# Implementação e Experimentação de Políticas de Escalonamento Baseadas na Aleatoriedade e nos Estados dos Recursos

**Irving M. Rodrigues[1], Renato P. Ishii[1], Antônio Tadeu A. Gomes[2], Valéria Q. dos Reis[1]**

[1]Faculdade de Computação – Universidade Federal do Mato Grosso do Sul
Caixa Postal 549 – Campo Grande – MS – Brasil

[2]Laboratório Nacional de Computação Científica
Av. Getúlio Vargas, 333 - Petrópolis, Brasil

irving_mr_@hotmail.com, renato@facom.ufms.br, atagomes@lncc.br, valeria@facom.ufms.br

***Abstract.*** *This work aims at implementing and evaluating metascheduling policies that employ little or no information about the resources in the infrastructure they act upon. We have conducted some experiments with such policies in a simulation setting that models the one found on the Brazilian SINAPAD network. As expected, these experiments have shown that policies which employ promptly available resource information—such as CPU load—lead to better scheduling decisions than randomly selecting resources. Among the evaluated policies, the one with better performance was based on the history of previous job executions and on the expected CPU usage. Such policy, on the other hand, was the one with the largest decision time. Even so, we have identified a trend in the time spent by this policy for the decision making phase to be mitigated by the gains in the overall execution time of jobs scheduled by this policy.*

***Resumo.*** *Este trabalho tem como objetivo a implementação e a avaliação de políticas de metaescalonamento que utilizam pouca ou nenhuma informação a respeito dos recursos na infraestrutura onde elas atuam. Para isso, foram realizados experimentos que simulam o sistema encontrado no SINAPAD, uma rede nacional de processamento de alto processamento disponível a comunidade científica nacional. Esses experimentos, como esperado, mostraram que políticas que utilizam informações facilmente obtidas, como a taxa de ocupação de CPU, levam a decisões de escalonamento melhores que a escolha aleatória de recursos. Dentre as políticas avaliadas, a política que obteve melhor desempenho foi baseada em um histórico de execuções prévias de tarefas e na estimativa de uso de CPU dos recursos. Tal política, por outro lado, foi uma das que necessitou de maior tempo de decisão. No entanto, o tempo gasto no momento do escalonamento tende a ser facilmente recuperado pelo ganho no tempo de execução que essa política é capaz de gerar.*

## 1. Introdução

Grades computacionais têm se destacado ao longo dos anos entre as infraestruturas utilizadas para o processamento de diversas aplicações científicas [Hey and Trefethen 2003, Emmott 2005]. Nessas infraestruturas, *middlewares* de gerenciamento escondem a natureza altamente dinâmica e heterogênea dos recursos e, com frequência, implementam

módulos de escalonamento responsáveis por decidir quando e em quais recursos uma dada aplicação deve executar [Frey et al. 2002, Foster 2005, Cirne et al. 2006].

Um bom escalonamento é capaz de aumentar o desempenho obtido pelas aplicações. Por outro lado, otimizar a atribuição de tarefas em uma grade é uma tarefa complexa e de difícil solução, dada a quantidade de variáveis envolvidas, incluindo o grande número de máquinas disponíveis e a diferença de capacidade entre elas, a variação da carga de trabalho nos recursos e o perfil de cada aplicação [Foster et al. 2001, Xhafa and Abraham 2010].

No Brasil, o Ministério da Ciência, Tecnologia e Inovação criou o Sistema Nacional de Processamento de Alto Desempenho, SINAPAD,[1] com a finalidade de prestar serviços a comunidade brasileira usuária de processamento de alto desempenho. O SINAPAD consiste em uma rede formada por centros de processamento distribuídos por todo o território nacional. Os centros que compõem essa rede (os CENAPADs) são acessíveis diretamente pelos usuários, por meio de terminal remoto seguro (ssh), ou utilizando portais Web.[2] Nesse último caso, para esconder do usuário a heterogeneidade de recursos providos pelos CENAPADs, um *software* de gerenciamento de recursos distribuídos é empregado. Atualmente, o módulo responsável pelo escalonamento das aplicações científicas submetidas ao SINAPAD nesse *software* escolhe os centros de computação com base na taxa de ocupação dos processadores dos nós executores presentes nos CENAPADs. No entanto, outras políticas de escalonamento podem ter desempenho melhor em diversos cenários (por exemplo, no caso de aplicações intensivas de E/S). Dessa maneira, este trabalho tem como objetivo a construção e comparação de um conjunto de políticas de escalonamento direcionadas ao SINAPAD e sistemas similares. Neste trabalho são empregadas políticas que utilizam pouca informação a respeito dos recursos, o que permite uma rápida tomada de decisão. Apesar da simplicidade dessas políticas, demonstramos que com elas conseguimos reduzir de forma significativa o tempo de execução das aplicações dos usuários e também aumentar o nível de ocupação desses sistemas.

O restante deste artigo está dividido da seguinte maneira: a Seção 2 apresenta a arquitetura e o funcionamento dos portais Web do SINAPAD e do software de gerenciamento de recursos distribuídos usado pelos mesmos; a Seção 3 descreve o funcionamento das novas políticas implementadas; a Seção 4 explica como foram conduzidos os experimentos e descreve os resultados obtidos; por fim, a Seção 5 apresenta as conclusões e os trabalhos futuros.

