Análise de desempenho de algoritmos de escalonamento em simuladores de grades

Alvaro Henry Mamani Aliaga

December 17, 2009

Abstract

In this days, the Grid environments are very important. Many applications need increasing processing capabilities. More now, that the cloud computing [PL09] is offered and need a lot of processing capabilities in the cloud. One solution to this problem is the Grid Computing. But until now there are many solutions, middlewares to do Grid Computing. Grid environments is too important, but in this context appear the "Scheduler", if a scheduling algorithm works right, the scheduler and the Grid Middleware works much better. In this paper, we studied several scheduling algorithms and their simulations, this will help to better understand them in different environments to develop work related.

keywords: *grid computing, scheduling algorithm, grid simulation*

Resumo

Na atualidade, o ambiente das grades são muito importantes. Muitas aplicações precisam alta capacidade no processamento. Mais agora, que a computação na nuvem [PL09] é oferecida e precisa de boa capacidade no processamento. Uma solução para este problema é a computação em grade. Até agora temos muitas soluções(*middlewares*) para fazer computação grade. Ambientes em grade são muito importantes, mas em este contexto aparecem os escalonadores, se um algoritmos de escalonamento trabalha bem, o escalonador e o *middleware* trabalham ainda melhor. Em este trabalho, nós estudamos diferentes algoritmos de escalonamento e suas simulações, isto vai ajudar ao melhor entendimento deles em diferentes ambientes para desenvolver trabalhos relacionados.

Palavras chave: *computação grade, algoritmos de escalonamento, simuladores em grades*

1 Introdução

As ciências da computação, estão presentes em todos os ambientes profissionais, tanto academicamente quanto industrialmente, a medida que o tempo passa numerosas aplicações sofisticadas em todos os níveis de pesquisa precisam de poder computacional cada vez maior.

Nos recentes anos, tem sido acrescentado a disponibilidade de computadores poderosos e redes de alta velocidade. Este fato tem feito isto possível, para agregar recursos geograficamente dispersados para execução de tarefas de aplicações de longa escala e recursos intensivos. Esta agregação de recursos tem sido chamado de **Computação em Grade** (*Grid Computing*) [FJMJBO07], assim, a computação em grade é uma alternativa para obter grande capacidade de processamento.

A Computação Grade tem o potencial para fazer computação a larga escala, usando esse valor adicional, que é o poder computacional ao permitir computadores domésticos ou de escritórios conectadas ao internet. A disponibilidade do poder computacional na grade muda o tempo todo porque esse valor adicional subministrado por esses computadores é dinâmico e varia como o pico de desempenho destes variedade de computadores. É assim que o uso de um adequado algoritmo de escalonamento é tão importante, respeitando as políticas de um determinado cenário.

No trabalho pretende analisar alguns algoritmos de escalonamento, cuja área de ação é a computação grade. Eles serão avaliados através da simulação, para finalmente propor certo algoritmo segundo um determinado cenário, que ofereça bom desempenho, neste caso o nosso cenário serão aplicações seqüenciais, portanto a nossa avaliação será para algoritmos de escalonamento para tarefas seqüenciais.

Os algoritmos escolhidos, avaliados neste trabalho são:

- Dynamic FPLTF.
- Sufferage.
- Workqueue
- Workqueue with Replication.
- XSufferage
- Storage Affinity

2 Algoritmos de escalonamento

Graças aos avanços em tecnologias de redes de longa distância e um custo baixo de recursos computacionais, a Computação em Grade surgiu e é atualmente uma área de investigação ativa.

Uma motivação de Computação em Grade, é a agregação de poder de recursos amplamente distribuídos, e fornecer serviços não triviais aos usuários. Para melhorar o alvo, um sistema eficiente de escalonamento em grade é uma parte essencial da grade.

Em sistemas tradicionais paralelos e distribuídos, os quais geralmente são executados sobre recursos dedicados homogêneos, por exemplo, *clusters*, os recursos computacionais são geralmente gerenciados por um único ponto de controle, o escalonador (*Scheduler*). O escalonador não só tem total informação acerca de todas as tarefas *running/pending* e utilização dos recursos, mas também gerência a **fila de tarefas** e o conjunto de recursos. Assim, pode facilmente prever o comportamento dos recursos, e é capaz de atribuir tarefas aos recursos, de acordo com determinados requisitos. Em uma Grade, no entanto, os recursos são usualmente autônomos e o escalonador da grade não tem controle total dos recursos. Ele não pode violar as políticas locais de recursos, o que torna difícil para o escalonador da grade estimar o custo exato da execução de uma tarefa em locais diferentes. Assim, num escalonador da grade, é necessário para se adaptar a diferentes políticas locais.

Na atualidade existem algoritmos de escalonamento em diferentes cenários. É bem sabido que a complexidade do problema geral de escalonamento é NP-Completo [MOM+07]. O problema de escalonamento se torna mais desafiador devido a algumas características únicas pertencentes à computação em grade [DA07], onde se apresenta a seguinte taxonomia dos algoritmos de escalonamento, mostrado na figura 1.

Num ambiente de computação em grade, cada recurso é dinâmico, recursos novos podem ser agregados e outros podem ser retirados, neste tipo de ambientes dinâmicos podem ser considerado um escalonador que trabalhe com algoritmos de escalonamento, dinâmicos.

