Universidade de São Paulo

Pró-Reitoria de Graduação

Curso de Ciências Moleculares

Ciclo Avançado

Relatório de Iniciação Científica

Modificação de DAGs de Aplicações para Nuvens Computacionais

1º semestre de 2012

Lucas Virgili
Turma 18
no. USP: 5916831
luvirgili@gmail.com

Prof. Dr. Daniel Macêdo Batista
Instituto de Matemática e Estatística
Departamento de Ciência da Computação
Telefone: 3091-5177
batista@ime.usp.br

1 Introdução

Grades computacionais foram desenvolvidas para prover recursos computacionais em larga escala para ciência e engenharia. Nas grades, os recursos estão espalhados ao redor do mundo e são conectados por elementos de redes, compondo organizações virtuais para a execução de aplicações com alta demanda de recursos.

Devido ao sucesso das grades, o paradigma das nuvens computacionais foi desenvolvido. Apesar de serem diferentes, diversos mecanismos e protocolos usados em nuvens foram herdados das grades e, por isso, melhoras em mecanismos de grades podem ter impacto em diversas nuvens já implantadas na Internet.

Aplicações de grades podem ser decompostas em um conjunto de programas chamados de tarefas. Escalonar tais tarefas eficientemente é fundamental para a operação da grade.

Aplicações formadas por um conjunto de tarefas também são executadas em nuvens. De forma similar ao que ocorre com as grades, também é importante realizar um escalonamento eficiente das aplicações nas nuvens. Os requisitos para uma aplicação podem incluir softwares específicos, o que demanda que máquinas virtuais (MVs) sejam instanciadas nos hosts da nuvem antes que as aplicações sejam utilizadas. A instanciação das MVs tem impacto na utilização dos links da rede, já que as imagens delas devem ser transferidas de um repositório para os hosts. Além disso, o tempo para transferir e inicializar as MVs pode aumentar o tempo de execução das aplicações, afetando a qualidade do serviço para os usuários.

Nesse semestre foi estudado um algoritmo para embutir os requisitos de software na descrição das aplicações que serão executadas em nuvens. Esse algoritmo possibilita que as MVs sejam compartilhadas por tarefas com os mesmos requisitos de software, caso tal compartilhamento não prejudique o tempo de execução das aplicações. O algoritmo é destinado à tarefas dependentes que possam ser descritas por digrafos acíclicos (DAGs). Tarefas deste tipo transferem dados entre si e demandam ainda mais recursos da rede do que tarefas sem tal dependência.

2 Algoritmo

Para podermos escalonar os DAGs das aplicações com requisitos de software, é preciso que tais dependências sejam embutidas no DAG original. Assim, as dependências devem ser transformadas em novas tarefas e colocadas nos primeiros níveis do DAG.

Cada nova tarefa representa, então, a instanciação de uma MV que contém o software requerido pela tarefa original. A transferência da imagem do repositório para o host também deve ser incluída no novo DAG. Isso é feito adicionando uma nova tarefa que representa o repositório de MVs. Cada arco da tarefa repositório para as tarefas de instanciação representam a transferência das imagens das MVs. As transferências não podem ser negligenciadas uma vez que consomem banda da rede.

Uma solução para o problema está ilustrada na fig. 1. No DAG da esquerda da figura, cada tarefa é representada por um rótulo $x : y$, no qual x é o ID da tarefa

e y é o ID do software que a tarefa x necessita. O DAG na direita mostra o DAG modificado, no qual foram acrescentadas tarefas que representam a instanciação das MVs que contém o software $S1$ e $S2$. As novas tarefas representam as instanciações e a tarefa R o repositório. Nesse caso, a MV que contém $S1$ será instanciada duas vezes, uma para a tarefa 1 e outra para a tarefa 2. A MV que contém $S2$, por sua vez, será instanciada 3 vezes, para as tarefas 3, 4 e 5. O DAG modificado pode ser usado em qualquer escalonador, já que não há diferença entre entre as tarefas originais e as adicionadas pelo algoritmo.