## 2. Portais Web no SINAPAD

Para esconder do usuário dos portais Web do SINAPAD a heterogeneidade de recursos providos pelos CENAPADs, o SINAPAD emprega um software de gerenciamento de recursos distribuídos que adota um estilo arquitetural orientado a serviços (*Service-Oriented Architecture* – SOA). Na arquitetura desse software, ilustrada na Figura 1, um componente servidor denominado OpenBus funciona como um barramento virtual, centralizando o registro de todos os demais componentes do sistema e respondendo a consultas dos portais Web ou de outros componentes do sistema. Cada CENAPAD oferece acesso aos seus recursos através de um componente OpenDreams, o qual faz parte de um gerenciador de

[1] http://www.lncc.br/sinapad
[2] http://www.lncc.br/sinapad/portais.php

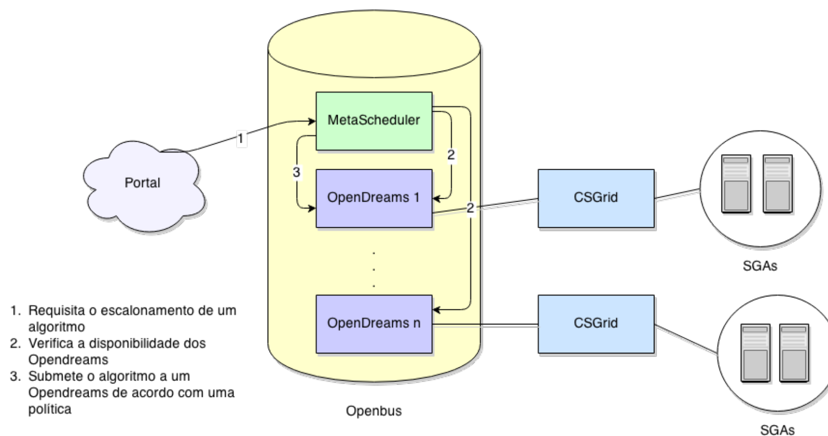


**Figura 1. Metaescalonamento no SINAPAD.**

recursos local denominado CSGrid [Lima et al. 2005]. Esse gerenciador regula o funcionamento dos nós executores de processos de seu CENAPAD por meio de *daemons* denominados SGA e instalados em cada nó do CENAPAD. Por fim, a arquitetura apresenta um componente MetaScheduler, responsável pela distribuição de tarefas, através dos componentes OpenDreams, entre os gerenciadores CSGrid de cada CENAPAD.

O metaescalonamento do SINAPAD está implementado da seguinte maneira: 1) um portal do SINAPAD invoca o método *selectOpenDreamsService* de um componente Metascheduler passando o nome e a versão do arquivo executável (*algoritmo* na nomenclatura adotada pelo gerenciador CSGrid) e os critérios para seleção dos OpenDreams; 2) o metaescalonador consulta os OpenDreams disponíveis no barramento e que estão de acordo com os critérios escolhidos; 3) os Opendreans são ordenados de acordo com a política de escalonamento ativa e 4) o melhor OpenDreams é escolhido para receber a submissão. É nessa última fase do escalonamento que as políticas investigadas neste trabalho estão inseridas.

## 3. Políticas de escalonamento

Políticas de escalonamento devem produzir bons resultados de acordo com os critérios que as definem. No caso do SINAPAD, o critério utilizado é minimizar o tempo de execução das aplicações. Além disso, essas políticas devem ser de fácil implementação e ter um tempo de decisão relativamente baixo a fim de evitar a sobrecarga do metaescalonador. Nesse aspecto, políticas que usam pouca ou nenhuma informação a respeito dos recursos e das aplicações a serem executadas se destacam. No SINAPAD, existem duas políticas atualmente implementadas com esse perfil: a *RandomScheduler* e a *CPULoadScheduler*. A primeira seleciona aleatoriamente os componentes OpenDreams para executar uma aplicação. A ideia principal dessa política assume que a seleção é tão aleatória que os escalonamentos bons e ruins irão se sobrepor um ao outro, e assim, as decisões de seleção serão aceitáveis. A segunda política submete uma aplicação ao componente OpenDreams que anuncia a menor carga de trabalho por núcleo de processamento em seus nós.

Políticas muitos simples tendem a realizar escalonamentos piores do que os obtidos por políticas que consideram as informações dos *jobs* e dos recursos. Todavia, estimar

o tempo de execução de cada aplicação, assim como propõem as políticas MCT, MET, Sufferage, Min-Min e Dynamic FPLTF [Maheswaran et al. 1999, Silva et al. 2003, Izakian et al. 2009, Xhafa and Abraham 2010] é uma tarefa difícil e custosa. A replicação de execuções, assim como proposto pela política WQR [Cirne et al. 2007b], é uma alternativa para a estimativa do tempo de execução das aplicações, mas acarreta na desvantagem de aumentar a utilização de recursos, o que poderia impactar negativamente no uso da infraestrutura SINAPAD.

Dessa maneira, neste trabalho, foram implementadas três novas políticas que utilizam somente as informações dos recursos computacionais durante o escalonamento: a *LotteryScheduler*, a *LotterySchedulerWithLoad* e a *WallTimeEstimatedScheduler*. Tais políticas realizam escalonamento imediato visando o alto desempenho de aplicações intensivas de CPU formadas por um único *job*. A escolha desse tipo de aplicação alvo é justificada pela sua simplicidade e significativa incidência nos problemas de cunho científico.

