Aprendizado por Reforço aplicado a escalonamento em Grids

Bernardo Fortunato Costa
COPPE/Sistemas, UFRJ, Brasil
bernardofcosta@yahoo.com

Inês Dutra
DCC, Universidade do Porto, Portugal
ines@dcc.fc.up.pt

Marta Mattoso
COPPE/Sistemas, UFRJ, Brasil
marta@cos.ufrj.br

Resumo

Aprendizado por reforço é uma técnica simples que possui aplicação em várias áreas. Um ambiente real de grid, em geral dinâmico e heterogêneo, oferece um ambiente interessante para sua aplicação. Neste trabalho, utilizamos esta técnica para classificar os nós disponíveis em um grid, dando suporte assim a dois algoritmos de escalonamento, AG e MQD. Um ambiente de grid real foi montado e experimentos foram realizados com estes dois algoritmos, de maneira a verificar seu impacto em um ambiente real, com e sem a presença de reescalonamento.

1. Introdução

Grids têm atraído muito interesse da comunidade científica por oferecer uma poderosa ferramenta de computação em grande escala. Dentro de grids, a estratégia de alocação de tarefas nos recursos é um problema que permanece em aberto. Um ponto a ser destacado é a dificuldade intrínseca em monitorar os recursos. Nesse contexto, foram desenvolvidos algoritmos que vão desde estratégias ingênuas, como a alocação aleatória, até algoritmos mais complexos que prevêem estimativa de desempenho das tarefas nos nós computacionais, passando pela verificação da localidade de dados.

Estas estratégias são limitadas. O critério aleatório não produz otimização da execução. Localidade de dados é uma estratégia interessante apenas para aplicações intensivas em dados. E por fim, estimar desempenho das tarefas em nós computacionais é extremamente difícil de ser realizada pelas limitações que o ambiente de grid impõe ao monitoramento dos recursos. Há, ainda, a possibilidade de uso de metaheurísticas baseadas em técnicas de inteligência artificial. Uma destas técnicas possíveis de uso é o aprendizado por reforço [13], o qual possui algumas vantagens como adaptação às mudanças no ambiente e implementação pouco complexa.

Neste trabalho, mostramos duas maneiras de utilizar a técnica de aprendizado por reforço aplicada ao problema do escalonamento de tarefas em recursos computacionais num ambiente de grid, as quais resultarão em dois algoritmos. Seu propósito é verificar o impacto destes dois algoritmos no tempo total de execução, quando estes são utilizados em um ambiente de grid real para alocar um total de tarefas em um conjunto de nós computacionais disponíveis. Utilizamos a ferramenta GridbusBroker [14] para construir nosso próprio grid computacional e nela implementamos os algoritmos. De maneira a simplificar este estudo, trataremos do problema de escalonamento associado a tarefas independentes.

Este trabalho está organizado em cinco seções. A próxima seção apresenta o problema de meta-escalonamento e a ferramenta a ser utilizada. Na Seção 3, detalhamos o uso da técnica de aprendizado por reforço no contexto de meta-escalonamento, bem como os algoritmos dela decorrentes. Na Seção 4, apresentamos nossa metodologia e adaptações para implementação. Na Seção 5, mostramos os resultados obtidos e sua análise e, por fim, concluímos e oferecemos sugestões para trabalhos futuros.

2. Meta-escalonamento

Meta-escalonamento é um termo que designa o escalonamento de um conjunto de tarefas sobre um conjuntos de sítios, possivelmente estabelecidos em domínios diferentes, onde cada qual possui um escalonador local utilizado para acessar seus recursos. O meta-escalonamento é, portanto, uma instanciação do problema de escalonamento de tarefas em recursos onde não temos precisão nem certeza sobre a informação de monitoramento fornecida pelos recursos utilizados, sendo eles heterogêneos em geral.

Uma boa fonte de revisão sobre os trabalhos já realiza-

dos no campo do meta-escalonamento é o realizado por Dong e Akl [3] que sintetizam as heurísticas mais importantes utilizadas para alocação de tarefas nos recursos, categorizando-as pelas suas características. Dentro de sua classificação, o trabalho aqui desenvolvido localiza-se na área de escalonadores globais, dinâmicos ou híbridos, baseados em heurísticas e cuja solução encontrada seja um sub-ótimo atingido por aproximação.

