Uma Heurística de Agrupamento de Caminhos para Escalonamento de Tarefas em Grades Computacionais

Bittencourt, L. F.; Madeira, E. R. M.; Cicerre, F. R. L.; Buzato, L. E.

[1]Instituto de Computação – Universidade Estadual de Campinas (UNICAMP)
Caixa Postal 6176 – Campinas – São Paulo – Brasil

{bit,edmundo,fcicerre,buzato}@ic.unicamp.br

Abstract. *Scheduling is fundamental to achieve good performance in process execution in distributed systems. The task scheduling problem is, in general, NP-Complete, leading to the development of heuristic-based algorithms. In a heterogeneous and dynamic system as a computational grid, this problem acquires new variables and becomes more complex. In this paper we present an algorithm for scheduling processes on a grid that supports dependent task execution through hierarchical control sctructures called* controllers. *The performed simulations show that the algorithm gives good performance and, jointly with the middleware, provides scalability and dependability for process execution in a grid environment.*

Resumo. *Escalonamento é fundamental para atingir bom desempenho na execução de processos em sistemas distribuídos. O problema de escalonamento de tarefas é, em geral, NP-Completo, levando ao desenvolvimento de algoritmos baseados em heurísticas. Em um sistema heterogêneo e dinâmico como uma grade computacional, esse problema adquire novas variáveis e torna-se mais complexo. Neste trabalho apresentamos um algoritmo de escalonamento de processos em uma grade computacional que suporta execução de tarefas dependentes através de estruturas de controle hierárquicas chamadas* controladores. *As simulações conduzidas mostram que o algoritmo fornece bom desempenho e, em conjunto com o middleware, provê confiabilidade, alta disponibilidade, escalabilidade e recuperação de falhas para execução de processos em um ambiente de grade computacional.*

1. Introdução

Uma grade computacional é um sistema heterogêneo, colaborativo, geograficamente distribuído, multi-institucional e dinâmico, onde qualquer recurso computacional ligado a uma rede, local ou não, é um potencial colaborador. Grades computacionais são atualmente um grande foco de estudos relacionados à execução de aplicações paralelas, tanto aquelas que demandam alto poder de processamento quanto aquelas que se adaptam bem a ambientes distribuídos. Como os recursos de uma grade pertencem a domínios administrativos diferentes, cada qual com a sua política de uso, cada recurso tem autonomia para participar ou deixar de participar da grade. Essa característica dinâmica e a heterogeneidade tornam o escalonamento de tarefas um problema importante a ser estudado no âmbito das grades computacionais.

O problema de escalonamento envolve uma aplicação e um ambiente alvo, onde a aplicação será executada. O escalonador é a entidade que manipula a aplicação e atribui

cada um de seus componentes a um recurso. O principal objetivo de um escalonador é minimizar o tempo de execução das aplicações (*makespan*), escalonando seus componentes de forma a maximizar o paralelismo na execução das tarefas e minimizar a comunicação, conseqüentemente otimizando a utilização dos recursos. No escalonamento de tarefas uma aplicação (ou processo) é composta de tarefas que têm dependências de dados, onde a execução de cada tarefa deve respeitar as precedências impostas por tais dependências. Então, o escalonador deve tratar das dependências de dados, custos de comunicação entre tarefas e custos de computação das tarefas componentes do processo. O ambiente alvo é uma grade computacional baseada no middleware Xavantes [Cicerre et al. 2005] e seu modelo de programação. Os processos são representados por *grafos acíclicos direcionados* (*directed acyclic graphs* - DAGs).

O algoritmo aqui apresentado visa o escalonamento de tarefas em uma grade composta pelo middleware Xavantes [Cicerre et al. 2005], que através de elementos de processo chamados *controladores* fornece alta distribuição de processos, alta disponibilidade, escalabilidade, tolerância a falhas e recuperação, confiabilidade e desempenho.

O restante deste artigo é organizado da seguinte maneira. O Xavantes é brevemente descrito na Seção 2. Algumas definições acerca do problema de escalonamento são expostas na Seção 3, enquanto trabalhos relacionados são descritos na Seção 4. O algoritmo proposto é apresentado na Seção 5. Resultados experimentais são mostrados na Seção 6. Conclusão e trabalhos futuros finalizam o artigo na Seção 7.

2. Descrição da Infra-estrutura

O Xavantes foi desenvolvido especificamente para a execução de workflows [Cicerre et al. 2005], diferentemente da grande maioria dos middlewares para grades. Estes foram desenvolvidos visando a execução de sacos de tarefas e, quando existente, o suporte a execução de tarefas acopladas depende de extensões desenvolvidas posteriormente, muitas vezes sem suporte completo para workflows. O Xavantes é composto por um modelo de programação, para especificação de processos estruturados, e por uma infra-estrutura, que gerencia tais processos em um ambiente de grades computacionais composto por grupos de recursos altamente distribuídos, heterogêneos e autônomos.

2.1. Modelo de Programação

O modelo de programação do Xavantes permite a especificação de aplicações como processos estruturados, contendo uma hierarquia de estruturas de controle para determinar o fluxo de controle. Um processo pode ser composto por outros processos, controladores e atividades, chamados de elementos de processo. Uma atividade é uma tarefa atômica de um processo, a ser executada em uma única máquina. Um controlador é uma estrutura de controle que organiza a ordem de execução de elementos de processo interiores a ele. Controladores podem ser aninhados, permitindo especificação hierárquica de controle, similarmente às linguagens de programação estruturadas.

Existem vários tipos de controladores, sendo estes classificados como paralelos ou seqüenciais, quanto ao paralelismo, e como simples, condicionais ou interativos, quanto ao controle de fluxo. Os tipos de controladores seqüenciais são similares às estruturas de controle das linguagens de programação estruturadas: *block*, *switch*, *for* e *while*. Os tipos paralelos equivalem aos seus respectivos tipos seqüenciais, mas executam seus elementos

internos simultaneamente: *par*, *parswith*, *parfor* e *parwhile*. Quando aninhados eles podem representar controles mais expressivos, como execução paralela de várias seqüências de atividades. Nos controladores paralelos é possível especificar a condição de junção de fluxos (*join*). Na Figura 1, os rótulos dos nós e arestas representam custos de computação e comunicação, respectivamente. Nessa figura, o retângulo 1 representa um controlador paralelo *par*, contendo as atividades (ou tarefas) 7 e 8, e o retângulo 4 representa um controlador seqüencial *block*, contendo as atividades 2, 5, 10 e o controlador *par* 1.

