Lilian Felix de Oliveira

Ferramenta para Simulação de Escalonamento em *Grids* Computacionais

Monografia apresentada ao Departamento de Ciências de Computação e Estatística do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", como parte dos requisitos necessários para aprovação na disciplina Projeto Final.

São José do Rio Preto
2008

Lilian Felix de Oliveira

Ferramenta para Simulação de Escalonamento em *Grids* Computacionais

Monografia apresentada ao Departamento de Ciências de Computação e Estatística do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", como parte dos requisitos necessários para aprovação na disciplina Projeto Final.

Orientadora
Profa. Dra. Renata Spolon Lobato

São José do Rio Preto
2008

Lilian Felix de Oliveira

Ferramenta para Simulação de Escalonamento em *Grids* Computacionais

Monografia apresentada ao Departamento de Ciências de Computação e Estatística do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", como parte dos requisitos necessários para aprovação na disciplina Projeto Final.

Profa. Dra. Renata Spolon Lobato   Lilian Felix de Oliveira

Banca Examinadora:
Prof. Dr. Aleardo Manacero Junior
Prof. Dr. Maurílio Boaventura

São José do Rio Preto
2008

Dedicatória

Aos meus pais Cicero e Dozolina,
aos meus irmãos Everton, Marcos e
Cicero Junior e aos irmãos que ganhei
durante esses cinco anos de faculdade.

Agradecimentos

Primeiramente, agradeço a Deus por não me deixar desistir nunca, me dando forças sempre e também por me fazer escolher o melhor curso de todos.

Aos meus pais Cícero e Dozolina, pelo carinho, dedicação, educação, apoio e por sempre acreditarem em minhas capacidades. E também pelo patrocínio durantes esses cinco anos de curso. AMO VOCÊS.

Aos meus irmãos Everton, Marcos e Junior, pelo apoio, carinho, amizade, críticas e por existirem. Adoro vocês.

À minha orientadora Profª Renata Spolon Lobato, uma profissional excepcional e uma pessoa maravilhosa, pelo apoio, orientação, paciência e principalmente por acreditar. Sem tal apoio teria sido impossível a realização deste trabalho. MUITO OBRIGADA.

Aos irmãos e irmãs que ganhei nesses anos, Mirinha, Claudinha, Serginho, Afofonso, Yoki, Paraguai, Edinho, Zara, Rose e Gisele, pela amizade verdadeira. Nunca esquecerei vocês, até porque estaremos sempre juntos. Amo muito todos.

À Mirinha e seus pais João e Sumie pelo abrigo nos dias em que fico em Rio Preto. Muito obrigada mesmo.

Aos amigos das duas turmas na qual fiz parte, a de Rio Preto e a de Prudente. Foi muito bom conviver esses anos com vocês. Em especial ao Eder, companheiro das madrugadas passadas em claro para concluir este trabalho, me mantendo acordada com suas divertidas conversas pelo MSN, agüentando os meus desabafos e também por torcer para que eu conseguisse terminar essa monografia.

E não podia faltar um grande agradecimento aos mestres que tive durante esses anos, culpados por quase todo o conhecimento adquirido como universitária. Os professores do Departamento de Computação do IBILCE e os professores do Departamento de Computação da FCT.

Resumo

Grids Computacionais abstraem o conceito de que máquinas geograficamente distribuídas compartilham recursos. Tais recursos pertencem, na maioria das vezes, a diversos proprietários. Os donos dos recursos devem manter a autoridade e a prioridade sobre os mesmos, exigindo assim a criação de políticas de escalonamento apropriadas. O estudo de escalonamento em grids computacionais é cada vez mais alvo de pesquisas, mostrando a importância da área.

Este trabalho consistiu na construção de um simulador de algoritmos de escalonamento em *grids* computacionais. O simulador tem uma interface gráfica e possibilita ao usuário criar suas próprias políticas de escalonamento. Permite também, perante um conjunto de dados de entrada, a comparação das políticas de escalonamento previamente implementadas. Para que o usuário possa criar suas políticas de escalonamento, uma linguagem foi desenvolvida. A mesma permite ao usuário mesclar as políticas de escalonamento contidas na ferramenta.

Abstract

Computational Grids abstract the concept that machines geographically distributed share resources. These resources are, in most cases, the various owners. The owners of the resources to maintain the authority and the priority on them, thus requiring the creation of appropriate scheduling policies. The study of computational grids scheduling is increasingly the target of researches, showing the importance of the area.

This work was to build a simulator of scheduling algorithms in computational grids. The simulator has a graphical interface and allows users to create their own scheduling policies. It also allows, a set of input data, comparing the policies of escalation previously implemented. To enable the users create their scheduling policies a language was created. It allows the user to merge the scheduling policies within the tool.

Índice

Lista de Figuras ... **iii**
Lista de Tabelas ... **iv**
Lista de Abreviaturas e Siglas ... **v**

Capítulo 1 – Introdução ... **1**
- **1.1 Considerações Iniciais** ... **1**
- **1.2 Objetivos** ... **2**
- **1.3 Organização da Monografia** ... **2**

Capítulo 2 – Revisão Bibliográfica ... **3**
- **2.1 Considerações Iniciais** ... **3**
- **2.2 *Grids* Computacionais** ... **3**
  - **2.2.1 Tipos de *Grids*** ... **5**
  - **2.2.2 Escalonamento em *Grids*** ... **7**
  - **2.2.3 Políticas de Escalonamento para *Grids*** ... **10**
- **2.3 Compiladores** ... **15**
  - **2.3.1 *Front-end*** ... **16**
- **2.4 Simulação** ... **19**
- **2.5 Ferramentas de Simulação Existentes para *Grids*** ... **21**
- **2.6 Considerações Finais** ... **24**

Capítulo 3 – Desenvolvimento da Ferramenta ... **25**
- **3.1 Considerações Iniciais** ... **25**
- **3.2 Simulação das Políticas de Escalonamento** ... **27**
  - **3.2.1 Simulação das Aplicações** ... **28**
  - **3.2.2 Simulação dos Recursos** ... **30**
- **3.3 Políticas** ... **32**
  - **3.3.1 *Workqueue*** ... **32**
  - **3.3.2 *Workqueue with Replication*** ... **35**
  - **3.3.3 *Sufferage*** ... **38**
- **3.4 Comparação das Políticas** ... **40**
- **3.5 Criação de uma Política** ... **40**
  - **3.5.1 Linguagem** ... **41**
  - **3.5.2 Compilador** ... **45**
- **3.6 Considerações Finais** ... **50**

Capítulo 4 – Testes e Resultados ... **51**
- **4.1 Considerações Iniciais** ... **51**
- **4.2 Simulação das Políticas de Escalonamento** ... **51**
- **4.3 Compilador** ... **58**
- **4.4 Considerações Finais** ... **61**

Capítulo 5 – Conclusões ... **62**
- **5.1 Conclusões** ... **62**

5.2 Propostas de Trabalhos Futuros 63
Referências Bibliográficas 64

Lista de Figuras

Figura 2.1 – Relacionamento e heterogeneidade dos recursos do *grid* 4
Figura 2.2 – Relacionamento entre escalonador, recursos e usuários 9
Figura 3.1 – Relação entre os módulos da ferramenta 26
Figura 3.2 – Formato dos arquivos de *traces* utilizados na simulação 27
Figura 3.3 – Tela da simulação, informando os resultados no quadro *Statistcs*..34
Figura 4.1 – Tela inicial da ferramenta 52
Figura 4.2 – Tela de entrada de dados para comparação de políticas 53
Figura 4.3 – Tela de simulação 54
Figura 4.4 – Desempenho das políticas em vários níveis de heterogeneidade de máquinas 55
Figura 4.5 – Desempenho das políticas com diferentes tamanhos médios das Tarefas 56
Figura 4.6 – Tela de resultados da comparação 57
Figura 4.7 – Tela do compilador 58
Figura 4.8 – Código com erro semântico 59
Figura 4.9 – Mensagens de erros resultantes da compilação 59
Figura 4.10 – Resultado da compilação de um código aceito pela linguagem 60

Lista de Tabelas

Tabela 3.1: palavras reservadas e constantes da linguagem desenvolvida 41

Tabela 4.1: Desempenho obtido por cada uma das políticas em 5 níveis de heterogeneidade de recursos 55

Tabela 4.2: Granulosidade das Aplicações 56

Lista de Abreviaturas e Siglas

CT: ***Completion Time***
FPLTF***: Fastest Processor to Largest Task First***
MI: Milhões de Instruções.
MIPS: Milhões de Instruções por Segundo.
NWS: ***Network Weather Service***
UCP: Unidade Central de Processamento.
VPN: ***Virtual Private Network***
WQR: ***Workqueue with Replication.***
WQR2x: ***Workqueue with Replication*** **com 2 réplicas no máximo.**
WQR3x: ***Workqueue with Replication*** **com 3 réplicas no máximo.**
WQR4x: ***Workqueue with Replication*** **com 4 réplicas no máximo.**

Capítulo 1 – Introdução

1.1 Considerações Iniciais

Com a consolidação da rede mundial de computadores tornou-se possível à construção de sistemas envolvendo máquinas geograficamente distribuídas. Assim nasceu o conceito de *Grids* Computacionais, no qual máquinas, espalhadas fisicamente compartilham recursos [1].

Uma das maiores vantagens no uso de *grids* computacionais está no fato de utilizarem recursos ociosos das máquinas pertencentes ao seu domínio. Essa característica resulta na formação de supercomputadores virtuais, possibilitando assim a execução de aplicações que exigem grande poder computacional. Contudo, os proprietários dos recursos devem manter a autoridade sobre os mesmos, exigindo assim a criação de políticas de escalonamento apropriadas. Portanto, pesquisas na área de escalonamento de processos em *grids* computacionais são de suma importância, como as feitas no GSPD (Grupo de Pesquisas em Sistemas Paralelos e Distribuídos do Departamento de Ciências de Computação e Estatísticas do Instituto de Biociências, Letras e Ciências Exatas da UNESP) [2], um exemplo de trabalho desenvolvido é o artigo [3] apresentado no Wperformance 2007.

1.2 Objetivos

O objetivo deste trabalho consistiu na construção de um simulador de algoritmos de escalonamento em *grids* computacionais. Foram implementadas três políticas, a *Workqueue*, *Workqueue with Replication* e *Sufferage*. O simulador tem uma interface gráfica e possibilita ao usuário criar suas próprias políticas de escalonamento. Permite também, perante um conjunto de dados de entrada, a comparação das políticas de escalonamento pré-implementadas. Para que o usuário possa criar suas políticas de escalonamento, uma linguagem foi desenvolvida. Ela possibilita ao usuário mesclar as políticas de escalonamento já implementadas.

Os simuladores existentes não simulam componentes específicos do *grid*, apenas existem os que simulam um *grid* computacional em geral, estes de grande porte e de difícil instalação e configuração, requisitando em sua maioria, máquinas com grande poder de processamento e exigindo do usuário um conhecimento prévio do assunto.

Assim, a ferramenta desenvolvida tem o intuito de auxiliar no estudo de políticas de escalonamento em *grids e* possibilita a um iniciante no estudo de *grids* criar, comparar e simular tais políticas.

1.3 Organização da Monografia

No capítulo dois é descrita toda a fundamentação teórica necessária para a realização do projeto, começando com o estudo de *grids* computacionais, passando por compiladores e por fim abordando o tema da simulação em ambientes computacionais.

As etapas de desenvolvimento do projeto, incluindo sua especificação e implementação são descritas no capítulo três.

No capítulo quatro são apresentados os testes e verificação da funcionalidade do sistema.

Finalmente, no capítulo cinco apresentam-se as conclusões e os possíveis trabalhos futuros para esse projeto.

Capítulo 2 – Revisão Bibliográfica

2.1 Considerações Iniciais

Neste capítulo são abordados tópicos relevantes para o desenvolvimento do projeto. Na sessão 2.2 são descritos o funcionamento, os tipos e o escalonamento em *grids* computacionais. Na sessão 2.3 é mostrado o funcionamento de um compilador por meio da descrição de suas fases, na sessão 2.4 é tratado o assunto de simulação em ambientes computacionais e na sessão 2.5 é feita uma breve descrição das ferramentas para simulação de *grids* existentes. Por fim são feitas considerações finais sobre os temas citados.

2.2 *Grids* Computacionais

Segundo [1] um *grid* computacional pode ser definido como sendo uma infra-estrutura formada por hardware e software que provê segurança, consistência, difusão e um baixo custo de acesso a altas capacidades computacionais.

Grids permitem o compartilhamento de recursos como, por exemplo, dados, capacidade de processamento e armazenamento. Esse compartilhamento é feito por

meio de uma rede de computadores, na maioria dos casos essa rede é a Internet. Sendo assim, os recursos podem estar geograficamente distribuídos e dinamicamente disponíveis, além de comumente pertencerem a múltiplas organizações, fazendo com que recursos sejam bastante heterogêneos. O fato de recursos serem multi-institucionais constitui um dos maiores problemas encontrados na implementação de um sistema de computação em *grid*, tal problema se encontra na administração de recursos pertencentes a múltiplos domínios, que por sua vez, possuem suas próprias políticas de administração. A figura 2.1 ilustra o relacionamento e a heterogeneidade dos recursos em um *grid*.

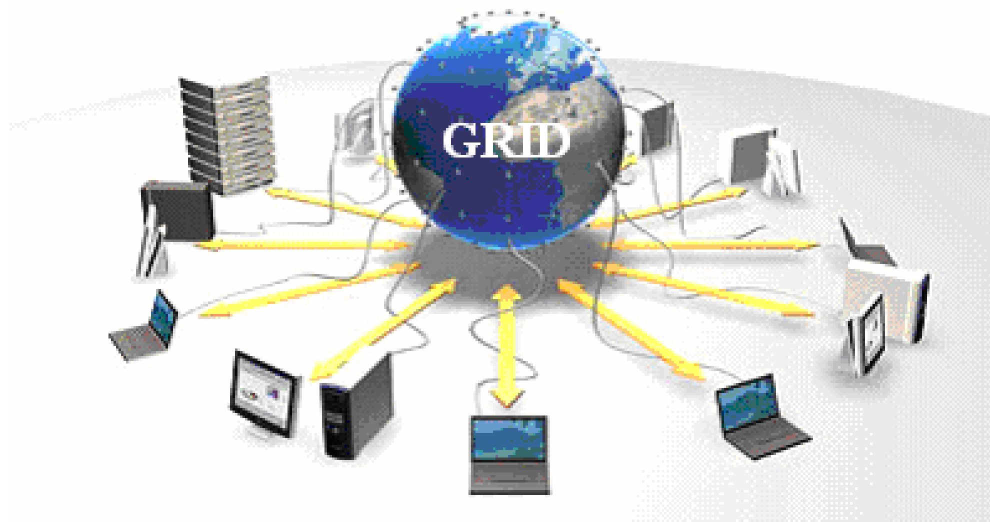

Figura 2.1: relacionamento e heterogeneidade dos recursos do *grid* [1].

Pela característica da Internet de possuir uma alta latência na comunicação, as aplicações submetidas ao *grid* devem evitar trocar informações entre suas tarefas. Assim, aplicações compostas por tarefas independentes, ou seja, que não necessitam de qualquer troca de informações entre as mesmas, são mais aplicáveis a *grids* computacionais.

Um modelo de aplicações compostas por tarefas independentes é chamado de *Bag-of-tasks* [4]. Neste modelo as tarefas independem da execução das outras, tornando assim, desnecessária a comunicação entre elas.

Em casos onde a rede utilizada para a interconexão não for a Internet e sim uma rede com máquinas fortemente acopladas e com uma alta velocidade, aplicações com alto grau de comunicação entre as tarefas podem ser executadas.

O fato de *grids* aumentarem o poder computacional e de armazenamento, sem altos investimentos em infra-estrutura de hardware, incentiva pesquisas nessa área e também o aparecimento cada vez maior de aplicações para esse ambiente [5].

2.2.1 Tipos de *Grids*

Com o intuito de contextualizar a relação entre coordenação de recursos e os tipos de grids, as principais definições e explicações para diferentes tipos de grids são apresentadas nesse tópico. Têm-se adotado diversas nomenclaturas. Alguns definem *grids* baseados na estrutura da organização (virtual ou outra) que é atendida por um *grid*, enquanto outros definem pelos principais recursos utilizados no *grid* [6]. Seguem as classificações dessas várias definições.

