GRAND: Um Modelo de Gerenciamento Hierárquico de Aplicações em Ambiente de Computação em Grade

Patrícia Kayser Vargas
Centro Universitário La Salle
Canoas, RS – Brasil
kayser@unilasalle.edu.br

Inês de Castro Dutra
COPPE/Sistemas – UFRJ
Rio de Janeiro,RJ – Brasil
ines@cos.ufrj.br

Cláudio F. R. Geyer
Instituto de Informática – UFRGS
Porto Alegre, RS – Brasil
geyer@inf.ufrgs.br

Resumo

*Gerenciamento de dados e controle da sobrecarga das máquinas de submissão são dois problemas desafiadores na computação em grade. A tese apresentada neste artigo aborda estes dois problemas, focando em aplicações compostas por um número muito grande de tarefas. Neste contexto, foi proposto e avaliado um novo sistema de gerenciamento de aplicações denominado GRAND (***G***rid* ***R****obust* ***A****pplicatio****n*** ***D****eployment). Algumas das contribuições do modelo proposto são (1) um particionamento flexível da aplicação; (2) a nova linguagem de descrição de aplicação GRID-ADL; e (3) uma hierarquia de gerenciadores para a realização da submissão de tarefas. Um protótipo foi implementado e avaliado mostrando bons resultados relacionados ao gerenciamento de recursos e dados.*

1 Introdução

Uma grade [12] é uma infra-estrutura computacional distribuída utilizada principalmente para aplicações científicas e de engenharia. Um ambiente de grade apresenta vários desafios tais como o controle de um grande número de tarefas e a sua alocação nos nodos da grade. Suportar a submissão simultânea de centenas ou milhares de tarefas tem se tornado um aspecto importante em trabalhos relacionados à grade [25]. Isto é natural considerando-se que quanto mais recursos se tornam disponíveis, maior será a possibilidade de o usuário realizar experimentos mais complexos e computacionalmente intensivos. Suportar este tipo de submissão tem um impacto direto no projeto da arquitetura de escalonamento.

Vários trabalhos têm sido propostos [1, 23], porém, no momento de concepção desta tese, não existia nenhum trabalho na literatura que tratasse apropriadamente todos os aspectos relacionados ao gerenciamento de aplicações e que fosse ao mesmo tempo escalável, ou seja, suportasse o gerenciamento e a submissão das centenas de milhares de tarefas que podem compor uma aplicação. Aplicações compostas de centenas de milhares de tarefas são muito comuns em aplicações tais como simulações de Monte Carlo ou sequenciamento de DNAs. De fato, a maior parte dos trabalhos tem se preocupado mais com a gerência de recursos [19, 13, 28, 5, 27, 6], embora recentemente tenha havido um aumento no número de trabalhos que possuem uma ênfase no controle do ponto de vista da aplicação [17, 3]. Dentre os problemas que merecem maior atenção, procuramos tratar de aspectos relacionados ao gerenciamento de dados e da possível sobrecarga da máquina de submissão. A máquina de submissão (*submit machine*) refere-se à máquina na qual o usuário inicia a submissão ao ambiente de grade de uma aplicação composta por várias tarefas.

Assim, a tese sumarizada neste artigo [29] trata do gerenciamento de aplicações que disparam um número muito grande de tarefas (na ordem de milhares) em um ambiente de grade, focando em aspectos de gerenciamento de dados e sobrecarga de submissão. O principal resultado obtido foi a proposta do modelo de gerenciamento de aplicação GRAND (**G**rid **R**obust **A**pplicatio**n** **D**eployment) [30, 31, 32] e a implementação e avaliação de um protótipo denominado AppMan (**App**lication **Man**ager) [33, 34]. A referida tese foi defendida na COPPE/Sistemas, UFRJ, em março de 2006. Outros resultados foram obtidos após a defesa da mesma [35] e trabalhos relacionados a monitoramento de recursos [21] estão sendo desenvolvidos visando o aprimoramento do escalonamento no AppMan.