2.2. Infra-estrutura

O middleware Xavantes organiza os recursos da grade em grupos, considerando que máquinas e repositórios de dados que estão próximos, tais como recursos em redes locais e clusters, compõem cada grupo. O principal objetivo desse middleware é oferecer alto desempenho e confiabilidade na execução de processos. Ele gerencia processos estruturados em um ambiente de grade escalonando, distribuindo e executando seus subprocessos, controladores e atividades hierarquicamente nos recursos disponíveis, em um ou mais grupos, de acordo com a estrutura aninhada desses elementos de processo.

Em cada grupo há um *Group Manager* (GM), um ou mais *Process Managers* (PM) e um ou mais *Activity Managers* (AM). O GM é responsável por manter informações sobre a disponibilidade de recursos dentro do grupo, como capacidade de processamento atual, largura de banda e tempo estimado de fila de execução. Além disso, o GM mantém referências aos GMs adjacentes e informações sobre disponibilidade de parte dos recursos desses grupos. Em cada grupo existem um ou mais PMs, que são responsáveis por instanciar e executar os processos, escalonar os elementos de processo e enviá-los para outros recursos. Se um processo contém subprocessos, estes podem ser enviados a outros PMs ou podem ser executados pelo próprio PM que está gerenciando a execução do processo pai. As atividades são enviadas para serem executadas nos recursos (AMs) escolhidos pelo escalonador, que também são responsáveis por manter informações sobre as atividades atribuídas a ele e informar o GM do grupo sobre sua disponibilidade. A Figura 2 mostra como os recursos são organizados pelo Xavantes.

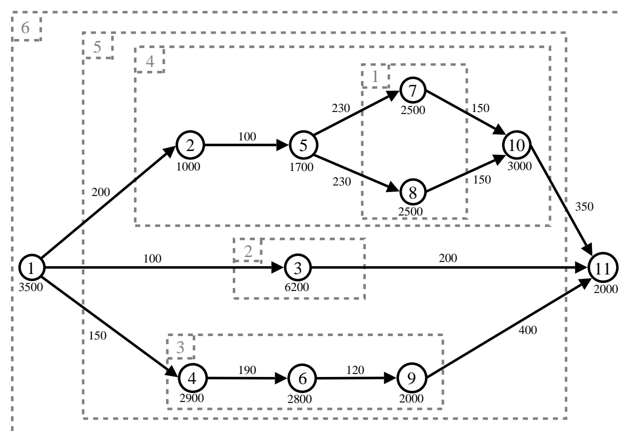

Figura 1. Exemplo de DAG representando tarefas, dependências e controladores.

Segundo [Cicerre et al. 2005], a execução hierárquica de processos estruturados em vários grupos autônomos e distribuídos de recursos ampliam a escalabilidade da grade.

O suporte explícito à execução de processos permite mais eficiência no fornecimento de tolerância a falhas e no escalonamento de processos, melhorando a confiabilidade e o desempenho.

2.3. Escalonamento no Xavantes

Para que um processo possa ser executado potencialmente em qualquer recurso da grade, os grupos são conectados entre si, direta ou indiretamente, através dos GMs. Cada GM mantém informações da disponibilidade de parte dos recursos dos grupos adjacentes, para que o escalonador possa decidir se é vantajoso executar um processo ou tarefa em recursos de outro grupo. A Figura 2 também mostra a conexão entre os GMs.

Dado um processo especificado pelo modelo de programação, o primeiro passo para escaloná-lo é transformá-lo em um DAG. Os grafos que podem ser especificados pelo modelo de programação do Xavantes são aqueles que têm um *join* para cada *fork*, em seqüência. Um exemplo de grafo é mostrado na Figura 1, onde cada *fork* tem seu *join* correspondente. Ainda, *forks* e *joins* devem ser aninhados, isto é, um *fork* deve sempre ter seu *join* correspondente interno ao *fork-join* pai. Essa é uma restrição do modelo de programação análoga à restrição encontrada nas linguagens de programação estruturadas, onde um comando *for* aninhado a um comando externo deve iniciar e terminar dentro desse comando externo. Essa restrição pode ser superada utilizando variáveis compartilhadas dentro dos controladores na especificação do processo.

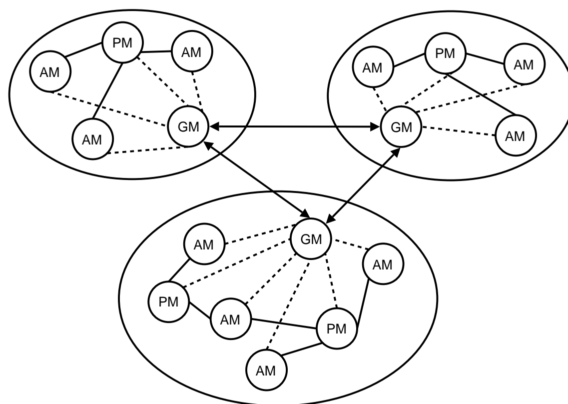

Figura 2. Organização dos recursos no middleware Xavantes.

O modelo de programação fornece para o escalonador o número de instruções de cada tarefa e a quantidade de dados que será transmitida entre as tarefas (dependência de dados). Para determinar a capacidade dos recursos, *benchmarks* são executados. Durante a execução de um processo, as tarefas estão subordinadas a controladores, que são os elementos de controle responsáveis pela execução das tarefas. Controladores são elementos especificados no modelo de programação que permitem alta distribuição dos processos. O *escalonamento de controladores* é o passo do algoritmo proposto que visa minimizar o *overhead* de comunicação gerado pelos controladores. O *overhead* gerado por esses elementos é justificado pelas vantagens oferecidas por eles, como alta disponibilidade, escalabilidade, tolerância a falhas e recuperação, confiabilidade e desempenho. Qualquer

comunicação entre duas tarefas é feita através de seus controladores. Maiores detalhes sobre o Xavantes estão em [Cicerre et al. 2005].

