o EasyGrid AMS utilizando-se da política as, e sem concorrência com outros jobs, o tempo aumenta para *90,17s*. O aumento criado pelo EasyGrid AMS com a política SA deve-se uma subutilização dos recursos durante a inicialização do EasyGrid AMS ao executar os *pilot tasks* para a configuração da Política SA. A aparência de uma inicialização lenta é devido a necessidade de executar apenas um *job* por recurso afim de processar algumas métricas que serão utilizada ao longo da execução. Embora, posteriormente sua vazão é normalizado, isso o torna o sobrecarga percentualmente custoso quando utilizando aplicações com poucos segundos de execução, como é o caso da aplicação utilizada nos testes.

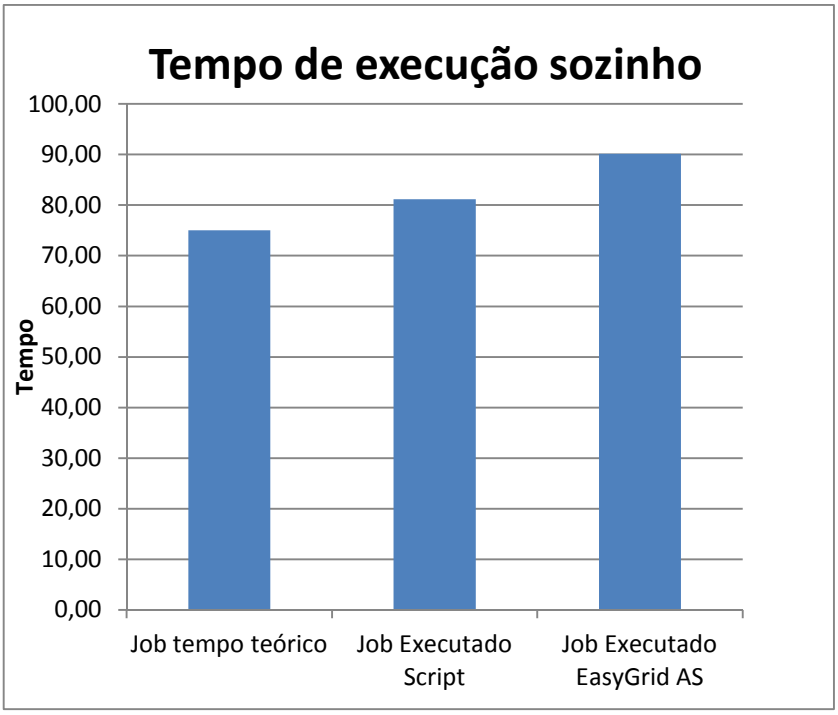

Figura 6.9 Média do tempo de execução sozinha

Teste 1: Teste base para os demais testes. Executamos três *jobs* (APP1, APP2 e APP3) trabalhando sem o controle de vazão do host e sem metas. Queremos verificar o quão justo é o balanceamento entre os jobs.

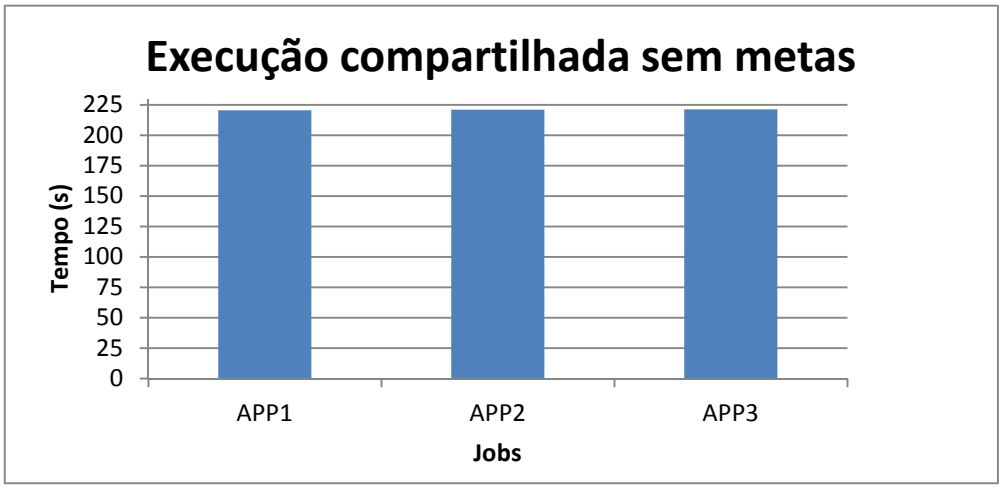

Figura 6.10 Média de tempo de execução política estática

Podemos observar no Teste 1 que o EasyGrid AMS foi capaz de executar as aplicações distribuindo as cargas igualmente entre nós. O que demonstra a eficiência do escalonador que balanceou as cargas igualmente entre os hosts, permitindo o término das tarefas em tempos próximos. Também é possível observar a eficiência do SO, que mesmo com a existência de

diversas tarefas dos três *jobs* (APP1, APP2, APP3), balanceou corretamente as cargas permitindo que os jobs ganhassem quantia de processamento bem próximas, sendo justo em seu escalonamento. Observe que todos os três *jobs* terminam em tempos similares, incapazes de serem diferenciados na Figura 6.10 e distinguíveis somente pelos valores na Tabela 6.6. Neste teste podemos observar que o compartilhamento foi bem saudável para aplicação, a concorrência de *jobs* permitiu que os tempos compartilhados fossem menores que a soma dos tempos da execução sequencial, Figura 6.9. Fazendo com que as aplicações preenche-se as lacunas de ociosidades entre o termino e inicialização das tarefas, utilizando melhor os recursos.

Agora o que aconteceria se os *jobs* tivessem requisito de tempos diferentes? Como este requisito pode ser refletido na execução?

Tabela 6.6 Resultado execução política estática

Jobs	**Tempo médio de execução em segundos**	**Desvio Padrão**
APP1	220,34	2,49
APP2	220,86	0,76
APP3	221,15	1,34

Teste 2: Neste teste, iremos avaliar se o middleware conseguirá atender as demandas de tempo, priorizando as aplicações com metas menores. Utilizamos três jobs (APP1, APP2 e APP3) trabalhando com as respectivas metas de tempo 170s, 280s e 380s, que representam percentualmente as folgas de tempo em relação tempo sozinho de 88,5%, 210,5% e 321,5%. A proposta é que as aplicações se auto-organizarão para criarem a ordenação de execução entre si sem um controlador central externo.

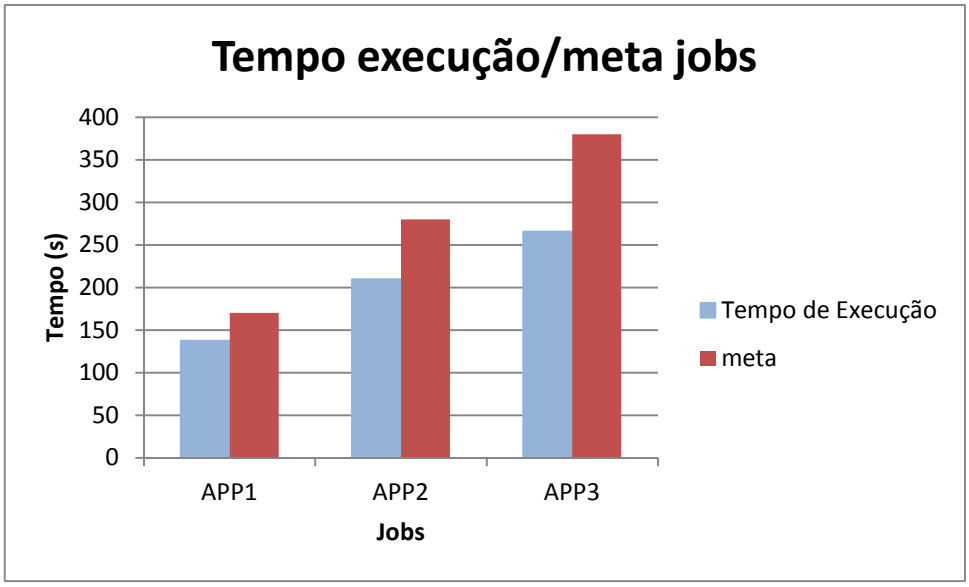

Figura 6.11 Meta x Tempo médio de execução dos jobs - Teste 2

Tabela 6.7 Meta x Tempo médio de execução - Teste 2

Jobs	Meta	Tempo médio de execução em segundos	Desvio Padrão
APP1	170	137,50	5,46
APP2	280	209,48	8,28
APP3	380	265,81	3,37