Para esta classificação, há um conjunto de algoritmos já conhecidos. Em se tratando de escalonamento de tarefas independentes, os mais conhecidos são o Min-min, Min-max, Sufferage e Xsufferage, os quais utilizam algoritmos como o MCT (*Minimum Completion Time*) ou o MET (*Minimum Execution Time*) em sua base. Estes algoritmos pertencem a uma classe de algoritmos chamada PIDA (*Performance Information Dependent Algorithms*), pois necessitam de uma previsão de desempenho da tarefa no nó.

Uma outra maneira de se atacar o problema é duplicar a execução das tarefas nos nós computacionais, terminando as restantes assim que a primeira finalizar. Tal estratégia, obviamente, pressupõe a existência de recursos computacionais em abundância. Alguns exemplos de utilização desta estratégia podem ser vistos em Silva *e*t al. [6] no sistema OurGrid e no trabalho de Lee e Zomaya [11].

Em escalonadores dinâmicos de grid, uma forma usualmente encontrada são os escalonadores hierárquicos ou organizados por federação onde as escolhas realizadas são baseadas em ótimos de Pareto alcançados localmente para cada escalonador local. Um enfoque interessante foi mostrado em simulações por Galstyan *e*t al. [10] com uma estratégia baseada em aprendizado por reforço que consegue coordenar a submissão de tarefas dos usuários distribuindo sua carga pelo grid sem que estes se comuniquem explicitamente.

Alguns sistemas de gerenciamento de grid possuem seus respectivos escalonadores, tais como GrADS [7], GridWay [4], EasyGrid [5] e GridbusBroker [14]. Outros escalonadores, como APPLeS [8] ou GRAND/AppMan [9], são mais simples. GridbusBroker e APPLeS estão focados em aplicações com troca de parâmetro (*parameter sweep applications*) e possuem também algoritmos para lidar com localidade de dados. O GridbusBroker também trabalha com tarifação de recursos computacionais, sendo esta utilizada como um possível parâmetro de otimização. GrADS classifica os recursos por meio de uma ponderação de pesos entre o tempo estimado para computação e o custo de transferência dos dados, além de realizar reescalonamento. GridWay também utiliza reescalonamento e é capaz de se adaptar dinamicamente às mudanças no ambiente. EasyGrid está focado em ambientes voltados a aplicações paralelas com base em MPI. GRAND é um modelo hierárquico para administrar grandes submissões de aplicações em grids, o qual foi implementado como protótipo pelo AppMan. Este último funciona como administrador do nó do grid, escolhendo os recursos tendo em vista a localidade dos dados.

A maioria destes sistemas assume a existência de um sistema de filas de tarefas (ou *batch job queue*, ou ainda *resource management system*) para cada nó do grid, o qual estará acessível para submissão. Desta maneira, a estratégia de escalonamento trata de escolher para qual fila será enviada a tarefa a ser escalonada. Sendo assim, apresentemos agora a ferramenta de trabalho utilizada para acessar estas filas.

2.1. GridbusBroker

O GridbusBroker (GBB) é uma extensão do Nimrod/G [1], um escalonador mais antigo voltado para aplicações de troca de parâmetros ou que utilizem grandes volumes de dados. A sua escolha como ferramenta para trabalho se deve a simplicidade de sua arquitetura baseada em SOA (*Service Oriented Architecture*) e java, além de licença código aberto que possibilita a realização de alterações. Além disso, dá suporte à utilização de vários sistemas de filas de tarefas por já ter suas interfaces com eles desenvolvidas, e tem como pressuposto de utilização ter o mínimo possível de restrições sobre o ambiente utilizado. A única restrição encontrada nos experimentos será a necessidade de possuir um acesso por ssh a cada um dos nós do grid, cada qual possuindo um sistema de filas de tarefas disponível para submissão.

O GBB em si é um aplicativo java, aqui utilizado como aplicativo de linha de comando, o qual é chamado passando-se como parâmetro o nome de três arquivos de configuração em formato XML. Estes arquivos dão suporte à descrição dos nós computacionais disponíveis, entendidos por este como serviços, além de descreverem a maneira pela qual estes serviços estarão disponibilizados para uso (p.ex: conexão ssh) e a descrição das tarefas que se deseja executar no ambiente de grid.

Como um aplicativo multitarefa, este possui uma *thread* específica para monitorar o estado dos serviços, o estado de cada tarefa, uma thread para escalonar tarefas e outra para despachá-las aos seus sítios de computação. Toda comunicação realizada entre estas threads é realizada por meio de uma base de dados comum que guarda informações das tarefas e dos serviços computacionais disponíveis a todo momento, inclusive após terminada a sua execução.