As principais premissas assumidas na concepção do modelo proposto são as seguintes: **(1)** o ambiente de execução é heterogêneo; **(2)** um grande número de tarefas pode ser submetido; **(3)** tarefas não se comunicam por troca de mensagens; **(4)** tarefas podem ter dependências devido ao compartilhamento de arquivos, permitindo a modelagem da aplicação como um grafo; **(5)** um grande número de arquivos pode ser manipulado pelas tarefas; **(6)** arquivos grandes podem ser necessários para a computação; **(7)** a infra-estrutura de grade utilizada é segura; **(8)** a infra-estrutura de

grade utilizada permite a descoberta dinâmica de recursos; **(9)** cada nodo da grade possui o seu sistema gerenciador de recursos (RMS ou *resource management system*) local; **(10)** tarefas submetidas para um RMS são executadas localmente ou a migração é transparente.

As idéias centrais na concepção do modelo GRAND são (1) prover um particionamento flexível e automático da aplicação tanto na execução quanto na submissão, e (2) criar uma organização hierárquica onde a carga de gerenciamento da aplicação seja distribuída entre máquinas do mesmo nó da máquina de submissão evitando gargalos e sobrecargas desnecessárias. Uma estrutura hierárquica é a mais apropriada para um ambiente de grade devido a sua escalabilidade. Além disso, obtivemos alguns resultados experimentais [34, 35] para verificar o comportamento da submissão distribuída. Nesses experimentos realizados em um *cluster* gerenciado com o Condor [28], foi possível constatar que, para cargas acima de 1000 tarefas, é significativamente diferente da centralizada, com 95% de confiança. Para esses experimentos, a submissão distribuída produziu um desempenho entre 10% e 31,6% melhor do que a submissão centralizada, onde uma única máquina é responsável por gerenciar e monitorar todas as tarefas submetidas. Recentemente, os grupos que utilizam Condor também constataram a limitação da submissão centralizada e em seus tutoriais sempre aconselham o usuário a submeter tarefas a partir de várias máquinas da rede local [7]. Uma das grandes vantagens do modelo GRAND e do protótipo AppMan é que esta submissão distribuída e sua orquestração são feitas de forma completamente automática.

O restante deste artigo encontra-se organizado da seguinte forma. Inicialmente, na Seção 2 apresentamos e analisamos o modelo GRAND. Na Seção 3 descrevemos o protótipo AppMan, a metodologia experimental e alguns resultados. Na Seção 4 apresentamos alguns trabalhos relacionados. Finalmente, concluímos este texto com considerações finais e trabalhos futuros.

2 GRAND: um modelo de gerenciamento de aplicações para grades

O modelo GRAND trata de gerenciamento de aplicações (*application management*). Gerenciamento de aplicações consiste em preparar, submeter e monitorar o progresso da execução de todas as tarefas que compõem uma aplicação do usuário. A Figura 1 apresenta de forma esquemática todos os passos necessários para gerenciar uma aplicação distribuída no GRAND. Os próximos itens explicam essas etapas, bem como os componentes da arquitetura responsáveis pela execução de cada uma delas.

Inferência do DAG (*DAG Inference*) O primeiro passo é chamado *DAG Inference*: dada uma aplicação descrita em GRID-ADL, o sistema detecta dependências entre as tarefas e obtém um grafo orientado acíclico (DAG ou *directed acyclic graph*). Assume-se que o usuário descreveu sua aplicação já com tarefas determinadas, utilizando a linguagem de descrição proposta chamada GRID-ADL (**Grid A***pplication* **D***escription* **L***anguage*), que se encontra descrita tem detalhes na tese [29].

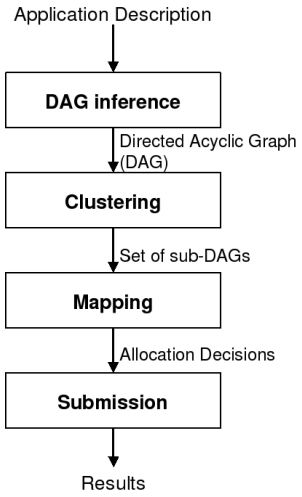

Figura 1. GRAND: etapas para execução de aplicações distribuídas

A Figura 2 apresenta um exemplo ilustrativo de aplicação descrita em GRID-ADL. A aplicação em questão tem três passos: (1) executa gnuplot com um arquivo de entrada (*in1.dat*) e produz um arquivo de saída (*out1.png*); (2) em seguida executa 4 instâncias diferentes de gnuplot, onde cada uma utiliza o seu arquivo de entrada e o seu arquivo de saída; (3) o último passo utiliza os arquivos de saída produzidos pelos passos (1) e (2) para fazer a execução final do gnuplot gerando a saída final *data.ps*.