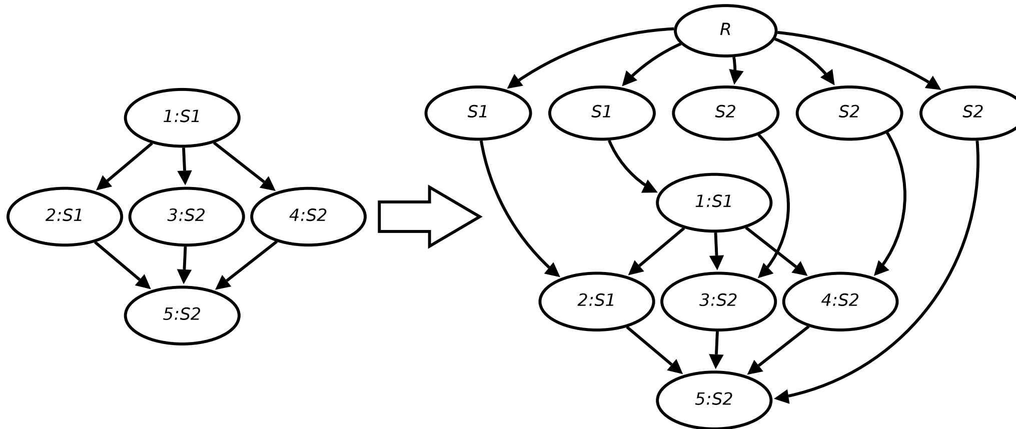

Figura 1: Exemplo de modificação do DAG - uma MV para cada tarefa.

O problema com essa abordagem é que ela não garante que o DAG modificado terá um tempo de execução baixo. Isso acontece pois existem várias maneiras que podemos modificar o DAG, cada uma delas resultando em um DAG modificado diferente. A fig. 2 mostra uma modificação equivalente do DAG original da fig. 1. Nesse caso, a modificação da fig. 2 agrega todas as tarefas que dependem do mesmo software de forma que apenas uma instanciação é necessária. Assim, a MV1 que contém $S1$ é instanciada em um host para a execução da tarefa 1, e pode ser usada, na sequência, pela tarefa 2. A MV que contém $S2$ pode ser compartilhada pelas tarefas 3, 4 e 5 de forma semelhante.

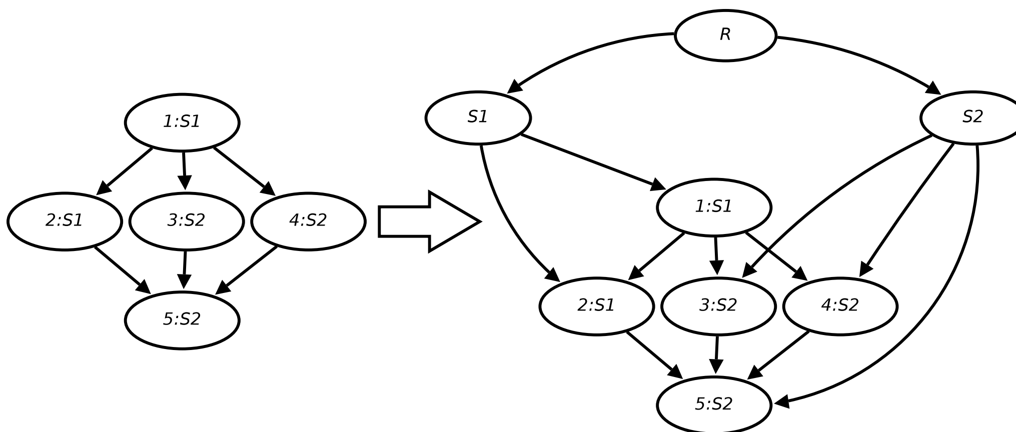

Figura 2: Outro exemplo de modificação do DAG - uma MV para tarefas dependentes do mesmo software.

Cada DAG modificado potencial pode ter um tempo de execução diferente. Por exemplo, para o DAG da aplicação Montage, uma aplicação real executada em grades, existem 877 possíveis DAGs modificados quando 7 tarefas dependem do mesmo software. Quando esses DAGs são escalonados em uma grade com 400 hosts usando-se o escalonador HEFT, o tempo de execução da aplicação varia no intervalo $(321, 518)$ segundos. Portanto, a modificação do DAG não pode ser realizada de forma aleatória, já que não há garantia de que o tempo de execução da aplicação será perto do mínimo.

A modificação ideal deveria transformar o DAG original no DAG que demanda o menor tempo de execução quando escalonado. Uma solução seria gerar todos os DAGs modificados possíveis, escaloná-los e depois selecionar o com o menor tempo de execução. Isso, contudo, é computacionalmente impraticável, como será mostrado abaixo.