### 3.1. LotteryScheduler

A *LotteryScheduler* é uma política que utiliza aleatoriedade no escalonamento, porém, diferentemente da *RandomScheduler*, as chances de um componente OpenDreams ser escolhido para executar uma aplicação $j$ pode ser distinta dos demais. Isso porque ao receber um pedido de execução de uma aplicação $j$, essa política fornece uma certa quantidade de bilhetes a cada componente OpenDreams e escolhe aleatoriamente um desses bilhetes. Depois disso, submete a aplicação $j$ ao recurso que possuir o bilhete sorteado. O número de bilhetes, $nb_i$, dados a um componente OpenDreams $i$ em um escalonamento é igual ao valor da divisão do maior tempo de CPU médio divulgado no conjunto de componentes OpenDreams pelo tempo de CPU médio divulgado pelo componente OpenDreams $i$. Esse cálculo é apresentado na Equação 1.

$$nb_i = \begin{cases} \parallel (\frac{\max_{tcm_{ij} \in TCM}}{tcm_{ij}}) \parallel & \text{Se } tcm_{ij} > 0 \\ \infty & \text{Se } tcm_{ij} = 0 \text{ ou } tcm_{ij} \text{ for indefinido} \end{cases} \quad (1)$$

Em (1), $tcm_{ij}$ é o tempo de CPU médio de uma aplicação $j$, que é escalonada em um componente OpenDreams $i$ e $TCM$ é o conjunto de todos os tempos de CPU médios de uma aplicação divulgados por cada componente OpenDreams. Segundo a Equação 1, o componente OpenDreams que tiver o divulgado o maior tempo de CPU receberá somente um bilhete, enquanto os demais receberão um ou mais bilhetes. Para os componentes OpenDreams que não executaram nenhuma vez a aplicação $j$, é dado a eles um número muito alto de bilhetes, com o intuito de informar a política sobre esses recursos.

O tempo de CPU funciona como uma métrica de comparação da capacidade computacional dos recursos. Se um recurso pode executar uma aplicação com um tempo de CPU menor que outro, então assume-se que o primeiro possui uma maior capacidade de processamento do que o segundo. Para medir a capacidade computacional ofertada por um OpenDreams, a *LotteryScheduler* utiliza uma média dos tempos de CPU observados até o momento. O cálculo da média, $tcm_{ij}$, é obtido da seguinte forma:

$$tcm_{ij} = 0.75 * tcm_{ij} + 0.25 * tn_{ij} \quad (2)$$

onde $tn_{ij}$ é o novo tempo de CPU de uma aplicação $j$ em um componente OpenDreams $i$. Como pode ser observado na Equação 2, os tempos antigos têm maior peso do que os novos, pois assim, variações temporárias não modificam demasiadamente o tempo médio de CPU.

A *LotteryScheduler* é uma política que, ao contrário da *RandomScheduler*, leva em conta a heterogeneidade de recursos, pois quanto mais potente for um recurso em relação aos demais, maior será a sua probabilidade de executar um *job*. Assim, os *jobs* têm uma maior probabilidade de serem submetidos aos recursos com maior capacidade computacional e, por consequência, serem executados mais rapidamente. Todavia, essa heurística ignora um importante elemento que está ligado intimamente ao tempo de execução: a carga de trabalho. Pode não ser interessante enviar um *job* a um recurso muito mais rápido do que outro se o primeiro apresentar uma carga de trabalho muito maior do que o segundo, visto que a aplicação submetida ao segundo recurso terá que concorrer com outros processos pelo uso do processador.

### 3.2. LotterySchedulerWithLoad

A *LotterySchedulerWithLoad* utiliza o mesmo mecanismo de bilhetes que a *LotteryScheduler*. A diferença está no modo de calcular o número de bilhetes dado a um OpenDreams $i$ em um escalonamento de uma aplicação $j$. Esse cálculo é realizado da seguinte forma:

$$nb_i = \begin{cases} \| (\frac{\max_{tee_{ij} \in TEE}}{tee_{ij}}) \| & \text{Se } tee_{ij} > 0, \\ \infty & \text{Se } tee_{ij} = 0 \text{ ou } tee_{ij} \text{ for indefinido} \end{cases} \quad (3)$$

onde $tee_{ij}$ é o tempo de execução estimado de uma aplicação $j$ escalonada em um componente OpenDreams $i$ e $TEE$ é o conjunto de todos os tempos de execução estimados de uma aplicação $j$ em cada componente OpenDreams. O tempo de execução estimado representa a quantidade de tempo que a aplicação demora para terminar sua execução de acordo com seu tempo de CPU médio no recurso e a carga de trabalho atual desse recurso. Esse tempo é definido pela Equação 4.

$$tee_{ji} = \begin{cases} tcm_{ij} * (cen_i) & \text{Se } cen_i \geq 1, \\ tcm_{ij} & \text{Se } cen_i < 1 \end{cases} \quad (4)$$

onde $tcm_{ij}$ é o tempo de CPU médio de uma aplicação $j$ em um componente OpenDreams $i$ e $cen_i$ é a carga de trabalho estimada por núcleo de processamento em um componente OpenDreams $i$. Quando a carga de trabalho estimada por núcleo é igual ou menor a 1, isso quer dizer que não existe concorrência das aplicações pelo processador, e por isso, o tempo de execução da aplicação será igual ao tempo de CPU médio. Porém, o contrário mostra um cenário de concorrência que afeta o tempo de execução.