Podemos ver, através dos resultados na Figura 6.11, que a heurística conseguiu com que o comportamento estivesse dentro do esperado durante a sua execução, pois priorizou o *job* com a menor meta, *APP1* que terminou em média com *137,50s*, e a cada *job* concluído os recursos restantes foram sendo reorganizados entre as aplicações restantes. O que permitiu uma diferença de tempo de término menor entre as aplicações que terminaram mais tardiamente (*APP2 e APP3*). Isso demonstra que mesmo em um ambiente compartilhado, conseguimos priorizar a aplicação *APP1*. Reduzindo seu tempo total de execução de *220,34s* para *137,50s*, 62,4% do tempo anterior. Outro aspecto interessante no compartilhamento foi que conseguimos com que o tempo de término do APP3, última tarefa a terminar, inferior ao tempo da execução sequencial das três *jobs*, com base na Figura 6.9. Ou seja, conseguimos manter a priorização das tarefas no ambiente computacional com todas as execuções batendo suas metas, que é um dos comportamentos almejados, e ainda sim melhoramos o tempo de execução para os *jobs* mais prioritários.

Podemos observar também na Figura 6.11 que acabamos bem abaixo das metas estabelecidas. Sendo assim, vamos estreitar ainda mais os tempos de meta do *job* APP1 para observamos como a aplicação se comporta.

Teste 3: Utilizamos os mesmos três jobs (APP1, APP2 e APP3) trabalhando com as respectivas metas de tempo. 130s, 280s e 380s. Neste teste iremos avaliar como a APP1 se comporta quando sua meta fica mais estreita.

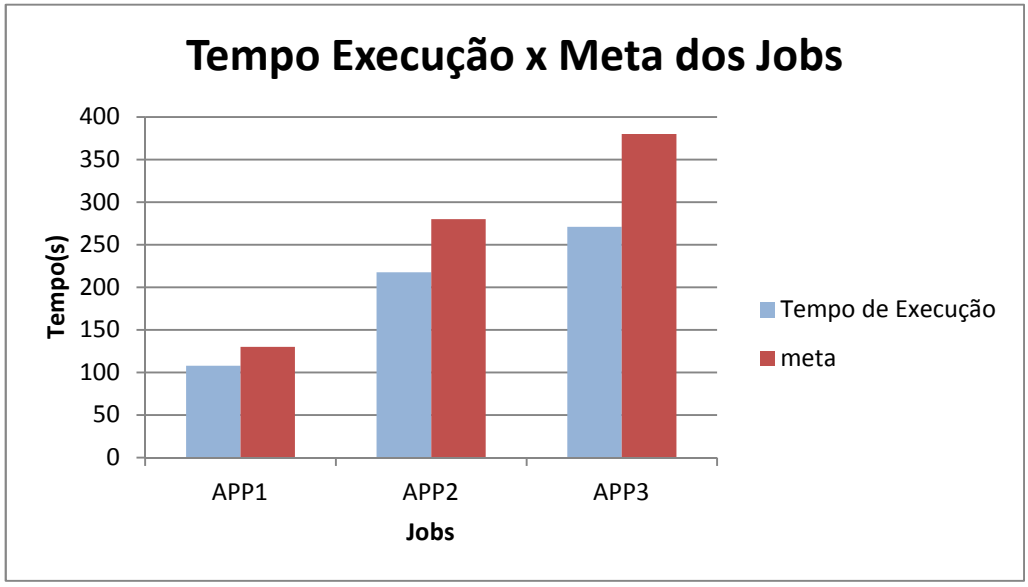

Figura 6.12 Meta x Tempo médio de execução - Teste 3

Tabela 6.8 Meta x Tempo médio de execução - Teste 3

Jobs	Meta	Tempo médio de execução em segundos	Desvio Padrão
APP1	130	107,61	3,57
APP2	280	217,60	5,61
APP3	380	270,91	1,62

Neste experimento, conseguimos verificar um resultado bem interessante para o ambiente de execução. Estreitamos as metas do job APP1 em relação ao teste anterior, e notamos que o tempo de término do job acompanhou sua meta, sendo priorizada a sua execução, e com variações bem menores que nos testes anteriores. Como consequência observamos que os jobs APP2 e APP3 tiveram seus tempos médios acrescidos em alguns segundos. Sugerindo que

uma parte maior do processamento foi utilizada pelo job APP1 para sua execução, mas sem grandes consequências, pois os jobs APP2 e APP3 ainda acabaram dentro das suas metas e com o tempo bem abaixo. Porém, ainda temos folga nas nossas execuções, para tal iremos realizar o próximo teste com metas bem justas para verificar os limites da heurística.

Teste 4: Utilizamos os três *jobs* (APP1, APP2 e APP3) trabalhando com as respectivas metas de tempo. 120s, 240s e 300s. Neste experimento, iremos avaliar se o comportamento altruísta se mantém com metas consideravelmente mais estreitas do que os testes anteriores.

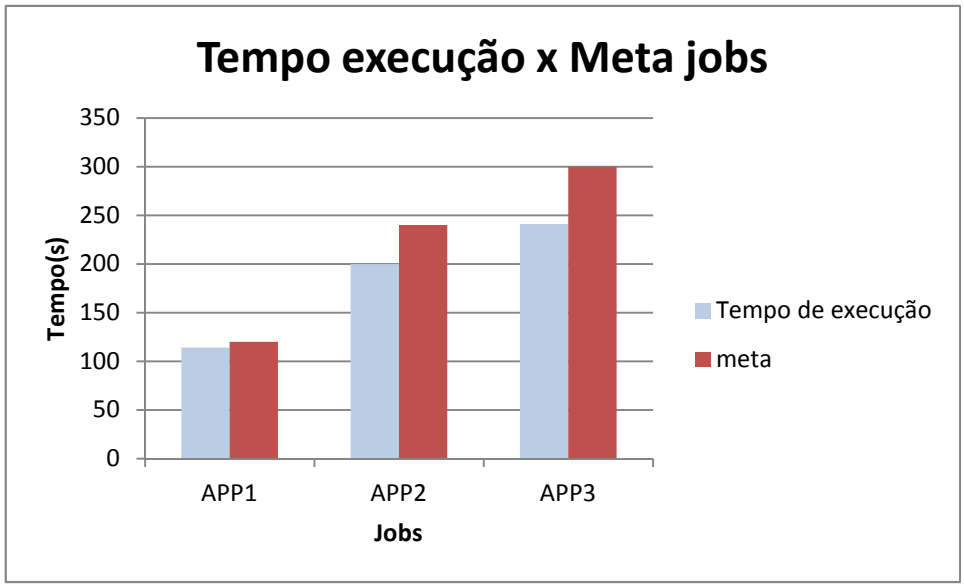

Figura 6.13 Meta x Tempo médio de execução dos jobs - Teste 4

Tabela 6.9 Meta x Tempo médio de execução - Teste 4

Jobs	Meta	Tempo médio de execução em segundos	Desvio Padrão
APP1	120	113,86	2,88
APP2	240	199,79	9,47
APP3	300	240,83	1,93

A cada novo teste, os *jobs* demonstram-se mais adaptativos às metas. Neste teste, o job APP1 aumentou levemente seu tempo, mas ainda se manteve dentro de sua meta, contudo os *jobs* APP2 e APP3 conseguiram reduzir seus tempos para se adaptarem às novas metas. Para

avaliarmos melhor como foi a utilização do processamento nos host ao longo das execuções iremos avaliar o trace de uma das execuções em um dos hosts.