***Grids* Departamentais**: São desenvolvidos para resolver problemas de um particular grupo de pessoas dentro de uma empresa. Os recursos não são compartilhados por outros grupos internos a empresa. Duas definições são encontradas:

- *Cluster Grids:* termo usado pela Sun Microsystems [7], consiste de um ou mais sistemas trabalhando juntos para prover um único ponto de acesso a usuários. É tipicamente usado por um time, para um único projeto, e pode ser usado pra prover tanto alto *throughput* quanto alto desempenho na execução de tarefas.
- *Infra Grids:* termo usado pela IBM [8], para definir um grid que otimiza recursos dentro de uma empresa e não envolve qualquer outro parceiro interno.

Grids* Empresariais:* Consiste em recursos espalhados por toda a empresa e prover serviços para todos os usuários da empresa. Três definições são encontradas:

- *Enterprise Grids:* de acordo com Platform Computing [9], é desenvolvido dentro de corporações com grande presença global ou por empresas que precisam acessar recursos externos a um único local corporativo.
- *Intra Grids:* de acordo com a IBM, consistem em diversos recursos compartilhados por diferentes grupos dentro da empresa.
- *Campus Grids:* de acordo com a Sun Microsystems, *campus grids* permite que múltiplos projetos ou departamentos compartilhem recursos de uma forma cooperativa. Consiste em diversas *workstations* (estações de trabalho) dispersas e servidores como fonte centralizada de recursos, localizada em múltiplos domínios administrativos, em departamentos ou por toda a empresa.

Grids* Extra-Empresariais*: são estabelecidos entre empresas companhias, parceiros e clientes. Os recursos do *grid* são disponibilizados, de uma ponta a outra, por meio de uma rede privada virtual [10] (VPN – *Virtual Private Network*).

- *Extra Grids:* de acordo com a IBM, *extra grids* possibilita o compartilhamento de recursos com parceiros externos.
- *Partner Grids:* Platform Computing define estes como *grids* entre organizações dentro de indústrias com o mesmo ramo de atividade, que necessitam colaborar em projetos e usam os recursos uns dos outros como meio de atingir uma meta comum.

Grids* Globais*: são grids estabelecidos sobre a Internet pública. Eles podem ser estabelecidos por organizações para facilitar negociações ou compras. Seguem duas definições encontradas para essa categoria:

- *Global Grids:* segundo a Sun Microsystems, *global grids* provê o poder de distribuir recursos para usuários em qualquer lugar do mundo para computação e colaboração.
- *Inter Grids:* de acordo com a IBM, *Inter Grids* têm a capacidade para compartilhar recursos de computação e armazenamento de dados pela Web pública. Isto pode envolver o compartilhamento de recursos com outras empresas, ou por meio da compra e venda desses recursos.

***Grids* de Computação:** *grids* criados pela simples proposta de prover acesso a recursos computacionais. Eles são classificados de acordo com o *hardware* computacional implantado.

- *Desktop Grids:* são *grids* que compartilham recursos de computadores pessoais.
- *Server Grids:* apenas incluem servidores, usualmente rodando um sistema operacional Unix/Linux.
- *High-Perfomance/Cluster Grids:* grids constituídos de supercomputadores e clusters.

***Grids* de Dados:** *grids* desenvolvidos para requerer acesso ou processar dados. Aperfeiçoam operações orientadas a dados e embora possam consumir uma grande capacidade de armazenamento, eles não confundidos com provedores de serviço de armazenamento.

***Grids* de Utilidades:** são definidos como sendo recursos computacionais comerciais mantidos e gerenciados por um provedor de serviço. Consumidores com necessidade de computação podem comprar ciclos de um *grid* de utilidades. Além disso, os consumidores podem escolher utilizar este *grid* para continuidade de negócios e para propostas de recuperação de desastres. Um exemplo de *grid* que segue essa definição são os *service grids.*

- *Service Grids:* fornecem acesso a recursos que podem ser comprados por empresas para aumentar seus próprios recursos.

2.2.2 Escalonamento em Grids

Com o surgimento dos Sistemas Distribuídos, houve a necessidade de criar novos escalonadores, tais escalonadores chamados de Globais, em contraste aos escalonadores locais de sistemas operacionais convencionais. Recebem esse nome, pois em um ambiente distribuído existem diversos recursos disponíveis, sendo assim

sua tarefa agora passa a ser escalonar processos entre um conjunto acoplado de máquinas diferente de um sistema convencional, onde apenas existe um recurso. Um escalonamento é feito objetivando diversas metas de desempenho e performance, por isso os escalonadores globais agregaram a função de escolher quando e quais processos têm acesso a quais recursos do sistema [11][12][13]. Entre as metas existentes as principais são:

Aumentar o *throughtput* do sistema: também chamado de vazão do sistema, é a medida feita a partir do número de processos finalizados por unidade de tempo.

Diminuir o tempo de resposta: tal medida é definida pela diferença entre o momento de término da execução da tarefa e seu instante de chegada na fila de processos, ou seja, essa medida é soma dos tempos gastos em fila de espera por recursos e na execução propriamente dita dos processos.

Aumentar a utilização de recursos: o escalonador pode fazer com que os recursos do sistema, tais como UCP, memória ou rede, sejam utilizados ao máximo, mesmo que para atingir tal meta seja necessário esquecer outros critérios.

Balancear a carga do sistema: consiste em não subutilizar recursos enquanto outros estão trabalhando em sua capacidade máxima. A intenção é distribuir os processos para os recursos de acordo com a capacidade dos mesmos.

Os escalonadores globais têm a necessidade de selecionar recursos e escalonar processos ao mesmo tempo em que requisições de usuários chegam, garantindo tempo de execução e custo de recursos. A figura 2.2 ilustra o relacionamento entre usuário, escalonador e recursos.

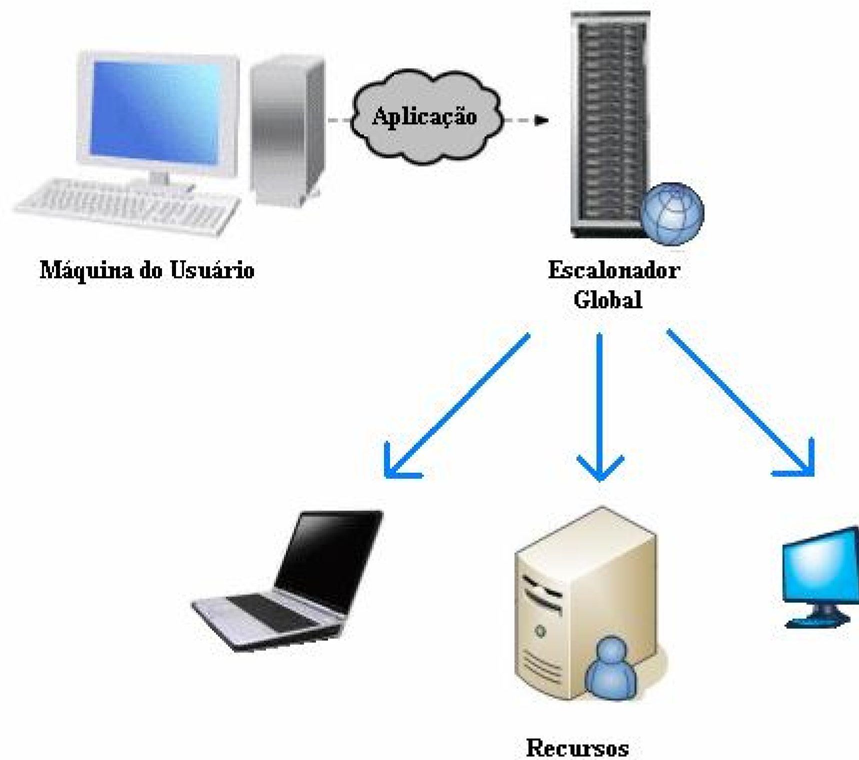

Figura 2.2: relacionamento entre escalonador, recursos e usuários [12].

G*rids* computacionais utilizam recursos ociosos de máquinas pertencentes ao seu domínio, essas máquinas por sua vez podem apresentar diferentes configurações [14][15]. Essa característica resulta em alguns problemas inexistentes em sistemas de um único processador, como:

Grande quantidade de recursos: a grande disponibilidade de recursos torna-se um problema para o escalonador, que pode se tornar um gargalo do sistema, pois ele deve escolher de forma apropriada, qual recurso irá executar um certo processo.

Grande heterogeneidade de recursos: máquinas pertencentes ao *grid* podem apresentar configurações heterogêneas. Entre as configurações, as principais são poder de processamento, interconexões e Sistemas Operacionais.

Alto compartilhamento de recursos: a variação de carga nas máquinas causada pela submissão de novos processos ao sistema é proporcional ao número de usuários do *grid*, isto é, quanto mais usuários, maior será a variação de carga do sistema, e isso pode fazer com que políticas que não prevêem tal fato atinjam um resultado negativo.

Movimentação e Consistência de dados: em *grids* deve-se evitar a submissão de aplicações que realizem muita comunicação, pois a baixa latência da rede de interconexão dos recursos pode causar prejuízos ao escalonamento.

2.2.3 Políticas de Escalonamento para *Grids*

A realização de um escalonamento nada mais é que a aplicação de regras a um conjunto de tarefas e recursos, utilizando ou não informações do sistema. Deve-se adotar a política de escalonamento correta para cada aplicação distinta. Aplicações *Bag-of-tasks* facilitam o escalonamento, por sua característica de independência entre as tarefas, o que permite o uso de políticas baseadas em apenas alguns dados do sistema, raramente necessitando de informações sobre a infra-estrutura do *grid*, como latência da rede e largura de banda. As políticas podem ser estáticas ou dinâmicas, a diferença está no momento em que o escalonamento é feito. Se ele acontecer no momento da compilação, então é dito estático, caso contrário é chamado de dinâmico. A dinamicidade e heterogeneidade presentes em *grids* caracterizam as soluções estáticas como não muito apropriadas, por isso neste capítulo só serão descritas políticas dinâmicas.

Workqueue

Essa política não precisa de nenhuma informação sobre o ambiente ou sobre a aplicação para atribuir tarefas a recursos disponíveis. Assim que recursos se tornem disponíveis, tarefas são atribuídas a eles. A distribuição das tarefas entre os recursos é feita de forma aleatória, bastando que o recurso esteja disponível e que exista tarefa

para ser escalonada. Após o término da execução da tarefa, o processador envia o resultado de sua execução ao escalonador, que por sua vez lhe atribui uma nova tarefa, isso persiste até que todas as tarefas da fila tenham sido executadas. Assim, processadores mais velozes receberão mais tarefas, por se tornarem disponíveis mais rapidamente. A grande vantagem do *Workqueue* é que ele independe de informações do sistema para escalonar as tarefas. Contudo, o problema com o *Workqueue* surge quando uma tarefa grande é alocada para uma máquina mais lenta no final da execução da aplicação, isso faz com que o tempo de resposta aumente significativamente.

Workqueue with Replication

Para lidar com a heterogeneidade de máquinas e tarefas presentes no *grid* o algoritmo *Workqueue with Replication* (WQR) utiliza a técnica de replicação de tarefas. Vale ressaltar também que o WQR assume que as tarefas são idempotentes, ou seja, não geram efeitos colaterais, como alterações em banco de dados, de modo a prevenir a inconsistência de dados que réplicas poderiam causar. Essa hipótese é válida em ambientes como *grids* pelo fato de máquinas estarem bastante dispersas geograficamente e em ambientes assim não é comum a utilização de aplicações cujas tarefas façam acesso em bancos de dados, por ser algo muito custoso.

A política *WQR* é similar a *Workqueue*, diferindo apenas a partir do momento em que todas as tarefas da fila de tarefas forem alocadas para execução. A partir desse momento, se existirem processadores ociosos, tarefas de uma mesma aplicação são escolhidas para serem replicadas nos processadores disponíveis [16]. As tarefas são replicadas até que um número máximo de tarefas predefinido seja alcançado. Para expressar o limite de réplicas do algoritmo é usado um complemento em seu nome, WQR2x, por exemplo, permite a criação de até duas réplicas, ou seja, duas tarefas iguais em execução, WQR3x permite a criação de três réplicas e assim por diante. Vale ressaltar ainda que a n-ésima réplica de uma tarefa só é criada quando todas as

tarefas em execução tiverem pelo menos n-1 réplicas criadas, objetivando ter um maior controle no desperdício de ciclos de UCP.

Quando uma réplica termina sua execução, todas as outras são abortadas para que não haja desperdício de processamento. Com essa abordagem o desempenho tende a ser melhorado em situações em que tarefas grandes atrasam o término da aplicação por terem sido alocadas para máquinas lentas, pois a réplica possui chances de ser alocada para uma máquina mais rápida.

A vantagem do algoritmo, assim como no *Workqueue*, está no fato de não necessitar de informações do sistema para realizar o escalonamento, isso faz com que a política atinja performance equiparável a políticas que utilizam informações do sistema, mas em contrapartida seu lado negativo é que para atingir uma grande performance essa abordagem gasta mais ciclos de UCP com as réplicas que são abortadas, esses ciclos são contabilizados como desperdício, pois não foram úteis para o processamento da aplicação.

***Dynamic* FPLTF**

O *Dynamic Fastest Processor to Largest Task First* (*Dynamic* FPLTF) [17] possui uma grande habilidade de se adaptar à dinamicidade e heterogeneidade de ambientes como *Grids*. O *Dynamic* FPLTF, diferentemente das políticas descritas anteriormente necessita de três informações do sistema para escalonar as tarefas: *Task Size*, *Host Load* e *Host Speed*. *Host Speed* representa a velocidade relativa da máquina. Uma máquina com *Host Speed* igual a dois, executa uma tarefa duas vezes mais rapidamente que uma máquina com *Host Speed* igual a um. *Host Load* representa a fração de UCP da máquina que não está disponível para a aplicação, ou seja, está sendo usada por outros usuários ou aplicações. Vale ressaltar que o valor de *Host Load* varia com o tempo, dependendo da carga que está sendo imposta à máquina por outros usuários ou aplicações. Finalmente, *Task Size* representa o tempo necessário para uma máquina com *Host Load* igual a zero, ou seja, totalmente disponível, complete a execução da tarefa.

No início do algoritmo uma variável TBA (*Time to Become Available*) é inicializada com zero. Tal variável representa o tempo necessário para que um determinado recurso esteja disponível. As tarefas são organizadas em ordem decrescente de tamanho, desse modo a maior tarefa é a primeira a ser alocada. Cada tarefa é alocada para o *host* que provê o menor tempo de execução CT (*Completion Time*) para ela, onde:

$$\text{CT} = \text{TBA} + \text{Task Cost}$$

e

$$\text{Task Cost} = (\text{Task Size} / \text{Host Speed}) / (1 - \text{Host Load}).$$

Quando uma tarefa é alocada para uma máquina, o valor de TBA correspondente é incrementado com *Task Cost*. As tarefas são alocadas até que todas as máquinas do *grid* estejam ocupadas. Após isso, a execução da aplicação é iniciada. Assim que uma tarefa completa sua execução, todas as tarefas que estão executando no momento são desescalonadas e re-escalonadas até que cada máquina do *grid* tenha pelo menos uma tarefa alocada. Isso continua até que todas as tarefas sejam completadas. Essa estratégia tenta minimizar os efeitos do dinamismo do *grid*. Uma máquina, que a princípio, é mais rápida e não está sobrecarregada recebe mais tarefas que uma máquina também rápida, porém com bastante carga. Essa variação na carga pode comprometer a execução da aplicação já que as tarefas escalonadas para essa máquina iriam executar mais lentamente que o previsto. O processo de re-escalonar tarefas tenta corrigir este problema alocando as tarefas maiores para as máquinas mais rápidas no momento do escalonamento. Esta política apresenta bom desempenho, contudo é muito complicado de ser implementado na prática, pois necessita de muita informação do ambiente. Tais informações geralmente são difíceis de obter com precisão e frequentemente não estão disponíveis devido a restrições administrativas, tornando a solução inviável em vários casos.