Podemos resolver o problema da geração dos DAGs modificados encontrando as partições de um conjunto. Uma partição de um conjunto $\mathcal{S}$ é um conjunto $\mathcal{P}$ não vazio de subconjuntos de $\mathcal{S}$ tal que

$$\bigcup \mathcal{P} = \mathcal{S} \tag{1}$$

Cada software $\mathcal{S}_i$ demandado pelas aplicações da grade pode ser vista como um conjunto contendo os IDs das tarefas dependentes de $\mathcal{S}_i$. A partição de $\mathcal{S}_i$ nos dá todas as possíveis instanciações das MVs que contém o software $\mathcal{S}_i$. Por exemplo, nas figs. 1 e 2, os conjuntos são $S_1 = \{1, 2\}$ e $S_2 = \{3, 4, 5\}$. As partições são, na fig. 1, $\{\{1\}, \{2\}\}$ e $\{\{3\}, \{4\}, \{5\}\}$, e na fig. 2 são $\{\{1, 2\}\}$ e $\{\{3, 4, 5\}\}$.

Entretanto, o número de partições de um conjunto de tamanho n pode ser calculado pela seguinte equação:

$$\begin{aligned} B_0 &= 1 \\ B_{n+1} &= \sum_{k=0}^{n} \binom{n}{k} B_k \end{aligned} \tag{2}$$

Pode-se ver que o número de partições cresce exponencialmente. Logo, não é viável gerar todos os possíveis DAGs, encontrar o tempo de execução de cada um e escolher o que tiver o menor.

O seguinte algoritmo foi proposto para modificar um DAG com requisitos de software. Seu objetivo é reduzir a utilização dos links da rede e evitar aumentar o tempo de execução das aplicações. Isso é feito procurando-se tarefas que tenham as mesmas dependências de software nos caminhos do DAG, de forma que só exista uma instanciação de uma VM para cada dependência em um caminho.

Algoritmo 1 Modificador de DAG

Entrada: DAG $\mathcal{D}$ com requisitos de software $\mathcal{S}$ não incluídos no DAG.
Saída: DAG $\mathcal{M}$ modificado com os requisitos de software incluídos no DAG.

1: **para** cada caminho $h \in \mathcal{D}$ **faça**
2:     **para** cada tarefa $t \in h$ **faça**
3:         Inclua t no conjunto $\mathcal{P}_{h,s}$, onde $\mathcal{S}_s$ é a dependência de software de t.
4:     **fim**
5: **fim**
6: Crie uma nova tarefa inicial R com custo 0
7: **enquanto** existir pelo menos um conjunto $\mathcal{P}$ **faça**
8:     $\mathcal{P}_s$ recebe o maior $\mathcal{P}_{h,s}$ {O custo de $\mathcal{P}_{h,s}$ é medido em bytes e é calculado somando todos os custos dos arcos no caminho h}
9:     Crie uma nova tarefa p_s com custo equivalente ao tempo de inicialização da MV que contém o software $\mathcal{S}_s$
10:     Crie um novo arco da tarefa p_s para cada uma das tarefas em $\mathcal{P}_s$ com custo ∞. {Assumindo o custo como ∞, as tarefas executarão no mesmo host em que a MV foi instanciada}
11:     Crie um novo arco da tarefa R para p_s com custo equivalente ao tamanho da MV que contém o software S_s
12:     Apague o $\mathcal{P}_{h,s}$ mais custoso.
13:     Remova cada tarefa em $\mathcal{P}_s$ de todos os conjuntos $\mathcal{P}$ e, se algum $\mathcal{P}$ ficar vazio, remova-o.
14: **fim**

Considerando-se que existam m tarefas dependentes da instanciação da mesma MV, $m-1$ transferências da MV são evitadas. Não só, por agregar somente tarefas no mesmo caminho, o paralelismo da aplicação não é perdido. Na fig. 2, por exemplo, na qual todas as tarefas foram agregadas, é criada uma dependência artificial entre tarefas, o que impede sua execução em paralelo.

A fig. 3 exemplifica a modificação do DAG quando o algoritmo é utilizado. Apesar das tarefas $3, 4$ e 5 dependerem do software $S2$, elas não dependem da mesma instanciação pois não estão no mesmo caminho do DAG. Nesse exemplo, as tarefas 4 e 5 dependem da mesma instanciação, ao invés das tarefas 3 e 5, devido ao comprimento dos caminhos.