2.1 Dynamic FPLTF

Um bom representante de escalonamento estático é *Fastest Processor to Largest Task First (FPLTF)* [MSP+95]. Contudo, a dinamicidade e heterogeneidade de recursos presentes sobre grades fazem que escalonadores estáticos não sejam uma boa solução. Para lidar com este problema o FPLTF possui uma versão dinâmica, o Dynamic FPLTF.

Dynamic FPLTF precisa de três tipos de informação para escalonar. tamanho da tarefa (*Task Size*), carga da máquina (*Host Load*) e velocidade da máquina (*Host Speed*).

- Velocidade da máquina: representa a velocidade da máquina e o valor relativo dele. Por exemplo, uma máquina que tem uma $VelocidadeDaMaquina = 2$ executa uma tarefa duas vezes mais rápido que uma máquina com $VelocidadeDaMaquina = 1$.
- Carga do máquina: representa a fração da máquina que não está disponível para a aplicação.

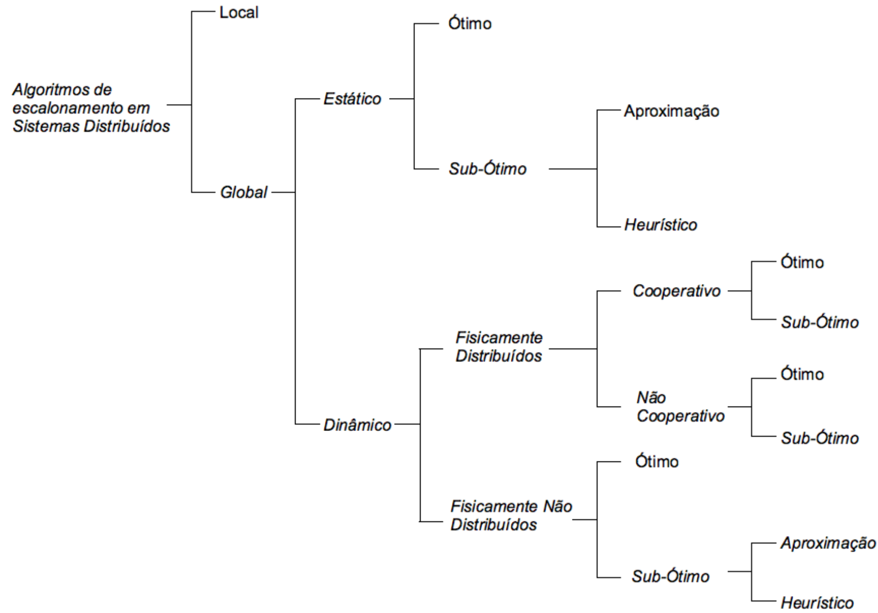

Figura 1: A taxonomia hierárquica para os algoritmos de escalonamento. Os algoritmos para a computação em grade, estão indicados em itálico. Fonte: [DA07].

- Tamanho da tarefa: é o tempo necessário para uma máquina com $VelocidadeDaMaquina = 1$ completar a tarefa dela quando $CargaDaMaquina = 0$.

No começo do algoritmo, temos um **tempo para se tornar disponível** (TBA, *Time to Become Available*), o TBA de cada máquina é inicializado com 0, e as tarefas são ordenadas pelo tamanho (ordem não crescente). Portanto, a tarefa mais longa é alocada primeiro. Uma tarefa é alocada a uma máquina que fornece o melhor tempo de conclusão (CT, *Completion Time*), assim temos:

$$CT = TBA + CustoDaTarefa \tag{1}$$

onde na ecuação 1, o *CustoDaTarefa* pode ser calculado assim:

$$CustoDaTarefa = \frac{\frac{TamanhoDaTarefa}{VelocidadeDaMaquina}}{1 - CargaDaMaquina} \tag{2}$$

Quando uma tarefa é alocada a uma máquina, o valor do *TBA* corresponde a esta máquina é acrescentado pelo *CustoDaTarefa*. Tarefas são alocadas até

todas as máquinas da grade serem usadas. Depois, a execução da aplicação começa. Quando uma tarefa termina, todas as tarefas que não estão rodando são desalocadas e escalonadas novamente até todas as maquinas chegarem a ser usadas. Este cenário continua até todas as tarefas serem completadas.

Esta estratégia tenta minimizar os efeitos da dinamicidade da grade. A máquina que, em um primeiro momento, é muito rápida e carga ligeira, toma mais tarefas que uma máquina muito rápida mas com carga pesada. Este problema de variação de carga comprometeria a aplicação inteira desde as tarefas escalonadas a esta máquina poderiam serem executadas muito mais lento. A tarefa reescalonada processa corretamente este problema por alocar tarefas longas priorizando a máquinas atualmente rápidas.

Este cenário deixa um bom desempenho o qual será avaliado, mas é complicado implementá-lo na prática. *Dynamic FPLTF* precisa muita informação acerca do ambiente (Tamanho da tarefa, Carga da máquina e Velocidade da máquina). Este tipo de informação algumas vezes é difícil obter e freqüentemente não está disponível devido a restrições administrativas fazendo dela uma solução não viável em muitos casos [SCBG03].

2.2 Sufferage

Sufferage [CZBL00] é uma heurística dependente de dados sobre as tarefas e os recursos da grade.

No começo, para cada máquina da grade, é calculado seu tempo de conclusão mínimo (MTC, *Minimum Completion Time*). O cálculo do tempo de conclusão mínimo é similar como para o Dynamic FPLTF.