Sufferage

O *Sufferage* é um algoritmo dinâmico que tem mostrado atingir bom desempenho em ambientes heterogêneos e dinâmicos quando considerado a existência de informação completa [18].

A idéia básica da heurística de escalonamento *Sufferage* [12][19], como o próprio nome reflete, é determinar quanto cada tarefa seria prejudicada (“sofreria”) se não fosse escalonada para o processador que a executaria de forma mais eficiente. O valor de *sufferage* de uma tarefa é definido pela diferença entre seu segundo melhor e seu melhor CT, este calculado da mesma forma como mostrado na descrição da política *Dynamic* FPLTF. Em outras palavras, a *sufferage* consiste na idéia de atribuir a melhor máquina à aplicação que mais seria prejudicada se essa atribuição não fosse feita. A tendência é que cada tarefa seja atribuída à máquina que oferece o menor tempo para completá-la.

No início do algoritmo, quando todas as máquinas estão livres, o valor de *sufferage* é calculado para cada tarefa da fila. A tarefa que possuir o maior valor de *sufferage* é alocada para a máquina que a executaria de forma mais eficiente, ou seja, a que proporcionou o melhor *Completion Time* para a tarefa. Em seguida, o procedimento é repetido várias vezes até que todas as tarefas sejam alocadas. Se uma máquina proporcionar o melhor CT para uma tarefa, mas estiver ocupada executando outra tarefa, os valores de sufferage de cada uma das tarefas é comparado e a que possuir o maior fica alocada na máquina, a que possuir o menor valor volta para a fila de tarefas prontas. Com o dinamismo do *grid*, como variação de carga dos processadores, os valores de sufferage das tarefas variam durante toda a execução da aplicação. Dessa forma, o algoritmo Sufferage é invocado diversas vezes durante a execução da aplicação, até que todas as tarefas tenham sido escalonadas. Toda vez que uma tarefa é completada, todas as tarefas que ainda não começaram a executar, são desescalonadas e o algoritmo é invocado novamente, usando os valores de sufferage atuais, fazendo com que o algoritmo re-escalone as tarefas restantes, mas dessa vez com a nova carga das máquinas. Isso se repete até que todas as tarefas tenham sido alocadas para alguma máquina.

XSufferage

XSuferage é uma extensão da heurística de escalonamento *Sufferage*. A principal diferença é o método utilizado para calcular o valor de *sufferage*. Agora existe o conceito de *site*. Inicialmente é calculado para cada tarefa o seu melhor tempo de execução em cada *site*, esses valores, são chamados de *site-level completion time*, em seguida o *site-level sufferage* é calculado. O valor de *site-level sufferage* é encontrado calculando a diferença entre os dois melhores valores encontrados para *site-level completion time* da tarefa. Assim, a tarefa que apresentar o maior *site-level sufferage* terá prioridade e será escalonada no *site* que a executaria de forma mais rápida. Além da idéia de *site*, existe outro aspecto distinto entre as políticas *XSufferage* e *Sufferage*. *XSufferage* considera a transferência dos dados de entrada da tarefa. Isso implica no uso das informações usadas por *Sufferage* mais a largura de banda disponível na rede.

A definição de *site-level sufferage* permite que no momento de alocação dos recursos, a localização dos dados de entrada da tarefa seja considerado. O efeito esperado é a redução do impacto das transferências de dados desnecessárias no tempo total de execução da aplicação. Evitando transferências desnecessárias de dados, o desempenho da aplicação pode ser melhorado. Contudo, os cálculos para obtenção do valor de *site-level sufferage* são complicados, pois exigem muito conhecimento prévio da aplicação e do ambiente, inclusive da infra-estrutura e em alguns casos, esses valores nem sequer se encontram disponíveis.

2.3 Compiladores

São tradutores que mapeiam instruções em linguagem simbólica para programas em linguagem de máquina, de forma que viabilize sua execução [20]. Em outras palavras, um compilador é um programa que recebe como entrada um programa fonte e produz como saída um programa, semanticamente equivalente, porém escrito em uma linguagem simbólica, que por sua vez é traduzida para linguagem de máquina por meio do uso de montadores.

O compilador agrupa as fases em *front-end* e *back-end* [21]. O *front-end* consiste na verificação e análise, enquanto o *back-end* na otimização e geração de códigos. Todas as fases são auxiliadas por uma tabela de símbolos e o *front-end* por um tratador de erros.

Tabela de Símbolos

Uma tabela de símbolos é uma estrutura de dados contendo um registro para cada identificador.

Este módulo não consiste em uma fase, mas compreende um conjunto de tabelas e rotinas associadas que são utilizadas pelas fases do compilador.

A tabela armazena informações sobre declaração de variáveis, declaração de funções e seus parâmetros.

Tratador de Erros

Cada fase de análise do compilador pode produzir erros. Estes erros devem ser tratados para que o processo de compilação continue.

O tratamento de erros é realizado por meio de mecanismos de recuperação, encarregados de ressincronizar a fase com o ponto do texto em análise. A perda desse sincronismo faria a análise prosseguir de forma errônea, propagando o efeito do erro.

2.3.1 *Front-end*

O *front-end* deve descobrir a estrutura sintática de um programa e convertê-la em uma estrutura de alto nível denominada árvore de sintaxe abstrata. Essa fase é dividida em três outras, as análises léxica, sintática e semântica.

Análise Léxica

É a primeira fase do compilador. Este analisador converte a entrada em um fluxo de *tokens* a ser analisado pelo analisador sintático. Faz a leitura do programa fonte, caractere a caractere, e traduz o grupo de símbolos lidos em uma seqüência de símbolos léxicos (*tokens*).

O token é representado internamente por três informações:

Classe do *token*: representa o tipo do *token* reconhecido. Pode ser identificador, constante, operadores, palavras reservadas e assim por diante.

Valor do *token*: depende da classe. Por exemplo, se for uma constante inteira, o valor assumido pode ser 90.

Posição do *token*: indica o local do texto onde ocorreu o *token*.

As principais funções do analisador léxico são:

Remoção dos espaços em branco e comentários: bastando para isso ignorá-los na leitura do código fonte.

Reconhecer Constantes: uma seqüência de dígitos ou de caracteres entre aspas representa uma constante. O que é relevante neste caso é identificar a classe a que essa constante pertence.

Reconhecer identificadores e palavras reservadas: palavras reservadas têm prioridade de reconhecimento, se uma seqüência de caracteres satisfaz as regras de construção de identificadores, mas é equivalente a uma palavra reservada, essa seqüência será reconhecida como palavra reservada, caso contrário é reconhecida como um identificador.

Poucos são os erros reconhecidos nessa fase, pois o analisador trabalha apenas com símbolos.

Análise Sintática

Sua função é verificar se as construções gramaticais estão corretas. O analisador sintático, também chamado de *parser*, recebe do analisador léxicos uma seqüência de *tokens* que constitui a sentença e produz como resultado uma árvore de derivação se a sentença for válida, caso contrário uma mensagem de erro deve ser gerada. Há dois métodos de análise, o *Top-Down* e o *Bottom-up*.

O método *Top-Down* gera a árvore de derivação a partir do símbolo inicial da gramática, fazendo a árvore crescer até as folhas. Já a estratégia *Bottom-up* realiza a análise no sentido inverso, a partir dos tokens (que devem estar nas folhas) até o símbolo inicial da gramática (representado na raiz).

Na estratégia de detecção de erros é desejável que o compilador continue o processo de análise, mesmo após encontrar algum erro. Desta forma poderá detectar outros erros que possam existir no código ainda não analisado. Para isso é necessário que ocorra a recuperação ou reparação do erro. Existem muitas estratégias para tratar erros sintáticos, entre as principais estão a Modalidade do Desespero, Recuperação de Frases e Produções de Erros.

Análise Semântica

Nesta fase é feita a verificação de tipos, ou seja, a certificação de que variáveis e constantes que aparecem em cada expressão possuam tipos compatíveis. Também é feita a verificação de escopo, isto é, determinar os pontos do programa para os quais uma variável está definida.

Verificação de escopo

- Na verificação do escopo o compilador verifica se existe declaração para o identificador utilizado.
- Normalmente o escopo é definido por blocos. Um identificador declarado em um bloco é valido dentro de seus sub-blocos. Caso exista uma outra declaração de identificador com o mesmo nome no sub-bloco, a declaração mais recente tem prioridade de uso. Existem

diversas formas de se implementar a verificação de escopos, entre elas a numeração hierárquica e a pilha de escopos.

Verificação de tipos

Esta parte do analisador semântico verifica:

- Se a referência indireta é aplicada a um ponteiro;
- Se as variáveis e constantes são compatíveis entre si;
- Se as variáveis e constantes são compatíveis com os operadores;
- Se a variável do lado esquerdo da atribuição é compatível com o tipo resultante do lado direito da expressão.
- Se a função está sendo chamada com o número de parâmetros corretos e com tipos adequados.
- Se a estrutura indexada é um vetor.

2.4 Simulação

Com um custo reduzido, por meio da simulação, pode-se predizer o comportamento futuro de um projeto ou mesmo de algumas alterações em um sistema existente [22].

O processo de desenvolvimento de uma simulação envolve diversas etapas. Primeiro é necessário especificar o modelo, abstraindo as características mais importantes do sistema. A partir do modelo é preciso transformá-lo em um programa de simulação.

Simular é o processo de desenvolver um modelo matemático ou lógico de um sistema real e então realizar experimentos de modo a prever o comportamento real do sistema [23]. Para que uma simulação seja realizada [24], algumas etapas devem ser seguidas, tais etapas são descritas a seguir.

Formulação do problema: descrever o sistema sem dubiedade, não negligenciando características relevantes nem considerando detalhes desnecessários que podem causar erros no futuro.

Ajuste dos objetivos e planos: traçar objetivos e planos. Os objetivos indicam quais serão as questões que a simulação irá responder e os planos devem incluir as várias possibilidades que serão analisadas.

Construção do modelo: após o sistema a ser simulado estar bem definido, é necessário abstrair um modelo do mesmo. Um modelo deve ser de fácil compreensão, mas sem perder as características essenciais para a determinação do desempenho.

Coleta de dados: dados que iram fluir pelo sistema. Estes são de entrada e servirão de parâmetros para o modelo. Podem ser valores hipotéticos ou baseados em alguma análise preliminar.

Codificação: a escolha da linguagem depende de preferências do usuário, sistema a ser simulado ou ainda de recursos disponíveis. Três abordagens podem ser utilizadas para descrever o comportamento dinâmico de sistemas discretos: atividades, processos e eventos. Normalmente as linguagens são orientadas a uma dessas abordagens [25], portanto para a implementação do modelo, pode-se seguir um dos seguintes enfoques

- *Desenvolvimento do programa de simulação em uma linguagem convencional:* vantajosa no fato de um usuário não precisar aprender uma nova linguagem, mas em contrapartida há a inexistência de ferramentas para auxiliar o desenvolvimento da simulação.
- *Utilização de um pacote de uso específico para o desenvolvimento do programa de simulação:* são poucos flexíveis a mudanças, mas são de fácil utilização para o objetivo proposto.

- *Uso de linguagens de simulação de uso geral:* existe a necessidade de aprendizado de uma nova linguagem por parte do desenvolvedor do sistema, mas pode livrar o programador de certos detalhes da implementação.
- *Uso de uma extensão funcional de uma linguagem de programação de uso geral:* são bibliotecas inseridas em linguagens convencionais, formando um ambiente completo para o desenvolvimento de uma simulação.

Verificação: consiste em verificar se o programa de simulação é uma implementação válida do modelo [26].

Validação: deve-se demonstrar que o modelo proposto é uma representação do sistema a ser simulado.

Período experimental: nesta fase são tomadas decisões com relação ao tamanho da simulação, o número de replicações e a maneira como a simulação é inicializada, de modo a manter um custo mínimo.

Produção das rodadas e análise: após o inicio das simulações dados a serem analisados começam a ser coletados. Os métodos para análise de resultados baseiam-se em estimar o intervalo de confiança dos valores médios.

Emissão de relatório: deve-se emitir ao usuário do programa de simulação, resultados claros e de forma concisa. Para isso um relatório é gerado.

2.5 Ferramentas de Simulação Existentes para *Grids*

A análise do comportamento de *grids* computacionais pode ser feita através do uso de simulações ou por meio de experimentos feitos em ambientes de *grids* reais. Experimentos em plataformas reais podem resultar em dados confiáveis, mas apresentam uma série de limitações como escalabilidade, mínima possibilidade de reconfiguração de *software*, a dependência a um conjunto de condições reais, entre

outros. Com isso resultados obtidos a partir de uma plataforma dificilmente representaram dados de outras plataformas.

Os simuladores, por apenas executarem um modelo do sistema real, independem da plataforma de execução e permitem abordar comportamentos específicos de um sistema distribuído.

Para estudar o comportamento das políticas de escalonamento de tarefas existentes para *grids* computacionais, o uso de simuladores é de extrema importância. Por meio das ferramentas de simulação, pode-se avaliar e comparar o desempenho de diferentes algoritmos em diferentes cenários. Várias ferramentas foram desenvolvidas com esse propósito, entre elas SimGrid, GridSim e GangSim.

SimGrid

Foi desenvolvida por Henri Casanova em um projeto de pós-doutorado em 1999. SimGrid [27][28] foi construída pela necessidade de se utilizar simulação ao invés de experimentos reais, no estudo prático de algoritmos de escalonamento.

Fornece um conjunto de ferramentas e funcionalidades para a simulação de aplicações distribuídas em ambientes heterogêneos distribuídos. Provê alguns ambientes de programação construídos sobre um único núcleo de simulação. Cada ambiente é destinado a um usuário alvo.

O poder computacional é definido neste simulador como sendo o número de unidades de trabalho por unidade de tempo. Não faz nenhuma distinção entre transferência de dados e computação, ambos são vistos como tarefas e é de responsabilidade do usuário garantir que tarefas de computação sejam escalonadas em processadores e transferência de dados para conexões de rede. Assume que todas as tarefas são *CPU-bound* e que transferência de dados são *bandwidth-bound*.

A implementação das políticas de escalonamento é feita através da programação, com a utilização da API em C por ele fornecida. Tal API permite manipular tipos de dados para recursos e para tarefas. Um recurso é descrito pelo nome, um conjunto de métricas relacionadas a desempenho, traços e constantes. Uma tarefa é descrita pelo nome, custo e estado. Além de funções básicas como criação, inspeção e destruição,

são fornecidas funções que descrevem possíveis dependências entre tarefas e funções, para designar tarefas para recursos.

Seu principal objetivo é facilitar a pesquisa na área de escalonamento de aplicações paralelas e distribuídas em plataformas computacionais distribuídas.

Atualmente está na versão 3.2.

GridSim

GridSim [29][30] permite a modelagem e simulação de entidades em paralelo e sistemas computacionais distribuídos, aplicações, recursos, além de escalonadores para o projeto e validação de algoritmos de escalonamento. Facilita a criação de diferentes classes de recursos heterogêneos que podem ser agregados através do uso de escalonadores para solução de aplicações intensivas de dados e de computação. Um recurso pode ser um simples processador ou um multiprocessador com memória distribuída ou compartilhada e gerenciada por escalonadores de tempo ou espaço compartilhado. Os nós de processamento podem ser heterogêneos em termos de capacidade de processamento, configuração e disponibilidade. Os escalonadores usam algoritmos de escalonamento e políticas para mapear trabalhos para recursos com o intuito de otimizar o sistema ou os objetivos do usuário dependendo das suas necessidades.

Atualmente o GridSim está na versão 4.2.

GangSim

GangSim [31] foi construída para auxiliar estudos de escalonamento em *grids* computacionais. Tais estudos visam avaliar o impacto das políticas de alocação de recursos adotadas por *sites* e organizações virtuais (VO). Sendo assim a ferramenta simula grandes grupos de usuários. Em g*rids* computacionais, uma VO é definida como um conjunto de instituições que compartilham recursos de forma coordenada e que atendem a determinados requisitos [32]. Requisitos estes envolvendo um único

método de autenticação, autorização, acesso aos recursos, descoberta de recursos, entre outros.

Permite combinar componentes e simulação com instâncias da ferramenta de monitoração VO-Ganglia executando em recursos reais.