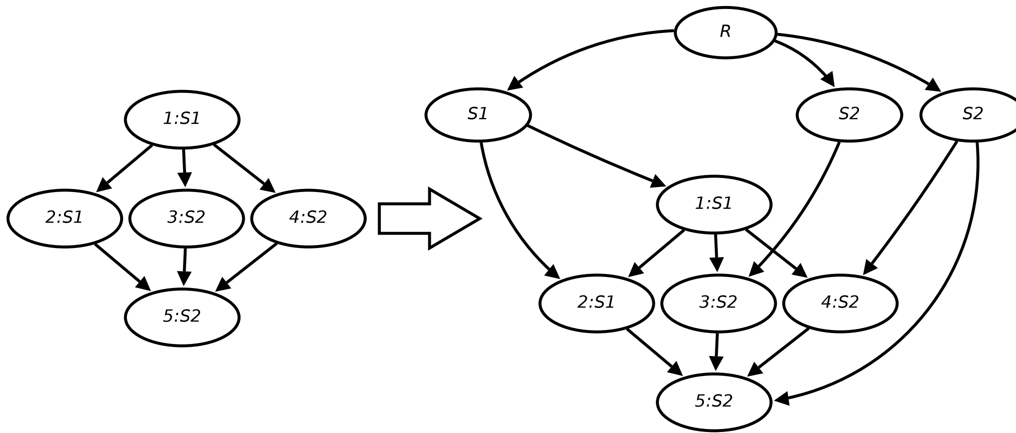

Figura 3: Outro exemplo de modificação do DAG - uma MV para tarefas dependentes do mesmo software que estão em um mesmo caminho.

3 Implementações[1]

3.1 Classe base

Definiu-se uma classe que representa um digrafo de aplicação. O código está abaixo.

Código 1 Classe que representa o DAG

```
1   class app_dag {
2     private:
3     int _ntasks;         //Number of tasks.
4     int _nvm;            //Number of VMs.
5     int *_I;             //Weight of each task.
6     int *_S;             //Software demand of each tasks, denoted by the VM's id.
7     int **_B;            //Number of bytes transmitted between tasks.
8     vector<vector<int> > _paths;
9     map<pair<int,int>, vector<int> > _P;
10    vm_info *vinfo;
11    int _max_vm;                //The max id of the VMs.

12    map<pair<int,int>, vector<int> >::iterator find_heavy();
13    void gen_paths();
14    void find_path(int cur_vertex, vector<int>& curr);
15    void gen_Phs();
16    int find_weight(map<pair<int,int>, vector<int> >::iterator it);
17    void remove_tasks(map<pair<int,int>, vector<int> >::iterator cur_it);
18    void printDAG(const char *, int, vector<vector<int> >,
19                  vector<int>, vector<int>);

20    public:
21    app_dag(const char *, const char *);
22    ~app_dag();
23    void dagmdf_path(const char *);
24    void dagmdf_path();
25    void dagmdf_oneeach(const char *);
26    void dagmdf_oneeach();
27    void dagmdf_onlyone(const char *);
28    void dagmdf_onlyone();
29  };
```

[1]O código completo pode ser visto em
https://github.com/lvirgili/dag_modification/tree/master/src

Os membros mais importantes da classe são:

- ```
  int **_B
  ```

  Uma matriz de adjacências, representando dependências entre as tarefas;

- ```
  vector<vector<int> >_paths
  ```

  Um vetor que armazena os caminhos entre a tarefa 0 e a tarefa $n-1$; e

- ```
  map<pair<int,int>, vector<int> > _P
  ```

  Um mapa que armazena os conjuntos $P_{h,s}$.

Cada um dos métodos públicos da classe, com excessão do construtor e do destrutor, realizam uma modificação diferente no DAG:

- ```
  void dagmdf_path()
  ```

  A modificação explicitada acima;

- ```
  void dagmdf_oneeach()
  ```

  Cada tarefa tem uma MV instanciada somente para ela; e

- ```
  void dagmdf_onlyone()
  ```

  Apenas uma instância de cada MV é feita.

3.2 Funções principais do algoritmo proposto

As funções mais relevantes e suas implementações estão abaixo.