A lógica por trás *Sufferage* é que uma máquina deve ser atribuída à tarefa que vai "sofrer" mais se não for atribuída a essa máquina. Para cada tarefa, o seu valor *sufferage* é definida como a diferença entre o seu melhor templo de conclusão mínimo e do seu segunda melhor templo de conclusão mínimo. Tarefa com maior valor *sufferage* prevalece [CZBL00].

Se outra tarefa foi previamente atribuída à máquina, o valor *sufferage* da tarefa previamente atribuída e da nova tarefa são comparadas. A tarefa que esteja na máquina é a tarefa que tem o valor *sufferage* maior, a outra retorna para ser alocada depois [SCBG03].

É claro que o valor *sufferage* de cada tarefa muda durante a execução da aplicação, a dinamicidade de carga é intrínseca em ambientes como grades. Para lidar com isto, nós executamos ao algoritmos descrito acima muitas vezes durante a execução da aplicação. Cada vez que uma tarefa termina, todas as tarefas que ainda não começaram serão desalocadas e o algoritmo é invocado novamente, usando valores atuais de *sufferage*. Consequentemente, o algoritmo roda outra vez escalonando as tarefas restantes, mas nesta vez com a nova carga de maquinas. Este cenário é repetido até que todas as tarefas tenham sido completadas.

O problema deste algoritmo é o mesmo problema do *Dynamic FPLTF*; este precisa de muita informação para calcular o tempo de conclusão (*Completion Time*) das tarefas, além de conhecer *tamanho da tarefa, carga da máquina e velocidade da máquina* [SCBG03].

2.3 Algoritmo Workqueue

Workqueue é uma escalonamento de conhecimento livre, no sentido que não precisa de qualquer tipo de informação para escalonar tarefas. Tarefas são escolhidas em uma ordem arbitrária e enviadas ao processador, tão logo eles chegarem a ficar disponíveis. Depois do término de uma tarefa, o processador envia o resultado e o escalonador atribui uma nova tarefa ao processador. Isto é, o escalonador começa enviando uma tarefa para cada máquina disponível. Uma vez que a máquina termina sua tarefa, o escalonador atribui outra tarefa à máquina (se ainda existem tarefas para serem processadas).

A idéia atrás disto é que mais tarefas serão atribuídas para máquinas rápidas ou ociosas enquanto que maquinas lentas ou ocupadas processarão uma pequena carga. A vantagem aqui é que *Workqueue* não depende de informação de desempenho [SCBG03].

O problema com *Workqueue* surge quando uma grande tarefa é atribuída a uma máquina lenta no final do escalonamento. Quando isso ocorre, a conclusão da aplicação será adiada até a completa execução desta tarefa.

2.4 Algoritmo WorkQueue with Replication

O algoritmo *Workqueue with Replication*(WQR), foi criado para solucionar problemas que não precisam de informação dos recursos da grade para fazer o escalonamento.

Este algoritmo de escalonamento é similar a *Workqueue*, diferindo apenas a partir do momento em que todas as tarefas da fila de tarefas forem alocadas para execução.

As tarefas são enviadas para execução nos processadores que se encontram disponíveis em uma ordem aleatória. Quando um processador finaliza a execução de uma tarefa, este recebe uma nova tarefa para processar.

No momento em que um processador se torna disponível e não há mais nenhuma tarefa pendente para executar. Neste momento, *Workqueue* já finalizou seu trabalho e apenas aguarda a finalização de todas as tarefas. Porém, o WQR inicia sua fase de replicação para as tarefas que ainda estão executando. Note que, na fase de replicação, quando uma tarefa original finaliza sua execução primeiro que suas réplicas, estas são interrompidas. Caso contrário, quando alguma réplica finaliza primeiro , a tarefa original e as demais réplicas dela são interrompidas [dSN04].

Note que a replicação assume que as tarefas não causam efeitos colaterais. Esta hipótese é razoável quando se fala de ambientes da computação em grade,

porque não é comum ter este tipo de aplicação em ambientes como esse, devido à sua ampla distribuição. [SCBG03].

2.5 Algoritmo XSufferage

O algoritmos de escalonamento *XSufferage* é uma heurística de escalonamento que se baseia nas informações sobre o desempenho dos recursos e da rede que os interliga.

XSufferage é uma extensão de heurística de escalonamento *Sufferage*. A idéia básica da heurística *Sufferage* é determinar quanto cada tarefa seria prejudicada "sofreria" se não fosse escalonada no processador que a executaria de forma mais eficiente. Portanto, *Sufferage* prioriza as tarefas de acordo com o valor que mede o prejuízo "sofrimento" de cada tarefa. Este valor é denominado de *sufferage* e definido como a diferença entre o melhor e o segundo melhor tempo de execução previsto para a tarefa, considerando todos os processadores da grade [dSN04].

A principal diferença entre *Sufferage* e *XSufferage* é o método usado para calcular o valor do "sufferage". Na heurística *XSufferage*. No *XSufferage* o valor do "sufferage" não é calculado só com o **tempo de conclusão mínimo**, mas sim considerando o nível de **tempo de conclusão mínimo** dos clusters (*cluster-level MCTs*), ou seja, **tempo de conclusão mínimo** pelo cálculo do mínimo sobre todos as máquinas em cada cluster. Assim para cada tarefa é calculado o valor do "sufferage" no cluster, esse valor é a diferença entre o melhor e o segundo melhor valor *cluster-level MCTs* [CZBL00].

Portanto o algoritmo de escalonamento *XSufferage* usa informações sobre os níveis do cluster, os quais precisam estar disponíveis no momento em que o algoritmo vai alocar as tarefas. Algumas das informações que ele precisa são:

- disponibilidade de CPU;
- disponibilidade da rede;
- Os tempos de execução das tarefas.