O algoritmo de inferência detecta tarefas que irão representar os nodos e, através da verificação do fluxo de dados, obtém as arestas que representam dependências entre tarefas através de acesso a arquivos de dados. A Figura 3 representa o DAG gerado para o exemplo da Figura 2.

Agrupamento (*clustering*) Na etapa de *clustering*, uma aplicação composta por várias tarefas é divida em subgrafos. O algoritmo consiste em agrupar tarefas que sejam dependentes ou, caso não exista na aplicação, obter grupos de tarefas independentes, de modo que as tarefas de uma grupo possam ser submetidas ao mesmo tempo. Tarefas agrupadas nesta etapa serão submetidas para um mesmo nodo da grade (*cluster* ou rede local). O objetivo é buscar um agrupamento que minimize a transferência de dados, ao maximizar a localidade dos dados, e maximize o desempenho da aplicação.

```
graph loosely-coupled
OUTPUT = "out1.png"
gnuplot= "/usr/local/bin/gnuplot"
task 1 -e ${gnuplot} -a "1.txt" -i in1.dat
     -o out1.png
foreach TASK in 2..5 {
   task ${TASK} -e ${gnuplot} -a ${TASK}".txt"
        -i in${TASK}.dat -o out${TASK}.png
   OUTPUT = ${OUTPUT} + ";out"+${TASK}+".png"
}
task 6 -e prepare_print -i ${OUTPUT} -o data.ps
transient ${OUTPUT}
```

Figura 2. Exemplo de uma aplicação descrita com GRID-ADL

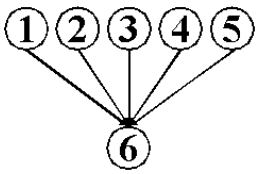

Figura 3. Exemplo de DAG

Como não seria possível obter um agrupamento ótimo, criou-se algoritmos utilizando heurísticas conforme descrito em [29]. Existe um algoritmo para cada um dos tipos de aplicações distribuídas, de acordo com a taxonomia de aplicações proposta neste trabalho [32, 30]: **independent tasks**: tipo mais simples, também referenciado na literatura como *bag-of-tasks*, caracteriza aplicações nas quais todas as tarefas são independentes; **loosely-coupled tasks**: aplicações cujo DAG apresenta poucos pontos de compartilhamento e com formato bem conhecido, representa aplicações divididas em fase (*phase*) ou com seqüências de tarefas dependentes (*pipeline*); **tightly-coupled tasks**: grafos complexos cujo particionamento é mais difícil de ser realizado manualmente, devido ao grande número de arestas do DAG.

O usuário pode decidir armazenar os resultados destas duas etapas iniciais, isto é, a descrição das tarefas, a estrutura do DAG e as informações de *clustering*. Para isso, o GRAND permite a exportação destas informações em formato XML. Para isso foi proposta uma extensão da linguagem padrão do Open Grid Forum, antigo Global Grid Forum, denominada JSDL[20]. A especificação da JSDL (*Job Submission Description Language*) [4] não prevê a dependência entre tarefas, algo vital para o escalonamento de certas aplicações suportadas pelo GRAND.

Embora existam linguagens de descrição em XML como GXML [2], AGWL [11] e XPWSL [10], acreditamos que XML é uma linguagem apropriada para troca de informações entre programas e processamento de dados, mas não é a mais adequada para codificação direta por usuários. A inclusão deste tipo de saída no GRAND tem o objetivo de simplificar futuras interações com outros RMSs que também utilizem XML.

Mapeamento (*mapping*) Uma vez que os sub-DAGs (*clusters*) foram determinados, eles serão atribuídos a recursos na fase de mapeamento (*mapping*). O mapeamento ocorre de acordo com os recursos disponíveis usando critérios detalhados na tese e aqui resumidos. Um dos componentes chaves nesta etapa, conforme será detalhado a seguir (parágrafo "Componentes"), é o *Submission Manager* (*SM*). Inicialmente, o *SM* busca na sua base local quais nodos atendem aos requisitos da aplicação. Para que um nodo da grade possa ser considerado como candidato para a realização de mapeamento ele tem em primeiro lugar que atender aos requisitos mínimos da aplicação. Exemplos de requisitos são quantidade de memória (manipulação de estruturas de dados muito grandes em memória), espaço livre em disco (certas aplicações precisam de uma quantidade de memória mínima para gravar os resultados em forma de arquivo) e software (tanto um sistema operacional específico quanto um programa ou biblioteca podem ser essenciais para permitir a execução de uma aplicação).