Código 2 Funções que encontram caminhos no DAG

```
1  void app_dag::gen_paths() {
2    for (int i = 1; i < _ntasks-1; ++i) {
3      vector<int> curr;
4      curr.push_back(0);
5      if (_B[0][i] > 0) {
6        find_path(i, curr);
7      }
8    }
9  }

10 void app_dag::find_path(int cur_vertex, vector<int>& curr) {
11   curr.push_back(cur_vertex);
12   if (cur_vertex == _ntasks-1 && curr[0] == 0) {
13     _paths.push_back(curr);
14     curr.clear();
15     return;
16   }
17   for (int i = cur_vertex+1; i < _ntasks; ++i) {
18     if (_B[cur_vertex][i] > 0) {
19       find_path(i, curr);
20     }
21   }
22 }
```

O par de funções do código 2 é responsável por encontrar os caminhos entre os vértices 0 e $n-1$ no digrafo.

Para tanto, uma busca em profundidade é feita no digrafo e, toda vez que o vértice $n-1$ é atingido, o caminho é armazenado no vetor $_paths$.

Código 3 Função que constrói os conjuntos $P_{h,s}$

```
void app_dag::gen_Phs() {
  gen_paths();
  for (unsigned i = 0; i < _paths.size(); ++i) {
    for (unsigned j = 0; j < _paths[i].size(); ++j) {
      int vm = _S[_paths[i][j]];
      if (_P.count(make_pair(i,vm)) == false) {
        vector<int> aux;
        _P.insert(make_pair(make_pair(i,vm),aux));
      }
      _P[make_pair(i,vm)].push_back(_paths[i][j]);
    }
  }
}
```

A função no código 3 constrói os conjuntos $P_{h,s}$. Para armazená-los, um mapa entre pares de inteiros, representando o caminho e máquina virtual, e vetores, que armazenam as tarefas do digrafo que pertencem ao conjunto é contruído.

Código 4 Funções auxiliares

```
1  int app_dag::find_weight(map<pair<int,int>,vector<int> >::iterator it) {
2    int weight = 0;
3    vector<int> path(_paths[it->first.first]);
4    for (unsigned i = 0; i < path.size()-1; ++i) {
5      weight += _B[path[i]][path[i+1]];
6    }
7    return weight;
8  }

9  map<pair<int,int>, vector<int> >::iterator app_dag::find_heavy() {
10   map<pair<int,int>, vector<int> >::iterator it, max_it;
11   int max = 0;
12   for (it = _P.begin(); it != _P.end(); ++it) {
13     int cur_weight = find_weight(it);
14     if (cur_weight > max) {
15       max_it = it;
16       max = cur_weight;
17     }
18   }
19   return max_it;
20 }
```

As duas funções no código 4 são utilizadas para encontrar o conjunto de $P_{h,s}$ mais pesado, isto é, com caminho entre os vértices mais custoso.

Por fim, a função dagmdf, definida no código 5, utilizando as funções acima, gera o digrafo modificado.

Código 5 Função que gera o DAG modificado.

```
1  void app_dag::dagmdf(const char *outfile) {
2    gen_Phs(); // Creates the set P_{h,s}
3    map<pair<int,int>, vector<int> >::iterator it;
4    vector<int> newS, newI;
5    vector<vector<int> > newB;
6    int vms_added = 0;

7    newS.push_back(0); newI.push_back(0);
8    vector<int> repo(1,0); newB.push_back(repo);

9    while (_P.empty() == false) {
10     map<pair<int,int>, vector<int> >::iterator heavy = find_heavy();
11     ++vms_added;
12     newS.push_back(0);
13     newI.push_back(vinfo->TV(heavy->first.second - 1));
14     newB[0].push_back(vinfo->BV(heavy->first.second - 1));
15     vector<int> vm(_ntasks, 0); newB.push_back(vm);
16     for (unsigned i = 0; i < heavy->second.size(); ++i) {
17       newB[vms_added][heavy->second[i]] = 0x7FFFFFFF;
18     }
19     remove_tasks(heavy);
20   }

21   for (int i = 0; i < _ntasks; ++i) {
22     newB[0].push_back(0);
23     newI.push_back(_I[i]);
24     newS.push_back(_S[i]);
25   }
26   for (unsigned i = 1; i < newB.size(); i++) {
27     vector<int> aux(vms_added+1, 0);
28     newB[i].insert(newB[i].begin(), aux.begin(), aux.end());
29   }
30   for (int i = 0; i < _ntasks; ++i) {
31     vector<int> task(vms_added+1, 0);
32     for (int j = 0; j < _ntasks; ++j) {
33       task.push_back(_B[i][j]);
34     }
35     newB.push_back(task);
36   }
37   int new_ntasks = _ntasks + vms_added + 1;

38   printDAG(outfile, new_ntasks, newB, newI, newS);
39 }
```

É interessante notar que o código é relativamente simples devido ao uso de contêineres da STL de C++, que permitem que todas as manipulações dos conjuntos sejam feitas de forma fácil.