Supondo que o sistema operacional distribui de forma uniforme o uso de um núcleo de processamento, e sabendo que o valor da carga de trabalho estimada por núcleo representa o número de processos que estão usando o núcleo ou estão na fila para isso, então, nesse caso, o uso do núcleo por uma aplicação será igual a $\frac{1}{cen_i}$. Como, no cenário de aplicações intensivas de CPU (foco deste trabalho), o uso do núcleo de processamento

é inversamente proporcional ao tempo de execução, uma aplicação que estiver usando $\frac{1}{cen_i}$ do núcleo terá o tempo de execução estimado em $tcm_{ji} * cen_i$.

A carga de trabalho estimada por núcleo de processamento, $cen_i$, assim como apresenta a equação 5, é calculada pela adição de uma unidade à carga de trabalho estimada, $ce_i$, e pela divisão desse montante pelo número de núcleos de processamento, $np_i$[3], que existem em um componente OpenDreams $i$. É acrescentada uma unidade à carga de trabalho estimada pois sendo a aplicação escalonada intensiva em CPU, ela tentará usar o núcleo de processamento constantemente, aumentando em uma unidade o número de processos disputando por recursos da máquina.

$$cen_i = \frac{ce_i + 1}{np_i} \tag{5}$$

Os valores das cargas de trabalho coletadas nos *daemons* SGAs e apresentadas ao seu componente OpenDreams correspondente são referentes à carga de trabalho média (*jobs* que estão na fila de execução ou executando no processador) no período de um minuto. Assim, quando uma aplicação for submetida a um componente OpenDreams, a sua execução só será refletida por completo na carga de trabalho depois de um minuto após a submissão da aplicação. Supondo que a reflexão seja uniforme, pode-se dizer que a carga de CPU sobe $\frac{1}{60000}$ em 1 milissegundo quando uma aplicação intensiva em CPU é executada. O mesmo raciocínio é válido quando ocorre o término da aplicação, porém, ao invés de adicionar, há a diminuição da carga. Portanto, as cargas coletadas não irão representar valores condizentes com a carga real de um componente OpenDreams, e por isso, é realizada a estimativa da carga de trabalho utilizando a seguinte fórmula:

$$ce_i = \sum_{s \in S_i} c_s + \sum_{ae \in Ae_i} exec_ratio(ae, ta) - \sum_{at \in At_i} fin_ratio(at, ta), \tag{6}$$

onde $S_i$ é o conjunto de todos os *daemons* SGAs de um componente OpenDreams $i$, $Ae_i$ é o conjunto de todas as aplicações $ae$ que estão sendo executadas em um componente OpenDreams $i$, $At_i$ é o conjunto de aplicações $at$ que terminaram de executar em um componente OpenDreams $i$, $ta$ é o valor do tempo atual e $c_s$ é o valor da carga de trabalho média coletada em um *daemon* SGA $s$. A função $exec_ratio$ retorna o valor que falta ser incluído na carga de trabalho média para que essa carga represente a carga de trabalho real. Quando uma aplicação intensiva em CPU está sendo executada a um tempo menor que 1 minuto, então a sua execução não é incluída por completo no valor da carga de trabalho média. Para que esse valor represente a carga real, é necessário que seja acrescentado à carga de trabalho média o produto da multiplicação de $\frac{1}{60000}$ pelos milissegundos que faltam para que o tempo de execução seja igual a 1 minuto. Todavia, se o tempo de execução for maior ou igual a 60 segundos, então a execução da aplicação como um todo já está sendo representada na carga, e logo, não é necessária qualquer adição. Assim, a função $exec_ratio$ é representada da seguinte maneira:

$$exec_ratio(ae, ta) = \begin{cases} \frac{60000-(ta-ti_j)}{60000} & \text{Se } ta - ti_j \leq 60000, \\ 0 & \text{Se } ta - ti_j > 60000 \end{cases} \tag{7}$$

[3]Somatório do número de núcleos informados por todos os *daemons* SGAs geridos por um componente OpenDreams.

onde $ti_j$ consiste no tempo em que a aplicação $j$ começou a ser executada. Processo inversamente similar ocorre quando uma aplicação é finalizada, pois no instante que a aplicação termina, o valor da carga média inclui a aplicação finalizada e, por isso, não corresponde à carga real. A função $fin_ratio$ é a responsável pelo cálculo do valor a ser removido da carga de trabalho informada pelo componente OpenDreams.

A *LotterySchedulerWithLoad* considera a heterogeneidade e a carga de trabalho estimada dos recursos. A carga de trabalho não constitui uma representação exata da carga atual do sistema, pois, como dito anteriormente, ela é estimada. Além disso, o componente OpenDreams atualiza somente os valores da carga de trabalho coletados por cada *daemon* SGA em intervalos de tempo pré-definidos. Mesmo assim, essa carga sugere o que está ocorrendo em cada CENAPAD. De maneira geral, a maior desvantagem da *LotterySchedulerWithLoad* está no fato de que ela permite, através de sorteios, que aplicações sejam submetidas a recursos que irão executá-las de forma muito mais lenta do que se as mesmas fossem enviadas a outros recursos escolhidos por políticas mais cautelosas.