Estas informações devem ser conhecidas antes de escalonamento e são utilizadas para decidir qual tarefa deve ser escalonada em qual processador.

Um ponto importante a ser observado é que o algoritmo considera somente os recursos livres no momento em que vai escalonar uma tarefa, pois caso contrário sempre o recurso mais rápido e com a melhor conexão de rede receberia todas as tarefas [FJMJBO07].

2.6 Algoritmo Storage Affinity

O algoritmo *Storage Affinity* [SNCB+04] foi feito com o intuito de explorar a reutilização de dados a fim de melhorar o desempenho de aplicações que utilizam grandes quantidades de informação. O método de escalonamento de tarefas é definido sobre o conceito fornecido pelo próprio nome do algoritmo, a **afinidade**,

o valor da afinidade entre uma tarefa e uma máquina determina quão próximo da máquina esta tarefa está. A semântica do termo *próximo* está associada à quantidade de bytes da entrada da tarefa que já está armazenada remotamente em uma dada máquina, assim, quanto mais bytes da entrada da tarefa estiver armazenado na máquina, mais próximo a tarefa estará da máquina, pois possui mais *storage affinity*. [dSN04] [SNCB+04].

Alem de aproveitar a reutilização de dados, o *Storage Affinity*, também realiza replicação de tarefas, tratando da dificultade de obtenção de informações dinâmicas sobre a grade, e sobre o tempo de execução das tarefas. A idéia é que as replicas tenham a chance de serem submetidas a processadores mais rápidos do que aqueles associados à tarefas originais, reduzindo o tempo de execução da aplicação [dSN04] [SNCB+04].

3 Simuladores

A pesquisa de algoritmos de escalonamento (*scheduling algorithms*) para aplicações distribuídas tem sido o foco dos esforços de pesquisas ao longo dos anos.

A estratégia mais simples para comparar a eficácia dos algoritmos seria realizar experimentos reais, programando aplicações reais em recursos reais. Mas isto, não é apropriado por três razões:

- Aplicações reais devem rodar aplicações por longos períodos de tempo, e não é viável para executar um grande número de experimentos tipo simulação neles.
- A utilização de recursos reais, torna difícil a tarefa de explorar uma grande variedade de configurações de recursos.
- Variações na carga de recursos ao longo do tempo tornam difícil a obtenção de resultados reproduzíveis.

Por meio das ferramentas de simulação, torna-se possível avaliar e comparar o desempenho de diferentes algoritmos em diferentes cenários. Assim a simulação é a abordagem mais viável para avaliar efetivamente aos algoritmos de escalonamento. Neste cenário existem diversos simuladores, por exemplo:

- SimGrid [CLQ08].
- GridSim [BM02].
- OGST [Fil09]
- Alea [KMR07]

Neste trabalho tomamos os dois primeiros simuladores.

3.1 SimGrid

SimGrid [CLQ08] é uma ferramenta que fornece de funcionalidades chaves para simulação de aplicações distribuídas em ambientes distribuídos heterogêneos. O objetivo específico do simulador é facilitar a pesquisa na área de aplicações de escalonamento paralelas e distribuídas em plataformas de computação distribuída que vão desde simples redes de estações até Grades computacionais.

3.1.1 Componentes

Como descrito na figura 2, a ferramenta SimGrid é basicamente de três camadas.

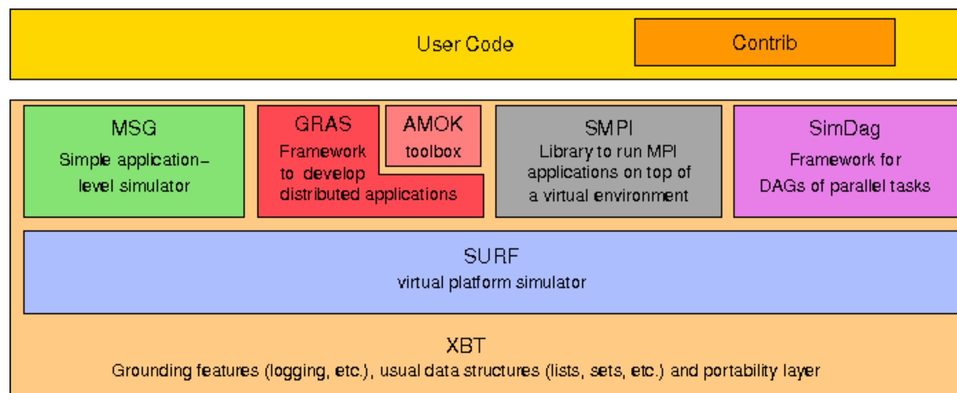

Figura 2: Módulos do *SimGrid*

Camada: Ambientes de Programação SimGrid fornece diversos ambientes de programação, construído sobre um único núcleo(*kernel*). Cada ambiente objetiva um alvo específico e constituí um paradigma diferente. Para escolher qual usar, tem que se pensar sobre que se quer fazer e qual seria o resultado do trabalho. A figura 3 mostra os componentes da camada.