A especificação da carga de trabalho, que caracteriza o conjunto de tarefas a serem simuladas, e do ambiente do *grid* é realizada por meio de ferramentas específicas oferecidas pelo GangSim. A modelagem do programa de simulação baseia-se na especificação das políticas de alocação de recursos nos *sites* e nas Vos.

2.6 Considerações Finais

Foi apresentada neste capítulo uma descrição aprofundada sobre alguns conceitos relevantes envolvendo *grids* computacionais, sendo o principal deles o tópico relacionado a escalonamento. Além de *grids*, foi feita uma breve descrição sobre compiladores, descrevendo suas fases de análise. Por fim, foi apresentada uma breve descrição de simulação em ambientes computacionais e algumas ferramentas de simulação existentes. No próximo capitulo será descrita a implementação da ferramenta.

Capítulo 3 – Desenvolvimento da Ferramenta

3.1 Considerações Iniciais

Neste capítulo será tratado todo o desenvolvimento da ferramenta, incluindo a implementação das políticas de escalonamento *Workqueue*, *Workqueue with Replication* e *Sufferage*, a simulação das mesmas e o desenvolvimento de uma linguagem para criação de políticas. É descrito também, como o ambiente é preparado para simular a variação de carga dos recursos presentes em *grids* computacionais.

Para a implementação da ferramenta foi utilizada a linguagem Java. Java [33] é uma linguagem de programação desenvolvida na década de 90 por uma equipe de programadores na empresa *Sun Microsystems*. É orientada a objetos, distribuída com um vasto conjunto de bibliotecas, facilita a criação de programas distribuídos e por ser independente de plataforma, funciona em qualquer sistema operacional que suporte a máquina virtual Java (JVM).

A ferramenta é dividida em quatro grandes módulos, a simulação, a comparação e a criação de políticas e um módulo com a implementação das políticas. Um esquema da relação entre esses módulos é ilustrado na figura 3.1.

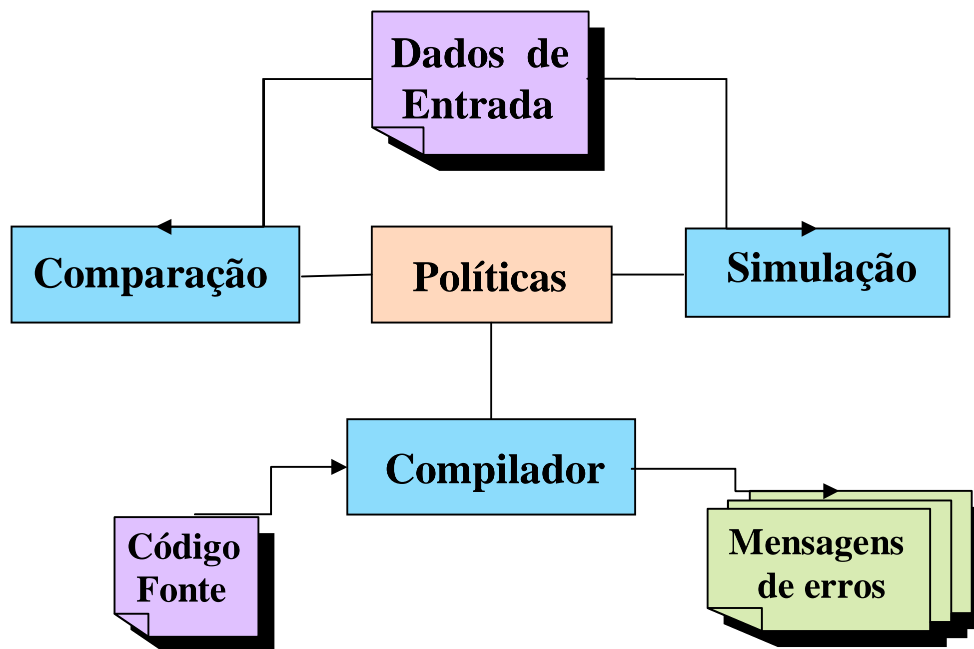

Figura 3.1: Relação entre os módulos da ferramenta.

Os módulos Simulação, Comparação e Compilador se relacionam com o módulo Políticas. No módulo Políticas foram implementados os três algoritmos de escalonamento escolhidos para este trabalho. A interface Dados de Entrada fornece os dados necessários para iniciar a Simulação ou então a Comparação de políticas. Os dados de entrada pedidos ao usuário foram escolhidos com base nas ferramentas para simulação de *grids* já existentes. O módulo Simulação prepara o ambiente de simulação e depois disso faz chamadas ao módulo Políticas apresentando ao usuário, em tempo de execução, dados da simulação. A comparação também faz chamadas ao módulo Políticas, mas apresenta resultados ao usuário apenas após o término da execução dos algoritmos.

No módulo Compilador é inserido um Código Fonte e depois de concluídas as análises sobre esse código são mostradas ao usuário mensagens de erros. Caso a análise não tenha encontrado erros o processo continua, sendo agora necessárias chamadas ao módulo Políticas para que o código possa ser interpretado.

3.2 Simulação das Políticas de Escalonamento

O módulo Simulador tem por objetivo simular algumas das principais políticas de escalonamento existentes para *Grids* Computacionais. Para que a simulação gere bons resultados, ou seja, mais próximos possíveis de resultados obtidos em ambientes de *grids* reais, a variação de disponibilidade de UCP para as máquinas do *grid*, foi feita com o auxílio de arquivos de traces obtidos a partir do NWS (*Network Weather Service*).

O NWS é um sistema distribuído que realiza medições e faz previsões da disponibilidade de vários tipos de recursos computacionais, tais como memória, processador e recursos de comunicação [34].

Foram utilizados *traces* coletados de 81 máquinas [35] da Universidade da Califórnia, Santa Bárbara [36]. Estes *traces* contém informações sobre a disponibilidade de UCP em um certo *timestamp*. A figura 3.2 ilustra como é formado o conteúdo de um arquivo de *trace*. Na primeira coluna tem-se o *timestamp* e na segunda o valor de UCP disponível naquele *timestamp*.

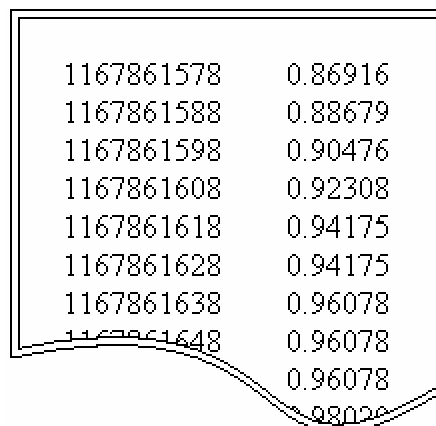

Figura 3.2: Formato dos arquivos de *traces* utilizados na simulação.

Esses arquivos de traces foram baixados previamente e armazenados em uma pasta de Traces no pacote da ferramenta. Os nomes dos arquivos são formados da seguinte maneira:

traceID.txt

onde, ID varia de 0 a 80 representando os traces utilizados de cada uma das 81 máquinas.

Os dados sobre a quantidade de UCP disponível foram armazenados em memória com o auxílio de uma lista, onde cada nó dessa lista representa a taxa de UCP livre em um *timestamp*. Usou-se essa abordagem, pois é mais eficiente acessar dados previamente alocados em memória do que fazer acesso a disco a todo momento, o que seria feito se os dados fossem coletados em tempo de execução diretamente dos arquivos.

3.2.1 Simulação das Aplicações

Aplicações são formadas por um conjunto de tarefas. Tais tarefas têm seu tamanho medido em milhões de instruções (MI) necessárias para completar sua execução. Para cada tarefa, um objeto Tarefa é instanciado, abaixo segue um trecho de código contendo a classe Tarefa e seus atributos:

```
public class Tarefa {
    int id;
    int tam;
    int chegada;
    int saida;
    int falta_executar;
    int tempo_execucao;
    DadosSufferage suffer;
    int replicas;
}
```

Segue descrição detalhada de cada atributo:

- **id:** utilizado para identificar cada tarefa de uma aplicação. Inicialmente 0 e incrementado em uma unidade a cada tarefa criada.
- **tam:** medida em número de milhões de instruções necessárias para que uma tarefa seja executada em sua integra.
- **chegada:** medida em segundos, informa o momento em que a tarefa foi alocada para alguma máquina do *grid*, ou seja, informa o início de seu processamento.
- **saída:** medida em segundos, informa o momento em que a tarefa conclui sua execução.
- **falta_executar:** medida em número de milhões de instruções que faltam ser executados para que a tarefa conclua sua execução. Inicialmente tem conteúdo igual a **tam**.
- **tempo_execução:** medida em segundos, informa a quanto tempo a tarefa está em execução. Utilizada para calcular a média de tempo de execução das tarefas.

  suffer: objeto da classe DadosSufferage, objeto este utilizado apenas para escalonamento com a política *Sufferage*, portanto será detalhado mais adiante, na descrição dessa política.
- **replicas:** indica o número de réplicas existentes da tarefa. Utilizado para políticas que replicam tarefas, para as que não replicam esse valor se mantém 0 durante toda a simulação.

Entre tarefas de uma mesma aplicação pode existir heterogeneidade. Para simular tal heterogeneidade é pedido ao usuário que informe o tamanho médio das tarefas e o valor da heterogeneidade. Assim o tamanho das tarefas é gerado segundo esse valor de heterogeneidade, mas mantendo constante o tamanho médio das tarefas. Para que a média se mantenha constante foi usada uma distribuição probabilística que propicia tal característica, assim como a Distribuição Uniforme U(mínimo, máximo). Por exemplo, tarefas com tamanho médio de 1000 MI e heterogeneidade 10% tem a

distribuição ditada por U(950, 1050), onde 950 e 1050 consistem nos tamanhos mínimo e máximo das tarefas, respectivamente. Foi usado um objeto **Random** e seu método **nextInt** do pacote **java.util.Random** como gerador de números para essa distribuição. O método **nextInt** gera números segundo uma distribuição Uuniforme

A fila de tarefas é criada como descrito abaixo:

```
para (i de 0 até qtde_tarefas){
    min_tar = tam_medio – (tam_medio * (heterogeneidade/200));
    max_tar = tam_medio + (tam_medio * (heterogeneidade/200));
    tam = gerRandom(rd, min_tar, max_tar);
    tarefa_aux = novo abjeto da classe Tarefa com os atributos
i, tam, 0, 0, tam, null, 0;
    adiciona tarefa_aux ao final da fila de tarefas;
}
```

onde, **gerRandom** é um método que gera o próximo número inteiro no intervalo [min_tar, max_tar] seguindo uma distribuição uniforme.

Para cada tarefa, o valor dos atributos chegada, saída e replica são inicialmente zero.

3.2.2 Simulação dos Recursos

As máquinas do *grid* têm seu poder de processamento medido de acordo com a quantidade de milhões de instruções que podem executar por segundo (MIPS), quando estão 100% disponíveis. Para cada máquina um objeto da classe Maquina é criado. Segue a definição da classe Maquina.

```
public class Maquina {
    int id;
    int poder;
    double cpu_livre;
    int status;
    ArrayList<Double> traces;
    Tarefa tarefa;
}
```

- **id:** utilizado para identificar cada máquina do *grid*. Inicialmente 0 e incrementado em uma unidade a cada máquina acrescentada.
- **poder:** número de milhões de instruções que a máquina pode processar por segundo, quando está totalmente disponível.
- **cpu_livre:** quantidade de UCP disponível para processamento, atualizado a cada 10 segundos, com base na lista de *traces* gerada a partir de um arquivo de *trace*. Valores variam entre 0 e 1.
- **status:** indica o estado da máquina. Pode assumir dois valores, 0 para máquina indisponível e 1 para máquina disponível.
- **traces:** lista contendo todos os valores de cpu_livre, cada nó representando o valor em um certo *timestamp*.
- **tarefa:** objeto da classe Tarefa, contém os dados da tarefa alocada para a máquina no momento atual.

A heterogeneidade de recursos, assim como a de aplicações, é bastante presente em *grids*. Para simular tal heterogeneidade foi utilizado o conceito da Lei de Moore [37], que diz que a cada 18 meses a velocidade de processamento das máquinas praticamente dobra. Sendo assim, os valores de heterogeneidade possíveis variam de acordo com a diferença de idade entre os recursos do *grid*. Por exemplo, para uma diferença de 3 anos, os valores possíveis para heterogeneidade são 1, 2, 4, ou seja, computadores com até 4 vezes mais capacidade de processamento que os mais lentos. Para implementar tal variação de poder de processamento, foi novamente utilizada a distribuição uniforme, a fim de manter constante a média de velocidade de processamento das máquinas do *grid*, média essa informada pelo usuário antes do início da simulação.

Para gerar a lista de máquinas, de acordo com a quantidade desejada pelo usuário, uma lista de objetos da classe Maquina se faz necessária. A criação dessa lista é descrita abaixo.

```
para(i de 0 até qtde_maquinas){
      traces = InicializaTraces(i);
      cpu_livre = traces(0);
      poder = gerRandom(rd, min_maq, max_maq);
      maquina_aux = novo objeto da classe Maquina com atributos i,
poder, cpu_livre, 1, traces, null;
      adiciona maquina_aux ao final da lista_maquinas;
}
```

O método **InicializaTraces(i)** carrega os dados do arquivo de traces com ID **i** para a lista **traces**. O atributo **cpu_livre** é inicializado com o valor de UCP disponível equivalente ao primeiro *timestamp*. O poder de processamento da máquina é gerado seguindo uma distribuição uniforme. Todas as máquinas da lista são criadas com **estado disponível**.

Simulação das Políticas

Depois de definidas a lista de máquinas e a fila de tarefas, deve-se escolher com qual política será simulado o escalonamento. Como os algoritmos de escalonamento foram implementados no módulo Políticas, após escolhida a política chamadas são feitas ao módulo Políticas.

3.3 Políticas

Como dito anteriormente três políticas foram implementadas nesse módulo, a *Workqueue*, a *Workqueue with Replication* e a *Sufferage*. Nos tópicos a seguir são detalhadas as implementações de cada um dos algoritmos.

3.3.1 *Workqueue*

Como visto no capítulo 2 a política *Workqueue* atende a um conjunto de tarefas, escolhendo aleatoriamente qual tarefa alocar sempre que houver recurso disponível.

A implementação consiste basicamente de um laço que só termina quando todas as tarefas da fila forem alocadas.

```
enquanto(existir tarefas na fila){
    . . .
}
```

Ao mesmo tempo uma rotina de verificação é invocada, atualizando o estado das máquinas ao verificar se uma tarefa já concluiu sua execução e atualizando também o valor de UCP disponível para cada máquina do *grid*.

```
enquanto(existir tarefas na fila){
        atualizaDadosMaquinas()
        . . .
}
```

Após o término da atualização, é verificado se existe máquina disponível, caso exista, a próxima tarefa da fila de tarefas é alocada para a máquina. O status da máquina é alterado para indisponível e o valor de chegada da tarefa é atualizado para o valor atual do relógio. A tarefa é retirada da fila de tarefas e o número de tarefas em execução é incrementado em uma unidade. O valor do relógio também é acrescido em uma unidade.

```
enquanto(existir tarefas na fila){
        atualizaDadosMaquinas()
        se(houver máquina disponível){
                alocar primeira tarefa da fila de tarefas para a máquina;
                atualizar o status da máquina para ocupada;
                valor de chegada da tarefa recebe valor do relógio;
                retirar tarefa da fila de tarefas;
                incrementar número de tarefas em execução;
        }
        incrementa em uma unidade o valor do relógio
}
```

Assim que todas as tarefas forem alocadas, o programa irá sair do laço, mas não pode ser finalizado, pois existe a possibilidade de ainda haver tarefas em execução. Para concluir é necessário um novo laço, mas agora o critério de parada passa a ser o número de tarefas em execução, ou seja, quando não houver mais tarefas executando o programa termina. Interno a esse laço, o relógio é atualizado.

```
. . .
enquanto(existir tarefas em execução){
   incrementa relógio em uma unidade;
}
```

A cada instante da simulação, dados estatísticos como número de tarefas concluídas, porcentagem de desperdício de UCP, tempo médio de execução de tarefas concluídas e número de máquinas ocupadas são atualizados e informados ao usuário, como pode ser visto na figura 3.3. Ao término da simulação é informado o tempo total de execução da aplicação, dado este importante para que mais adiante possamos comparar as políticas.