### 3.3. WallTimeEstimatedScheduler

A política *WallTimeEstimatedScheduler* prioriza a utilização do componente OpenDreams com menor tempo de execução estimado pela Equação 4. Em suma, essa política é uma versão da *LotterySchedulerWithLoad*, porém sem a aleatoriedade do sorteio de bilhetes.

## 4. Experimentos

A fase de experimentação foi realizada através da simulação de um ambiente heterogêneo que pudesse, de alguma forma, se aproximar ao encontrado no SINAPAD. O ambiente de testes foi composto por 6 máquinas, sendo cada uma com duas *CPUs* de 64 bits do modelo *AMD Opteron(tm) Processor 246* com 2 GHz, 8 GB de memória RAM e 1 TB de disco rígido. Um dos computadores foi escolhido para ser o *front-end* e nele foram instalados os componentes OpenBus e MetaScheduler. Como as grades são ambientes heterogêneos e o ambiente de testes tenta simular um ambiente de grade, então era necessário que fosse possível controlar a heterogeneidade dos recursos. A solução encontrada foi utilizar máquinas virtuais usando o Oracle VM VirtualBox juntamente como o programa CpuLimit[4], o qual limita a percentagem de uso do processador que uma determinada máquina virtual pode ter.

Com o uso de máquinas virtuais, também foi possível aumentar o número de recursos, possibilitando a criação de mais alguns cenários de testes. Para cada máquina, com exceção do *front-end*, foram criadas duas máquinas virtuais com as seguintes configurações: 1 processador virtual de 32 bits com 1,9 GHz, 2 GB de memória RAM e 200 GB de disco rígido. Em cada máquina virtual foram instalados um gerenciador CSGrid com seu componente OpenDreams e um *daemon* SGA associado. Todas as máquinas virtuais e físicas possuíam instalados o sistema operacional Debian 6 e foram interligadas por uma rede *Ethernet* de um 1 Gb. A configuração do ambiente do testes é ilustrada na Figura 2.

Nos testes existiam quatro tipos de recursos quanto à capacidade computacional: r1, r2, r3 e r4. Os recursos r1 eram duas, quatro e oito vezes mais potentes do que r2, r3 e

[4]http://cpulimit.sourceforge.net

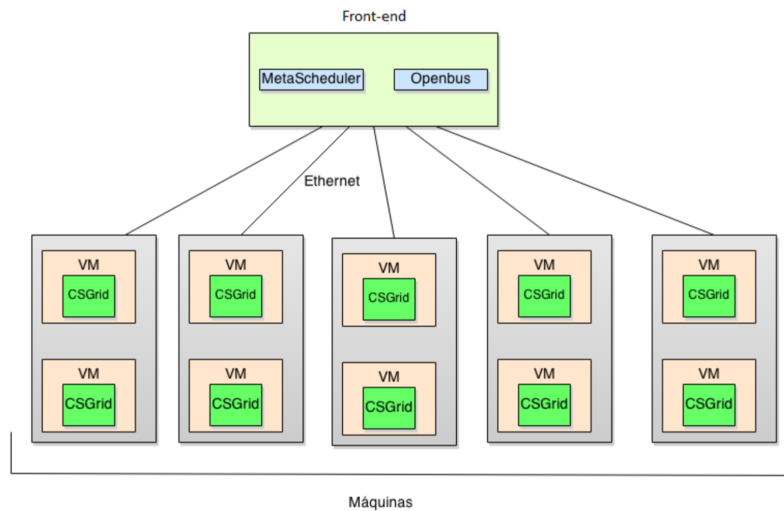


**Figura 2. Arquitetura do ambiente de testes.**

r4, respectivamente. Da mesma maneira que no trabalho feito por Cirne et al. 2007a, essas diferenças entre os recursos foram baseados na lei de Moore, que diz que a capacidade de processamento dos computadores dobra a cada 18 meses [Schaller 1997]. Com essas variações de recurso foram criados os cenários da Tabela 4.

| Cenário | Heterogeneidade recursos | Composição |
|---|---|---|
| RH | Homogênea | 10 recursos do tipo r1 |
| RBU | Baixa | 5 recursos do tipo r1 e r2 |
| RBD | Baixa | 2 recursos do tipo r1 e 8 recursos do tipo r2 |
| RBE | Baixa | 8 recursos do tipo r1 e 2 recursos do tipo r2 |
| RAU | Alta | 2 recursos para os quatros tipos de recursos |
| RAB | Alta | 4 recursos do tipo r2 e r3 e 2 recursos do tipo r4 e r1 |
| RAD | Alta | 2 recursos do tipo r1 e 8 recursos do tipo r4 |
| RBE | Alta | 8 recursos do tipo r1 e 2 recursos do tipo r4 |

**Tabela 1. Cenários simulados.**

A aplicação utilizada nos testes foi uma versão em C do *benchmark* Linpack [Menninger 1993], com uma alteração para tornar o número de repetições do código constante (5.000 repetições). Esse *benchmark* apresenta tempo de CPU próximo a 60 segundos em recursos do tipo r1, com variação de 10%.

Os testes foram divididos de acordo com o modo como as aplicações eram submetidas para serem escalonadas. Um desses modos enviava todos os *jobs* simultaneamente, enquanto o outro submetia um *job* a cada 20 segundos, ou seja, de forma intercalada. O primeiro tipo serve para simular uma grade com grande fluxo de pedidos em um curto espaço de tempo, ao passo que o segundo simula uma grade em que não existem tantas requisições durante um dia. Esse último tem um comportamento mais parecido com do SINAPAD.