- *MSG*: Foi o primeiro ambiente de programação disponibilizado e é o de uso mais difundido. É usado para modelar aplicações como processos seqüenciais concorrentes(*Concurrent Sequential Processes*). Útil para modelar problemas teóricos e para comparar diferentes heurísticas.
- **SMPI**: (*Simulated MPI*), para simulações de códigos MPI. Simulação do comportamento de uma aplicação MPI usando técnicas de emulação.
- **GRAS**: (*Grid Reality And Simulation*), possibilita a execução de aplicações reais para o estudo e teste deles. Acima da API do GRAS, tem uma *toolkit* chamada AMOK (*Advanced Metacomputing Overlat Kit*) que implementa em alto nível diversos serviços necessários a várias aplicações distruibuídas.

- **SimDag**: ambiente dedicado à simulação de aplicações paralelas, por meio do módelo DAG(*Direct Acyclic Graphs*). Com este modelo é possível especificar relações de dependência entre tarefas de um programa paralelo.

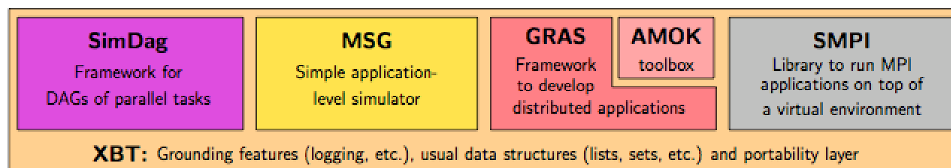

Figura 3: Camada: Ambientes de Programação(*Programmation environments*)

Camada: Núcleo de simulação O núcleo das funcionalidades, para simular uma plataforma virtual é fornecida pelo módulo chamado *SURF*. É muito baixo nível e não se destina a ser utilizado como tal pelos utilizadores finais. Em vez disso, servirá de base para a camada de nível superior.

Uma das principais características do *SURF* é a capacidade de mudar de forma transparente o modelo utilizado para descrever a plataforma. Isto facilita grandemente a comparação dos vários modelos existentes na literatura.

Camada: Base A base da ferramenta é constituída pelo XBT(*eXtended Bundle of Tools*).

É uma biblioteca portável, fornecendo alguns recursos como suporte de registro, suporte e apoio de exceção de configuração. Além disso, o XBT provê suporte à portabilidade da ferramenta.

3.1.2 Implementação

O *SimGrid* é implementado em linguagem C. Otimizações efetuadas no código melhoram o uso de memória e a velocidade de execução. Algumas efetuadas das técnicas de otimização são empregadas na manipulação dos *traces*: eles podem ser compartilhados pelos recursos; conjuntos grandes de *traces* são carregados em memória apenas quando necessários, e descartados após usados.

3.2 GridSim

GridSim [BM02] permite modelagem e simulação de entidades em sistemas de computação paralela e distribuída, como usuários, aplicações, recursos e *resource brokers* ou escalonadores, para o desenho e avaliação de algoritmos de escalonamento.

GridSim fornece um mecanismo global para a simulação de diferentes classes de recursos heterogêneos, usuários, aplicações, e *resource brokers*. Pode ser usado para simular aplicações de escalonamento para simples ou numerosos domínios de sistemas de computação distribuída como clusters e Grades.

Algumas das características do *GridSim*, são:

- Permite a modelagem de tipos de recursos heterogêneos.
- Recursos podem ser modelados sob uma política de espaço ou tempo compartilhado.
- Capacidade dos recursos podem ser definidos (na forma de MIPS (*Million Instruction per Second*) ou como SPEC (*Standard Performance Evaluation Corporation*)).
- Tarefas podem ser heterogêneos e podem ser CPU ou I/O *intensive*.
- Pode ser especificada a velocidade da rede entre recursos.
- Permite simulação de carga de traces tomadas de supercomputadores reais.
- Suporta mecanismos baseados em reserva ou subasta de alocação de recursos;
- Atribui uma tarefas nova baseado no modo de espaço ou tempo compartilhado;
- Fornece claras e bem definidas interfaces para implementação de diferentes algoritmos de alocação de recursos;

3.2.1 Arquitetura do GridSim

A arquitetura do simulador foi concebida de forma modular e em camadas. Esta arquitetura multi-camadas e suas aplicações são mostradas na figura 4.

De baixo para cima:

A primeira camada refere-se a interfaces escaláveis entre o simulador e a máquina virtual Java (*JVM*), cuja implementação está disponível para sistemas mono e multiprocessados.

A segunda camada é composta pela infra-estrutura básica de simulação orientada a eventos, construída usando as interfaces providas na camada inferior.

Uma implementação de infra-estrutura de simulação orientada a eventos, de uso bastante difundido e que serve de base para o *GridSim*, é o *SimJava*.
A terceira camada diz respeito à modelagem e simulação das entidades básicas do grid, como recursos e serviços de informação. Nesta camada, também é realizada a modelagem da aplicação, o acesso uniforme a interface, e existe um framework para a criação de entidades em alto nível.

A quarta camada diz respeito à simulação de agregadores de recursos do grid, chamados de *resource brokers* ou escalonadores.

A camada mais acima na arquitetura do *GridSim* está focada na aplicação e na modelagem dos recursos sob diferentes cenários, e faz uso dos serviços providos pelas duas camadas inferiores para avaliar as políticas de escalonamento e gerenciamento de recursos.