Depois, para cada um dos nodos da grade que atendem aos requisitos, o *SM* calcula quatro propriedades: (1) **distance**: inteiro que indica distância relativa, objetivando avaliar a latência (0 – nodo da submissão; 1 – conexões regionais; e 2 – conexões internacionais;) (2) **capacity**: assume um número indicando a capacidade de banda do canal; (3) **cpu**: indica o poder de computação calculado pela seguinte fórmula:

$$\text{cpu} = (\sum_{i=1}^{n} \text{number\_of\_cpus}_i * \text{individual\_power}_i)/\text{node\_load}$$

onde, number_of_cpus é o número de CPUs ou máquinas disponiveis no nodo, e individual_power é o poder computacional expresso em alguma medida padrão (MFlops ou MIPS). O node_load é o percentual de ocupação. (4) **memory**: capacidade de memória média.

Então cada nodo da grade recebe um valor chamado *preference*, o qual é calculado segundo a fórmula: $\text{preference} = [(\text{distance} * 0.25) + (\text{capacity} * 0.25)] + [(\text{cpu} * 0.25) + (\text{memory} * 0.25)]$ Este valor é usado para ordenar os nodos de forma decrescente, e esta fila gerada é usada para realizar o mapeamento.

Submissão (*submission*) A etapa de *submission* é a responsável por alocar os recursos escolhidos, garantindo que ocorra corretamente (a) a transferência dos dados (*data staging*), (b) instalação/localização dos executáveis e (c) execução de todas as tarefas. Cada *TM* está ligado a um nodo específico da grade e possui o algoritmo que permite mapear a descrição das tarefas para o formato de submissão do RMS correspondente. Este mapeamento pode tanto ser feito diretamente para o formato de um RMS específico

como Condor [28], PBS [24] ou SGE[18], ou utilizando uma interface padrão como o DRMAA [8].

Componentes O controle da submissão e da execução são gerenciados por uma hierarquia de três componentes, cujos relacionamentos estão ilustrados na Figura 4. O *Application Manager* (*AM*): é o controlador de mais alto nível que cuida de uma aplicação específica, sendo responsável inicialmente pela inferência do DAG e *clustering*. Ao longo da execução da aplicação, além de interagir com os *Submission Managers*, fornece ao usuário um retorno (*feedback*). O *Submission Manager* (*SM*): componente responsável pelo mapeamento dos agrupamentos (*clusters*) para os nodos da grade. Uma rede local pode ter ao mesmo tempo vários *SMs* ativos, mas apenas um por máquina para garantir o balanceamento da carga de submissão. Finalmente, o *Task Manager* (*TM*): é um *wrapper* capaz de submeter tarefas para um nodo da grade específico.

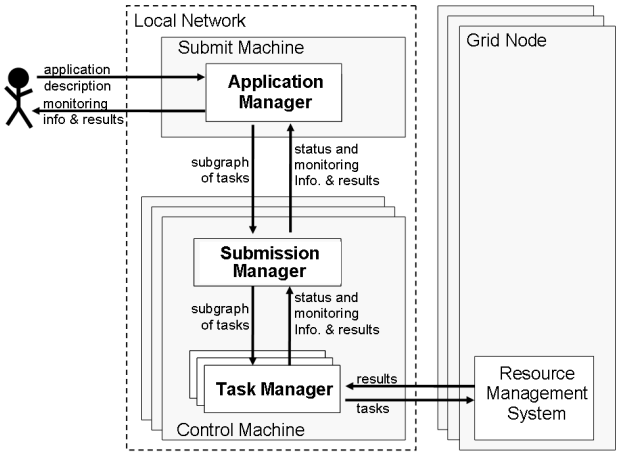

Figura 4. GRAND: principais componentes

Quando um usuário submete sua aplicação na máquina de submissão (*submit machine*), o *AM* é inicializado devido a esta requisição. Quando o *AM* se torna ativo, ele envia uma mensagem do tipo *broadcast* para a sua rede local. Todos os *SMs* ativos respondem a esta mensagem informando sua localização e estado. De posse destas informações, o *AM* constrói uma base privada a qual consulta para decidir quando e para qual *SM* enviar um determinado *cluster*. Note que um *SM* poderá tratar de requisições vindas de vários *AMs*. O GRAND instancia um *AM* por aplicação, para simplificar este componente e porque em princípio não há necessidade de sincronizações entre aplicações. Cada *AM* pode usar um ou mais *SMs* disponíveis na rede local. Há no máximo um *SM* por máquina. Cada *SM* se comunica com um ou mais *TMs*. Cada *TM* pode se comunicar com um nodo da grade e se reportar a um *SM*.