Tabela 6.10 Tempo de termino de cada uma dos jobs do trace analisado

APP	Meta	Tempo médio de execução em segundos
APP1	120	117,02
APP2	240	210,63
APP3	300	244,23

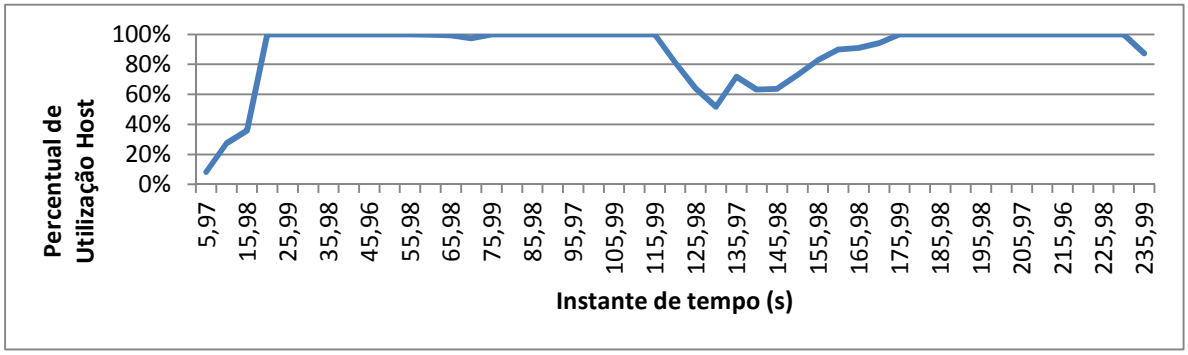

Figura 6.14 Utilização de CPU de um dos traces da execução em um dos hosts- teste 3

Ao longo do trace apresentado na Figura 6.14, podemos observar que a utilização manteve-se no máximo durante a maior parte da execuções dos *jobs*. Apresentando apenas uma queda na sua utilização perto do instante de tempo 115s, momento no qual termina o job APP1, o que faz com que a utilizanção do host caia para um valor próximo dos 80%. A que na utilização do host permanece até o momento em que o escalonador detectar a baixa utilização do host, através do componente "completa carga", iniciando o processo de ajuste de vazão e restabelencendo o consumo para o máximo no tempo próximo do instante 175s. O mesmo fato acontece no término do job APP2, entretanto não é perceptível no gráfico a queda da utilização do CPU, isso porque o mecanismo que detectar a execução sozinha do host rapidamente ajusta a vazão para o máximo, não transparecendo assim a ociosidade do host.

Nos testes até aqui, a priorização das metas foram bem sucedidas, pelo menos para tarefas pequenas. No próximo teste iremos avaliar o funcionamento da priorização do teste para tarefas com granularidade mais grossa.

Teste 5: Utilizamos dois *jobs* (APP1, APP2) trabalhando com as respectivas metas de tempo de 4000s, 7500s. Agora com tarefas requer 200s de processamento, e vamos avaliar o funcionamento da implementação com estas tarefas com maior duração.

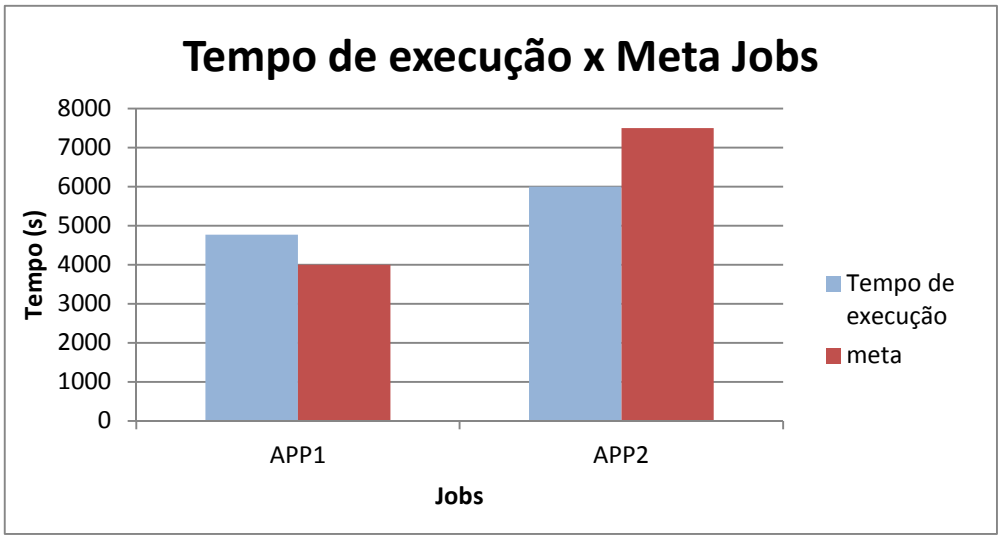

Figura 6.15 Meta x Tempo médio de execução - Teste 5

Tabela 6.11 Meta x Tempo médio de execução - Teste 5

Jobs	Meta	Tempo médio de execução em segundos	Desvio Padrão
APP1	4000	4774,12	91,65
APP2	7500	6005,85	55,35

Neste experimento, temos um resultado desfavorável. Podemos observar que o escalonador não foi eficiente o suficiente na priorização das tarefas do job para a meta definida. Apresentando que a heurística ainda precisa melhorar a priorização das tarefas quando a granularidade da tarefa é consideravelmente grossa. Podemos observar na Tabela 6.12 que em todos os testes realizados o tempo do job APP1 foi superior a sua meta de tempo de 4000 segundos.

Tabela 6.12 Tempos de execução - Teste 5

Execução	APP1	APP2
1	4819,36	6054,96
2	4907,68	5970,92
3	4725,97	5954,17
4	4746,61	5973,36
5	4670,97	6075,83
Média	**4774,12**	**6005,85**
Desvio	**91,65**	**55,35**
Meta	**4000,00**	**7500,00**

6.2.5 Priorização sobre cenários com folgas de tempo

Neste experimento, iremos avaliar alguns cenários onde as metas dos *jobs* tendem a gerar ociosidade nos recursos, e como a política SA adapta a vazão dos jobs existentes para utilizar-se destes recursos. A proposta neste experimento é avaliar como o ambiente de execução comporta-se no cenário onde os *jobs* em execução estão adiantados em suas metas.

6.2.5.1 Configuração dos jobs

Neste experimento, para todos os testes realizados, utilizamos a aplicação "random-float" junto ao middleware EasyGrid AMS. Utilizamos neste experimento 512 tarefas *CPU bound,* onde cada uma requer 5s de processamento, e uma tarefa mestre responsável pelo recebimento dos resultados. Esta aplicação quando executada sozinha possui um tempo médio de **90,17s**, como vista em Figura 6.9. Foram usados quatro hosts para a execução dos testes, um para executar GG e GS de cada aplicação e os três recursos restantes como trabalhadores.

6.2.5.2 Testes

Teste 1: Utilizamos dois jobs (APP1, APP2) trabalhando com as mesmas metas de tempo, 300s. O que para os *jobs* submetidos é uma meta bem folgada, um pouco mais de 3 vezes o tempo sozinho de execução. O quão justo será a divisão do poder computacional recebido por cada job.