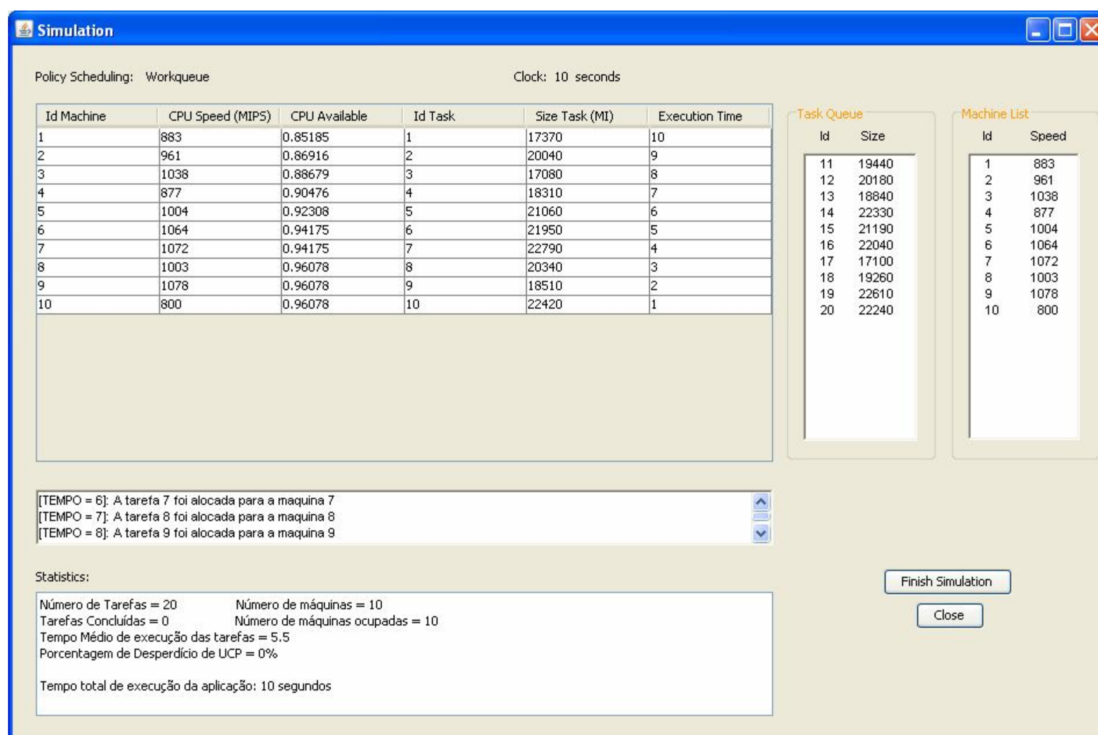

Figura 3.3: Tela da simulação, informando os resultados no quadro *Statistcs*.

3.3.2 *Workqueue with Replication*

A política *Workqueue with Replication* é similar a *Workqueue*, diferindo apenas a partir do momento em que todas as tarefas da fila de tarefas forem alocadas para execução. Uma alteração no código da política *Workqueue* faz-se necessária, para que seja possível reaproveitar seu código na implementação da WQR. Essa alteração inclui a criação de uma lista de tarefas em execução. A implementação de tal lista é muito simples, bastando apenas inserir um novo objeto da classe **DadosReplicas** na lista de tarefas em execução a cada momento em que uma tarefa é alocada para execução. A classe **DadosReplicas** é descrita abaixo:

```
public class DadosReplicas {
    int id_tarefa;
    int num_replicas;
    ArrayList<Maquina> maquinas;
}
```

Onde:

- **id_tarefa:** id da tarefa em execução.
- **num_replicas:** número de réplicas existentes da tarefa. Inicialmente vale 1, e a cada réplica criada é incrementado em uma unidade.
- **maquinas:** lista de objetos da classe Maquina, utilizada para armazenar as máquinas onde a tarefa com id id_tarefa foi replicada.

Sendo assim, o código modificado fica como descrito a seguir:

```
Workqueue(){
    . . .
    enquanto(existir tarefas na fila){
        se(houver máquina disponível){
            alocar a primeira tarefa da fila de para a máquina;
         -> inserir dados da tarefa na lista de tarefas em execução;
                . . .
        }
```

```
        }
    }
```

A lista de tarefas em execução é necessária para que seja possível o gerenciamento de réplicas das tarefas, pois sem ela não existiriam informações sobre tais tarefas, já que uma vez alocadas para execução, são retiradas da fila de tarefas.

Após concluir a alocação de todas as tarefas, o programa invoca uma função para verificar a disponibilidade de máquinas pertencentes ao *grid*, enquanto existirem tarefas em execução.

```
AtualizaDadosMaquinasWQR(){
    para(i de 0 até qtde_maquinas){
        se(maquina[i] estiver ocupada){
            atualiza dados da tarefa em execução na maquina;
            se(tarefa concluiu a execução){
                nr = tarefas_execução[tarefa].replicas;
                para (j de 0 ate nr){
                    atualiza status da maquina para disponível;
                }
                retira tarefa da lista de tarefas em execução;
                . . .
            }
            . . .
}
```

Tal função analisa máquina a máquina do *grid* verificando se a tarefa em execução na máquina já foi concluída. Caso a tarefa tenha sido concluída, todas as réplicas existentes da tarefa são abortadas, evitando desperdício excedente de UCP.

Após o término da execução da função de atualização das máquinas, caso exista máquina disponível, verifica-se, para a primeira tarefa da lista de tarefas em execução, seu número de réplicas, por meio de seu atributo **num_replicas** e compara-se esse valor com o número máximo permitido de réplicas, tal valor é passado como parâmetro na chamada do método WQR. Se o número de réplicas da tarefa for menor que o valor máximo de réplicas permitido então a tarefa é replicada. Seu número de

replicas é acrescido em uma unidade e para manter a consistência dos dados, esse valor é atualizado em todas as réplicas existentes da tarefa, réplicas alcançadas por meio da lista de máquinas. Uma tarefa pode apenas ser replicada pela segunda vez, se todas as outras tarefas em execução já tiverem sido replicadas ao menos uma vez. Isso é garantido, pois a lista de tarefas em execução é uma lista circular e sempre que uma tarefa é replicada o início da lista é atualizado para o próximo nó, ou seja, para a referência da próxima tarefa em execução.

```
faça {
     AtualizaDadosMaquinasWQR();
     se(existir máquina disponível){
          se(tarefas_execução.replica < max de réplicas permitido){
               atualiza número de réplicas;
               para (i de 0 até número de réplicas atual){
                    atualiza número de réplicas da tarefa;
               }
               replica tarefa;
               altera status da máquina para ocupada;
               atualiza ponteiro da lista de tarefas em execução;
               . . .
          }
     }
     . . .
}
enquanto(existir tarefas em execução);
```

Assim como na *Workqueue*, na WQR dados estatísticos também são impressos ao usuário. Mas aqui um dos dados se destaca, a porcentagem de desperdício de UCP, pois existindo a replicação de tarefas, ciclos de UCP serão utilizados para a execução de tarefas que mais adiante serão abortadas pelo término de alguma de suas réplicas, desperdiçando assim os ciclos de UCP que utilizaram até o momento.

3.3.3 *Sufferage*

Sufferage é uma política dependente de dados sobre as tarefas e os recursos do *grid*. Para implementá-la primeiro foi necessário o desenvolvimento de uma função que fosse capaz de calcular o valor de *sufferage* para cada tarefa da fila. Tal função foi escrita como abaixo:

```
DadosSufferage CalculaSufferage(tarefa){
	para(i de 0 ate qtde_maquinas){
	 ct = tarefa.tam / maquina(i).poder;
	 se( ct < dados.melhor_ct ){
		dados.segundo_ct = dados.melhor_ct;
		dados.melhor_ct = ct;
		dados.id_maquina = maquina(i).id;
	 }
	 senao{
		se( ct < dados.segundo_ct ){
			dados.segundo_ct = ct;
		}
	}
	dados.valor_sufferage = dados.segundo_ct - dados.melhor_ct;
	 retorne(dados);
}
```

Onde DadosSufferage é uma classe definida como:

```
public class DadosSufferage {
	double melhor_ct;
	double segundo_ct;
	int id_maquina;
	double valor_sufferage;
}
```

- **melhor_ct:** melhor *completion time* para a tarefa.
- **segundo_ct:** segundo melhor *completion time* para a tarefa.

- **id_maquina:** maquina com a qual a tarefa obteve o melhor *completion time*.
- **valor_sufferage:** diferença entre o segundo melhor e o melhor *completion time* encontrados para a tarefa.

A princípio, para cada máquina do *grid*, a função calcula o melhor e o segundo melhor *completion time* das tarefas em execução nas respectivas máquinas. Para calcular o melhor *completion time* da tarefa, utiliza dados como poder e cpu_livre das máquinas e tamanho da tarefa. Após isso, para encontrar o valor de *sufferage* para aquela tarefa, é feita a subtração do valor do segundo melhor *completion time* pelo melhor valor encontrado.

Depois de encontrado o valor de *sufferage* para cada tarefa, escolhe-se a tarefa com o maior valor dentre eles. Essa tarefa A é escalonada para a máquina que a executaria em menor tempo, desde que a máquina esteja disponível, caso a máquina esteja ocupada, o valor de *sufferage* da tarefa em execução nela e o valor de *sufferage* da tarefa A são comparados e a que possuir maior valor ganha a disputa. Se a tarefa A for escolhida para ser alocada para a máquina, a tarefa que estava executando é desescalonada e retorna para a fila de tarefas. Isso é feito até que todas as tarefas da fila de tarefas tenham sido alocadas para execução, e devido à heterogeneidade do *grid*, a cada laço os valores de *sufferage* das tarefas da fila são recalculados.

```
Sufferage(){
    DadosSufferage maior;
    i = 0;
    enquanto(existir tarefas na fila de tarefas){
        atualizaDadosMaquinas();
        para(todas as tarefas da fila de tarefas){
            dados = CalculaSufferage(tarefa[i]);
            se(dados.valor_sufferage > maior.valor_sufferage){
                maior = dados;
                tarefaA = id tarefa atual;
            }
        }
```

```
se(maquina[maior.id_maquina] estiver disponivel){
    aloca tarefa[tarefaA] para a máquina[dados.id_maquina];
    retira tarefa[tarefaA] da fila de tarefas;
    marca máquina como ocupada;
}
senão{
    se(maquina[maior.id_maquina].tarefa.suffer <
                                  tarefa[tarefaA].suffer){
        desescalona tarefa da maquina e insere na fila de tarefas;
        aloca tarefa[tarefaA] para a máquina;
    }
    . . .
}
```

Assim como nas duas políticas descritas anteriormente, dados estatísticos são impressos ao usuário. No caso da política *Sufferage*, também é possível visualizar o valor de *sufferage* de cada tarefa em execução.

3.4 Comparação das Políticas

Perante um conjunto de dados de entrada inseridos pelo usuário, como quantidade de máquinas, quantidade de tarefas, média de poder de processamento, tamanho médio das tarefas, além dos valores de heterogeneidade para máquinas e tarefas são invocados os algoritmos referentes às três políticas implementadas na ferramenta. Dados como tempo total de processamento e porcentagem de ociosidade da UCP são impressos ao usuário para que ele possa tirar suas conclusões sobre qual política é mais adequada ao ambiente por ele fornecido.

3.5 Criação de uma Política

O sistema desenvolvido permite ao usuário criar sua própria política de escalonamento. Para isso uma linguagem teve de ser desenvolvida, além disso um compilador para a linguagem também foi implementado.

3.5.1 Linguagem

A linguagem desenvolvida é bastante simples, constituída basicamente de palavras reservadas, constantes numéricas inteiras, operadores, símbolos e do comando de seleção IF-ELSE. É uma linguagem Livre de Contexto.

Tabela 3.1: palavras reservadas e constantes da linguagem desenvolvida.

Palavras Reservadas	
NUM_TAREFAS	MAIOR
NUM_MAQUINAS	SUFFER
TAM_TAREFA	CT
PODER_MAQUINA	TAM
HETEROGENEIDADE_TAREFAS	PODER
HETEROGENEIDADE_MAQUINAS	REPLICA
POLITICA	TEMPO
RANDOM	IF
SUFFERAGE	ELSE
MENOR	
Operadores e Símbolos	
>	;
<	(
=	)
==	{
!=	}
Constantes Numéricas Inteiras	
Ex: 100, 560, 1, 58, 12548, ...	

Abaixo segue uma breve descrição do que cada palavra reservada representa:

- **NUM_TAREFAS:** define o número de tarefas pertencentes a aplicação a ser executada no *grid*.
- **NUM_MAQUINAS:** define o número de máquinas que formarão os recursos do *grid*.
- **TAM_TAREFA:** valor médio do tamanho das tarefas.

- **PODER_MAQUINA:** valor médio de poder de processamento das máquinas.

- **HETEROGENEIDADE_TAREFAS:** define o quão heterogêneas serão as tarefas da aplicação. Se o valor for zero as tarefas serão consideradas homogêneas, sendo todas definidas com o mesmo tamanho definido por TAM_TAREFA, caso contrário o tamanho das tarefas irá variar em torno da média TAM_TAREFA, o valor definido por essa variável.

- **HETEROGENEIDADE_MAQUINAS:** define o quão heterogêneos serão os recursos do *grid*. Se o valor for zero, os recursos terão o mesmo poder de processamento, formando assim um *grid* homogêneo. Caso contrário o poder de processamento das máquinas irá variar de acordo com a heterogeneidade sugerida, seguindo a Lei de Moore.

Todas as palavras reservadas descritas acima são constantes que representam os dados iniciais da simulação e podem ser omitidas no desenvolvimento da política. Caso alguma, ou todas sejam omitidas, a ferramenta usará como base para esses dados valores pré-definidos. Abaixo é apresentada a lista de valores pré-definidos para essas constantes.

  - NUM_TAREFAS = 720
  - NUM_MAQUINAS = 100
  - TAM_TAREFA = 5000 MI
  - PODER_MAQUINA = 10 MIPS
  - HETEROGENEIDADE_TAREFAS = 25%
  - HETEROGENEIDADE_MAQUINAS = 4

- **POLÍTICA:** palavra obrigatória, define o início do bloco de escolha das políticas. As opções para políticas são: RANDOM, SUFFERAGE, MENOR(opção) e MAIOR(opção), onde opção pode ser: SUFFER, CT, TAM ou PODER.

- **RANDOM:** define que a política será como a *Workqueue*, distribuindo as tarefas de forma randômica.

- **SUFFERAGE:** define que a política será como a *Sufferage*, escolhendo qual tarefa alocar de acordo com o valor de *sufferage*.

- **MENOR(opção):** define que a política dará prioridade ao menor valor, definido pela macro opção.

- **MAIOR(opção):** define que a política dará prioridade ao maior valor, definido pela macro opção.

A macro opção pode ser representada por SUFFER, CT, TAM ou PODER. Uma breve descrição de cada uma das macros é feita abaixo:

- **SUFFER:** define que o valor utilizado para dar prioridade será o valor de *sufferage* da tarefa.

- **CT:** define que o valor utilizado para dar prioridade será o valor do *completion time* da tarefa.

- **TAM:** define que o valor utilizado para dar prioridade será o tamanho da tarefa.

- **PODER:** define que o valor utilizado para dar prioridade será o poder de processamento da máquina.

As opções de políticas MAIOR e MENOR podem ser usadas em conjunto ou replicadas, como por exemplo:

```
MAIOR(TAM);
MAIOR(PODER);
```

Neste caso deseja-se um algoritmo que atribua a tarefa de maior tamanho para a máquina com o melhor poder de processamento.

- **REPLICA:** define se ocorrerá replicação de tarefas ou não. Se o valor for 1 tarefas não serão replicadas, caso contrário réplicas poderão ser criadas até o limite máximo definido por essa macro. O valor máximo permitido para essa macro é 4, permitindo no máximo a criação de 3 réplicas da tarefa original.