A Figura 1 mostra uma representação de um processo e seus controladores. Rótulos dos nós são números de instruções de cada tarefa (custos de computação) e rótulos das arestas são dados transmitidos (custos de comunicação). A unidade dos custos de computação é o número estimado de instruções da tarefa, considerando que cada operação no código de alto nível pode ser mapeada num número de instruções e que *benchmarks* são executados nos recursos. Os controladores são representados pelos retângulos e são obtidos do modelo de programação. Nesse exemplo, a tarefa 1 é a primeira a ser executada, tendo as tarefas 2, 3 e 4 como dependentes. Estas só podem iniciar as suas execuções após receberem os dados computados pela tarefa 1. As outras dependências se comportam de forma análoga. Em relação aos controladores, as tarefas 7 e 8 estão subordinadas ao controlador 1, a tarefa 3 ao controlador 2, as tarefas 4, 6 e 9 ao controlador 3 e assim por diante. Note que, por exemplo, a comunicação entre os nós 6 e 9 deve passar pelo controlador 3. Então, se esses nós executam no recurso r_m e o controlador 3 executa no recurso r_n, o nó 6 deve enviar os dados para o controlador 3, no recurso r_n, e este deve enviar os dados ao nó 9, no recurso r_m. Nesse caso a comunicação entre os nós 6 e 9, ambos executando no recurso r_m, não teria custo zero. O *escalonamento de controladores* promovido pelo algoritmo proposto tem como objetivo minimizar comunicações desse tipo, evitando assim um *overhead* alto de comunicação. O escalonamento de controladores é feito com todas as tarefas já escalonadas.

Os PMs do grupo são responsáveis por receber os processos e enviá-los ao escalonador. Em seguida, o escalonador obtém as informações sobre disponibilidade de recursos na grade, do seu e de outros grupos, através do GM do seu grupo. Finalmente, o PM envia as tarefas e controladores para serem executados nos AMs, de acordo com o escalonamento retornado pelo algoritmo aqui proposto. A alocação de recursos é requisitada ao GM do grupo, caso os recursos estejam no mesmo grupo, ou a um GM externo, caso os recursos estejam em outro grupo.

3. Definições

Um processo composto por tarefas é representado por um DAG $G = (N, E)$, onde:

- N é o conjunto de $n = |N|$ nós, e $n_i \in N$ representa uma tarefa atômica que deve ser executada em um recurso.
- E é o conjunto de $e = |E|$ arestas direcionadas, e $e_{i,j} \in E$ representa dependência de dados entre n_i e n_j. Isso implica que n_j não pode iniciar sua execução antes que n_i termine e envie os dados necessários a n_j.

Um nó sem predecessores é chamado de *nó de entrada* e um nó sem sucessores é chamado de *nó de saída*. Esses nós podem ser convenientemente criados com custos de computação e comunicação iguais a zero. Cada nó (ou tarefa) do grafo é rotulado com o custo de computação e cada aresta é rotulada com o custo de comunicação. Essa representação mostra os componentes que formam a aplicação e como esses componentes interagem entre si. O modelo de programação usado pelo Xavantes permite que essa representação seja utilizada, transformando os processos especificados pelo modelo em DAGs, com estes sendo utilizados como entrada do escalonador.

O ambiente computacional considerado é um conjunto R de $r = |R|$ recursos heterogêneos que podem executar as tarefas representadas pelo DAG. Esses recursos estão interconectados por uma rede e são capazes de transmitir e processar dados simultaneamente, porém só podem executar uma tarefa de cada vez. O custo de comunicação entre duas tarefas que executam no mesmo recurso é igual a zero.

Existem, ainda, os DAGs condicionais e os DAGs dinâmicos. Os primeiros determinam se uma tarefa será executada ou não dependendo dos resultados de outras tarefas, enquanto os últimos mudam de acordo com resultados dos nós executados, permitindo aumentar a complexidade dos processos suportados.

4. Trabalhos Relacionados

O problema de escalonamento de tarefas é NP-Completo, exceto em alguns casos restritos [El-Rewini et al. 1995], então os esforços relacionados ao desenvolvimento de algoritmos de escalonamento de tarefas estão concentrados em heurísticas. O escalonamento de tarefas é um problema amplamente estudado em sistemas homogêneos [Hakem e Butelle 2005, Kwok e Ahmad 1996, Yang e Gerasoulis 1994, El-Rewini et al. 1995]. Com o advento dos sistemas heterogêneos, passou a ser necessário considerar a heterogeneidade dos recursos e da largura de banda de comunicação [Boeres et al. 2004, Hagras e Janeček 2004, Topcuoglu et al. 2002, Kwok e Ahmad 1998, Sakellariou e Zhao 2004].

Uma grade computacional é um sistema heterogêneo, dinâmico e amplamente distribuído. Essas características dão às grades algumas peculiaridades em relação aos sistemas heterogêneos anteriores, que devem ser tratadas pelo escalonador e/ou pelo middleware. O estudo de questões intrínsecas às grades computacionais tem recebido grande atenção da comunidade científica atualmente [Cicerre et al. 2005, Goldchleger et al. 2004, Foster 2005, Cooper et al. 2004, Fujimoto e Hagihara 2003, Zhang et al. 2005, Ernemann et al. 2004, Fujimoto e Hagihara 2004].

O escalonamento de tarefas em grades computacionais é um escalonamento em um sistema heterogêneo. Entretanto, o escalonador e o middleware devem tratar problemas existentes em grades que não eram tratados por escalonadores desenvolvidos para sistemas heterogêneos não dinâmicos e restritamente distribuídos. Nestes, o algoritmo de escalonamento precisa apenas determinar em qual recurso cada tarefa será executada, sem levar em consideração a topologia ou a variação de desempenho dos recursos, que são dedicados. *List scheduling* [Topcuoglu et al. 2002], *clustering* [Boeres et al. 2004] e *task duplication* [Hagras e Janeček 2004] são algumas técnicas utilizadas em algoritmos de escalonamento de tarefas tanto para sistemas heterogêneos como homogêneos.

A dinamicidade da grade e sua característica amplamente distribuída requerem que o escalonamento de tarefas tenha uma estratégia que promova também confiabilidade, evitando perda de dados computados e de processamento já realizado. Em [Cooper et al. 2004], são propostas estratégias de escalonamento para o projeto GraDS, incluindo uma abordagem para escalonamento de workflows e re-escalonamento de aplicações. Composição de serviços em grades e escalonamento direcionados a workflow são estudados e algoritmos são propostos em [Zhang et al. 2005]. Em [Vianna et al. 2004] é apresentada uma ferramenta para auxiliar o projeto e avaliação de políticas de escalonamento adequados à execução de aplicações paralelas em grades com-

putacionais. Em [Fujimoto e Hagihara 2003], um algoritmo de aproximação para escalonamento de tarefas em grades é apresentado, utilizando como critério de desempenho e aproximação o consumo de recursos na grade.