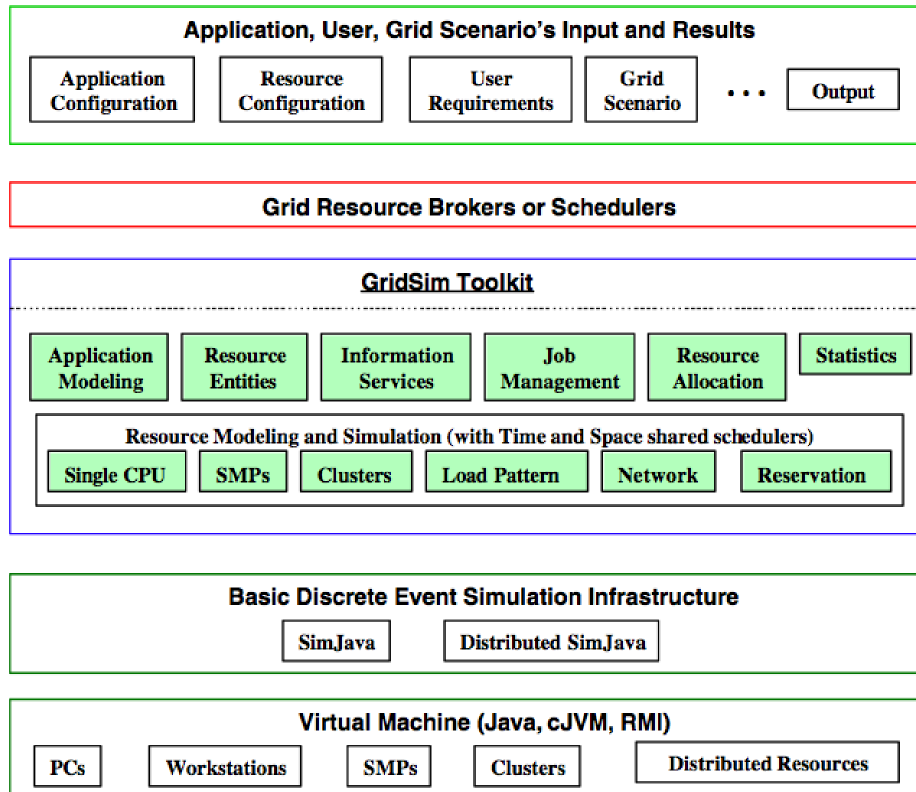

Figura 4: Arquitetura do GridSim

SimJava: Modelo de Eventos Discretos *SimJava*, é um pacote de propósito geral para simulação discreta orientada a eventos, implementada em *Java*. Simulações no *SimJava* contêm um número de entidades, cada uma sendo executada em uma thread própria. O comportamento de uma entidade é codificada em *Java* usando seu método *body()*.

Entidades *GridSim* *GridSim* suporta entidades que simulam as redes utilizadas para a comunicação entre os recursos. Durante a simulação, *GridSim* cria um número de entidades multi-threads, cada uma com se executa em paralelo em seu próprio segmento.

As simulações baseadas no *GridSim* contêm entidades para: usuários, *brokers*, recursos, serviço de informação, estatística.

- **User**: Cada instância da entidade *User* representa um usuário da grade. Cada usuário é diferenciado de outros usuários pelas seguintes características:
  - Tipos de tarefa criada.
  - Estratégia de otimização.

- *Time zone*

- **Broker**: Cada usuário é conectado a uma instância de uma entidade *Broker.* Cada tarefa de um usuário é primeiramente submetida ao seu *broker*, e este então escalona a tarefa de acordo com o algoritmo de escalonamento adotada pelo usuário. O algoritmo de escalonamento usado pelos *brokers* deve ser altamente adaptável para abastecer o mercado e situação de procura.

- **Recource**: Cada instância desta entidade representa um recursos da grade. Cada recursos se diferencia de outro de acordo às seguintes características:
  - Número de processadores.
  - Custo de processamento.
  - Velocidade de processamento.
  - Algoritmos de escalonamento.
  - Fator de carga local.
  - *time zone*

- **Grid Information Service**: Provê o serviço de registros dos recursos existentes na grade. Os *brokers* podem consultar esta entidade para obter informação sobre a disponibilidade, configuração e estado dos recursos.

- **Input e Output**: O fluxo de informações entre as entidades do *GridSim* se dá por meio das entidades de entrada e saída(I/O). O uso de entidades separadas para entrada e saída permite que as demais entidades modelem canais de comunicação *full-duplex* e *multi-usuário.*

3.2.2 Implementação

o *GridSim* é implementado em linguagem *Java*, a ferramenta implementa diversas classes que modelam componentes de sua arquitetura, além de classes e métodos úteis na parte estatística que facilitam a modelagem dos dados de entrada e a interpretaçnao dos resultados.

4 Resultados da Literatura

Na literatura existem diversos trabalhos, eles fazem avaliações com diferentes critérios em diferentes cenários. Assim temos os seguintes:

4.1 Do trabalho: Heuristic for Scheduling Parameter Sweep Applications in Grid Environments [CZBL00]

Neste trabalho, para avaliar a eficiência das heurísticas escolhidas, eles desenvolveram um simulador paramétrico, podemos ver comportamento dos algoritmos, como é mostrado na figura 5, ali mostra como o comportamento do algoritmo *Workqueue*, é mais ruim segundo o tamanho do arquivo submetido é maior.

São avaliados os seguintes algoritmos de escalonamento:

- Max-min [CZBL00]
- Min-min [CZBL00]
- Sufferage [CZBL00]
- XSufferage [CZBL00]
- Workqueue [SCBG03]

Neste cenário podemos ver como o comportamento do *XSufferage*, como ele melhora seu desempenho em comparação com os outros, isto porque ele toma informações tanto das tarefas como dos recursos (máquinas), para fazer o escalonamento.