- **TEMPO:** variável contendo o valor atual do relógio global da simulação.

- **IF-ELSE:** comando de seleção tem o seguinte formato:

```
IF(expressao){
}
ELSE{
}
```

Onde *expressão* pode ser qualquer combinação de operadores, constantes numéricas e a variável TEMPO. Admite também comandos aninhados.

Uma maneira de formalizar linguagens é definir sua gramática. A seguir é descrita a gramática da linguagem desenvolvida. Os símbolos não terminais estão representados entre <> e os símbolos restantes são terminais da linguagem.

<inicio> ← <variáveis> | <comandos>
<comandos> ← <if_else> | <política> | REPLICA <sinal> <valor> <pv>
<variáveis> ← NUM_TAREFAS <sinal> <valor> <pv>
| NUM_MAQUINAS <sinal> <valor> <pv>
| TAM_TAREFA <sinal> <valor> <pv>
| PODER_MAQUINA <sinal> <valor> <pv>

| HETEROGENEIDADE_TAREFAS <sinal> <valor> <pv>

| HETEROGENEIDADE_MAQUINAS <sinal> <valor> <pv>

<política> ← POLITICA { <escolha_politica> }

<escolha_politica> ← RANDOM <pv>

| SUFFERAGE <pv>

| MAIOR (<opção>) <pv>

| MENOR (<opção>) <pv>

| MAIOR (<opção>) <pv> <outra>

| MENOR (<opção>) <pv> <outra>

outra ← MAIOR (<opção>) <pv> | MENOR (<opção>) <pv>

<opção> ← CT | SUFFER | TAM | PODER

<if_else> ← IF (<expressao>) { <comandos> }

| IF (<expressão>) { <comandos> } ELSE { <comandos> }

<operadores> ← > | < | != | ==

<valor> ← 0<valor> | 1<valor> | 2<valor> | 3<valor> | 4<valor>

| 5<valor> | 6<valor> | 7<valor> | 8<valor> | 9<valor>

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<expressao> ← <variáveis> <operadores> <variaveis>

| <variáveis> <operadores> <valor>

| TEMPO <operadores> <valor>

<sinal> ← =

<pv> ← ;

3.5.2 Compilador

Para desenvolver o compilador foram implementados os analisadores léxico, sintático e semântico e um interpretador simples para testes. Abaixo são descritos os analisadores e o interpretador.

Analisador Léxico

Utilizando as facilidades que a linguagem Java oferece, para implementar o autômato léxico foi utilizado um elemento do tipo *HashMap*, que mapeia para cada chave única, um atributo que pode ser de qualquer tipo aceito pela linguagem.

Neste caso, foram mapeados todas as palavras reservadas, os operadores e os símbolos da linguagem desenvolvida. Cada um desses representando a chave e seu respectivo símbolo especial a ser utilizado pelo analisador sintático, o atributo.

O analisador léxico verifica para cada palavra do código inserido, se ela existe no *HashMap*, caso isso seja verdade apenas copia o símbolo relativo a esse *token* para uma string auxiliar, que será utilizada pelo analisador sintático, caso contrário verifica se o *token* representa um valor numérico, se o resultado ainda continuar falso, significa que um erro léxico foi encontrado. Neste caso o erro é de identificador inválido, o único erro encontrado por esse analisador. O método de Análise Léxica é descrito abaixo.

```
Boleano AnaliseLexica() {
    String lexico recebe código inserido pelo usuário;
    troca caracteres de nova linha por & na string lexico;
    guarda todas as linhas em uma matriz de String;
    para( cada linha ){
        atualiza contador de linha;
        divide linha, guardando os tokens em um vetor de strings;
        para( cada token ){
            se( token existe como chave no HashMap){
                guarda respectivo símbolo em uma string aux;
            }
            senão{
                se( token é valor numérico ){
                    guarda valor;
                    associa valor com identificador;
                }
                senão{
                    erro léxico: identificador inválido;
```

```
                    incrementa numero de erros;
                }
            }
        }
    }
    se( numero de erros for maior que zero ){
        retornar Falso;
    }
    senao{
        retornar Verdadeiro;
    }
}
```

Se algum erro for encontrado, um caractere de erro é inserido na *string* auxiliar que servirá como base para o analisador sintático. Esse caractere é usado para que a análise possa prosseguir, podendo informar ao usuário no final da compilação, o resultado de todas as analises. Após o analisador léxico é chamado o analisador sintático, ele irá analisar a string auxiliar gerada pelo léxico.

Analisador Sintático

O analisador sintático recebe como entrada uma *string* advinda da análise léxica. Tal *string* é constituída de símbolos especiais entendidos pelo analisador sintático, facilitando bastante a análise. Foi construído um reconhecedor para a linguagem. Como a linguagem desenvolvida é Livre de Contexto, o reconhecedor é um autômato com pilha. Este autômato foi implementado através do método **AutomatoSintatico** e é invocado a cada símbolo lido, podendo retornar um estado intermediário ou um estado de erro, além de empilhar e desempilhar blocos e comandos de seleção. Um esquema do analisador sintático é mostrado abaixo.

```
Boleano AnaliseSintatica(String lexico){
    para ( cada linha em lexico ){
        atualiza contador de linha;
        para ( cada simbolo na linha ){
```

```
            estado = AutomatoSintatico(estado_anterior, simbolo);
            se ( estado for de erro ){
                grava erro sintático de acordo com o estado de erro;
                ignora o restante da linha;
            }
        }
    }
    se ( pilha não está vazia E topo = '{'){
        erro sintático: simbolo '}' esperado;
    }
    se ( NÃO houve erros ){
        retornar Verdadeiro;
    }
    senão {
      retornar Falso;
    }
}
```

Caso a chamada ao método **AutomatoSintatico** retorne um estado de erro, esse erro é guardado para posterior impressão e todo o restante da linha é ignorado. A linha não é retirada da string que será analisada pelo semântico, pois nela também pode existir algum erro semântico. Dentre os erros sintáticos estão:

- caractere ';' esperado
- comando ilegal, 'ELSE'sem 'IF'
- esperado '{' após 'IF'
- esperado '{' após 'POLITICA'
- operador de atribuição '=' esperado
- operador esperado na expressão do 'IF'
- esperado algum valor de comparação na expressão do 'IF'
- variável não permitida na expressão do 'IF'
- opção esperada
- esperado '('
- esperado ')'

Analisador Semântico

Para auxiliar na implementação da análise semântica foi construído um autômato semântico que verifica se os valores estão em uma faixa aceitável para cada variável da linguagem, além de verificar se os tipos nas expressões são compatíveis. O método **AnaliseSemantica** recebe como entrada uma *string* definida no analisador sintático. Lê símbolo a símbolo e para cada um deles faz uma chamada ao método **AutomatoSemantico** passando como parâmetro o estado anterior do sistema e o valor do símbolo atual. O método **AutomatoSemantico** retorna o estado atual, que pode ser um estado intermediário ou um estado de erro. Se for um estado de erro, o pedaço ainda não analisado da linha é ignorado e o estado do sistema passa a ser o inicial.

```
Boleano AnaliseSemantica(String sintatico){
    para ( cada linha em sintatico ){
        atualiza contador de linha;
        para ( cada símbolo na linha ){
            estado = AutomatoSemantico(estado_anterior, simbolo);
            se ( estado for de erro ){
                guarda erro semântico de acordo com o valor do erro;
                ignora o restante da linha;
                atualiza valor do estado para o inicial;
            }

        }
    }
    se ( NÃO houve erros ){
        retorna Verdadeiro;
    }
    senão{
        retorna Falso;
    }
}
```

Erros semânticos encontrados:

- valor definido fora da faixa aceita pela linguagem
- tipos incompatíveis na expressão

Interpretador

Nesta etapa do projeto, apenas foi desenvolvido um interpretador simples, a fim de testar algumas funcionalidades da linguagem desenvolvida.

Ele consistiu basicamente de chamadas aos métodos já implementados para simular as políticas descritas na seção 3.3.

Após o término do processo de compilação, se não houver nenhum erro, a string gerada pelo analisador léxico é passada para o interpretador, que por sua vez, analisa símbolo a símbolo interpretando-os se for o caso.

3.6 Considerações Finais

Neste capítulo foi descrito todo o processo de desenvolvimento da ferramenta. Nas seções 3.2.1 e 3.2.2 foi apresentado como o ambiente de simulação é montado de modo a parecer o mais próximo do real. Na seção 3.3 foram descritas as implementações das políticas de escalonamento propostas. Já na seção 3.4 foi apresentada a maneira como é feita a comparação das políticas.

O desenvolvimento da linguagem, sua especificação e a implementação do compilador foram descritos na seção 3.5.

No capítulo 4 serão apresentados alguns testes feitos e os resultados obtidos.

Capítulo 4: Testes e Resultados

4.1 Considerações Iniciais

Neste capítulo são apresentados os testes realizados e alguns resultados obtidos. Foram feitos testes para analisar a eficiência do simulador de políticas de escalonamento e também testes com o compilador. Na seção 4.2 são mostradas as simulações feitas e a comparação de eficiência dos algoritmos testados. Na seção 4.3 são apresentados os testes feitos com o compilador. Por fim na seção 4.4 são feitas considerações.

4.2 Simulação das Políticas de Escalonamento

Em simulações, diversos cenários para a execução de aplicações podem ser construídos. Isso é feito através da atribuição de diferentes valores aos parâmetros do programa [38]. Para avaliar o comportamento de políticas de escalonamento é necessária a reprodução de vários cenários.

Os cenários para a simulação foram ambientados em diferentes aspectos. Um cenário verifica o impacto da heterogeneidade dos recursos e um outro analisa o comportamento das políticas com diferentes tamanhos de tarefas.

Para atribuir parâmetros como quantidade de máquinas, quantidade de tarefas, média de tamanho das tarefas, média de poder de processamento, heterogeneidade

das tarefas e heterogeneidade dos recursos é fornecida a interface ao usuário mostrada na figura 4.1.

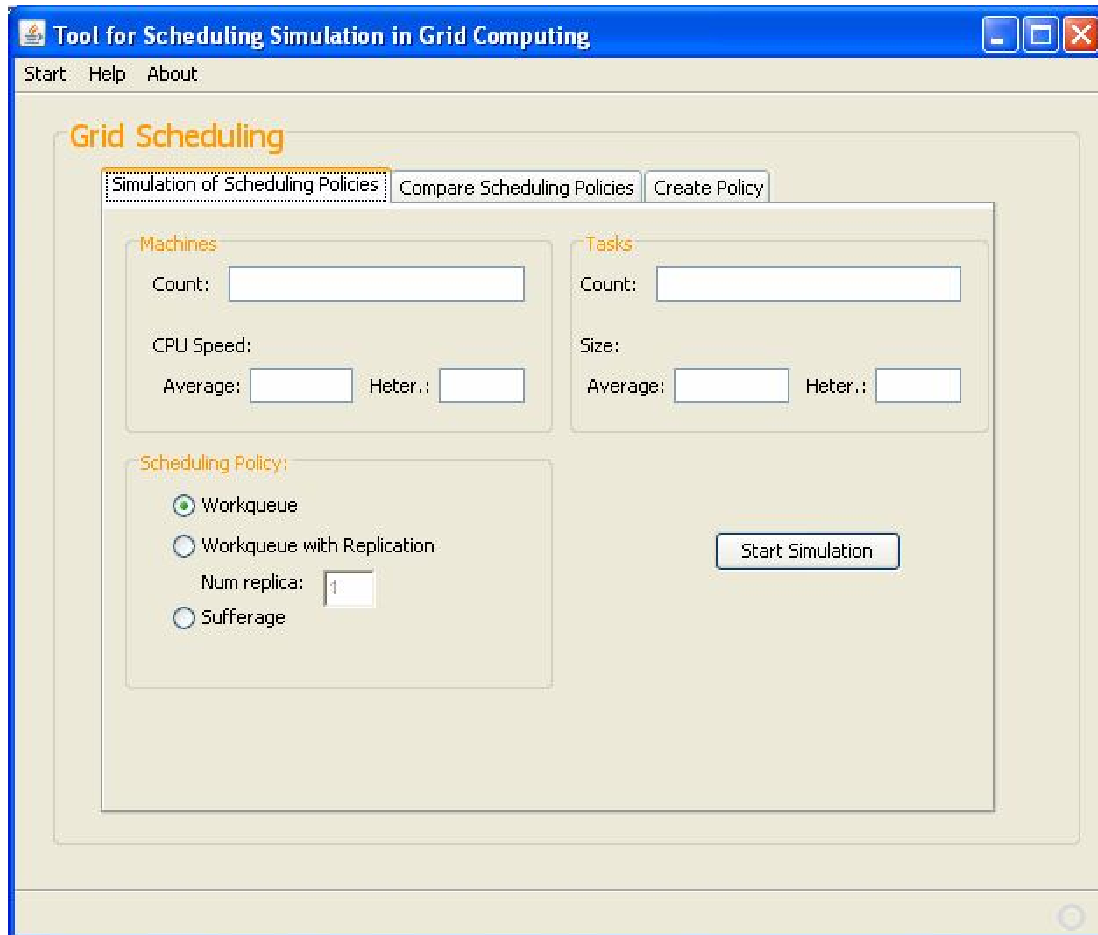

Figura 4.1: Tela inicial da ferramenta.

Neste caso, a tela mostrada é a de simulação das políticas, por isso é necessário também informar em qual política deseja-se simular. Para apenas comparar, sem visualizar as simulações pode-se entrar com os dados iniciais na segunda aba mostrada, a de comparação de políticas, tal tela é ilustrada na figura 4.2.

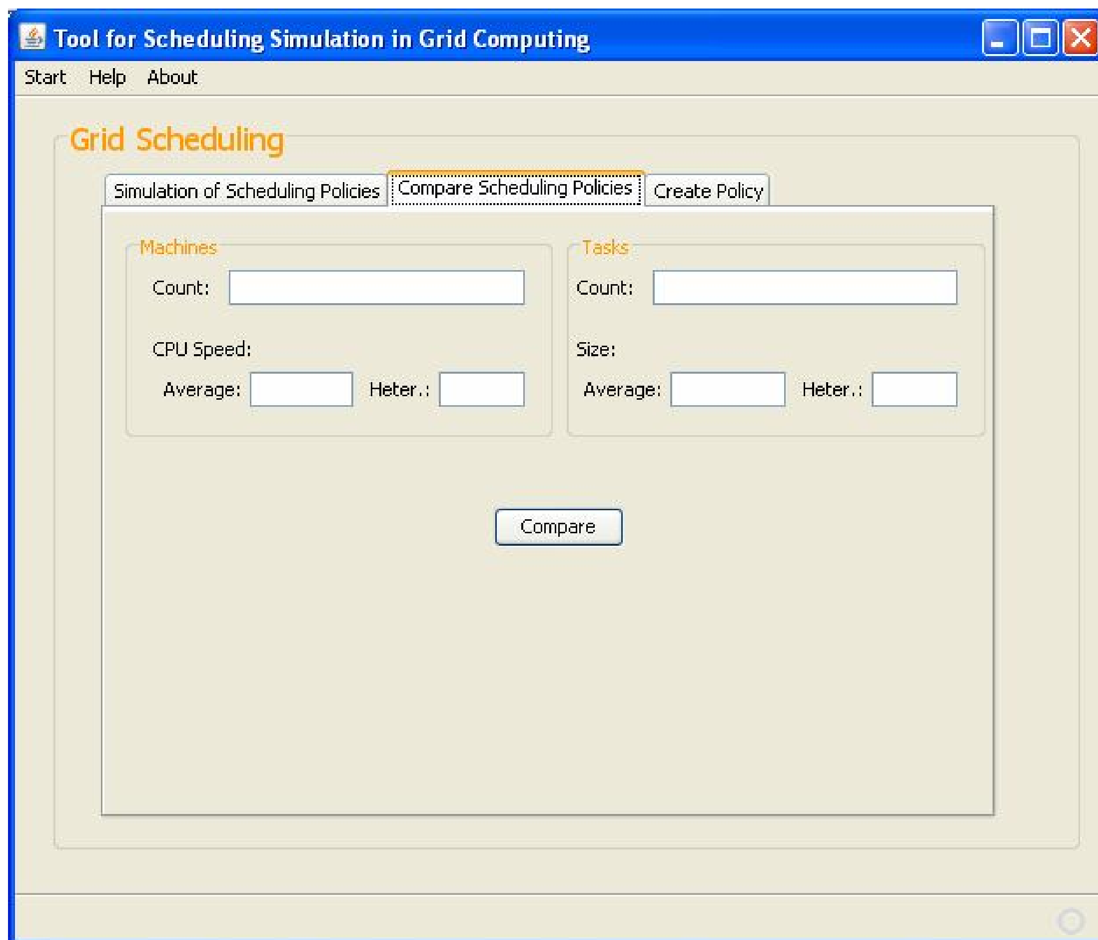

Figura 4.2: Tela de entrada de dados para comparação de políticas.