Uma tarefa pode estar associada a um total de até dez estados possíveis, sendo os mais comuns os estados de *ready*, *scheduled*, *stagein*, *pending*, *active*, *stageout* e *done*. Estes descritos nessa ordem perfazem um ciclo esperado de estados que uma tarefa em particular deve passar nesta ordem em seu ciclo de vida. Neste ambiente já estão implementados uma interface java de escalonador, a ser utilizada para desenvolvimento de futuras heurísticas, assim como alguns

escalonadores.

O GBB implementa uma estratégia de escalonamento *round-robin* e permite ao usuário escolher otimizações baseadas em tempo de execução, custo financeiro dos serviços ou ambos os parâmetros. Mais detalhes podem ser encontrados no manual do usuário do GBB [12].

3. Estratégias adaptáveis de escalonamento

Nesta seção, iremos apresentar a idéia por trás dos dois algoritmos estudados e implementados. Uma explicação com mais riqueza de detalhes será realizada na sessão 4.

3.1. Algoritmo de Galstyan

O trabalho de Galstyan *e*t al. [10] visa coordenar a alocação das tarefas de usuários sobre um grid, sem que haja comunicação explícita seja dos usuários ou dos nós do grid, respectivamente, entre si. Neste algoritmo, a idéia é utilizar uma técnica simples para enfrentar o problema da dificuldade de monitoramento dos recursos, seja pela imprecisão ou pela constante invalidação das informações obtidas em um ambiente de grid.

Seu usuário foi modelado como um agente egoísta que tem por objetivo ter os melhores recursos para si próprio e com isto diminuir o tempo total de execução de suas tarefas (*makespan*) e alcançar um bom balanceamento de carga no sistema. Estes tentam minimizar seu tempo de espera no sistema e para isto foram considerados importantes as informações de tempo de espera em fila e tempo de execução da tarefa.

A idéia por trás deste algoritmo (AG) pertence ao contexto da Inteligência Artificial Distribuída, onde foi utilizada uma metaheurística chamada de Aprendizado por Reforço [13]. Neste, um agente deve receber recompensas ou punições de maneira a guiar seu comportamento de procura por um determinado nó ou conjunto de nós do grid. Isto é feito atribuíndo-se um índice de eficiência para cada nó, o qual flutua de acordo com seu histórico de execuções realizadas.

A técnica de Aprendizado por Reforço utilizada no algoritmo é chamada de Aprendizado-Q (*Q-Learning*) [15]. Ali, fora mostrado por meio de simulações que o algoritmo utilizando a técnica de aprendizado por reforço distribui melhor as tarefas pelos recursos que um algoritmo baseado em seleção aleatória dos recursos ou que escolha o recurso com menor carga no sistema.

3.2. Filas múltiplas com duplicação

Lee e Zomaya [11] propuseram um algoritmo chamado de filas múltiplas com duplicação (*Multiple Queues with Duplication* - MQD), focado em diminuir o tempo total de execução das tarefas, também conhecido como *makespan*. A idéia por trás deste algoritmo é que tarefas mais trabalhosas devem ser alocadas em nós do grid que tenham mais facilidade de resolvê-las. Para isso, é necessário, além de se estimar a capacidade computacional dos recursos disponíveis, o que é realizado numa primeira fase do algoritmo, ter um meio de se ordenar as tarefas a serem submetidas de acordo com seu tempo esperado de execução.

Assim, em sua primeira fase, o algoritmo realiza uma alocação inicial de tarefas, distribuindo-as nos nós do grid de maneira crescente pelo seu tempo esperado de execução, de maneira a classificar e ordenar os recursos disponíveis segundo seu poder computacional. Feito isso, as tarefas restantes são divididas em grupos onde o critério de separação é a proximidade de tempo esperado de execução e o número de grupos é sempre igual ao número de nós disponíveis para execução no grid. Dessa forma, cada grupo de tarefas é associada a um nó do grid para o qual as tarefas serão submetidas em ordem decrescente de tempo esperado de execução. A associação é feita na proporção direta entre tempo esperado de execução e poder computacional do nó, de maneira que o nó com maior poder computacional seja responsável pelo grupo de tarefas mais longas, e assim por diante, até que o grupo de tarefas mais curtas seja associado ao nó com menor poder computacional.