DAG Manager (DAGMan) [Frey 2002] é o gerenciador de execução de DAGs do Condor [Litzkow et al. 1988], um conhecido gerenciador de recursos entre domínios utilizado em grades [Thain et al. 2002]. Dois trabalhos futuros citados no trabalho DAGMan são o suporte a DAGs condicionais e o suporte a DAGs dinâmicos. Essas são duas funcionalidades já oferecidas pelo middleware Xavantes e seu modelo de programação, mas que ainda não são tratadas pelo algoritmo proposto. Atualmente, nós em grafos condicionais cujas condições de execução são falsas podem ser simplesmente descartados, enquanto nós criados em grafos dinâmicos após o início da execução do processo podem ser escalonados trivialmente na melhor máquina disponível no momento. A incorporação de um tratamento mais elaborado a esses dois tipos de grafos no algoritmo proposto é considerada para trabalho futuro.

Utilizando técnicas de escalonamento em sistemas heterogêneos e tendo o middleware Xavantes como base, o algoritmo aqui apresentado considera a heterogeneidade do sistema para promover o escalonamento de tarefas, com recuperação e tolerância a falhas fornecidos pelo middleware Xavantes. Em uma visão geral, o algoritmo seleciona um caminho do DAG, fazendo uma busca em profundidade, e escalona seus nós no mesmo processador, com o objetivo de eliminar comunicação entre nós que têm dependência de dados e entre nós e controladores.

5. Algoritmo Proposto

O algoritmo proposto usa as seguintes definições:

- $suc(n_i)$ é o conjunto de sucessores do nó n_i no grafo acíclico direcionado.

- $pred(n_i)$ é o conjunto de predecessores do nó n_i no grafo acíclico direcionado.

- *Weight (Custo de Computação)*

$$w_i = \frac{instruções_i}{processamento_r}$$

Representa o custo de computação (tempo de execução) do nó i no recurso r, onde $processamento_r$ é o poder de processamento do recurso r, em instruções por segundo.

- *Custo de Comunicação*

$$c_{i,j} = \frac{dados_{i,j}}{banda_{r,t}}$$

Representa o custo de comunicação (tempo de transmissão dos dados) entre os nós i e j, usando o link entre os recursos r e t. Se $r = t$, $c_{i,j} = 0$.

- *Prioridade*

$$P_i = w_i + \max_{n_j \in suc(n_i)} (c_{i,j} + P_j)$$

Representa o nível de prioridade do nó n_i. A prioridade do nó de saída é $P_{\text{saída}} = w_{\text{saída}}$.

• *Earliest Start Time*

$$EST(n_i, r_k) = \max\{tempo(r_k), \max_{n_h \in pred(n_i)} (EST_h + w_h + c_{h,i})\}$$

Representa o menor tempo inicial possível para o nó n_i no recurso r_k. Nesta definição, $tempo(r_k)$ representa o tempo em que o recurso r_k estará disponível para executar uma tarefa. O EST da primeira tarefa do processo é o tempo em que o processador que contém $n_{entrada}$ estará pronto para executá-la.

• *Estimated Finish Time*

$$EFT(n_i, r_k) = EST(n_i, r_k) + \frac{instruções_{n_i}}{processamento_{r_k}}$$

Representa o tempo estimado de término do nó n_i no recurso r_k.

Os valores iniciais dos atributos citados acima são calculados supondo um sistema homogêneo virtual composto por um número ilimitado de processadores. Esses processadores têm o poder computacional do melhor processador disponível no sistema heterogêneo real, e todos os links têm a largura de banda mais alta disponível no sistema heterogêneo real. Cada tarefa é escalonada em um processador diferente nesse sistema virtual, então o algoritmo computa os valores dos atributos de cada nó.

5.1. Seleção de Tarefas e Agrupamento

Na seleção de tarefas e agrupamento (*clustering*), o algoritmo escolhe as tarefas que formarão cada cluster. O recurso que comportará cada cluster é escolhido de acordo com a estratégia de seleção de recursos apresentada na seção 5.2. A estratégia adotada no agrupamento de tarefas objetiva reduzir a comunicação entre tarefas acopladas e entre tarefas e controladores.

Computados os valores iniciais dos atributos, é possível utilizar a prioridade dos nós para selecionar as tarefas. O primeiro nó selecionado para compor um cluster cls_k é o nó n_i não escalonado com maior prioridade. O próximo nó selecionado para compor cls_k é o nó $n_s \in suc(n_i)$ com o maior $P_s + EST_s$. Essa soma representa o tamanho do maior caminho que inicia em $n_{entrada}$, passa por n_s e termina em $n_{\text{saída}}$.

No início do escalonamento o primeiro nó a ser selecionado será o nó de entrada, já que a função recursiva de cálculo de prioridade acumula os valores dos sucessores, ficando o nó de entrada com a maior prioridade. Após o nó de entrada ser selecionado, ele é adicionado ao cluster cls_0. Agora o algoritmo realiza uma busca em profundidade, partindo do nó de entrada, selecionando sempre o sucessor n_s que tem o maior $P_s + EST_s$ e adicionando-o ao cluster cls_0. A busca em profundidade continua até que o nó de saída seja alcançado, que também é adicionado ao cluster cls_0. Considerando os recursos da Tabela 1 e o grafo da Figura 1, o algoritmo gera os clusters mostrados na Tabela 2. O primeiro nó a ser adicionado ao cluster 0 é o nó 1, que é o nó de entrada. O próximo nó selecionado e adicionado ao cluster 0 é o nó 2, já que $P_2 + EST_2 > P_3 + EST_3$ e $P_2 + EST_2 > P_4 + EST_4$. Seguindo o mesmo raciocínio, os próximos nós a serem

adicionados ao cluster 0 são o nós 5, 7, 10 e 11, que é o nó de saída. Note que este primeiro cluster é composto por todos os nós que compõem o caminho crítico do DAG inicial. O caminho crítico é determinante no tempo de execução do processo, então é importante escaloná-lo em um recurso que proporcione um bom desempenho. Então, o cluster 0 é o primeiro a ser escalonado, de acordo com a estratégia de seleção de recursos.

Algoritmo 1 gera_próximo_agrupamento

```
1: n ⇐ nó não escalonado com a maior Prioridade.
2: cluster ⇐ cluster ∪ n
3: while (n tem sucessores não escalonados) do
4:    n_suc ⇐ sucessor_i de n com maior P_i + EST_i
5:    cluster ⇐ cluster ∪ n_suc
6:    n ⇐ n_suc
7: return cluster
```

Note que após o escalonamento do primeiro cluster, que contém o caminho crítico do grafo inicial, a supressão da comunicação entre os nós que compõem esse cluster pode fazer com que um outro caminho se torne o caminho crítico no novo grafo. Por esse motivo, a cada cluster escalonado o algoritmo atribui o valor 0 às arestas existentes entre tarefas no mesmo recurso e recalcula o Weight, o EST e o EFT de cada tarefa no grafo.