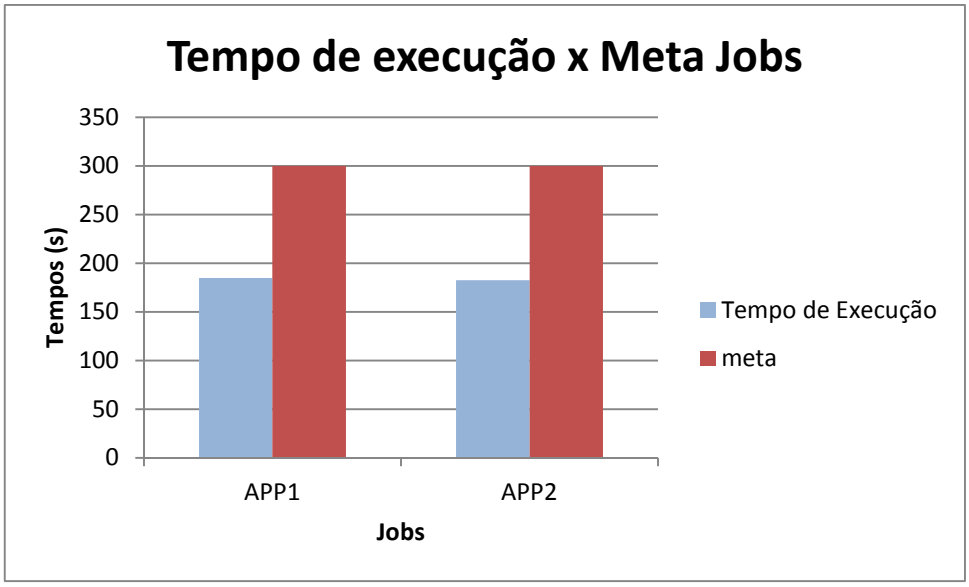

Figura 6.16 Meta x Tempo médio de execução - utilização recursos - Teste 1

Tabela 6.13 Meta x Tempo médio de execução - Teste 1

Job	Meta	Tempo médio de execução em segundos	Desvio Padrão
APP1	300	184,67	9,92
APP2	300	182,44	14,40

Nestes testes podemos observar que a nossa meta está bem folgada em relação aos jobs em execução. Entretanto, mesmo com a meta folgada, a execução foi capaz de detectar recursos disponíveis e ajustar a execução para terminar antes da meta e com tempos médios bem próximos. Como esperado, podemos concluir que os recursos foram utilizados igualmente quando possuímos metas iguais, mesmo que elas sejam bem folgadas. O que demonstra que nossa heurística é justa, mesmo sobre tempos com folgas. Ao longo dos experimentos coletamos o percentual de utilização dos recursos em instantes de tempos espaçados de 5s. Neste teste o percentual médio de ociosidade dos recursos ficou em torno de **17,78%**.

Teste 2: Neste teste avaliamos a justiça do escalonador. Para tal utilizamos dois *jobs* (APP1, APP2) trabalhando com as respectivas metas de tempo de 300s e 900s. Ambas as metas estão bastante folgadas em relação à quantidade de processamento necessária por cada *job* quando executada sozinha, ver Figura 6.9. Entretanto, queremos analisar justamente o comportamento

da execução em um cenário onde mesmo com as metas folgadas, como os *jobs* vão priorizar a utilização dos recursos existentes.

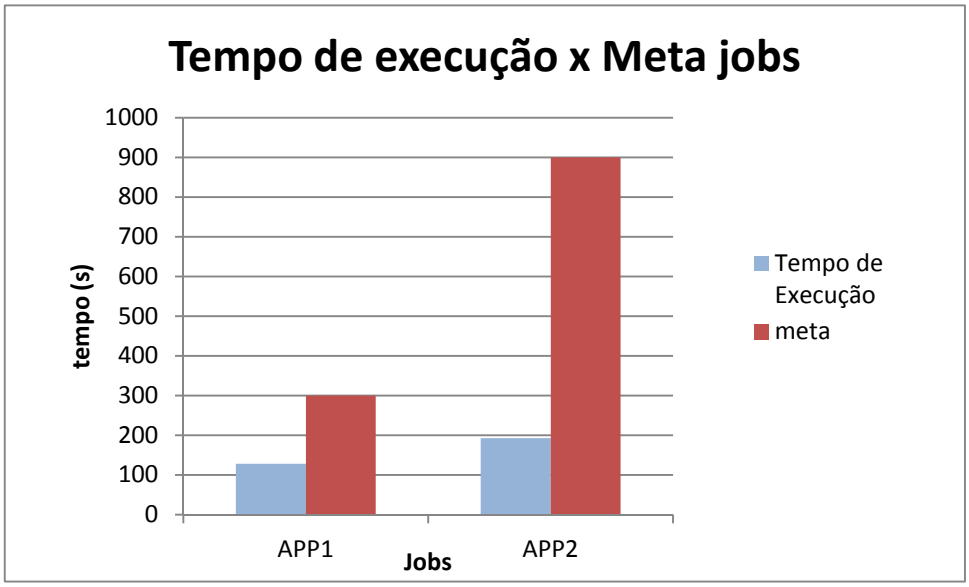

Figura 6.17 Meta x Tempo médio de execução - priorização da execução - Teste 2

Tabela 6.14 Meta x Tempo médio de execução - priorização da execução - Teste 2

Jobs	Meta	Tempo médio de execução em segundos	Desvio Padrão
APP1	300	128,59	26,59
APP2	900	193,01	13,60

Neste teste, podemos observar que o job APP1 foi priorizado na obtenção dos recursos, ganhando em média mais poder computacional que o job APP2. Terminando 33,4% antes do término do Job APP2. O que apresenta uma priorização da aplicação com as metas mais estreitas, mesmo que as metas estejam estremamente folgadas demonstrando uma justiça na priorização da execução, mas tambem um bom aproveitamento dos recursos disponiveis. Para podermos observar melhor o resultado da execução, vamos analisar um trace de um dos recursos, um dos três servidores que executaram as tarefas, que obteve os valores apresentados na tabela abaixo.

Tabela 6.15 Resultado da segunda execução teste 2 - ocioso

Jobs	Meta	Tempo médio de execução em segundos
APP1	300	106,37
APP2	900	193,01

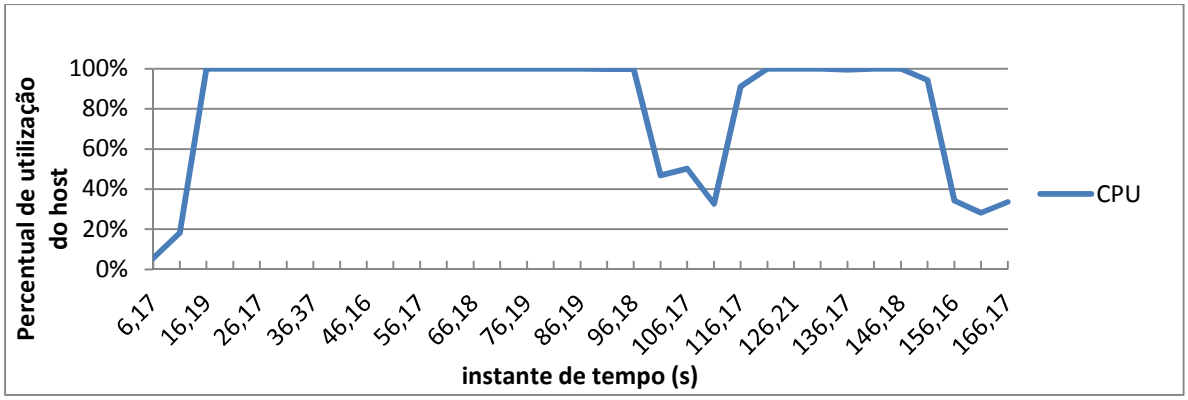

Figura 6.18 Trace de execução utilização CPU x Tempo

Através do trace de execução exibido na Figura 6.18 podemos observar que os *jobs* inicializam com uma baixa vazão de processos até que pelo menos 1 tarefa termine, devido os *pilot tasks*, o que ocorre por volta do instante 10s. Após este tempo, os *jobs* conseguem rapidamente se ajustar aos recursos disponíveis, tirando o máximo do ambiente. No instante 96,18s temos uma queda na utilização do recurso, isso porque o job APP1 está perto de sua finalização e já não possui mais tarefas prontas para executar, apenas as tarefas em execução que estão terminando. No instante 103,470s com o término do job APP1, o GM do job APP2 detecta a liberação do CPU e se reajustou para utilizar o processamento agora ocioso, demorando aproximadamente 13s para ajustar a vazão a nova realidade, a execução sozinha no ambiente. A ociosidade media deste trace foi de **12,66%**.