Para cada modo de envio e tipo de recursos foram submetidos 40 pedidos de execução de processos para o metaescalonador. Cada cenário foi simulado 5 vezes a fim de se obter uma média de tempo de execução mais precisa. Nos testes que usavam o modo de envio simultâneo das aplicações notou-se um problema: os componentes OpenDreams

que recebiam os primeiros *jobs* das simulações demoravam mais tempo para responder a consultas realizadas posteriormente pelo metaescalonador. Dessa maneira, políticas que consumiam informações a respeito dos recursos dos componentes OpenDreams atribuíam *jobs* aos recursos mais lentamente do que outras. Quanto maior o atraso para um *job* ser enviado a um OpenDreams, menor a concorrência sofrida pelos processos já em execução nesse componente e, consequentemente, menor o tempo de execução desses *jobs*. Tal característica tornaria injusta a comparação de heurísticas com diferentes tempos de envio de *jobs*.

Para comprovar essa suposição, foram realizados dois novos tipos de testes. No primeiro deles, um mecanismo que aumenta o tempo de decisão das políticas mais rápidas foi adicionado. Dessa maneira, os tempos de decisão foram uniformizados, permitindo que a medição dos tempos de execução gerados pelas políticas pudesse ser feita de maneira mais precisa. No segundo tipo de testes, esse mecanismo não foi ativado. Ambos os tipos de testes foram conduzidos nos cenários RAU e RH e utilizando as políticas *RandomScheduler* e *LotterySchedulerWithLoad*, pois elas representam as políticas que enviam tarefas, respectivamente, mais rapidamente e mais lentamente na infraestrutura de simulação. Os resultados obtidos confirmaram que há uma relação entre o aumento do tempo de envio de aplicações e a diminuição no tempo de execução das mesmas. Portanto, políticas que demoram mais para escalonar têm vantagem sobre as que demoram menos. Por isso, foi fundamental que os testes ocorressem com os mesmos tempos de envio. Nos testes com envio intercalado esse problema não foi encontrado.

As Figuras 3 e 4 apresentam os resultados dos testes e a comparação das políticas em cada cenário utilizando como métrica a média do tempo de execução. Na Figura 3 estão relacionados os testes em que as aplicações foram enviadas simultaneamente, enquanto na Figura 4, as aplicações foram enviadas a cada 20 segundos.

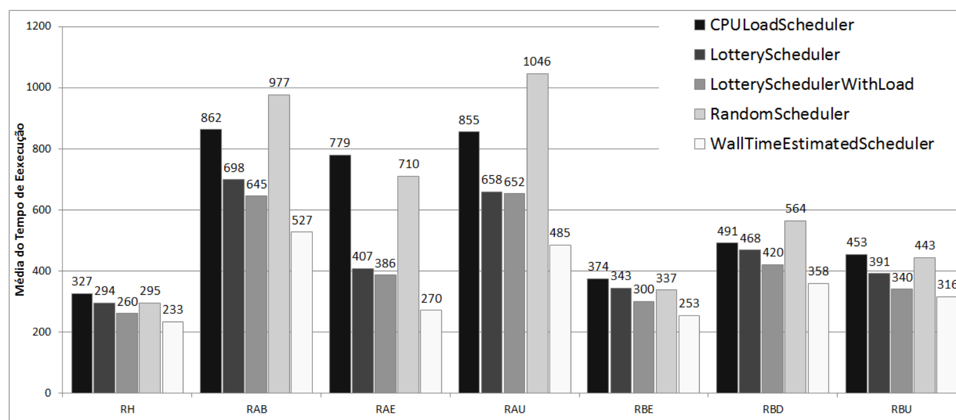


**Figura 3. Comparação das políticas nos testes feitos utilizando modo de envio simultâneo.**

Ao analisar os gráficos, a *RandomScheduler* é a política que teve o pior desempenho no geral. Nos cenários homogêneos, essa política não tem um desempenho tão ruim quanto as demais, porém, com o aumento da heterogeneidade há uma queda em seu rendimento. Como a *RandomScheduler* tenta distribuir a carga de forma balanceada, não importando a capacidade computacional dos recursos, todos os componentes OpenDreams recebem quantidades proporcionais de aplicações. Assim, quanto maior a diferença entre

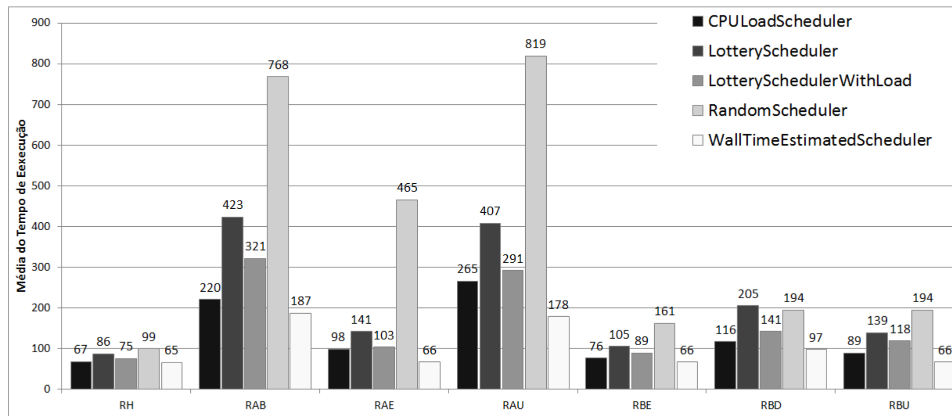


**Figura 4. Comparação das políticas nos testes feitos utilizando modo de envio intercalado.**

os recursos, maior será a penalização por enviar uma aplicação a um recurso mais lento, e por conseguinte, menor será o seu desempenho em relação às outras políticas.