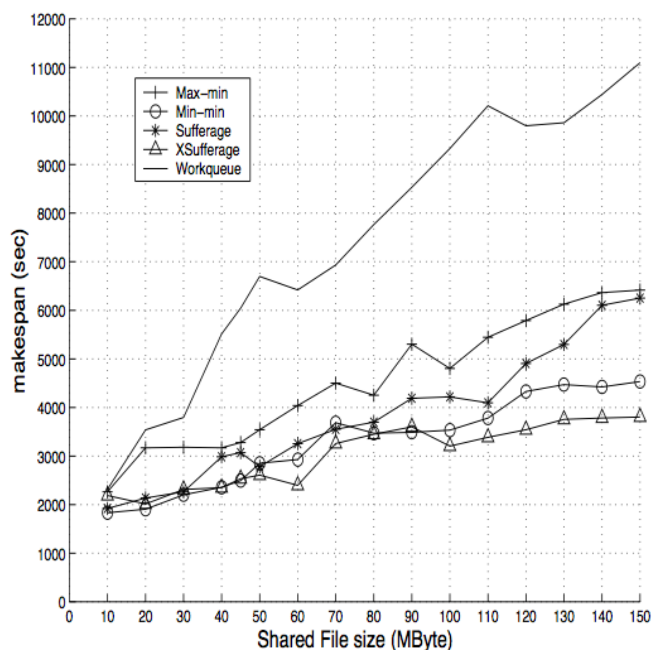

Figura 5: Comportamento dos algoritmos, fonte [CZBL00]

4.2 Do trabalho: Trading Cycles for Information: Using Replication to Schedule Bag-of-Task Applications on Computacional Grids [SCBG03]

Neste trabalho, para avaliar os algoritmos, é usado o simulador SimGrid [CLQ08], na figura 6 e na figura 7. A notação WQR 2.x, significa que a replicação acontece duas vezes, do mesmo jeito ocorre com WQR 3.x e WQR 4.x, na que as replicações ocorrem três e quatro vezes respectivamente.

- Dynamic FPLTF
- Sufferage

- Workqueue
- WQR 2.x
- WQR 3.x
- WQR 4.x

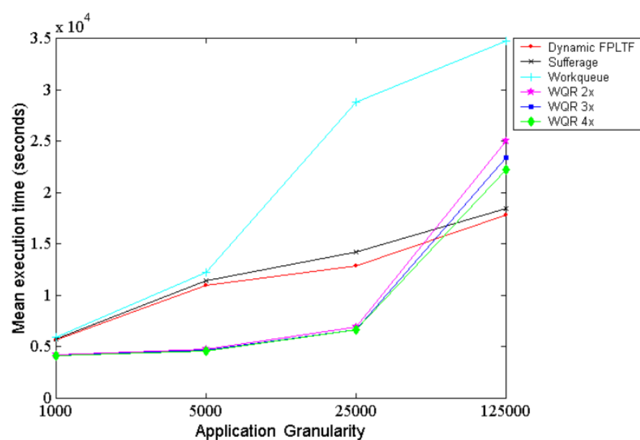

Figura 6: Comportamento dos algoritmos, com o simulador SimGrid

Em ambas figuras podemos ver o comportamento do ruim do *Workqueue*, além disso podemos notar que o algoritmo WQR, apresenta bom comportamento a medida que tem mais replicações.

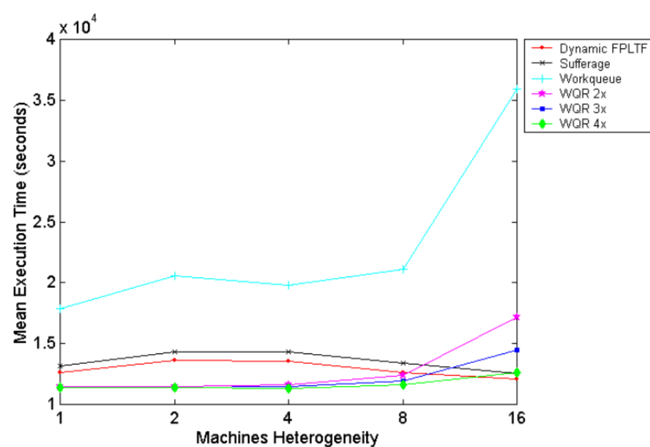

Figura 7: Comportamento dos algoritmos, com o simulador SimGrid

4.3 Do trabalho: trabalho: Avaliação de algoritmos de escalonamento para tarefas em Grids [FJMJBO07]

Neste trabalho, similar que o anterior, a simulação é feita com o SimGrid [CLQ08]. Como é mostrado na figura 8. Os algoritmos tomamos neste artigo são:

- WQR
- Dynamic FPLTF
- XSufferage
- Storage Affinity

Podemos ver o comportamento dos algoritmos, neste caso tanto o *XSufferage* quanto o *Storage Affinity* apresentam com desempenho que os outros.