A Figura 5 apresenta um exemplo de cenário de execução com o objetivo de sumarizar alguns detalhes. Apesar desta figura não explicitar isso, um *SM* pode mapear *clusters* de diferentes aplicações. Note que cada máquina com um *SM* possui um ou mais *TMs*. Uma máquina poderia ter mais de um *SM*, mas a idéia é que ao fixar um por máquina teremos um único processo controlando as informações sobre os nodos da grade diminuindo o consumo de recursos. Além disso, o protocolo de comunicação entre os componentes se torna mais simples.

Cada *TM* pode se comunicar com um RMS específico de um nodo da grade para submeter as tarefas. Se um *SM* decide alocar tarefas em dois nodos diferentes, ele utilizará dois *TMs* para efetuar a alocação de fato. Utilizar diferentes instâncias de *TMs* para diferentes tipos de RMSs permite a construção de componentes mais leves, uma vez que não precisa ter incorporado no seu processo todos os detalhes de comunicação de cada RMS suportado. Caso dois *SMs* queiram se comunicar com o mesmo RMSs, eles utilizam suas instâncias locais de *TM*, minimizando a comunicação entre as máquinas da rede local.

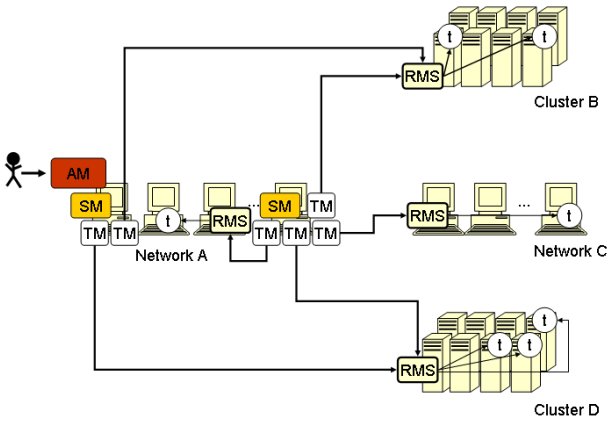

Figura 5. GRAND: um possível cenário

O modelo de submissão de tarefas proposto pode ser classificado como um escalonador de alto nível (*high-level scheduler*) [26] uma vez que ele consulta outros escalonadores para decidir o escalonamento.

3. AppMan: um sistema de gerenciamento de aplicações para um ambiente de grade

O AppMan é implementado em Java e utiliza a ferramenta JavaCC para implementar o *parsing* e a interpretação dos arquivos GRID-ADL. O ambiente de execução usa serviços disponíveis no *middleware* EXEHDA [36] que permite a execução remota e o monitoramento de recursos.

No AppMan, o componente *SM* é responsável por colocar os arquivos a serem utilizados pelas tarefas no nodo da grade (*stage in*) antes de requisitar que o *TM* instancie remotamente a tarefa. Em um cluster ou rede local com NFS, o *SM* cria um diretório em uma área temporária, que funcionará como uma *sandbox*, uma vez que as tarefas somente terão permissão de escrita nesta área. Então, todos os arquivos de entrada e executáveis exigidos pela tarefa são copiados para este diretório. Uma vez que algumas tarefas podem compartilhar arquivos de entrada, ou usar os mesmos executáveis, o *SM* verifica primeiro se os arquivos necessários já estão disponíveis na *sandbox* antes de buscá-los. Note que deste modo, o diretório passa a ter dupla função: além de *sandbox* ele serve como uma *cache* de dados para as tarefas que executam no mesmo nodo da grade. Alguns dados experimentais sobre o gerenciamento de dados são apresentados na Seção 3.1. Quando uma tarefa termina, os arquivos produzidos são enviados de volta *stage out* para a máquina onde está o *AM* que requisitou a execução desta tarefa. Nesta implementação do modelo GRAND, todos os arquivos precisam estar explicitamente definidos pelo usuário no código GRID-ADL. Isto porque a decisão de fazer *stage in* e *stage out* de arquivos é tomada pelo AppMan em função desta codificação. A única exceção é o executável que, embora utilizado pela tarefa, não precisa ser descrito, caso já esteja disponível e instalado nos nodos da grade. Além disso, arquivos produzidos, mas não explicitados como de saída, são considerados arquivos temporários e portanto não são transferidos de volta.