6.2.6 Avaliação da adaptação

Neste experimento queremos avaliar como *jobs* se comportam em um ambiente mais dinâmico. Verificando como o comportamento de um *job* que já esteja em execução sozinho no ambiente e com uma meta folgada se adaptará para executar concorrentemente com outro job que entrou depois e com uma meta estreita.

6.2.6.1 Configuração dos jobs

Neste experimento utilizamos a aplicação "random-float" junto ao middleware EasyGrid AMS. Utilizamos neste experimento 512 tarefas *CPU bound*, onde cada uma requer 10s de processamento, e uma task mestre responsável pelo recebimento dos resultados. Foram usados quatro hosts para a execução dos testes, um para executar GG e GS de cada aplicação e os três recursos restantes como trabalhadores.

6.2.6.2 Teste

Utilizamos dois *jobs* (APP1, APP2), ambos CPU Bound, trabalhando com as respectivas metas de tempo de 900s e 200s. Neste teste, a meta do *job APP1* está bastante folgada em relação à quantidade de processamento necessário, enquanto o *job APP2* possui sua meta consideravelmente justa. Ambos os *jobs* possuem uma carga de processamento 180s cada quando executado sozinho. Entretanto os tempos de inicialização dos *jobs* serão distintos, enquanto o *job APP1* iniciara no instante *0s*, o *job APP2* irá iniciar no instante *50s*. Neste experimentos, queremos verificar se o job APP1 irá modificar sua execução após a entrada do *job APP2*. Priorizando a meta mais curta e adaptando a execução as modificações do ambiente.

Tabela 6.16 Avaliação da adaptação – Teste 1

Jobs	Inicio	Meta	Tempo médio de execução em segundos	Tempo médio de termino	Desvio Padrão
APP1	0	900	329,69	329,69	9,80
APP2	50	200	203,43	253,43	8,75

Neste teste, podemos observar que o middleware teve um comportamento altamente adaptativo. Fazendo com que o *job* APP1 liberasse o processamento para o *job* APP2 e permitindo que o mesmo acabe bem próximo a sua meta. Este comportamento fica mais claro no trace de uma das execuções, apresentada na Figura 6.20. Onde observarmos, através da vazão de um dos hosts utilizados, que o *job* APP1 se adaptou para cede o processamento necessário.

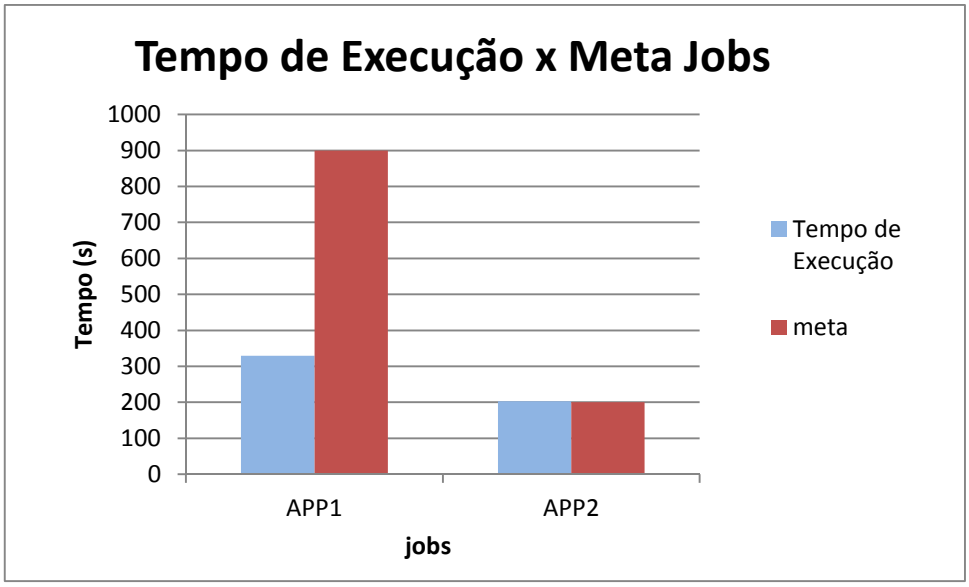

Figura 6.19 Meta x Tempo médio de execução

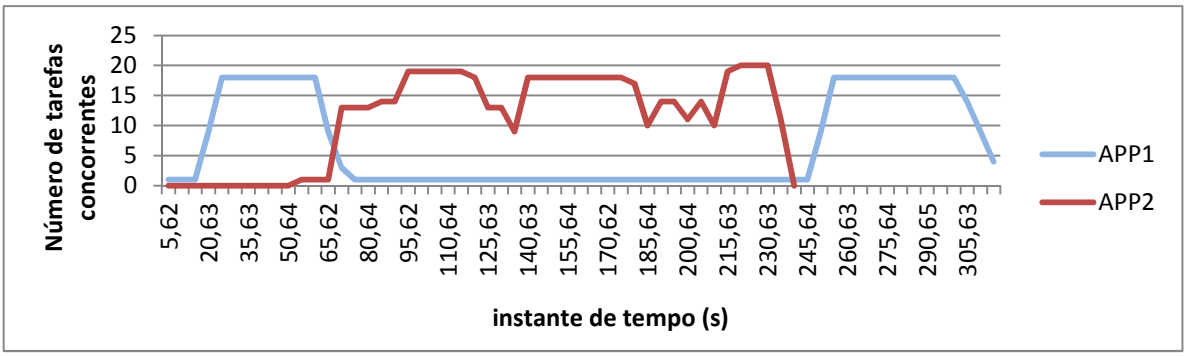

Figura 6.20 Trace de execução vazão de jobs por tempo corrente.

Através do trace de um dos hosts utilizado no teste, podemos verificar que pouco tempo após o inicio da execução do *job APP1* detecta que se encontra sozinho nos recursos e altera sua vazão para utilizar o máximo de processamento. Quando o *job APP2* inicia sua execução no instante *50s*, o comportamento imediatamente muda. Ciente de que está adiantado em sua meta o *job APP1* altera para uma baixa vazão, assumindo uma postura altruísta. Esta postura permite que o *job APP2* acabe dentro do seu prazo, e imediatamente após o termino o *job APP1* o *job APP2* volta a assumir um comportamento guloso, pois está novamente sozinho no sistema.

6.2.7 Simulação de um cenário HPC

Neste experimento iremos simular, em uma escala reduzida, um cenário de execução de *jobs* em ambientes HPC. Criaremos um conjunto de jobs (um total de 10 *job*s) com metas, tempos de inicialização e cargas distintas. O objetivo é avaliar como os *jobs* comportam durante a execução em um cenário compartilhado.

Para este teste utilizamos a aplicação "*random-float*", paralelizada e executada junto ao middleware *EasyGrid AMS.* Utilizaremos neste experimento um número variado de tarefas para cada job, tendo sua própria carga de processamento. Neste experimento, utilizamos 4 hosts para a execução dos testes, 1 para executar GM e SM de cada job, e os três recursos restantes com os trabalhadores, porém variando a alocação e quantidade de host *workers* para cada *job.* Na Tabela 6.17 estão os dados pertinentes de cada *jobs* que irá executar no teste.