A *LotteryScheduler* tem um comportamento similar a *RandomScheduler* quando os recursos são homogêneos, mas com o aumento da heterogeneidade, o seu desempenho acaba sendo geralmente melhor que o da *RandomScheduler*, pois a primeira política atribui as probabilidades de um *job* de acordo com a sua capacidade computacional em relação aos restantes. Dessa maneira, os *jobs* tendem a ser submetidos a componentes OpenDreams com maior capacidade. No entanto, a simples ação de permitir probabilisticamente que uma escolha ruim aconteça faz com que a *LotteryScheduler* ignore as direções que tendem a levar a um escalonamento bom e tome um sentido desfavorável, comprometendo o seu desempenho. A mesma fraqueza acomete a *LotterySchedulerWithLoad*. Entretanto, essa política apresenta desempenho melhor do que a *LotteryScheduler*, pois além de usar a capacidade computacional dos recursos, usa as cargas de trabalho desses recursos para atribuir as probabilidades de cada componente OpenDreams de receber uma determinada aplicação. Deve-se ressaltar que essa diferença de desempenho entre as duas políticas teve um grande contraste nos testes com envio intercalado nos cenários RAU e RAB.

A *CPULoadScheduler* teve desempenho melhor que a *RandomScheduler* somente nos testes em que os *jobs* eram enviados simultaneamente. Isso se deve ao fato de que as cargas dos *daemons* SGAs, por configuração, são atualizadas em intervalos de 15 segundos. Consequentemente, a *CPULoadScheduler* não identifica que a carga de um recurso está aumentando durante esse tempo e, então, segue enviando novas aplicações ao componente OpenDreams visto como o melhor da infraestrutura naquele intervalo. Porém, essa política tem o segundo melhor desempenho nos testes em que as aplicações eram submetidas intercaladamente. Esse resultado acontece devido aos tamanhos dos intervalos de envio de cada *job*, que são maiores que 15 segundos. Fazer com que os valores das cargas de trabalho dos recursos estejam sempre atualizados no momento do escalonamento permite que a política consiga balancear a carga de forma efetiva. Assim, a *CPULoadScheduler* envia novos trabalhos para o recurso que estiver mais ocioso.

A *WallTimeEstimatedScheduler* foi a política que teve o melhor desempenho nos testes em geral. Naqueles experimentos que tiveram os *jobs* enviados simultaneamente, a

política que chegou mais perto da *WallTimeEstimatedScheduler*, em termos de desempenho, foi a *LotterySchedulerWithLoad*. Porém, essa distinção de desempenho entre essas duas políticas cresce com o aumento da heterogeneidade.

Além da *WallTimeEstimatedScheduler* ser a melhor política em relação a média do tempo de execução, essa política conseguiu ter o menor desvio padrão dos tempos de execução das aplicações em todos os cenários.

Um outro ponto importante na comparação das políticas é o tempo de decisão de uma determinada política. Quando esse tempo é muito grande, há uma sobrecarga no metaescalonador, o que deve ser evitado. Para comparar as políticas quanto a isso, foi utilizada a média de três amostras do tempo de decisão de cada política no cenário RH. A Tabela 2 mostra o tempo médio de decisão das políticas.

| Política | Tempo Médio de Decisão (ms) |
|---|---|
| CPULoadScheduler | 315 |
| LotteryScheduler | 372 |
| LotterySchedulerWithLoad | 831 |
| RandomScheduler | 320 |
| WallTimeEstimatedScheduler | 774 |

**Tabela 2. Tempo médio de decisão das políticas.**

A *CPULoadScheduler* e a *RandomScheduler* têm aproximadamente o mesmo tempo de decisão e são as políticas que decidem mais rapidamente. A *WallTimeEstimatedScheduler* é a penúltima colocada e tem o tempo de decisão 2,4 vezes maior que as heurísticas mais rápidas. A política mais lenta é a *LotterySchedulerWithLoad*, que demora 2,6 vezes mais do que *CPULoadScheduler* para tomar uma decisão. Isso mostra que a estimativa de CPU é custosa. Porém, essa demora para decidir pode ser compensatória, visto que os tempos de decisão são aferidos em milissegundos, enquanto os tempos de execuções de aplicações em uma grade são medidos em minutos, horas ou até dias.

## 5. Conclusão

O escalonamento em uma grade computacional é um problema complexo, de difícil solução, devido à dinamicidade da infraestrutura e heterogeneidade de recursos e *jobs*. Este trabalho teve por objetivo a implementação e a experimentação de políticas de metaescalonamento para o SINAPAD, uma rede acadêmica nacional formada por diversos centros de alto desempenho. As políticas propostas têm maior complexidade do que a política utilizada por padrão no SINAPAD. No entanto, essas políticas não necessitam de informações de difícil obtenção tais como o perfil de uso de recursos pelas aplicações.