Para os experimentos descritos nesta seção, estamos considerando um *SM* que pode selecionar máquinas individuais e não nodos de uma grade. Com isso, o *SM* do AppMan pode ser considerado um RMS simplificado. Quando um *SM* recebe um *cluster* ou sub-DAG, ele deve selecionar uma máquina disponível no mesmo nodo da grade para executar a tarefa. O componente *TM* instancia o processo da tarefa diretamente em uma máquina do nodo local da grade.

Foi implementada uma política de escalonamento simples, uma vez que o modelo GRAND não tem por objetivo propor um novo RMS. No entanto, esta implementação do *SM* no protótipo AppMan, pode ser perfeitamente utilizada como uma alternativa de RMS, em nível do usuário, para uma rede local que não possua nenhum RMS instalado. A avaliação inicial mostrou bons resultados para tarefas pequenas em comparação com um RMS tradicional, como apresentado na Seção 3.2.

3.1 Experimentos sobre Gerenciamento de Dados

Para este experimento, cada uma das tarefas executava por um curto período de tempo (aproximadamente 3 segundos). Todas as tarefas eram determinísticas. Variamos o número de tarefas por aplicação e o tamanho dos arquivos de entrada utilizados pelas tarefas. Os experimentos foram realizados com duas versões do AppMan: (1) sem otimização, onde os arquivos de entrada eram sempre transferidos para os nodos de execução e (2) com otimização, onde o *SM* funciona como uma *cache* e os dados de entrada somente são buscados quando necessário. Também variamos a forma de obtenção dos arquivos de entrada: via NFS (arquivos na rede local) ou via ftp (de um site público).

Todos os experimentos foram executados em um ambiente com as seguintes características: (a) rede local Ethernet de 100 Mbps, não dedicada; (b) seis máquinas – quatro máquinas Pentium IV 1.8 GHz com 1 GByte de RAM e duas Atlhon XP 2GHz com 512 MByte de RAM.

A Tabela 1 mostra dados experimentais, variando o número de tarefas entre 50 e 200. A primeira coluna mostra o número de tarefas de cada execução. A segunda, mostra o tamanho dos arquivos de entrada para cada caso. A terceira, quarta e quinta colunas mostram o tempo total de execução para todas as tarefas, respectivamente, para transferências por NFS e FTP e por FTP com otimizações. Uma característica curiosa desta tabela é que o protocolo FTP teve melhor desempenho do que o NFS a medida que o tamanho dos arquivos de entrada aumentava. Isso indica o alto custo adicional (*overhead*) do protocolo NFS em uma rede local, bem como as contenções que podem ocorrer caso a execução não seja distribuída para outros nodos da grade.

Tabela 1. Resultados sobre gerenciamento de dados

número de tarefas	tamanho arqs. entrada	NFS	FTP	otimizado
50	10Kb	192,910	197,865	84,161
	500Kb	197,197	206,162	82,211
	1Mb	206,205	225,335	93,149
100	10Kb	408,950	461,429	156,657
	500Kb	408,834	438,488	233,906
	1Mb	509,584	406,442	223,649
200	10Kb	809,055	794,715	248,922
	500Kb	797,519	806,289	253,789
	1Mb	1234,024	826,37	260,501

A otimização da transferência de arquivos utiliza uma cache por *TM* para controlar os arquivos que já encontram-se disponíveis localmente. É possível observar o efeito positivo no desempenho a medida que o número de tarefas aumenta. Em alguns casos, pode-se obter um desempenho de mais de 50% quando da transferência de arquivos de 1 Mbyte de tamanho e com aplicações compostas por 200 tarefas. Estes resultados mostram a necessidade de gerenciamento de dados para arquivos de tamanho pequeno a médio, e para um número relativamente pequeno de tarefas.

O gerenciamento de dados é ainda mais importante quanto se aumenta o número de tarefas e o tamanho dos arquivos. A Figura 6 esboça os primeiros resultados obtidos para arquivos maiores (no grafo, até 5Mb por tarefa) e para um número um pouco maior de tarefas (até 300). Este gráfico indica que a abordagem é escalável. Este é um dos pontos que deve ser melhor avaliado no futuro, permitindo refinar o gerenciamento dos dados.