Tabela 6.17 Relação das caraterísticas de cada APP e seus tempos de inicio

Jobs	Carga tarefa	N° tarefas	N° Host	Tipo Meta	Tempo Meta (s)	Inicio (s)
APP1	20	1024	2(H1 e H2)	S	1600	0
APP2	30	128	2(H2 e H3)	H	280	0
APP3	20	1026	3(H1, H2 e H3)	S	1500	650
APP4	5	128	1(H1)	S	280	850
APP5	10	128	1(H2)	S	700	880
APP6	15	128	1(H3)	S	600	880
APP7	100	30	3(H1, H2 e H3)	H	800	900
APP8	30	64	2(H1 e H2)	S	1500	900
APP9	15	128	2(H2 e H3)	S	1200	930
APP10	20	2049	3(H1, H2 e H3)	S	2000	960

Cada uma das metas estipuladas para os *jobs* são metas viáveis de serem executadas dentro do cenário. O objetivo nesse teste é avaliar se em um ambiente dinâmico com múltiplas aplicações conseguiremos adaptar-se às metas dos *jobs*. Para iniciar o processo de avaliação, executamos o roteiro de execução das aplicações sem a existência da meta, apenas de modo estático para verificar os tempos de execução de cada *job*.

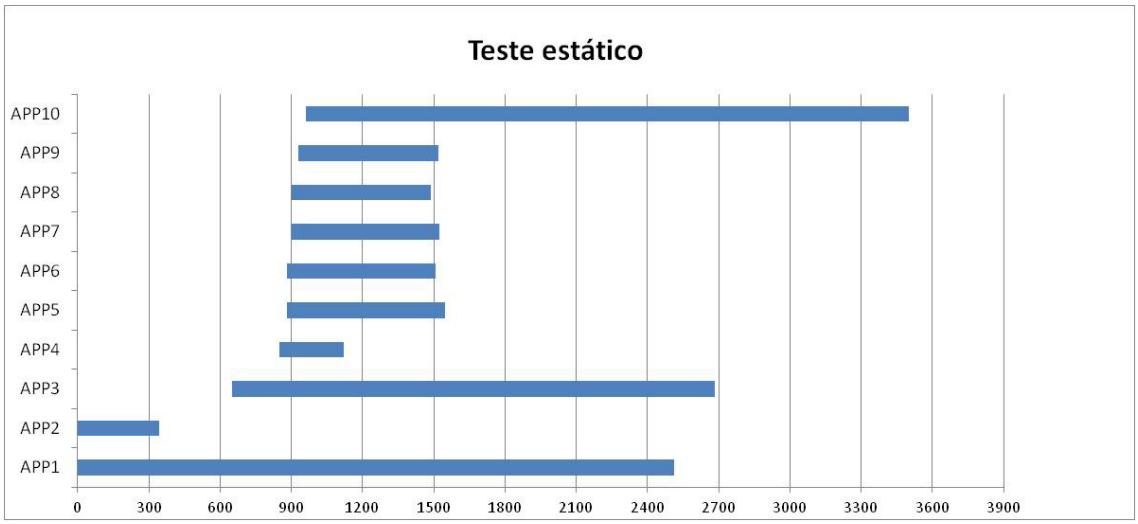

Figura 6.21 Resultado de execução teste estático

Avaliando o resultado da execução sem as metas e com a politica estática, podemos observar na Figura 6.21 e Tabela 6.18, que os jobs executaram livremente sem o controle do ambiente. Sendo o conjunto de tarefas de cada host gerenciado unicamente pela a alocação do SO, não tendo mecanismos para ajustar a meta oque consequentemente não atende aos requesitos de tempo apresentados na Tabela 6.18 para algumas das aplicações.

Tabela 6.18 Resultados execução política estática

Job	Inicio(s)	Fim(s)	Duração(s)	Tempo Meta(s)	Acabou no Prazo?
APP1	0	2513	2513	1600	**F**
APP2	0	342	342	280	**F**
APP3	650	2684	2034	1500	**F**
APP4	850	1123	273	280	**T**
APP5	880	1549	669	700	**T**
APP6	880	1508	628	600	**T**
APP7	900	1524	624	900	**T**
APP8	900	1486	586	1500	**T**
APP9	930	1521	591	1200	**T**
APP10	960	3502	2542	2000	**F**

Porém ao executarmos o mesmo cenário, com os mesmos *jobs* executando com a política SA, o qual irá realizar a priorização das tarefas baseados em suas metas, conseguimos os seguintes resultados:

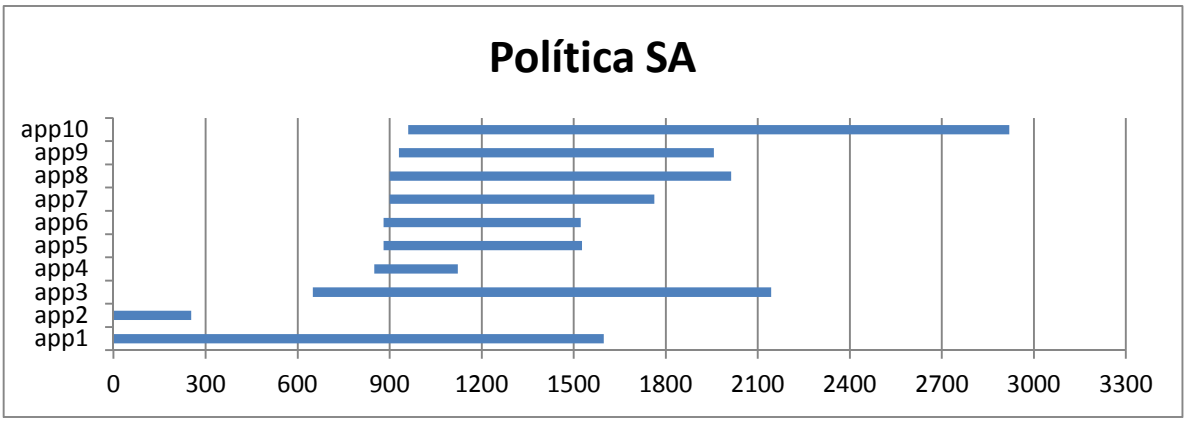

Figura 6.22 Execução política SA

Tabela 6.19 Resultado de execução política dinâmica

APP	Inicio(s)	Fim(s)	Duração(s)	Meta(s)	Acabou no Prazo?
APP1	0	1.597,76	1.597,76	1.600,00	**T**
APP2	0	252,99	252,99	280,00	**T**
APP3	650	2.143,63	1.493,63	1.500,00	**T**
APP4	850	1122,43	272,43	280,00	**T**
APP5	880	674,50	647,50	700,00	**T**
APP6	880	1527,50	642,40	700,00	**T**
APP7	900	1762,44	862,44	900,00	**T**
APP8	900	2013,02	1.113,02	1.500,00	**T**
APP9	930	1957,13	1.027,13	1.200,00	**T**
APP10	960	2.919,83	1959,83	2.000,00	**T**

6.3 Resumo

No conjunto dos experimentos realizados, nós tentamos refletir os principais cenários do ambiente compartilhado proposto e como nossa heurística adapta-se à execução. A proposta é uma abordagem de como escalonadores não centralizados com compartilhamento de recursos podem demonstra-se poderosos no escalonamento de tarefas, no aproveitamento dos hosts e ainda sim consegui manter o controle sobre a execução dos *jobs*.