Para formar o próximo cluster, o algoritmo seleciona o nó n_i não escalonado com maior prioridade, adicionando-o ao cluster cls_k. Partindo desse nó o algoritmo efetua uma busca em profundidade de forma análoga à realizada durante a formação do primeiro cluster, porém a busca cessa quando atinge um nó sem sucessores não escalonados, incluindo-o no cluster. Então o cluster é escalonado e os atributos do grafo recalculados.

No grafo da Figura 1, com o nó 1 já escalonado, o nó 4 é o nó com maior prioridade. O nó 4 é então adicionado ao cluster 1 e, de forma trivial, o cluster é completado com os nós 6 e 9. Note que, em virtude da estrutura dos grafos que são gerados pelo modelo de programação, onde *forks* e *joins* estão presentes em pares, cada cluster tem apenas uma tarefa que o sucede no grafo. Os próximos clusters gerados no exemplo são: cluster 2, com o nó 8, e o cluster 3, com o nó 3. O Algoritmo 1 mostra essa estratégia, que consome tempo $O(n)$.

Tabela 1. Recursos usados no exemplo da Figura 1. A largura de banda entre todos os recursos é fornecida.

Recurso	Processamento	Banda			
0	133	∞	10	5	5
1	130	10	∞	5	5
2	118	5	5	∞	10
3	90	5	5	10	∞

5.2. Seleção de Recursos

Na heurística de agrupamento de caminhos proposta na seção anterior, cada cluster contém um caminho do grafo que será escalonado no mesmo recurso. O fator que determina em qual recurso um cluster será escalonado é o EST do sucessor do último nó

Tabela 2. Atributos dos nós após executar o algoritmo para o grafo da Figura 1.

Nó	Recurso	Prioridade	EST	EFT	w	Cluster
1	0(133)	206.0	0	26.3	26.3	0
2	0(133)	159.7	26.3	33.8	7.5	0
3	2(118)	81.7	46.3	98.8	52.5	3
4	1(130)	143.9	41.3	63.6	22.3	1
5	0(133)	142.2	33.8	46.6	12.8	0
6	1(130)	103.1	63.6	85.1	21.5	1
7	0(133)	106.4	46.6	65.4	18.8	0
8	0(133)	106.4	65.4	84.2	18.8	2
9	1(130)	70.1	85.1	100.5	15.4	1
10	0(133)	72.6	84.2	106.8	22.6	0
11	0(133)	15.0	140.5	155.5	15.0	0

do cluster considerado. Um cluster cls_k é escalonado no recurso que minimiza o EST do sucessor de cls_k. Para calcular esse valor, o algoritmo primeiro calcula o Estimated Finish Time (EFT) de cada nó do cluster. É importante salientar que se o recurso contém tarefas do mesmo processo do cluster que está sendo escalonado, é necessário antes ordenar as tarefas para que suas precedências não sejam violadas, e então efetuar o cálculo dos EFTs. É fácil ver que se as tarefas estiverem ordenadas em ordem decrescente de prioridade, então suas precedências são satisfeitas.

Por exemplo, considere um recurso r_n com as tarefas (n_3, n_6, n_{10}) escalonadas, um cluster $cls_k = (n_8, n_{12}, n_{13})$, e as prioridades $p_3 > p_6 > p_8 > p_{10} > p_{12} > p_{13}$. Para que seja possível determinar os EFTs dos nós do cluster cls_k no recurso r_n, é preciso considerar o escalonamento $(n_3, n_6, n_8, n_{10}, n_{12}, n_{13})$.

Com os EFTs das tarefas determinados, o cálculo do EST do sucessor de cls_k que está sendo escalonado é direto: $EST(suc(n_{k_z})) = EFT(n_{k_z}) + c(n_{k_z}, suc(n_{k_z}))$, onde n_{k_z} é o último nó do cluster cls_k. O primeiro cluster, cls_0, não possui sucessores, então o recurso escolhido para executar o cluster cls_0 é o que propicia menor $EFT(n_{0_z})$. Os outros clusters têm somente um sucessor e esse sucessor já está escalonado, pois, por construção, o último nó de um cluster não tem sucessores não escalonados.

Usando a estratégia de seleção de recursos descrita, o cluster 0 gerado para o DAG da figura 1 é escalonado no recurso 0 e o cluster 1 no recurso 1. O cluster 2 é escalonado no recurso 0, com o nó 8 inserido antes do nó 10, obedecendo as precedências das tarefas. Finalmente, o cluster 3 é escalonado no recurso 2. A estratégia de seleção de recursos é mostrada no Algoritmo 2, que tem complexidade $O(rn)$.

5.3. Escalonamento de Controladores

Controladores têm um importante papel na especificação e execução dos processos, então devem ser escalonados procurando oferecer respostas imediatas às tarefas subordinadas a eles. Devemos atribuir cada controlador a um processador que minimize a comunicação com seus nós e com seus subcontroladores. A criação de clusters baseados em tarefas seqüenciais favorece na redução do *overhead* de comunicação dos controladores.

Algoritmo 2 seleciona_melhor_recurso

1: **for all** $r \in recursos$ **do**
2: $escalonamento \Leftarrow$ Insere $cluster$ em $escalonamento_r$
3: calcula_EFT($escalonamento$);
4: $tempo_r \Leftarrow$ calcula_EST(sucessor($cluster$))
5: **return** recurso r com menor $tempo_r$

O controlador a ser escalonado é uma escolha aleatória entre aqueles que têm todos seus subcontroladores já escalonados. Assim, o primeiro controlador a ser escalonado será um dos que não têm subcontroladores. Para decidir em que recurso um controlador irá executar, o algoritmo calcula a *comunicação externa* e a *comunicação interna* do controlador. Na comunicação externa estão incluídas as arestas que ultrapassam os limites do controlador, que na representação gráfica são aquelas que interceptam um dos lados do retângulo do controlador. Na comunicação interna estão incluídas apenas as arestas que não interceptam nenhum dos lados do retângulo do controlador considerado. Então, para calcular a comunicação de um controlador, os nós são separados em duas classes: nós apenas com comunicação interna e nós com comunicação externa ao controlador.