Para todos os cenários foram simuladas as políticas *Workqueue*, *Workqueue with Replication* com 2, 3 e 4 replicações e *Sufferage*.

Cenário 1: Recursos Heterogêneos.

Para tal cenário, considerou-se a diferença de 6 anos na idade dos recursos, portanto a heterogeneidade dos recursos pode assumir valores 1, 2, 4, 8 ou 16.

Foram feitas simulações para todos os valores de heterogeneidade, mantendo constante a quantidade e a média de tamanho das tarefas, neste caso foram submetidas 720 tarefas com média de tamanho igual a 5000 MI. Para todos os testes usou-se a mesma quantidade de máquinas, 100 no total.

Na figura 4.3 pode ser vista a simulação da política *Workqueue* com heterogeneidade das máquinas 4 e média de poder de processamento 10 MIPS.

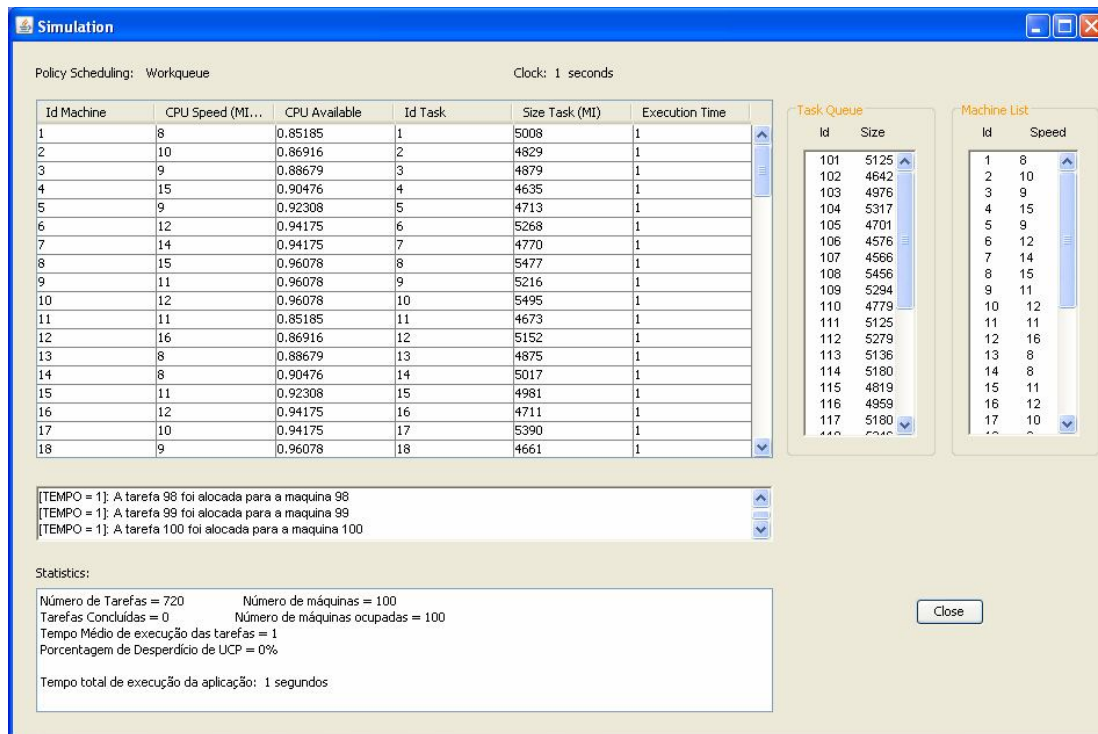

Figura 4.3: Tela de simulação.

Observou-se que com a variação da heterogeneidade dos recursos, as políticas WQR2x, WQR3x, WQR4x e *Sufferage* apresentaram comportamentos bastante estáveis, sendo que entre elas a WQR4x obteve a melhor média de tempo de execução das tarefas. Em contraste a esse comportamento estável, na política *Workqueue* observaram-se mudanças nos seus desempenhos proporcionais ao aumento da heterogeneidade dos recursos. A tabela 4.1 apresenta a média dos resultados obtidos para cada uma das políticas nos 5 níveis de heterogeneidade e a figura 4.4 ilustra os resultados no formato de um gráfico, sendo que cada ponto representa o tempo de execução da aplicação em seus respectivos níveis de heterogeneidade.

Tabela 4.1: Desempenho obtido por cada uma das políticas em 5 níveis de heterogeneidade de recursos.

Heterogeneidade	WQR4x	WQR3x	WQR2x	Sufferage	Workqueue
1	3201	3230	3260	4480	9520
2	3202	3232	3258	5277	12007
4	3207	3323	3440	5102	11120
8	3298	3501	3679	4008	12205
16	3606	5220	7005	3603	31518

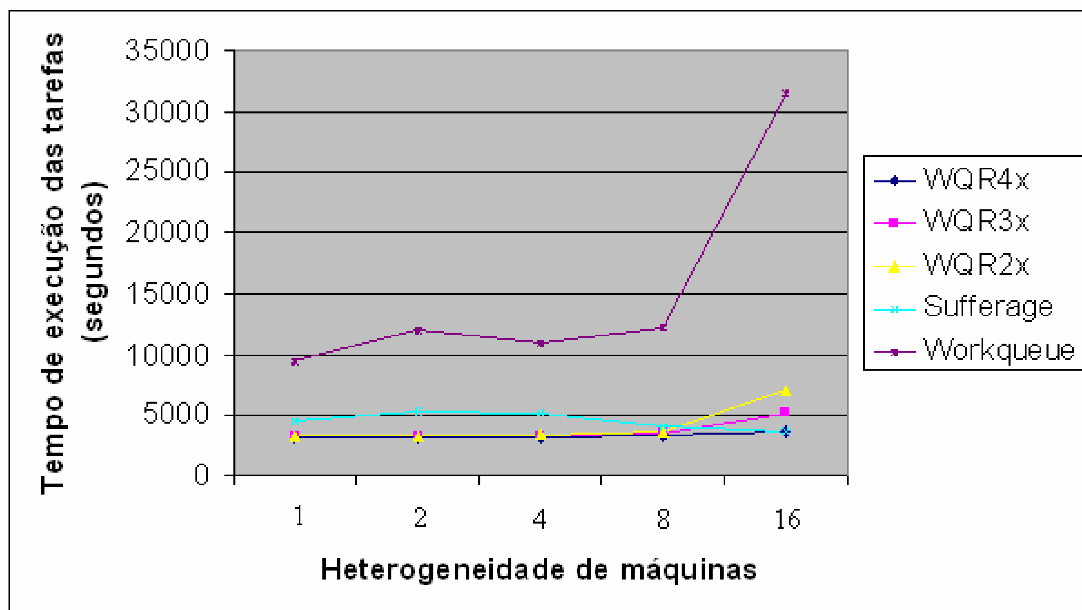

Figura 4.4: Desempenho das políticas em vários níveis de heterogeneidade de máquinas.

Cenário 2: Heterogeneidade de tarefas.

Neste cenário considerou-se o tamanho da aplicação fixo em 3600000 MI, variando apenas os tamanhos das tarefas pertencentes a essa aplicação, tal variação é conhecida como granularidade da aplicação. Os tamanhos médios considerados foram 1000, 5000, 25000, 125000 MI. A variação nos tamanhos das tarefas foi feita a propósito, a fim de verificar o comportamento das políticas com diferentes estados dos recursos, desde ociosos até saturados e não mostrar que um *grid* com recursos ociosos executa a aplicação de forma mais rápida. A aplicação com tarefas de

tamanho médio 1000 MI (grão pequeno) possui muito mais tarefas que a aplicação com tamanho médio 125000 MI (grão enorme).

A tabela 4.2 ilustra a relação das diferentes quantidades de tarefas contidas em uma aplicação com o número de máquinas por tarefa. Para este cenário considerou-se os mesmos recursos usados no cenário anterior, ou seja, 100 máquinas com poder de processamento médio igual a 10 MIPS.

Tabela 4.2: Granularidade das Aplicações

Tamanho Médio das Tarefas (MI)	Quantidade de Tarefas	Máquinas por Tarefa
1000	3600	36
5000	720	7,2
25000	144	1,44
125000	29	0,29

Foram feitas simulações para os diferentes valores de tamanhos das tarefas. O resultado da análise pode ser observado na figura 4.5.

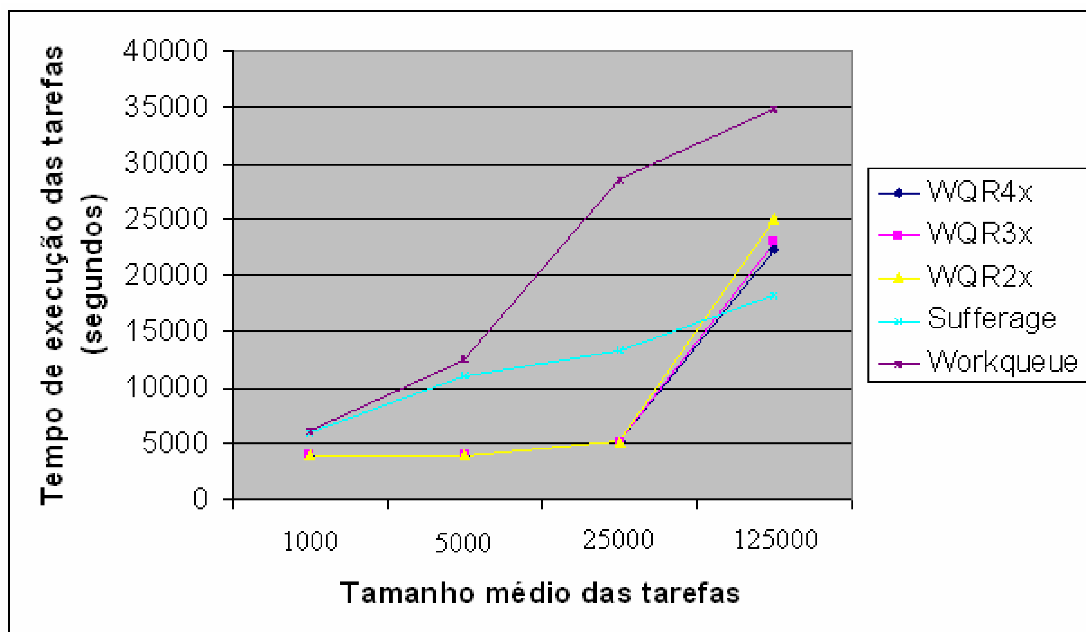

Figura 4.5: Desempenho das políticas com diferentes tamanhos médios das tarefas.

A grande diferença de desempenho das políticas, principalmente para tarefas com tamanho médio maior, mostra que a escolha adequada da política é muito importante. Observa-se que a granularidade das tarefas de uma aplicação é um fator determinante na escolha de qual política utilizar.

Os resultados de comparação foram obtidos através da interface de comparação disponibilizada na ferramenta. Na figura 4.6 é apresentada a tela de resultados da comparação.

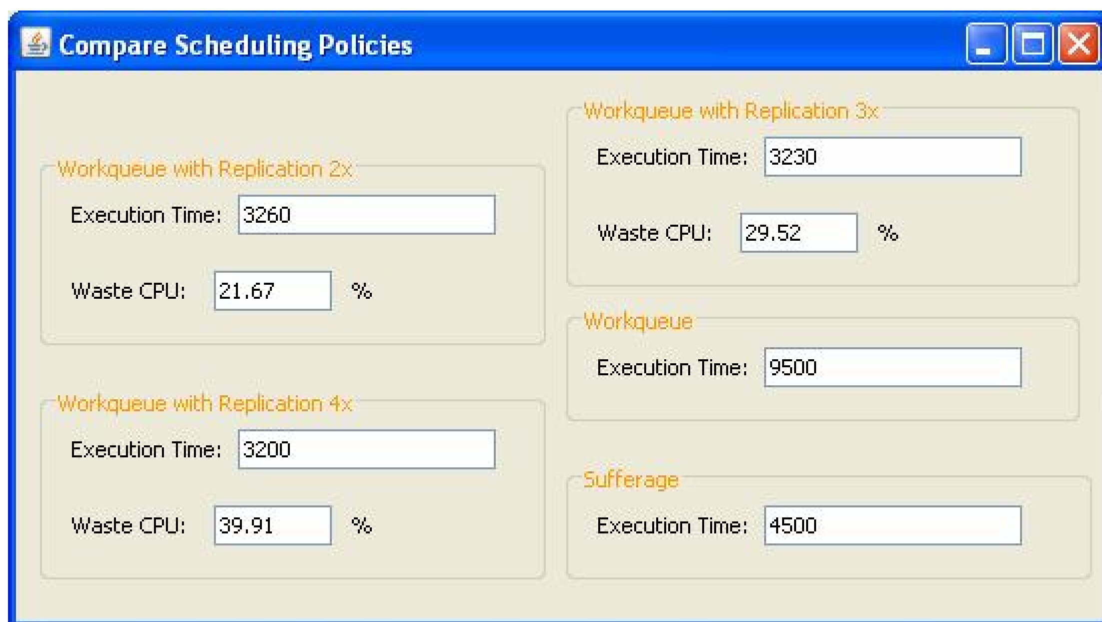

Figura 4.6: Tela de resultados da comparação.

Os resultados obtidos foram comparados com um trabalho relacionado [16] e mostraram-se bastante satisfatórios. Os valores de tempo de execução encontrados apresentam uma pequena diferença em relação ao trabalho utilizado como referência. Tal diferença pode ser explicada pela variação na carga dos recursos do ambiente simulado, variação no caso do presente trabalho feita através de arquivos de *traces* colhidos de uma universidade.

4.3 Compilador

Para o compilador foram feitos testes a fim de avaliar a eficiência dos analisadores implementados para a linguagem criada. Códigos com erros léxicos, sintáticos e semânticos foram inseridos em diferentes momentos, o resultado da compilação e os códigos inseridos podem ser visualizados nas figuras que se seguem. Na figura 4.7 pode ser visto um código com erros léxicos e sintáticos além de mensagens de erro para cada um dos erros.