Capítulo 7 – Conclusão

Os ambientes computacionais distribuídos são uma importante ferramenta na viabilização e execução de aplicações. Sua utilização nos permite a execução de aplicações que necessitam de alto poder computacional, tornando os tempos para a execução destes processos mais aceitáveis. Os paradigmas iniciais de fornecimento de computação de alto desempenho como os clusters e grids, tenham iniciado com modelos de negócio altamente custosos para usuários de médio e pequeno porte por necessitar a aquisição previa do ambiente e constante manutenção. Com a crescente subutilização dos parques computacionais de grandes empresas, surgiu um novo paradigma de computação, as nuvens computacionais. Usufruindo dos avanços tecnológicos de virtualização, a abordagem viabiliza um modelo de negocio para disponibilizar a capacidade sobrando em seus data-centers. Com um oferta de recursos sob demanda de cobrar somente o período utilizado, usuários podem alugar um ambiente computacional ou um serviço por a um preço bastante acessível.

Entretanto, viabilidade econômica não é suficiente para garantir que o usuário consiga a execução de sua aplicação em este ambiente distribuído e compartilhado. O processo de criação e execução de aplicações distribuídas é rodeado por inúmeros conhecimentos necessários. A começar pelo paradigma de programação paralela, que rever a forma de pensar sobre soluções de problemas simples no universo sequencial, ao framework de paralelização utilizado, serão necessários conhecimento de conceitos e técnicas da paralelização da aplicação. Também faz-se necessário ter conhecimentos adicionais sobre as heurísticas de escalonamento e conhecimentos sobre a infraestrutura a qual está utilizando. Todos estes requisitos tornam a vida do desenvolvedor da aplicação muito custosa, pois mesmo com todos esses conhecimentos, é possível que a aplicação não consiga utilizar todos os recursos que estão alocados para si, gerando ociosidade. Esta ociosidade que em um ambiente com um escalonador realiza reservas de recursos, representa uma maior espera na fila para a execução do seu job. Em ambientes como a nuvem, isso pode representar um acréscimo desnecessário na fatura da locação considerando que a cobrança por Host/Tempo é muito comum. A execução de aplicações distribuídas em ambientes com reserva de recursos tipicamente obriga a satisfazer duas condições para garantir a execução eficiente do *job*. Primeiro, é necessário reservar recursos suficiente para o pico da execução, e segunda, aloca esta quantidade de recursos exclusivamente à aplicação para duração de sua execução. Esse dois fatores juntos fazem com que a reserva de hosts para a execução dos *jobs* seja maior do que necessária.

Quando o problema se estende para os grandes ambientes computacionais com milhares de *jobs,* a obtenção de um bom escalonamento se torna uma tarefa tão custosa que pode passar a ser maior que o próprio tempo para execução dos *jobs* em alguns casos. Caracterizado como um problema de *job shop scheduling*, um dos problemas clássicos da área de programação distribuída, o problema consiste de: escalonar um dado conjunto de *n jobs* em *m* diferentes *hosts*, definindo uma ordem de execução que minimize o *makespan* total dos *n jobs*.

Existam ferramentas, como o EasyGrid AMS, que permitem melhorar esses aspectos, o da subutilização dos recursos alocados para um job, otimizando o processo de escalonamento das tarefas de um job, acelerando seu término e, consequentemente, liberando os recursos mais rapidamente. Os problemas como *Job Shop Scheduling* continuam existindo, e ainda pode agravar-se ao levarmos em consideração as características de execução da aplicação e a tendência de consolidar cada vez mais recursos num servidor, o que pode aumentar ainda mais a subutilização.

Neste projeto apresentamos uma proposta de um novo modelo de execução de *jobs* nos ambientes distribuídos. Um modelo que, diferente do usual, não realiza a reserva de recursos. Neste modelo, o papel do broker central se resume no processo de aceito do job a ganhar acesso aos recursos, deixando todo o resto da decisão de onde e quando rodar para ser tomado durante a execução pela própria aplicação. Os *jobs* por sua vez executaram nos diversos *hosts*, mesmo que existam outros jobs executando concorrentemente, podendo compartilhar parte ou todos os recursos que estejam utilizando e decidindo se usarão ou não os hosts disponíveis. Cada aplicação mantem duas diretivas em mente: preciso acabar dentro da minha meta, e; quando puder, preciso ajudar os demais *jobs* também terminar dentro de suas metas. Queremos com esta proposta sair da abordagem clássica, com reserva de recursos, comportamentos gulosos e utilizações relativamente baixas para um uso eficiente de ambientes compartilhados e altruístas. Tudo isso é claro, sem perder o controle existente sobre as prioridades dos *jobs* e com a redução da ociosidade dos recursos. Nossos meios para prover esse ambiente de execução é através da modificação do middleware EasyGrid AMS, projeto pertencente ao SGCLab do IC-UFF, com a adição de um conjunto de comportamentos (heurísticas) que monitoram a execução do *job* adaptando ele numa forma autônoma para que execute e termine dentro da meta definida.

No Capitulo 6, apresentamos alguns dos cenários de execução que acreditamos serem os mais prováveis com os quais devemos nos deparar no ambiente computacional proposto.

Cenários esses que foram definidos ao longo de experimentos de execução concorrente e análise de resultados, como: a execução de um único job no ambiente computacional distribuído, a priorização dos jobs pelas metas quando executando concorrentemente, a execução de múltiplas aplicações com metas folgadas, e a simulação do ambiente de teste proposto com mais diversos *jobs* onde todos esses comportamentos acontecem. Nestes cenários avaliamos como os *jobs* em execução se adaptaram à realidade do ambiente, que era sujeito a alterações, priorizando alguns jobs e detectando as modificações. Através dos testes realizados, podemos validar nossa proposta com um conjunto de métricas: verificação do altruísmo da aplicação, finalização dentro das metas definidas, ociosidade dos recursos e adaptação às variações. Métricas que demonstraram que a implementação apresentou-se eficiente no término da aplicação atendendo os requesitos. Conseguindo priorizar a execução das aplicações baseadas em sua meta, manter uma baixa ociosidade e adaptar-se ao longo da execução aos recursos existentes no ambiente.

Todos os testes realizados nesse trabalho foram realizados com uma modificação do middleware EasyGrid AMS para que alcance o comportamento proposto, o de uma sociedade de aplicações que cooperam entre si sem comunicação direta para coordenar o término dos *jobs* concorrentes em suas respectivas metas.

Embora tenhamos conseguido um grande avanço nesse trabalho na criação de um ambiente onde as aplicações autônomas possam executar concorrentemente, otimizando o consumo de CPU e ainda mantendo um controle sobre a execução, ainda existe viabilidade para vários trabalhos. Este projeto iniciou uma proposta de ambiente de execução novo, um ambiente distribuído onde as aplicações executam e comportam-se como em uma sociedade, sujeitos a regras e comportamentos. A implementação apresentada nesse trabalho é só um estudo de viabilidade e outro comportamentos devem ser investigados, inclusivo o compartilhamento de outros tipos de recursos (além de CPU), para realizar e analisar nosso conceito completo de representar ambientes de execução como uma sociedade de aplicações. Uma sociedade de aplicações onde cada aplicação se auto gerencia, pode ser estendida para prover um conceito de escalonamento muito mais amplo. Por ser um ambiente de execução mais escalável, altamente maleável e adaptável, onde as aplicações passariam a auto organizar durante sua execução, o potencial para explorar diferentes combinações de comportamentos de uma Sociedade de Aplicações seria bastante interessante.

REFERÊNCIAS

[1] MELL, Peter; GRANCE, Tim. The NIST definition of cloud computing. 2011.

[2] LAURE, Erwin et al. Middleware for the next generation Grid infrastructure. **Computing in High Energy Physics and Nuclear Physics (CHEP 2004)**, 2004.

[3] FOSTER, Ian; KESSELMAN, Carl (Ed.). **The Grid 2: Blueprint for a new computing infrastructure**. Elsevier, 2003.