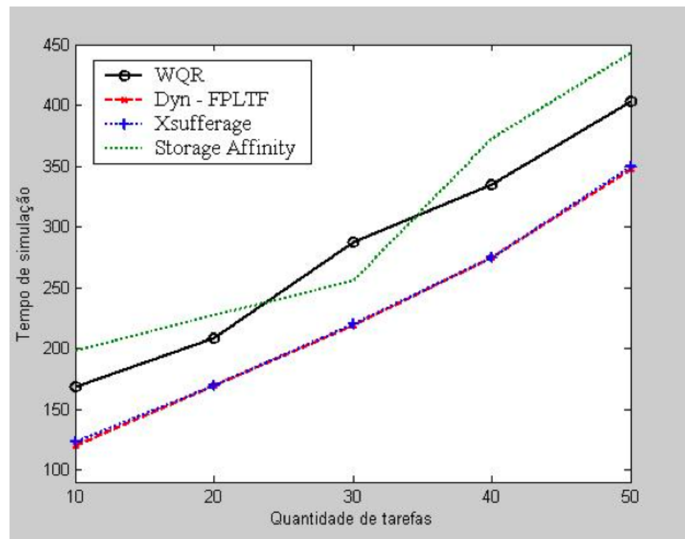

Figura 8: Comportamento dos algoritmos, com o simulador SimGrid

5 Conclusões e trabalhos futuros

Neste estudo, observamos e analisamos o comportamento dos principais algoritmos para escalonamento de tarefas em computação em grade, os quais fazem escalonamento para aplicações de tipo seqüenciais. Na literatura foram avaliados esses algoritmos principalmente com o simulador SimGrid. Até momento o outro simulador escolhido por nós, não foi utilizado para simular, isto fica como trabalho futuro, avaliação do desempenho tanto com o SimGrid quanto o GridSim.

Principalmente, os algoritmos WQR e Storage Affinity têm seu dinamismo caracterizado pela criação e distribuição de replicas. O algoritmo Dynamic

FPLTF analisa a carga de processamento, para que tarefas grandes não sejam alocadas a máquinas lentas. XSufferage não é tão dinâmico, mas até momento nas simulações estudadas, ele apresenta bom desempenho. Os outros algoritmos como Sufferage e Workqueue apresentam desempenho menor que estes algoritmos, isto porque estes algoritmos são melhorados por XSufferage e o WQR.

Para trabalhos futuros pretende-se fazer simulações, tomando diferentes características da grade, de acordo com o estudo, os simuladores tem as características para este propósito.

Além de fazer simulações em aplicações seqüenciais nosso objetivo é também fazer simulações com aplicações paralelas.

Referências

[BM02] Rajkumar Buyya and Manzur Murshed. Gridsim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing, 2002.

[CLQ08] Henri Casanova, Arnaud Legrand, and Martin Quinson. SimGrid: a Generic Framework for Large-Scale Distributed Experiments. In *10th IEEE International Conference on Computer Modeling and Simulation*, March 2008.

[CZBL00] Henri Casanova, Dmitrii Zagorodnov, Francine Berman, and Arnaud Legrand. Heuristics for scheduling parameter sweep applications in grid environments. In *HCW '00: Proceedings of the 9th Heterogeneous Computing Workshop*, page 349, Washington, DC, USA, 2000. IEEE Computer Society.

[DA07] Fangpeng Dong and Selim G. Akl. Scheduling Algorithm for Grid Computing State of the Art and Open Problems. January 2007.

[dSN04] Elizeu Lourenço dos Santos Neto. Escalonamento de Aplicações que Processam Grandes Quantidades de Dados em Grids Computacionais. Master's thesis, Coordenação de Pós-Graduação em Informática - Universidade Federal de Campina Grande, Março 2004. In Portuguese.

[Fil09] Gilberto Cunha Filho. OGST (Opportunistic Grid Simulation Tool): uma ferramenta de simulação para avaliação de estratégias de escalonamento de aplicações em grades oportunistas. Master's thesis, Universidade Federal Do Maranhão, São Luís, Brazils, Fevereiro 2009.

[FJMJBO07] José N. Falavinha Jr., A. Manacero Jr, D. R. Boccardo, and L. J. Oliveira. Avaliação de algoritmos de escalonamento em grids para diferentes configurações de ambiente. *WPerformance 2007*, CD - ROM:505–524, 2007.

[KMR07] Dalibor Klusácek, Ludek Matyska, and Hana Rudová. Alea - grid scheduling simulation environment. In *PPAM*, pages 1029–1038, 2007.

[MOM+07] Yoshiyuki Matsumura, Masashi Oiso, Masaki Matsuda, Noriyuki Fujimoto, Kenichi Hagihara, Kazuhiro Ohkura, Jeremy Wyatt, and Xin Yao. Application of grid task scheduling algorithm rr to medium-grained evolution strategies. In *ICNC '07: Proceedings of the Third International Conference on Natural Computation*, pages 223–227, Washington, DC, USA, 2007. IEEE Computer Society.

[MSP+95] Daniel A. Menascé, Debanjan Saha, Stella C. da Silva Porto, Virgilio A. F. Almeida, and Satish K. Tripathi. Static and dynamic processor scheduling disciplines in heterogeneous parallel architectures. *J. Parallel Distrib. Comput.*, 28(1):1–18, 1995.

[PL09] Ralph Lombreglia Peter Lucas, Joseph Ballay. The wrong cloud. *MAYA Design, Inc.*, 2009.

[SCBG03] Daniel Paranhos Da Silva, Walfredo Cirne, Francisco Vilar Brasileiro, and Campina Grande. Trading cycles for information: Using replication to schedule bag-of-tasks applications on computational grids. In *Applications on Computational Grids, in Proc of Euro-Par 2003*, pages 169–180, 2003.

[SNCB+04] Elizeu Santos-Neto, Walfredo Cirne, Francisco Brasileiro, Aliandro Lima, Ro Lima, and Campina Grande. Exploiting replication and data reuse to efficiently schedule data-intensive applications on grids. In *Proceedings of the 10th Workshop on Job Scheduling Strategies for Parallel Processing*, pages 210–232, 2004.