Os resultados obtidos com simulações realizadas sobre o SimGrid [2] mostraram um ganho de 3 até 10 % em termos de tempo total de execução (*makespan*) se comparado a um algoritmo de alocação aleatória (*round-robin*). O trabalho de Lee e Zomaya também realiza comparações entre seu algoritmo e outros mais conhecidos como Min-min, Min-max, Sufferage e Xsufferage. Este mostra que, caso haja um erro de cerca de 30 % na estimativa do desempenho dos recursos computacionais, seu algoritmo apresenta um ganho de até 20 % em relação a eles.

4. Metodologia, algoritmos e implementação

O trabalho realizado foi implementar estes dois algoritmos descritos na sessão anterior na ferramenta GridbusBroker. Ambos compartilham uma maneira comum de classificação dos recursos computacionais no grid em índices de eficiência, baseada em Aprendizado-Q como implementação de Aprendizado por Reforço. Esta implementação será melhor descrita a seguir. Posteriormente, mostraremos como os algoritmos em questão utilizam esta classificação para determinar a associação entre tarefa despachada e nó do grid escolhido para executá-la.

Cada tarefa tem a si associada um tempo total gasto (tto). Este pode ser entendido como a soma de tempo gasto em algumas operações básicas tais como espera em fila remota (tf), transferência de arquivos de entrada e saída (ttr) e execução propriamente dita no nó da grid (te). Podemos

então definir:

$$tto = te + tf + ttr \quad (1)$$

Podemos calcular, a partir do tempo total gasto (tto) de todas as tarefas, a sua média, definida como tempo total médio (ttm). Da mesma maneira, de todos os tempos de execução (te), temos a sua média definida como tempo de execução médio (tem). Estas considerações a respeito das componentes de tempo gasto pelas tarefas, de maneira individual e coletiva, é necessária para calcular um índice de tempo de cada tarefa (pi) e um índice médio de tempo das tarefas (pim), os quais determinarão a eficiência dos recursos computacionais. Assim sendo, podemos definir (pi) e (pim) como uma ponderação entre a parcela de tempo relativa a execução (te) ou (tem) respectivamente, e todas as demais parcelas restantes, como descrito em:

$$pi = te \cdot \alpha + (1 - \alpha) \cdot (tto - te) \quad (2)$$

$$pim = tem \cdot \alpha + (1 - \alpha) \cdot (ttm - tem) \quad (3)$$

Tanto em 2 quanto em 3, α é um peso que dará maior ou menor importância ao tempo de execução. Esta foi considerada como uma componente de pouca volatilidade, quando comparada a outras componentes de tempo, como tempo em fila ou de transferência de arquivos, onde se espera uma dispersão maior de valores.

Ao final da execução de uma tarefa, calculamos pi e pim e comparamos seus valores para determinar se o recurso computacional associado a tarefa terminada receberá uma recompensa ou uma punição. Isto se dá da seguinte forma. Seja $stdtem$ o desvio-padrão referente a média tem anteriormente calculada: se $pi < pim - stdtem$, então ao recurso será atribuída uma recompensa, e se $pi > pim + stdtem$, então ao recurso será atribuída uma punição. Caso nenhuma destas situações ocorram, a eficiência do recurso permanecerá inalterada.

Definimos assim a eficiência atualizada de um recurso computacional ($nrle$) como:

$$nrle = rle + l \cdot (r - rle) \quad (4)$$

onde rle é a valor anterior a atualização da eficiência do mesmo recurso, l é um parâmetro referente a velocidade do aprendizado na técnica utilizada e r é a recompensa ou punição atribuída ao recurso, a qual foi implementada como um valor unitário positivo ou negativo. No início da computação, o valor zero é atribuído a eficiência de todos os nós do grid. O algoritmo 1 resume o processo de atualização da eficiência dos recursos computacionais.

Com este processo comum de atribuição de índices de eficiência aos recursos computacionais, os algoritmos implementam suas políticas de escolha de recursos para tarefas. Primeiramente, tomemos o caso do algoritmo 2 (Algoritmo de Galstyan). Nele, a escolha do recurso é feita de modo guloso e probabilístico. É guloso porque sempre atribui o recurso disponível de melhor eficiência a tarefa a ser executada. E é probabilístico porque esta escolha gulosa não ocorre sempre, mas com uma probabilidade alta. Caso contrário, com uma probabilidade baixa é realizada a escolha aleatória do recurso a ser disponibilizado. Além disso, a ordem de submissão das tarefas é aleatória. Em nossa implementação, a probabilidade de escolha gulosa foi fixada em 85 %.