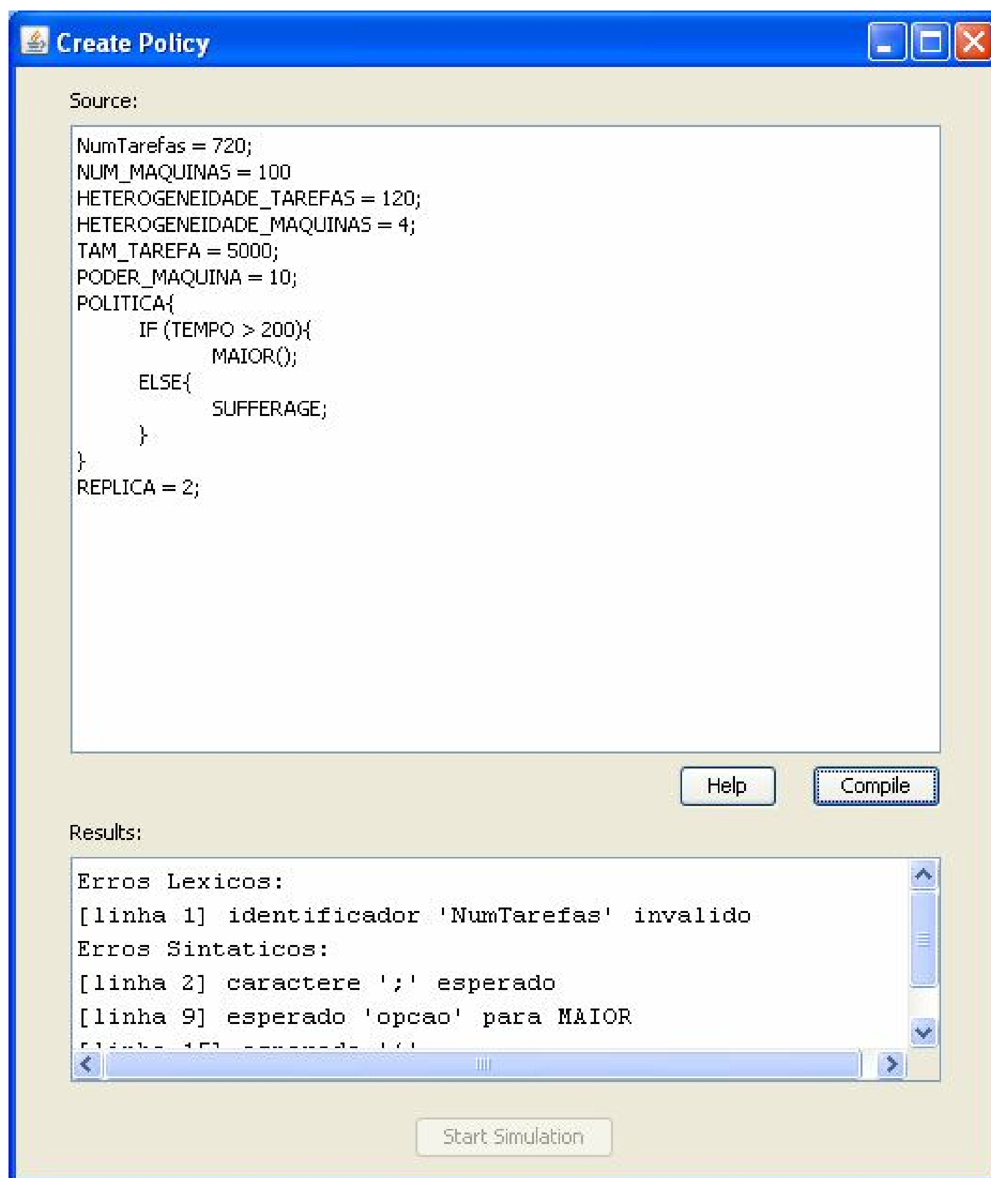

Figura 4.7: Tela do compilador.

Na figura 4.8 é ilustrado um código com erros semânticos e o resultado da compilação desse código, com as mensagens de erros é apresentado na figura 4.9.

```
HETEROGENEIDADE_TAREFAS = 120;
TAM_TAREFA = 5000;
PODER_MAQUINA = 10;
POLITICA{
	MAIOR(TAM);
	MAIOR(PODER);
}
REPLICA = TEMPO;
```

Figura 4.8: Código com erro semântico.

```
Results:
Erros Semanticos:
[linha 1] valor fora da faixa de valores aceitos para
essa variável.
[linha 8] tipos incompativeis na expressão.
2 erros encontrados.

Start Simulation
```

Figura 4.9: Mensagens de erros resultantes da compilação.

Percebe-se nas figuras 4.7 e 4.9 apresentadas anteriormente, que o botão para iniciar a simulação apresenta-se sempre desabilitado. Ele só será habilitado se a compilação não retornar nenhum erro, ou seja, após a inserção de um código sem erros.

Para finalizar os testes com o compilador, um código válido é inserido e pode-se observar na figura 4.10 o resultado obtido.

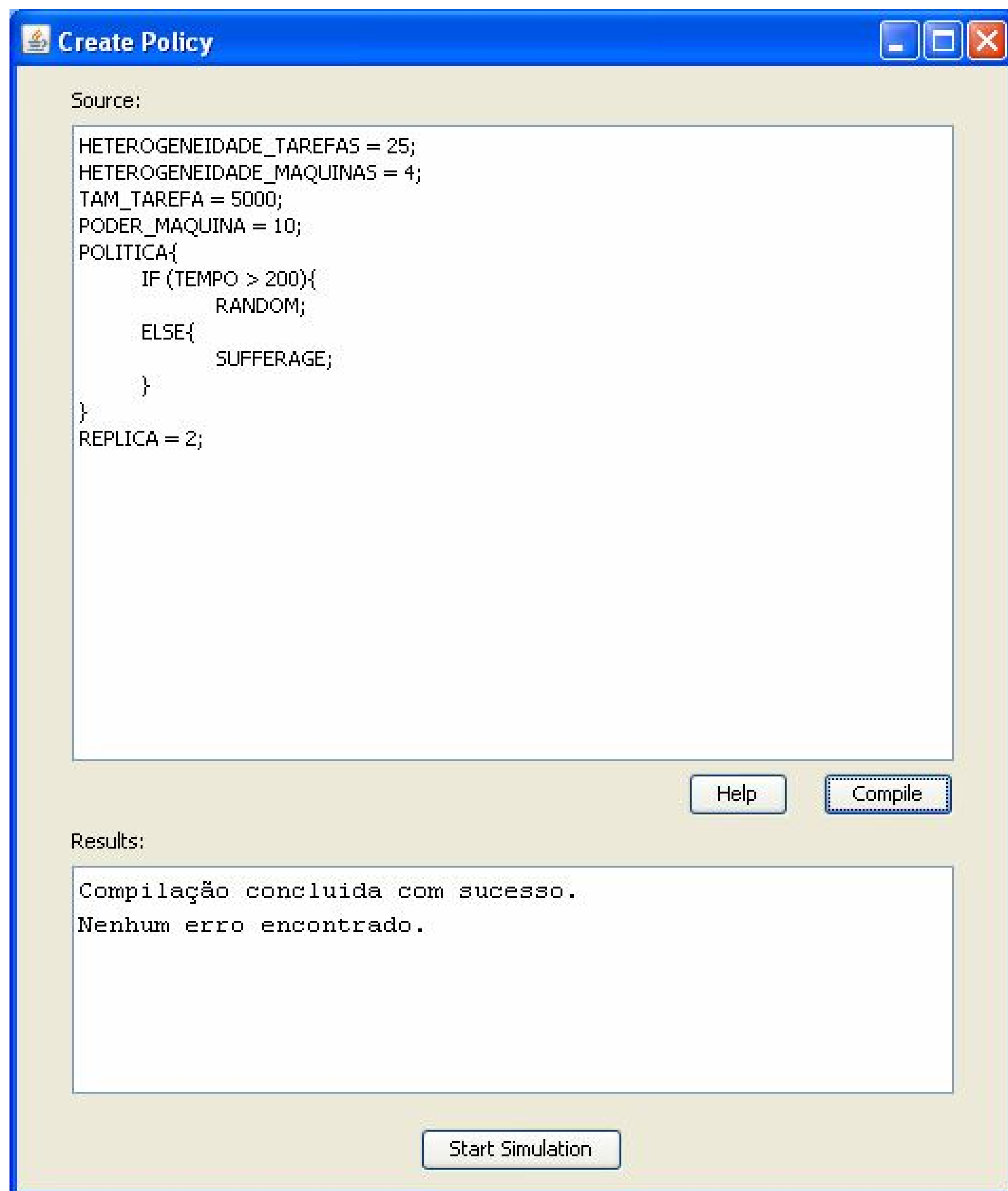

Figura 4.10: Resultado da compilação de um código aceito pela linguagem.

Nenhuma mensagem de erro foi exibida. Como o código é um código válido o botão para iniciar a simulação da política de escalonamento criada apresenta-se agora habilitado.

4.4 Considerações Finais

Neste capítulo foram apresentados os testes e validações feitos para a ferramenta implementada. Observou-se, através de simulações, que alguns fatores, como por exemplo, a granulosidade das tarefas de uma aplicação, são de fundamental importância para a escolha de um algoritmo de escalonamento adequado para cada situação.

Foi mostrada também a detecção de erros feita pelo compilador quando um código não aceito pela linguagem foi inserido. Em um outro exemplo foi possível visualizar o comportamento do compilador quando o código inserido era válido.

No capítulo 5 serão apresentadas as conclusões e algumas propostas de trabalhos futuros.

Capítulo 5: Conclusões

5.1 Conclusões

Grids mostram-se cada vez mais importantes para a realização de Computação Paralela e Distribuída. Aplicações submetidas a um *grid* devem ser escalonadas da melhor maneira possível a fim de se obter bons resultados. Por este motivo, a área de escalonamento em *grids* computacionais é hoje alvo de muitas pesquisas, aumentando o interesse de pessoas pelo assunto.

Com o aumento do número de interessados na área, a ferramenta desenvolvida se mostra útil, e poderá auxiliar novos estudantes no assunto a compreender melhor o funcionamento das políticas de escalonamento existentes.

Com o desenvolvimento deste trabalho a autora aprimorou seus conhecimentos na área de sistemas paralelos e distribuídos, estudando algo completamente novo para ela, a teoria de *Grids* Computacionais. Conseguiu, através da implementação da ferramenta compreender o funcionamento de algumas das principais políticas de escalonamento de tarefas existentes para *grids*.

5.2 Propostas de Trabalhos Futuros

Como continuidade do trabalho, propõe-se:

- A implementação de outras políticas de escalonamento existentes para *grids*. Isto possibilitará um aprofundamento maior no estudo de escalonamento.

- A inserção de novas funcionalidades, como por exemplo, a geração de gráficos para comparação de desempenho das políticas.

- O desenvolvimento de um interpretador para a linguagem, que possibilitará a visualização do funcionamento da política criada.

- A simulação da variação de carga dos recursos feita em tempo real, através do uso de *Webservices* que forneçam informações obtidas de um NWS (*Network Weather Service*).

- O desenvolvimento de uma interface mais visual, mostrando o relacionamento entre os recursos do *grid*.

Referências Bibliográficas

[1] FOSTER, I.; KESSELMAN, C. (1999) *The Grid: Blueprint for a New Computing Infrastructure*, Morgan-Kaufman.

[2] Site do Grupo de Sistemas Paralelos e Distribuídos, disponível em http://www.dcce.ibilce.unesp.br/spd/index.html.

[3] FALAVINHA JR., J.N.; MANACERO JR., A.; BOCCARDO, D.R.; OLIVEIRA, L.J. (2007). Avaliação de algoritmos de escalonamento em *Grids* para diferentes configurações de ambiente, em *Anais do Wperformance 2007*, Rio de Janeiro, CD-ROM, pp 505-524.

[4] *Sun Grid Engine* (2006), disponível em www.sun.com/software/gridware.

[5] CHTEPEN, M.; DHOEDT, B.; VANROLLEGHEM, P. (2005). *Dynamic scheduling in grid systems*, publicado no 6th FTW PHD Symposium, Gent, Belgium, pp. 95.

[6] ABBAS, A.; ABBAS, A. (2003). *Grid Computing: a Practical Guide to Technology and Applications*. Charles River Media, Inc.

[7] Página da *Sun Microsystems*. Disponível em www.sun.com.

[8] Página da IBM. Disponível em www.ibm.com.

[9] Página da *Plataform Computing*. Disponível em http://www.platform.com/

[10] *Virtual Private Network*, RFC 2764. Disponível em www.faqs.org/rfcs/rfc2764.html

[11] JAIN, H. (1991). *The Art of Computer Systems Performance Analysis: Tecniques for Experimental Design Measurement, Simulation, and Modeling*. Wiley.

[12] MAHESWARAN, M.; ALI, S.; SIEGEL, H. J.; HENSGEN, D. A.; FREUND, R. F. (1999). Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. *8th Heterogeneous Computing Workshop*, p.30-44.

[13] SILBERSCHATZ, A.; GALVIN, P.; GAGNE, G. (2001). *Sistemas Operacionais: Conceitos e Aplicações*. Campus.

[14] BERMAN, F. (1998). *The Grid: Blueprint for a New Computing Infrastructure*, chapter High-performance schedulers, p. 279-309. Morgan Kaufmann, San Francisco, USA

[15] CIRNE, W. (2002). Computational grids: Architectures, technologies and applications. *Terceiro Workshop em Sistemas Computacionais de Alto Desempenho*, Vitória, Brasil.

[16] SILVA, D. P. (2003). *Usando Replicação para Escalonar Tarefas Bag-of-Tasks em Grids Computacionais*. Dissertação (Mestrado), Universidade Federal de Campina Grande (UFCG), Campina Grande, Brasil.

[17] PARANHOS, D.; CIRNE, W.; BRASILEIRO, F. V.(2003). *Trading Cycles for Information: Using Replication to Schedule Bag-of-Tasks Applications on Computational Grids*.

[18] AHRNAD, I. (1995). *Resource Management in Parallel and Distributed Systems with Dynamic Scheduling*. Concurrency: Practice and Experience, v. 7, p. 587–590.

[19] CASANOVA, H.; LEGRAND, A.; ZAGORODNOV, D.; BERMAN, F. (2000). Heuristics for scheduling parameter sweep applications in grid environments. *9th Heterogeneous Computing Workshop (HCW)*, p. 349-363, Cancun, Mexico.

[20] AHO, A. V.; SETH, R.; ULLMAN, J. D. (1987). *Compilers: Principles, Techniques and Tools*, Addison - Wesley

[21] PITTMAN, T.; PETERS, J. (1992). *The Art of Compiler Design: Theory and Pratice*, Prentice Hall International.

[22] BAGRODIA, R.; MEYER, R.; TAKAI, M.; CHEN, Y.; ZENG, X.; MARTIN, J.; SONG, H. (1998). PARSEC: A Parallel Simulation Environment for Complex Systems. *IEEE Computer*, v.3, n.10, p.77-85.

[23] MENG, X. (1999). Distributed Simulations – Issues and Implementations in Clusters of Workstations Environment. *Computer Systems Science and Engineering*, v.14, n.1, p.27-57.

[24] BANKS, J. (1998). *Handbook of Simulation.* Georgia, Atlanta: John Wiley & Sons, Inc., p. 3-30.

[25] SANTANA, M.J. (1990). *An Advanced Filestore Architecture for a Multiple Lan Distributed Computing System*. Southampton, *Tese (Doutorado),* University of Southampton.

[26] MACDOUGALL, M.H. (1987). *Simulating Computer Systems Techniques and Tools*. The MIT Press.

[27] LEGRAND, A.; MARCHAL, L.; CASANOVA, H. (2003). *Scheduling distributed applications: The simgrid simulation framework*, in Proc. of the Third IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid'03), Tokyo, Japan.

[28] Página do Projeto SimGrid, disponível em http://simgrid.gforce.inria.fr

[29] BUYYA, R.;MURSHED, M. (2002). *GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing*, The Journal of Concurrency and Computation: Practice and Experience (CCPE), Volume 14, Issue 13-15, Wiley Press.

[30] Página do Projeto GridSim, disponível em http://www.gridbus.org/ raj/gridsim.

[31] DUMITRESCU, C. L.; FOSTER, I. (2002). Gangsim: a simulator for grid scheduling studies. In: *CCGRID '05: Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05)*. Washington, DC, Estados Unidos: IEEE Computer Society, v. 2, p. 1151. Disponível em: http://people.cs.uchicago.edu/cldumitr/docs/GangSim.pdf.

[32] FOSTER, I.; KESSELMAN, C.; TUECKE, S. (2001). The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *Lecture Notes in Computer Science*, v.2150.

[33] SIERRA, K.; BATES, B.; *Head First Java*, O'Reilly, 2ª edição, 2005.

[34] WOLSKI, R.; SPRING, N. T.; HAYES, J. (1999). The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Journal of Future Generation Computing Systems*, v.15, n5-6, p.757-768.

[35] Interface web, disponibilizada pela Universidade da Califórnia para acesso aos traces. Disponível em http://nws.cs.ucsb.edu/CGI/graphIt.cgi.

[36] Página da Universidade da Califórnia, Santa Bárbara. Disponível em http://www.cs.ucsb.edu/.

[37] SCHALLER, R. R. (1997). *Moore´s Law: Past, Present and Future*. IEEE Spectrum, v.34(6), p.52-59.

[38] REIS, V. Q. (2005). *Escalonamento em grids computacionais: estudo de caso*. Dissertação (Mestrado), ICMC-USP, São Carlos, Brasil.