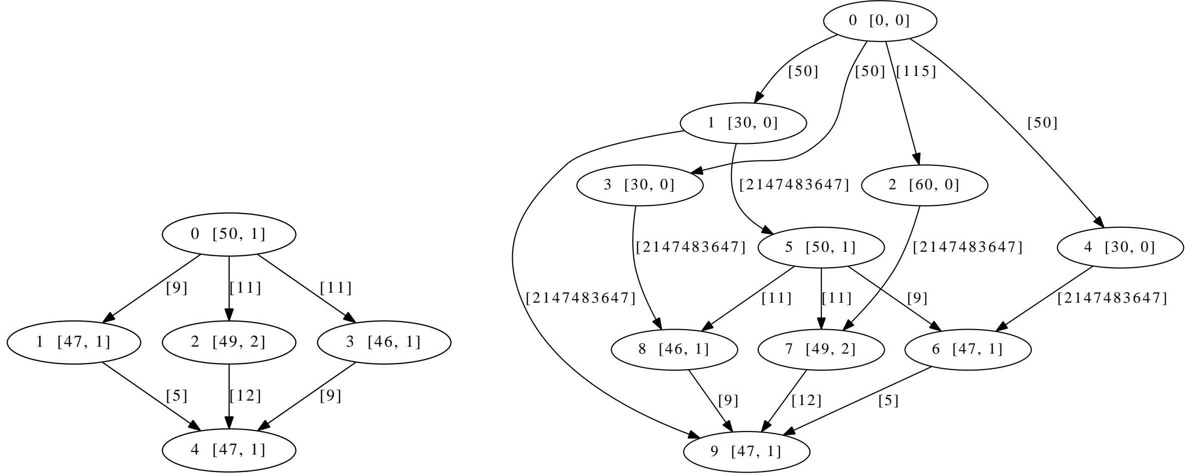

Figura 4: Um DAG simples e sua modificação pelo algoritmo proposto

3.3 Invariantes no DAG modificado pelo algoritmo proposto

De forma a avaliar se o DAG gerado pelo programa estava correto, foram implementadas as duas invariantes a seguir. Elas são importantes pois somente para grafos pequenos é possível analisar manualmente o DAG gerado para procurar possíveis erros.

3.3.1 Número de vértices

Na discussão a seguir, n denota o número de vértices no DAG original, n_m é o número de vértices no DAG modificado e $|P|$ o número de caminhos entre os vértices 0 e $n-1$.

O DAG modificado deve ter, no mínimo, uma máquina virtual para cada caminho no DAG original. Não só, o repositório também deve ser adicionado. Assim, temos que $n + |P| + 1 \leq n_m$.

Além disso, no pior caso, cada tarefa representada no DAG tem uma máquina virtual só para si. Então, $n_m \leq 2n + 1$.

Unindo as duas inequações temos:

$$n + |P| + 1 \leq n_m \leq 2n + 1$$

3.3.2 Propriedades das MVs

Sejam V o conjunto de vértices do DAG original e M o conjunto de vértices que representam máquinas virtuais no DAG modificado.

Seja G_m o DAG modificado. Temos que

$$G_m[M] = V \bigcup M$$

onde $G_m[M]$ denota o subgrafo induzido pelo conjunto M das MVs.

De forma menos técnica, cada $v \in V$ deve ter uma, e apenas uma, máquina virtual adjacente, dado que precisa ter suas necessidades de software atendidas de forma mínima.

4 Outras modificações

As outras duas modificações mencionadas acima também foram implementadas. Já existiam códigos que as implementavam, contudo eles apresentavam falhas e limitações grandes quanto ao consumo de memória, não sendo capazes de processar grafos com mais de 542 tarefas.

Com essas duas modificações implementadas, foi possível avaliar os ganhos em tempo de escalonamento em tarefas com até 3002 tarefas. Isso representou um aumento de 4800% sobre o limite de 62 tarefas apresentado em [1]. Dessa forma, todas as modificações podem ser avaliadas em uma quantidade muito maior de cenários.