Para calcular a comunicação de um nó n_i com comunicação interna é suficiente somar os custos de comunicação de n_i com seus predecessores e com seus sucessores. Note que se dois nós n_i e n_j estão subordinados diretamente ao mesmo controlador ctr_k e existe uma aresta entre eles, duas comunicações são necessárias: n_i envia os dados a ctr_k, então ctr_k envia os dados a n_j. Novamente, uma aresta entre dois elementos que executam em um mesmo recurso tem custo zero.

Para os nós na outra classe devemos considerar a comunicação entre controladores. Considere um nó n_i dentro dos controladores ctr_4, ctr_3, ctr_2, ctr_1, com $n_i \in ctr_1 \in ctr_2 \in ctr_3 \in ctr_4$. O custo de comunicação entre um nó $n_j \in ctr_4$ e n_i é $c_{j,ctr4} + c_{ctr4,ctr3} + c_{ctr3,ctr2} + c_{ctr2,ctr1} + c_{ctr1,i}$. Então, se cada um desses controladores estiver em um recurso diferente, o custo de comunicação pode ser proibitivo, tornando a execução seqüencial das tarefas mais eficiente. Ainda, quando um controlador ctr_k está sendo escalonado, o controlador que contém ctr_k ainda não foi escalonado, sendo desconhecida a largura de banda entre eles. Para a comunicação que entra ou sai de ctr_k, a comunicação considerada é aquela entre ctr_k e o nó externo, já escalonado, que comunica com ctr_k. O recurso selecionado para executar um controlador é aquele que minimiza a soma das comunicações dos nós nas duas classes.

Na Figura 1, o primeiro controlador a ser escalonado é uma escolha aleatória entre os controladores 1, 2 e 3. Vamos assumir que o controlador 1 é escolhido. Atribuindo-o ao recurso 0, o custo de comunicação é nulo, já que todos os nós que comunicam com o controlador 1 (nós 5, 7, 8 e 10) estão escalonados no recurso 0. Se o controlador 1 for atribuído ao recurso 1, o custo de comunicação seria $c_{n_5,ctr_1} + c_{ctr_1,n_7} + c_{n_5,ctr_1} + c_{ctr_1,n_8} + c_{n_7,ctr_1} + c_{ctr_1,n_{10}} + c_{n_8,ctr_1} + c_{ctr_1,n_{10}} = 152.0$. Para os recursos 2 e 3, a comunicação seria de 304.5. Então, o controlador 1 é escalonado no recurso 0. Seguindo o mesmo raciocínio, o controlador 2 é escalonado no recurso 0 e o controlador 3 no recurso 1. O controlador 4 é escalonado no recurso 0, pois todos os nós que comunicam com ele estão escalonados nesse recurso. O controlador 5 no recurso 0 teria comunicação de 55.0, no recurso 1 de 85.0 e de 280.0 nos recursos 2 ou 3. Então, o controlador 5 é atribuído ao recurso

0. Finalmente, o controlador 6 é escalonado no recurso 0, pois ele comunica somente com o controlador 5. O escalonamento de controladores, que consome tempo $O(rn^3)$ no pior caso, é mostrado no Algoritmo 3 e o algoritmo de escalonamento é mostrado no Algoritmo 4. O escalonamento resultante para o exemplo é mostrado na Figura 3.

Algoritmo 3 escalona_controladores

1: **while** existem controladores não escalonados **do**
2: $ctr_k \Leftarrow$ controlador sem subcontroladores não escalonados
3: **for all** $r \in recursos$ **do**
4: Atribui ctr_k a r
5: **for all** $n_k \in ctr_k$ **do**
6: **for all** n_j que comunica com n_k **do**
7: **if** $n_j \in ctr_k$ **then**
8: $intComunic_r = intComunic_r + c_{n_k,n_j}$
9: **else if** $n_j \in$ supercontrolador(ctr_k) **then**
10: $extComunic_r = extComunic_r + c_{n_j,ctr_k} + c_{ctr_k,n_k}$
11: $nósComunic_r = intComunic_r + extComunic_r$
12: **for all** $ctr_s \in$ subcontroladores(ctr_k) **do**
13: $subComunic_r = subComunic_r + c_{ctr_k,ctr_s} + c_{ctr_s,ctr_k}$
14: $comunic_r \Leftarrow nósComunic_r + subComunic_r$
15: Escalona ctr_k em recurso($\min\limits_{r \in recursos}(comunic_r)$)

Algoritmo 4 Algoritmo proposto

1: Atribui o DAG ao sistema virtual homogêneo
2: Calcula atributos iniciais das tarefas
3: **while** existem nós não escalonados **do**
4: $cluster \Leftarrow$ gera_próximo_agrupamento()
5: $recurso \Leftarrow$ seleciona_melhor_recurso($cluster$)
6: Escalona $cluster$ em $recurso$
7: Recalcula Weights, ESTs e EFTs
8: escalona_controladores()

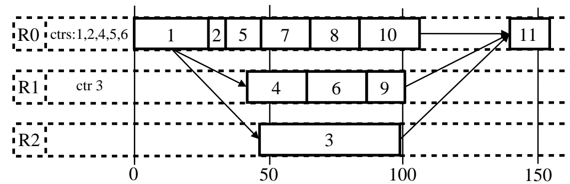

Figura 3. Escalonamento de tarefas e controladores para o grafo da Figura 1.

6. Resultados Experimentais

O algoritmo aqui proposto foi desenvolvido com o intuito de ter um bom desempenho no escalonamento do middleware Xavantes, buscando minimizar o tempo de execução dos processos na grade. A comparação mostrada nesta seção justifica o desenvolvimento de um novo algoritmo em detrimento da utilização de algoritmos de escalonamento existentes. Comparamos o algoritmo proposto com o algoritmo HEFT [Topcuoglu et al. 2002]

e com o algoritmo CPOP [Topcuoglu et al. 2002], que são amplamente utilizados em comparações na literatura, tornando possível a comparação indireta entre o PCH e outros algoritmos. Quinze grafos válidos no modelo de programação do Xavantes foram gerados de forma aleatória, sem a especificação de restrições, para o experimento. Cada grafo foi escalonado 1000 vezes utilizando cada algoritmo, com valores aleatórios em nós e arestas. Os recursos considerados são heterogêneos, assim como a largura de banda entre eles. Ambos algoritmos foram executados com e sem controladores, variando os custos de comunicação e computação. Comunicação baixa significa que todos os custos de comunicação são menores que todos os custos de computação. Comunicação média significa que os custos de computação e comunicação são gerados aleatoriamente dentro do mesmo intervalo. Comunicação alta significa que todos os custos de comunicação são maiores que todos os custos de computação. O algoritmo utilizado para o escalonamento de controladores nos cenários dos algoritmos HEFT, CPOP e Proposto é o mesmo.