Data:
tto, *// tempo total*
te, *// tempo execução*
tem, *// média da execução*
stdtem, *// desvio-padrão*
rle *// eficiência atual*
Result:
nrle *// nova eficiência*

atribuição de valores a α e l;
$pi \leftarrow te$ * α + (1 - α) * (tto - te);
$pim \leftarrow tem$ * α + (1 - α) * (ttm - tem);
if *(*pim - $stdtem$ *)* $> pi$ **then**
  $r \leftarrow 1$;
  $nrle \leftarrow rle + l$ * (r - rle);
else
  if *(*pim + $stdtem$ *)* $< pi$ **then**
    $r \leftarrow$ - 1;
    $nrle \leftarrow rle + l$ * (r - rle);
  else
    $nrle \leftarrow rle$;
  end
end
return $nrle$

Algorithm 1: calcular Eficiência

No caso do algoritmo MQD, a associação entre tarefa a ser executada e recurso computacional é determinística e não-gulosa, mas necessita de uma fase inicial para ser possível classificar e ordenar os recursos computacionais disponíveis. Nesta fase inicial, descrita no algoritmo 3, as menores tarefas são despachadas aos nós do grid, para que haja uma atualização inicial da eficiência dos recursos computacionais disponíveis.

Concluída esta fase com o preenchimento das filas nos nós do grid, realiza-se então o algoritmo 4 (MQD propriamente dito). A tarefas restantes são ordenadas de maneira decrescente pelo seu tempo esperado de computação, e posteriormente divididas, segundo o tempo esperado de computação, em grupos cujo tamanhos não excedam uma unidade entre si. A cada grupo é associado um nó do grid na proporção direta entre eficiência calculada do nó e tempo

Data:
Conjunto de tarefas a se escalonar,
Conjunto de nós disponíveis
Result:
Associação entre tarefas e nós

foreach *Job* i **do**
  // sorteio do tipo de escolha
  $p \leftarrow$ ponto flutuante no intervalo [0;1];
  if $p > 0.85$ **then**
    // escolha aleatória
    $r \leftarrow$ sorteia(nó);
    associa(r,i);
  else
    // escolha de nó mais eficiente
    $r \leftarrow$ maxEff(nó);
    associa(r,i);
  end
end

Algorithm 2: Algoritmo AG

Data:
Conjunto de tarefas a se escalonar,
Conjunto de nós disponíveis
Result:
Associação entre tarefas e nós

// tarefas postas em ordem ascendente
Tarefas $\leftarrow$ ordenarAscendente(Tarefas);
// escolha nó de maneira rotativa
foreach *Tarefas* i **do**
  if *estaVazio(Nós)* **then**
    // término da fase
    return
  else
    $r \leftarrow$ proximo(Nós);
    if *estaSaturado(r)* **then**
      // nós saturados são removidos da lista
      remove(r,Nós);
    else
      associa(r,i);
    end
  end
end

Algorithm 3: MQD: fase inicial

médio esperado de execução de suas tarefas. As tarefas de maior tempo esperado de execução dentro de cada grupo são despachadas até que não haja mais posições disponíveis em fila remota dos nós do grid ou que a fila de tarefas esteja vazia.

Caso esta não se esvazie por completo, refaz-se uma nova rodada de divisão de tarefas em grupos, sua associação aos nós do grid pelo critério do algoritmo e posterior despacho. Para se evitar uma injustiça no cálculo da eficiência dos nós do grid, durante a segunda fase do algoritmo MQD, o cálculo de ttm, tem e $stdtem$ é realizado considerando-se apenas as tarefas já computadas por um determinado nó do grid, ao invés do conjunto total de tarefas já computadas como feito anteriormente. Assim, cada uma destas grandezas médias é utilizada por seus respectivos sítios no algoritmo de cálculo da eficiência.

Cabe ressaltar outras duas diferenças que o algoritmo MQD aqui implementado tem em relação ao original. Na versão de Lee e Zomaya, foi utilizado como critério de classificação dos nós do grid o tempo total gasto por uma única tarefa. Nesta versão, utiliza-se um conjunto de tarefas para calcular um índice de eficiência relativo ao nó. Outra diferença é que, no algoritmo original, uma vez realizada a classificação dos nós, esta permanece inalterada até o fim do algoritmo. Já a versão aqui implementada permite a alteração desta classificação na segunda fase do algoritmo, ao continuar computando os índices de eficiência relativo aos nós do grid.