Por fim, das implementações feitas durante esse projeto poderão ser derivadas outras modificações de DAGs.

5 Resultados

Foram gerados 8 DAGs representando a aplicação de Montage desde 62 até 3002 tarefas. A diferença entre esses 8 DAGs é o número de tarefas dependentes da mesma MV no caminho mais à direita do DAG, variando de 1 a 8.

Esses DAGs foram, então, escalonados pelo escalonador HEFT e os tempos para cada uma das modificações foram comparados. Todos os códigos foram executados em um processador i5 com 2.67GHz e 4GB de ram rodando Linux Mint 12.

Os gráficos abaixo resumem o comportamento encontrado.

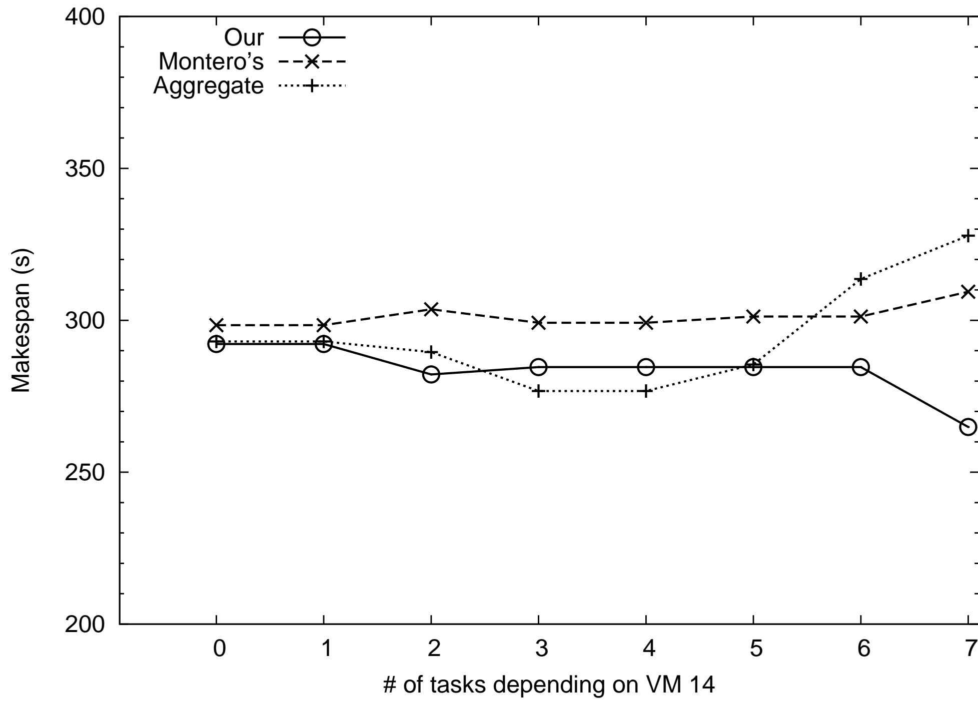

Figura 5: Tempos gerados pelo escalonador HEFT para os DAGS modificados a partir do Montage com 62 tarefas.