Experimentos mostraram que a aleatoriedade trouxe muitos aspectos negativos às políticas que a utilizam. A exclusão dessa característica na política *LotterySchedulerWithLoad* deu origem à *WallTimeEstimatedScheduler*, política que obteve o melhor desempenho entre as políticas investigadas. A *WallTimeEstimatedScheduler* apresentou a menor variação nos tempos de execução em todos cenários simulados. Por outro lado, o seu tempo de decisão foi superior aos das políticas mais simples. No entanto, entende-se que tal desvantagem é compensada pelo menor tempo de execução obtido.

A política *LotterySchedulerWithLoad* tem bom rendimento nos testes com grandes sobrecarregadas, porém não repetiu o seu desempenho quando a grade estava com

menos carga de trabalho. Nesses casos, a *CPULoadScheduler* conseguiu ter melhores escalonamentos, mesmo sem utilizar informações quanto a capacidade computacional dos recursos. Por fim, a política mais simples, a *RandomScheduler*, teve os piores resultados, mostrando que ela só deve ser utilizada quando os recursos da grade forem homogêneos.

Melhorias e novos experimentos envolvendo as políticas propostas podem ser realizados no futuro. Os cenários dos testes, por exemplo, não contemplaram heterogeneidade de *jobs*. Além disso, os *jobs* executados nesses cenários são intensivos de CPU. Entretanto, no SINAPAD existe uma diversidade de tipos de aplicações. Por esse motivo, seria interessante realizar testes com aplicações formadas, por exemplo, por um conjunto de *jobs* dependentes entre si ou que façam uso intensivo de dados.

## Agradecimentos

Este artigo é parte de um trabalho financiado pelo programa de iniciação científica e pesquisa da Universidade Federal de Mato Grosso do Sul (UFMS), processo nº 163978.669.167550.13122013; pela Fundação de Apoio ao Desenvolvimento do Ensino, Ciência e Tecnologia do Estado de Mato Grosso do Sul (FUNDECT), processos nº 23/200.700/2012 e nº 23/200.267/2014; e pelo Programa de Educação Tutorial (PET-MEC).

## Referências

Cirne, W., Brasileiro, F., Andrade, N., Costa, L. B., Andrade, A., Novaes, R., and Mowbray, M. (2006). Labs of the world, unite!!! *Journal of Grid Computing*, 4(3):225–246.

Cirne, W., Brasileiro, F., Paranhos, D., Góes, L. F. W., and Voorsluys, W. (2007a). On the efficacy, efficiency and emergent behavior of task replication in large distributed systems. *Parallel Computing*, 33(3):213–234.

Cirne, W., Paranhos, D., Brasileiro, F., Fabrício, L., and Góes, W. (2007b). On the efficacy, efficiency and emergent behavior of task replication. In *in Large Distributed Systems, Parallel Computing*, pages 213–234.

Emmott, S. (2005). Towards 2020 science. Technical report, Microsoft Research. Disponível em http://research.microsoft.com/en-us/um/cambridge/projects/towards2020science/downloads/T2020S_ReportA4.pdf. Acessado em: 02/06/2014.

Foster, I. (2005). Globus toolkit version 4: Software for service-oriented systems. In *Proceedings of the 2005 IFIP International Conference on Network and Parallel Computing*, NPC'05, pages 2–13, Berlin, Heidelberg. Springer-Verlag.

Foster, I., Kesselman, C., and Tuecke, S. (2001). The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of High Performance Computing Applications*, 15(3):200–222.

Frey, J., Tannenbaum, T., Livny, M., Foster, I., and Tuecke, S. (2002). Condor-G: A computation management agent for multi-institutional grids. *Cluster Computing*, 5(3):237–246.

Hey, T. and Trefethen, A. (2003). *The Data Deluge: An e-Science Perspective*, pages 809–824. John Wiley & Sons, Ltd.

Izakian, H., Abraham, A., and Snasel, V. (2009). Comparison of heuristics for scheduling independent tasks on heterogeneous distributed environments. In *Proceedings of the 2009 International Joint Conference on Computational Sciences and Optimization - Volume 01*, CSO '09, pages 8–12, Washington, DC, USA. IEEE Computer Society.

Lima, M. J. D., Melcop, T., Cerqueira, R., Cassino, C., Silvestre, B., Nery, M., and Ururahy, C. (2005). CSGrid: um sistema para integração de aplicações em grades computacionais. In *Salão de Ferramentas do 23o. Simpósio Brasileiro de Redes de Computadores*, Porto Alegre, Brazil. SBC.

Maheswaran, M., Ali, S., Siegel, H. J., Hensgen, D., and Freund, R. (1999). Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In *In Eighth Heterogeneous Computing Workshop*, pages 30–44. IEEE Computer Society Press.

Menninger, W. (1993). Código do Benchmark Linpack em C. Disponível em http://www.netlib.org/benchmark/linpackc.new. Acessado em: 20/06/2014.

Schaller, R. R. (1997). Moore's law: Past, present, and future. *IEEE Spectrum*, 34(6):52–59.

Silva, D. P. D., Cirne, W., Brasileiro, F. V., and Grande, C. (2003). Trading cycles for information: Using replication to schedule bag-of-tasks applications on computational grids. In *Applications on Computational Grids, in Proceedings of Euro-Par 2003*, pages 169–180.

Xhafa, F. and Abraham, A. (2010). Computational models and heuristic methods for grid scheduling problems. *Future Generation Computer Systems*, 26(4):608–621.