5. Resultado e Análise

Uma vez que estes algoritmos descritos puderam ser implementados no GridbusBroker, realizamos uma série de experimentos de maneira a medir seu desempenho em um ambiente de grid real. O ambiente de grid montado foi constituído de um conjunto de contas de usuário em domínios diferentes, onde uma máquina de entrada tinha acesso a um sistema de filas de tarefas. A tabela 1 mostra o poder computacional disponível nos sítios utilizados. Utilizamos sempre o maior número possível de sítios disponíveis, sendo utilizados no mínimo quatro sítios por experimento. Dentro de um experimento, não só o número como o próprio sítio estão fixos.

Utilizamos, aqui, como base para os nossos experimentos, aplicações de trocas de parâmetros sem relação de precedência ou dependência entre tarefas (também conhecidas como *bag-of-tasks*) para avaliar os algoritmos propostos. Este tipo de aplicação tem seu uso bastante difundido em ambientes de grid, tendo sido uma das primeiras utilizadas neste tipo de ambiente.

A aplicação stub, necessária para realizar os testes com os algoritmos, se trata de um executável simples, intensivo em uso de CPU, mas com pouco uso de outros recursos de máquina como memória, comunicação, ou grandes arquivos de entrada e saída. Este aplicativo necessita de um valor de entrada, o qual acaba por determinar seu tempo gasto em execução. Escolhemos uma aplicação curta e de poucos recursos computacionais para evitar alongar de-

```
Data:
Conjunto de tarefas a se escalonar,
Conjunto de nós disponíveis
Result:
Associação entre tarefas e nós
// Tarefas e nós serão listados na descendente
Tarefas ← ordenaDecrescente(Tarefas);
Nós ← ordenaEficienciaDecrescente(Nós);
// descobre tamanho dos grupos
grp ← tamanho(Tarefas) / tamanho(Nós);
j ← 0;
r ← proximoInsaturado(Nós);
foreach Tarefas i do
    if estaVazio(Nós) then
        // Todos os Nós indisponíveis ou saturados
        return
    else
        // Tarefas relativas a Nós saturados esperarão
        a próxima rodada
        if j ≥ grp then
            j ← 0;
            remove(r,Nós);
            r ← proximoInsaturado(Nós);
        end
        // Tente associar tarefa ao nó. Mesmo que
        associa falhe, conte as tarefas até o fim do
        grupo
        associa(r,i);
        j ← j + 1;
    end
end
```

Algorithm 4: MQD: fase principal

Tabela 1. Sítios disponíveis

Sítio	CPUs	Gerenciador de Filas
LabIA	24	Torque/Maui
LCP	28	SGE
Nacad	16	PBS_PRO
LCC	44	Torque
UERJ	144	Condor
UFRGS	4	Torque

mais a finalização dos experimentos, assim como para não penalizar os sítios que nos cederam seus recursos e seus usuários. Tomando-se por base uma máquina com processador Pentium IV com 2,8 GHz, 1GB de RAM, 80 GB de disco e sistema operacional linux SL 4.2, os valores mínimo e máximo de entrada para este aplicativo situaram-se próximos a 3 e 8 minutos, respectivamente. Dentro deste intervalo, a distribuição de valores de entrada gerados foi uniforme.

Dito isso, passemos a organização propriamente dita dos experimentos. Os experimentos foram conduzidos em duas fases. Na fase inicial, busca-se encontrar empiricamente um valor satisfatório para os parâmetros α e l no algoritmo que calcula a eficiência dos recursos computacionais. Utilizamos valores de α com 0,2, 0,5 e 0,8 para dar menor, igual ou maior importância ao tempo de execução no cálculo do índice de eficiência. Da mesma forma em l, utilizamos valores de 0,3, 0,5 e 0,7 para sucessivamente aumentar a importância da última tarefa executada no cálculo da eficiência. Numa segunda fase, com os valores de α e l ajustados, verifica-se o comportamento dos algoritmos na presença de reescalonamento.