As principais métricas de comparação utilizadas na literatura de escalonamento de tarefas são o *Schedule Length Ratio* (SLR) e o *Speedup*:

$$SLR = \frac{makespan}{\displaystyle\sum_{n_i \in CC} \frac{instruções_{n_i}}{processamento_{melhor}}}$$

onde, CC é o conjunto de nós do caminho crítico do grafo inicial e $processamento_{melhor}$ é a capacidade de processamento do melhor recurso disponível. Assim, a soma no denominador representa o custo de computação do caminho crítico no melhor recurso.

$$Speedup = \frac{\displaystyle\sum_{n_i \in V} \frac{instruções_{n_i}}{processamento_{melhor}}}{makespan}$$

O somatório no numerador representa o tempo de execução seqüencial de todas as tarefas no melhor recurso disponível. O número de vezes que um algoritmo resulta em escalonamentos com menor *makespan* também é uma métrica bastante utilizada. Essa métrica deve ser utilizada complementarmente a outras, já que apenas indica se um escalonamento é melhor que outro, mas não esclarece o quanto.

A Figura 4, que apresenta o número de melhores escalonamentos gerados por cada algoritmo, mostra que o algoritmo proposto gera escalonamentos melhores na maioria das execuções com controladores. Com comunicação baixa, em 52.84% das execuções o *makespan* do algoritmo proposto foi menor. Com comunicações média e alta, essas taxas são de 68.51% e 74.2%, respectivamente. Sem considerar controladores, o algoritmo HEFT gera escalonamentos melhores quando a comunicação é baixa em 44.81% do total de execuções, contra 26.54 do CPOP e 13.96% do algoritmo proposto. Com comunicação média e sem controladores, o algoritmo proposto é melhor em 37.03% dos escalonamentos, enquanto HEFT é melhor em 30.43% e CPOP em 16.24% das execuções. Quando a comunicação é alta, o algoritmo proposto é melhor em 44.37% dos casos, contra 20.58% do HEFT e 14.08% do CPOP. Os resultados de empate são representados de duas maneiras: *Empate 1* são escalonamentos onde o algoritmo proposto gerou o melhor *makespan*,

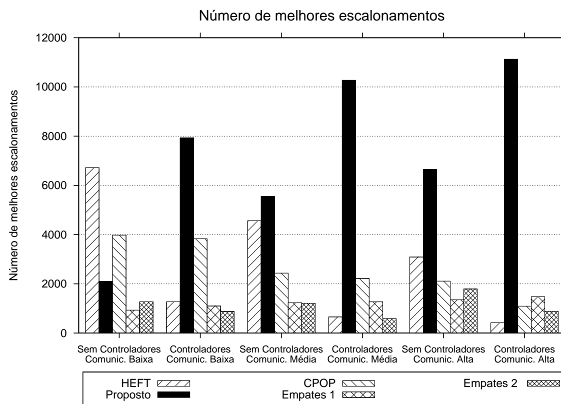

Figura 4. Número de escalonamentos com menor makespan.

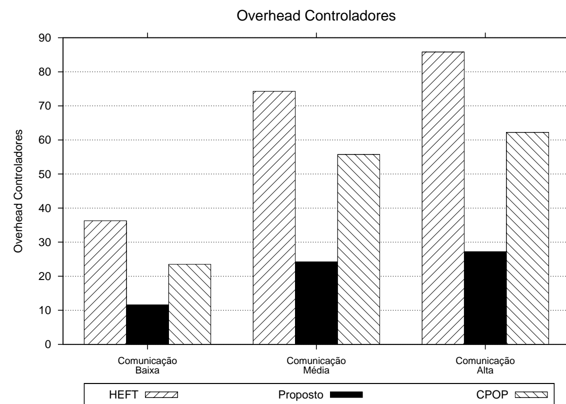

Figura 5. *Overhead* de comunicação dos controladores.

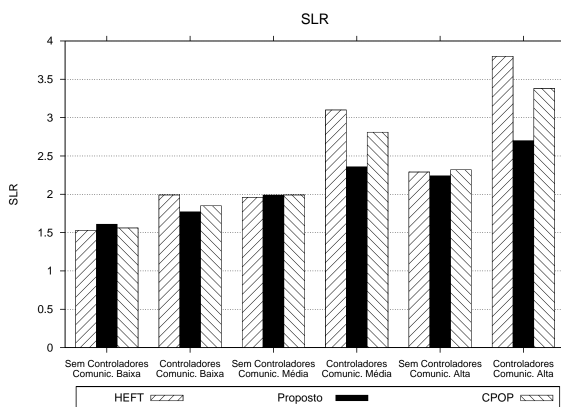

Figura 6. SLR médio.

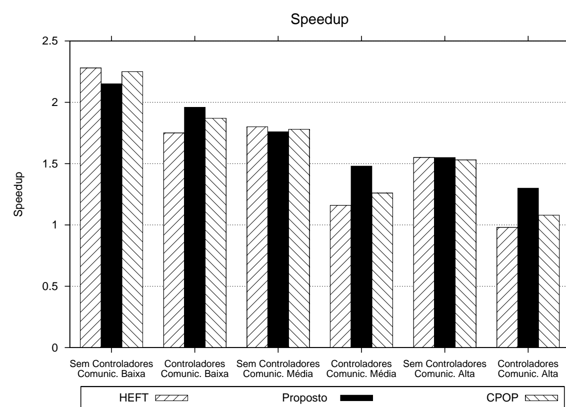

Figura 7. *Speedup* médio.

porém outro algoritmo também o fez, enquanto *Empate 2* são resultados onde CPOP e HEFT geraram escalonamentos iguais e melhores que o algoritmo proposto.

A Figura 5 mostra a porcentagem de *overhead* na execução com controladores em relação à execução sem controladores. O *overhead* nos escalonamentos gerados pelo HEFT é da ordem de 3 vezes o *overhead* gerado pelo algoritmo proposto, enquanto o *overhead* do CPOP é da ordem de 2 vezes o *overhead* do algoritmo proposto.

A Figura 6 mostra o SLR médio das execuções de cada algoritmo. Nessa figura podemos ver que a diferença entre os SLRs médios dos três algoritmos sem considerar controladores é baixa na maioria das execuções. Quando consideramos controladores, essa diferença é maior, com o algoritmo proposto resultando em SLRs menores. Isso significa que a maioria dos *makespans* gerados pelo algoritmo proposto é consideravelmente menor que os *makespans* gerados pelo HEFT e pelo CPOP.