[4] Amazon AWS,Elastic Cloud 2.Disponível em: *https://aws.amazon.com/ec2/?nc2=h_ls*, último acesso em 1 de Outubro de 2016.

[5] BOERES, Cristina; REBELLO, Vinod EF. EasyGrid: Towards a framework for the automatic grid enabling of legacy MPI applications. **Concurrency and Computation: Practice and Experience**, v. 16, n. 5, p. 425-432, 2004.

[6] COFFMAN, Edward Grady; BRUNO, John L. **Computer and job-shop scheduling theory**. John Wiley & Sons, 1976.

[7] AHMAD, Ishfaq; RANKA, Sanjay; KHAN, Samee Ullah. Using game theory for scheduling tasks on multi-core processors for simultaneous optimization of performance and energy. In: **Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on.** IEEE, 2008. p. 1-6.

[8] LI, Xinyu; GAO, Liang; LI, Weidong. Application of game theory based hybrid algorithm for multi-objective integrated process planning and scheduling. **Expert Systems with Applications**, v. 39, n. 1, p. 288-297, 2012.

[9] MORAD, Norhashimah; ZALZALA, A. M. S. Genetic algorithms in integrated process planning and scheduling. **Journal of Intelligent Manufacturing**, v. 10, n. 2, p. 169-179, 1999.

[10] SHAO, Xinyu et al. Integration of process planning and scheduling—a modified genetic algorithm-based approach. **Computers & Operations Research**, v. 36, n. 6, p. 2082-2096, 2009.

[11] LI, Xiaorong; PRODAN, Radu; DUAN, Rubing. Multiobjective Game Theory-based Schedule Optimization for Bags-of-Tasks on Hybrid Clouds. **IEEE Transactions on Cloud Computing**, n. 1, p. 1, 2014.

[12] YUAN, Wanghong; NAHRSTEDT, Klara. Energy-efficient soft real-time CPU scheduling for mobile multimedia systems. In: **ACM SIGOPS Operating Systems Review. ACM**, 2003. p. 149-163.

[13] STALLINGS, William; PAUL, Goutam Kumar. **Operating systems: internals and design principles**. Upper Saddle River, NJ: prentice hall, 1998.

[14] TANENBAUM, Andrew S.; VAN STEEN, Maarten. **Distributed systems**. Prentice-Hall, 2007.

[15] KERSTING, William H. **Distribution system modeling and analysis**. CRC press, 2012.

[16] VOLKERT, Jens. Austrian grid: Overview on the project with focus on parallel applications. In: **2006 Fifth International Symposium on Parallel and Distributed Computing**. IEEE, 2006. p. 14-14.

[17] FOSTER, Ian; KESSELMAN, Carl. Globus: A metacomputing infrastructure toolkit. **International Journal of High Performance Computing Applications**, v. 11, n. 2, p. 115-128, 1997. (1997): 115-128.

[18] KURDI, Heba; LI, Maozhen; AL-RAWESHIDY, Hamed. A classification of emerging and traditional grid systems. **IEEE Distributed Systems Online**, v. 9, n. 3, p. 1-1, 2008.

[19] The Data Grid Project, home. Disponível em: *http://eu-datagrid.web.cern.ch*, último acesso em 1 de Fevereiro de 2016.

[20] Globus Toolkit, home. Disponível em: *http://www.globus.org/toolkit*, último acesso em 1 de janeiro de 2016.

[21] Berkeley University California, Boinc. Disponível em: *https://boinc.berkeley.edu*, último acesso em 1 de janeiro de 2016.

[22] ARMBRUST, Michael et al. A view of cloud computing. **Communications of the ACM**, v. 53, n. 4, p. 50-58, 2010.

[23] Heroku, home. Disponível em: *https://www.heroku.com,* último acesso em 1 de junho de 2016.

[24] Google, Cloud Platform. Disponível em: *https://cloud.google.com/compute,* último acesso em 1 de junho de 2016.

[25] ASANOVIC, Krste et al. **The landscape of parallel computing research: A view from berkeley**. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[26] OpenMP, especifications. Disponível em: *http://openmp.org/wp*, último acesso em 1 de junho de 2016.

[27] MPI Forum, "A message-passing interface standart". Disponível em: . *http://www.mpi-forum.org/docs*, último acesso em 1 de Outubro de 2016.

[28] IBM, MPI-F. Disponível em: *http://www-03.ibm.com/systems/platformcomputing/products/mpi*, último acesso em 1 de junho de 2016.

[29] MPICH, High Performance Portable MPI. Disponível em: *https://www.mpich.org*, último acesso em 1 de junho de 2016

[30] Intel, MPI Libry. Disponível em: *https://software.intel.com/en-us/articles/intel-mpi-library-documentation*, último acesso 1 de junho de 2016

[31] Open MPI, Open source High Performance Computing. Disponível em: *https://www.open-mpi.org*, último acesso em 1 de junho de 2016.

[32] The University of Tennessee, FT-MPI. Disponível em: *https://icl.cs.utk.edu/ftmpi*, último acesso em 1 de junho 2016.

[33] Los Alamos National Laboratory, LA-MPI. Disponível em: *http://public.lanl.gov/lampi*, último acesso em 1 de junho2016.

[34] The University of Edinburgh, LAM/MPI. Disponível em: *http://www.dcs.ed.ac.uk/home/trollius/www.osc.edu/lam.html*, último acesso em 1 de junho 2016.

[35] TOP-500, lista de junho 2016. Disponível em:

https://www.top500.org/lists/2016/06, último acesso em 1 de junho de 2016.

[36] NASCIMENTO, A. P. **Escalonamento Dinâmico para Aplicações Autonômicas MPI em Grades Computacionais**. 2008. Tese de Doutorado. PhD thesis, Instituto de Computação, Universidade Federal Fluminense, Niterói, Brasil.

[37] NASCIMENTO, Aline P. et al. Distributed and dynamic self-scheduling of parallel MPI Grid applications. **Concurrency and Computation: Practice and Experience**, v. 19, n. 14, p. 1955-1974, 2007.

[38] NASCIMENTO, Aline P.; BOERES, Cristina; REBELLO, Vinod EF. Dynamic self-scheduling for parallel applications with task dependencies. In: **Proceedings of the 6th international workshop on Middleware for grid computing**. ACM, 2008. p. 1.

[39] BOERES, Cristina et al. An EasyGrid portal for scheduling system-aware applications on computational Grids. **Concurrency and Computation: Practice and Experience**, v. 18, n. 6, p. 553-566, 2006.

[40] SENA, A. **Um modelo alternativo para execuç ao eficiente de aplicaç oes paralelas MPI nas Grades Computacionais.** 2008. Tese de Doutorado. PhD thesis, Universidade Federal Fluminense.

[41] Bueno, H. R. **Grid SA: Uma Sociedade Autônoma**. 2009. Tese de Mestrado. Master thesis, Universidade Federal Fluminense.

[42] Microsoft, Azure. Disponível em:
http://azure.microsoft.com/en-us, último acesso em 1 de junho de 2016

[43] Wikipédia, sar (Unix). Disponível em:
https://en.wikipedia.org/wiki/Sar_(Unix), último acesso em 1 de junho de 2016.

[44] Oliveira, F. J. B. **Compartilhando Recursos Computacionais na Direção de uma e-Sociedade de Aplicações**. 2013. Trabalho de Conclusão Curso. Term Paper, Universidade Federal Fluminense.

[45] Ferreira, A. A., **Pela lente do CFS : um olhar sobre o escalonador Linux**. 2012. Tese de Mestrado. Master thesis, Universidade Federal Fluminense.

[46] Castellani, P. H. S., **Understanding Linux's CPU Resource Management** .2012. Tese de Mestrado. Master thesis, Universidade Federal Fluminense.