Cada experimento significa uma rodada com três execuções do GridbusBroker, onde os algoritmos RR, AG (Algoritmo de Galstyan) e MQD são executados de maneira consecutiva. RR (Round-robin) é um algoritmo do GridbusBroker sem princípio de otimização embutido. Ele foi utilizado como uma maneira de normalizar o makespan obtido nas execuções de AG e MQD. Utilizamos 500 como o total de tarefas a serem alocadas, salvo em um caso onde dobramos este número para verificar o comportamento dos algoritmos. Definimos um cenário de experimentos quando os parâmetros de execução α e l estão fixos. Neles realizamos um total de 15 experimentos finalizados com sucesso, de onde foram calculados os valores de média e desvio-padrão apresentados nas tabelas. Experimentos finalizados com sucesso excluem execuções onde foram identificados algum tipo de problema ou que se apresentaram como pontos fora da curva, principalmente devido a queda do acesso a rede ou com a infraestrutura interna dos sítios, entre outras ocorrências.

Sendo assim, as tabelas 2 e 3 mostram o resultado obtido para os experimentos da primeira fase. As porcentagens mostradas na tabela 2 são a média dos ganhos em relação ao RR e o respectivo desvio padrão. Na tabela 3, está a probabilidade de os algoritmos AG e MQD terem um resultado semelhante ao seu respectivo RR, ou seja, não produzirem otimização nenhuma. Este foi calculado tendo por base a comparação entre amostras de um teste T-Student.

Podemos, então, verificar que o melhor resultado para ambos os algoritmos é quando $\alpha = 0,5$ e $l = 0,3$. Verificamos também que, embora um aumento na importância do parâmetro l melhore o desempenho do Algoritmo de

Tabela 2. Fase I: Ganhos de Tempo de Execução

Parâmetros		Resultados		
α	l	Algoritmo	Média	Desvio Padrão
0.2	0.3	AG	4.65%	7.44%
		MQD	8.33%	7.11%
0.5	0.3	AG	8.36%	8.39%
		MQD	11.62%	6.94%
0.5	0.5	AG	5.10%	5.81%
		MQD	-0.49%	18.93%
0.5	0.7	AG	8.03%	9.86%
		MQD	-8.92%	17.28%
0.8	0.3	AG	4.69%	6.98%
		MQD	4.84%	5.42%

Tabela 3. Fase I: Probabilidade de não-otimização

Parâmetros		Resultados	
α	l	AG	MQD
0.2	0.3	23.69%	20.83%
0.5	0.3	14.67%	6.55%
0.5	0.5	4.74%	46.53%
0.5	0.7	0.74%	6.19%
0.8	0.3	22.58%	24.29%

Tabela 4. Fase II: Ganhos de Tempo de Execução

Parâmetros		Resultados		
Tarefas	Espera em Fila (sec.)	Algoritmo	Média	Desvio Padrão
500	500	AG	6.72%	15.18%
		MQD	11.56%	10.15%
500	750	AG	1.86%	6.16%
		MQD	6.98%	4.85%
1000	750	AG	4.02%	7.09%
		MQD	4.75%	5.66%
500	1000	AG	5.46%	5.49%
		MQD	9.53%	8.75%

Tabela 5. Fase II: Probabilidade de não-otimização

Parâmetros		Resultados	
Tarefas	Espera em Fila (sec.)	AG	MQD
500	500	15.48%	0.13%
500	750	23.16%	8.24%
500	1000	21.99%	0.52%
1000	750	7.90%	8.70%

Galstyan, por outro piora sensivelmente o desempenho de MQD. Assim, ao amplificar a importância dos resultados mais recentes, por um lado se torna mais fácil capturar mudanças do ambiente e por outro, mais difícil se torna a tarefa de comparar a capacidade computacional entre os recursos disponíveis.

Com os valores de α e l determinados, na segunda fase dos experimentos analisamos como os algoritmos se comportam na presença de reescalonamento por tempo máximo de espera em fila. Assim, tomamos os valores de 500, 750 e 1000 segundos para tempo de espera em fila. Se trata da média do tempo de espera em fila medido na primeira fase dos experimentos e o intervalo de uma desvio padrão em torno desta média. Fixamos também o tempo de espera em fila e dobramos a carga de tarefas para verificar se os algoritmos mantêm seu padrão de otimização.

Equivalentemente as tabelas 2 e 3 para a primeira fase, são as tabelas 4 e 5 para a segunda fase. As tabelas 2 e 3 nos mostram que na presença de reescalonamento, os algoritmos MQD e AG continuam a produzir otimizações de makespan, embora menores que as apresentadas sem reescalonamento. MQD, em particular, mostra até uma melhora quando o tempo em fila é de 500 segundos. Este algoritmo, aliás, teve desempenho melhor que AG, mostrando-

se uma heurística mais robusta para trabalhar com eventos de reescalonamento.