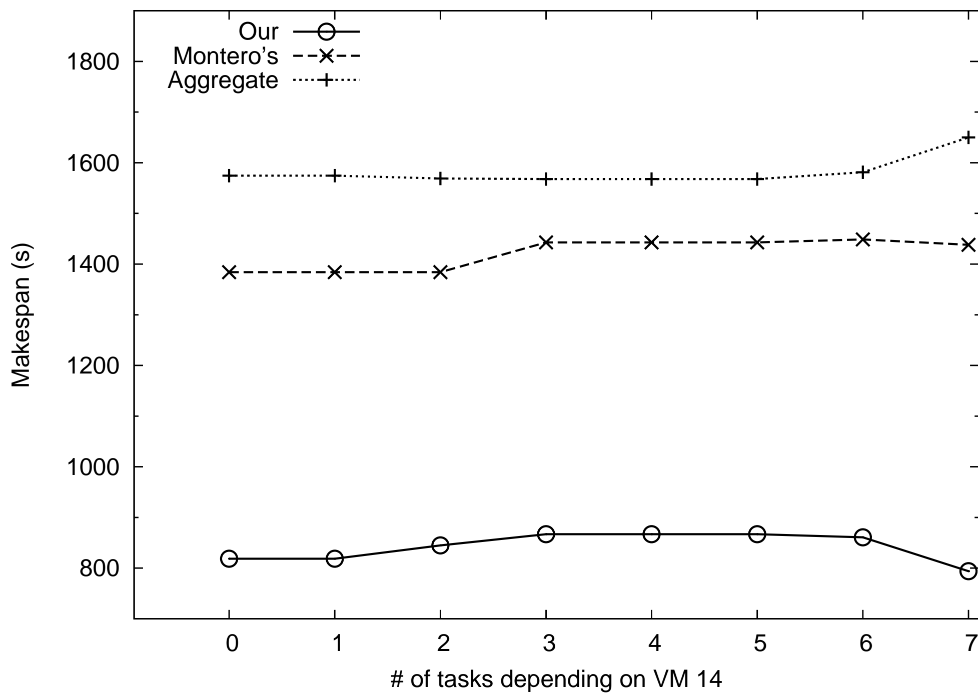

Figura 6: Tempos gerados pelo escalonador HEFT para os DAGS modificados a partir do Montage com 1562 tarefas.

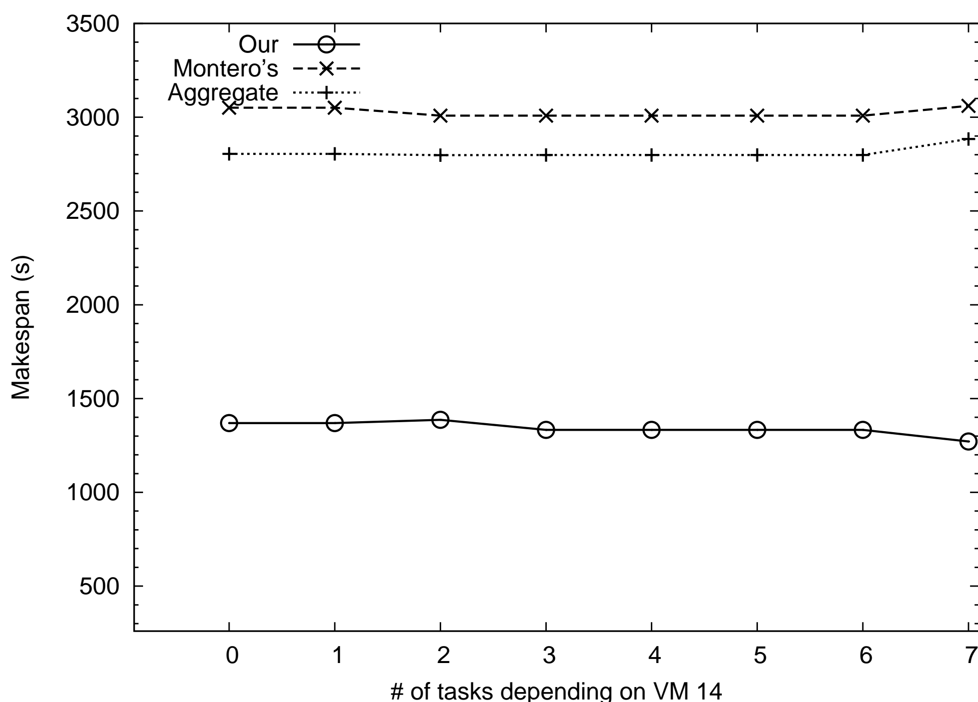

Figura 7: Tempos gerados pelo escalonador HEFT para os DAGS modificados a partir do Montage com 3002 tarefas.

Pode-se ver que, em uma aplicação com poucas tarefas, a abordagem proposta não obtém sempre os menores tempos de escalonamento. Contudo, em aplicações maiores, o tempo de execução dos DAGs modificado chegam a ser 50% menores.

Também, como esperado, o tempo de execução dos DAGs com mais tarefas dependentes da mesma MV é bastante menor.

A tabela abaixo resume as informações de todos os experimentos:

	Uma MV para cada tarefa	Uma única MV	Proposta
Média	1609.322	1556.28	938.5022
Desvio padrão	767.9224	771.2772	277.1883

As figuras e a tabela apresentadas nesta seção comprovam que os objetivos do projeto foram alcançados.

6 Bibliografia

[1] D. M. Batista, C. G. Chaves, and N. L. S. Fonseca. Embedding Software Requirements in Grid Scheduling. In *Proceedings of the ICC 2011: IEEE International Conference on Communications (ICC2011)*, pages 1-6, 2011.