A Figura 7 mostra a média dos *speedups* das execuções da cada algoritmo. Quando controladores não são considerados, na maioria dos casos HEFT gera *speedups* maiores que CPOP, que gera *speedups* maiores que o algoritmo proposto, porém com pouca diferença. Quando controladores são considerados, o algoritmo proposto tem uma média de *speedup* maior, com uma boa diferença. Então, grande parte dos *makespans* gerados pelo algoritmo proposto são consideravelmente menores que aqueles gerados pelo

HEFT e pelo CPOP, quando controladores são considerados. Sem considerar controladores, a diferença entre as médias de *speedup* é pequena, então há uma pequena diferença entre os *makespans* gerados por cada algoritmo em cada execução.

7. Conclusão

Este artigo apresenta uma heurística para escalonamento de tarefas no Xavantes [Cicerre et al. 2005], um middleware para grades computacionais desenvolvido para a execução de processos tipo workflow. O algoritmo aqui proposto pode ser utilizado para escalonamento de tarefas tanto em sistemas heterogêneos como homogêneos. Entretanto, seu objetivo é o escalonamento no middleware Xavantes considerando controladores, fornecendo alta disponibilidade, escalabilidade, tolerância a falhas e recuperação, confiabilidade e desempenho. As simulações realizadas mostram que, para os grafos de tarefas válidos no Xavantes, a estratégia de construir clusters de tarefas baseados em caminhos do grafo proporciona bom desempenho quando controladores são considerados e também quando a comunicação é alta e controladores não são considerados, com complexidade $O(rn^3)$, onde r é o número de recursos e n o número de nós do grafo.

Trabalhos futuros incluem escalonamento dinâmico de tarefas, onde apenas parte do grafo é escalonada inicialmente, deixando outros nós para serem escalonados durante a execução do processo. Nesse tipo de escalonamento devemos ponderar que porção do grafo deve ser escalonada inicialmente. A adaptação do algoritmo para tratamento de grafos condicionais e grafos dinâmicos também será alvo de estudo.

Agradecimentos

Os autores agradecem ao CNPq, à FAPESP, e ao projeto AgroFlow pelo apoio financeiro.

Referências

Boeres, C., Filho, J. V., e Rebello, V. E. F. (2004). A cluster-based strategy for scheduling task on heterogeneous processors. In *16th Symposium on Computer Architecture and High Performance Computing*, pages 214–221. IEEE Computer Society.

Cicerre, F. R. L., Madeira, E. R. M., e Buzato, L. E. (2006). A hierarchical process execution support for grid computing. *Concurrency and Computation: Practice and Experience*, 18(6):581–594.

Cooper, K., Dasgupta, A., Kennedy, K., et al. (2004). New grid scheduling and rescheduling methods in the grads project. In *18th International Parallel and Distributed Processing Symposium, EUA*. IEEE Computer Society.

El-Rewini, H., Ali, H. H., e Lewis, T. G. (1995). Task scheduling in multiprocessing systems. *IEEE Computer*, 28(12):27–37.

Ernemann, C., Hamscher, V., e Yahyapour, R. (2004). Benefits of global grid computing for job scheduling. In *5th International Workshop on Grid Computing, EUA*, pages 374–379. IEEE Computer Society.

Foster, I. (2005). Globus toolkit version 4: Software for service oriented systems. *IFIP International Conference on Network and Parallel Computing*, pages 2–13.

Frey, J. (2002). Condor dagman: Handling inter-job dependencies. http://www.cs.wisc.edu/condor/dagman/.

Fujimoto, N. e Hagihara, K. (2003). Near-optimal dynamic task scheduling of precedence constrained coarse-grained tasks onto a computational grid. In *2nd Int. Symp. on Parallel and Distributed Computing, Slovenia*, pages 80–87. IEEE Computer Society.

Fujimoto, N. e Hagihara, K. (2004). A comparison among grid scheduling algorithms for independent coarse-grained tasks. In *Symposium on Applications and the Internet Workshops, Japão*, pages 674–680. IEEE Computer Society.

Goldchleger, A., Kon, F., Goldman, A., Finger, M., e Bezerra, G. C. (2004). InteGrade: Object-Oriented Grid Middleware Leveraging Idle Computing Power of Desktop Machines. *Concurrency and Computation: Practice and Experience*, 16(5):449–459.

Hagras, T. e Janeček, J. (2004). An approach to compile-time task scheduling in heterogeneous computing systems. In *33rd International Conference on Parallel Processing Workshops, Canadá*, pages 182–189. IEEE Computer Society.

Hakem, M. e Butelle, F. (2005). Dynamic critical path scheduling parallel programs onto multiprocessors. In *19th International Parallel and Distributed Processing Symposium, EUA*. IEEE Computer Society.

Kwok, Y.-K. e Ahmad, I. (1996). Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 7(5):506–521.

Kwok, Y.-K. e Ahmad, I. (1998). Benchmarking the task graph scheduling algorithms. In *IPPS/SPDP, EUA*, pages 531–537.

Litzkow, M. J., Livny, M., e Mutka, M. W. (1988). Condor—A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems, EUA*, pages 104–111.

Sakellariou, R. e Zhao, H. (2004). A hybrid heuristic for dag scheduling on heterogeneous systems. In *18th Int. Parallel and Distributed Processing Symp., EUA*. IEEE.

Thain, D., Tannenbaum, T., e Livny, M. (2002). Condor and the grid. In Berman, F., Fox, G., e Hey, T., editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons Inc.

Topcuoglu, H., Hariri, S., e Wu, M.-Y. (2002). Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distrib. Syst.*, 13(3):260–274.

Vianna, B. A., Fonseca, A. A., Moura, N. T., Menezes, L. T., Mendes, H. A., Silva, J. A., Boeres, C., e Rebello, V. E. F. (2004). A tool for the design and evaluation of hybrid scheduling algorithms for computational grids. In *Proceedings of the 2nd workshop on Middleware for grid computing, Toronto, Canadá*, pages 41–46. ACM Press.

Yang, T. e Gerasoulis, A. (1994). Dsc: Scheduling parallel tasks on an unbounded number of processors. *IEEE Trans. Parallel Distrib. Syst.*, 5(9):951–967.

Zhang, S., Zong, Y., Ding, Z., e Liu, J. (2005). Workflow-oriented grid service composition and scheduling. In *International Symposium on Information Technology: Coding and Computing, EUA*, volume 2, pages 214–219. IEEE Computer Society.