6. Conclusão e Trabalhos Futuros

Neste artigo, estudamos duas estratégias de escalonamento em grid, as quais utilizaram aprendizado por reforço como técnica para escolha de recursos computacionais disponíveis. São alternativas ao que vem sendo tradicionalmente utilizado na área de grid, e ao mesmo tempo de simples implementação. Em um primeiro instante, nos dedicamos a estimar empiricamente alguns parâmetros utilizados na técnica. Posteriormente, verificamos o desempenho dos algoritmos na presença de reescalonamento. Os algoritmos aqui descritos são referentes a trabalhos anteriormente realizados em ambiente simulado de grid.

Neste trabalho, nos dispusemos a criar um ambiente real de grid e tomar nossas medidas neste. Pequenas adaptações foram realizadas para portar os algoritmos na ferramenta utilizada. A utilização da técnica de aprendizado por reforço no algoritmo MQD foi a maneira encontrada para estimar a capacidade computacional dos recursos disponíveis.

Os resultados mostram uma otimização dos algoritmos, que vem a fortalecer o trabalho anteriormente realizado nas simulações das fontes deste trabalho. Com relação a técnica de aprendizado por reforço, foi visto que uma taxa de aprendizado menor e mais conservadora é preferível para a tarefa de classificar e comparar os recursos. Em ambientes dinâmicos e heterogêneos como os grids, estratégias adaptativas são importantes e acreditamos ser necessário um esforço maior para compreender como estas se comportam em um ambiente real. Estamos agora concentrando esforços em realizar trabalho semelhante com outras estratégias de escalonamento, em especial, as já implementadas no GridbusBroker.

7. Agradecimentos

Gostaríamos de agradecer a todas as pessoas e instituições que gentilmente cederam seus laboratórios para nossos experimentos, assim como a comunidade desenvolvedora do GridbusBroker e SSHTools.

Referências

[1] R. Buyya, D. Abramson, and J. Giddy. Nimrod/g: An architecture for a resource management and scheduling system in a global computational grid. *hpc*, 01:283, 2000.

[2] H. Casanova. Simgrid: A toolkit for the simulation of application scheduling. In *CCGRID*, page 430. IEEE, 2001.

[3] F. Dong and S. Akl. Scheduling algorithms for grid computing: State of the art and open problems. Technical Report 2006-504, School of Computing, Queen's University, Jan 2006.

[4] R. M. E. Huedo and I. Llorente. The gridway framework for adaptive scheduling and execution on grids. *Scalable Computing - Practice and Experience*, 6(3):1–8, Sep 2005.

[5] C. B. et al. An easygrid portal for scheduling system-aware applications on computational grids. *Concurrency and Computation: Practice and Experience*, 18(6):553–566, 2006.

[6] D. P. S. et al. Trading cycles for information: Using replication to schedule bag-of-tasks applications on computational grids. In *Euro-Par*, pages 169–180, 2003.

[7] F. B. et al. New grid scheduling and rescheduling methods in the grads project. *Int. J. Parallel Program.*, 33(2):209–229, 2005.

[8] H. C. et al. The apples parameter sweep template: user-level middleware for the grid. In *Supercomputing*, page 60. IEEE, 2000.

[9] P. K. V. et al. Grand: Toward scalability in a grid environment. *Concurrency and Computation: Practice and Experience*, 19(14):1991–2009, 2007.

[10] A. Galstyan, K. Czajkowski, and K. Lerman. Resource allocation in the grid using reinforcement learning. In *AAMAS*, pages 1314–1315. IEEE, 2004.

[11] Y. C. Lee and A. Y. Zomaya. A grid scheduling algorithm for bag-of-tasks applications using multiple queues with duplication. *icis-comsar*, 0:5–10, 2006.

[12] K. Nadiminti, S. Venugopal, H. Gibbins, T. Ma, and R. Buyya. The gridbus grid service broker and scheduler (2.4.4) user guide. http://www.gridbus.org/broker/2.4.4/manualv2.4.4.pdf, Agosto 2007.

[13] R. Sutton and A. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

[14] S. Venugopal, R. Buyya, and L. Winton. A grid service broker for scheduling distributed data-oriented applications on global grids. In *MGC*. ACM Press, 2004.

[15] C. J. C. H. Watkins and P. Dayan. Technical note: Q-learning. *Mach. Learn.*, 8(3-4):279–292, 1992.