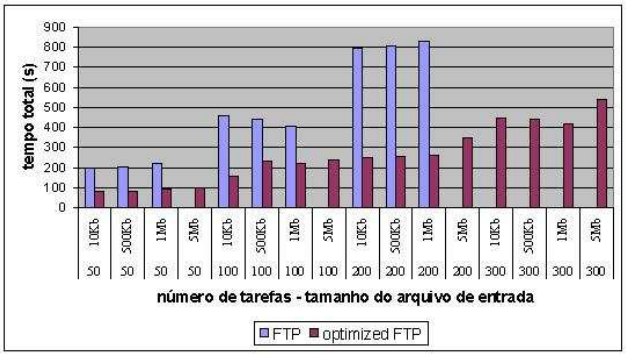

Figura 6. Resultados sobre gerenciamento de dados: escalabilidade

3.2 Experimentos sobre Gerenciamento de Recursos

A aplicação utilizada neste experimento consiste em um conjunto de tarefas que realiza um conjunto determinístico de operações matemáticas, e foi escolhida como um *exerciser* para o AppMan [34]. Os experimentos foram realizados em um ambiente com as seguintes características: (a) rede local giga Ethernet; (b) 4 máquinas Atlhon XP 2 GHz, 512 MB, with Linux kernel 2.4.21-20.EL. O *cluster* foi utilizado de forma praticamente dedicada.

A Tabela 2 mostra o tempo médio de execução das tarefas e o tempo de execução da aplicação usando tanto o AppMan quanto o Condor. A coluna "SM" indica o número de *Submission Managers* criados pelo AppMan para distribuir a tarefa de submissão (a1 corresponde a um *SM*, a2 corresponde a dois *SMs* e a3 corresponde a três *SMs*.). A letra "c" indica os tempos de execução para aplicações que foram disparadas pelo Condor. As colunas que indicam os tempos de execução (task exec. time) e submissão (task sub. time) apresentam a média e, dentro de parênteses, o desvio padrão. Todas as medidas de tempo são apresentadas em segundos. A coluna "task sub. time" no caso do AppMan, corresponde ao tempo para instanciar o *SM* e de fato selecionar um nodo para disparar uma tarefa. Já no caso do Condor, esta coluna inclui o tempo que a tarefa fica na fila de tarefas (*task queue*) aguardando para ser escalonada.

Tabela 2. Tempos de execução: Condor e AppMan

tasks	SM	tempo sub. tarefa		tempo exec. tarefa		tempo exec. apl.
10	a1	1,10	(0,88)	2,30	(0,48)	19,82
	a2	1,20	(0,79)	2,50	(0,53)	19,89
	a3	1,90	(1,10)	2,10	(0,74)	19,75
	c	12,80	(5,98)	0,30	(0,48)	22,00
50	a1	9,56	(4,09)	7,58	(2,98)	42,83
	a2	8,62	(4,23)	7,52	(2,89)	42,83
	a3	8,58	(3,88)	7,68	(2,88)	37,57
	c	53,52	(29,57)	0,02	(0,27)	103,00
100	a1	16,13	(8,30)	10,63	(2,64)	66,68
	a2	17,46	(8,50)	8,62	(2,90)	65,63
	a3	16,62	(7,92)	11,39	(3,18)	65,59
	c	104,83	(58,56)	0,08	(0,27)	205,00
200	a1	27,65	(12,58)	11,30	(2,61)	113,61
	a2	28,09	(11,89)	13,21	(2,86)	115,63
	a3	21,14	(12,66)	16,44	(8,25)	118,19
	c	206,39	(117,20)	0,10	(0,29)	407,00
300	a1	62,36	(42,30)	28,41	(15,32)	303,30
	a2	59,23	(39,71)	26,91	(13,58)	295,50
	a3	57,49	(39,27)	28,36	(14,19)	288,63
	c	305,70	(174,52)	0,11	(0,31)	606,00
500	a1	14,56	(9,05)	8,39	(3,49)	276,37
	a2	16,02	(10,53)	8,11	(2,89)	273,35
	a3	17,90	(11,67)	9,97	(4,22)	288,69
	c	508,60	(292,80)	0,11	(0,31)	1018,00

Esta tabela mostra que o AppMan completa a aplicação em bem menos tempo que o Condor. A medida que o número de tarefas submetidas aumenta, esta diferença se mostra ainda maior, chegando a ser quase cinco vezes melhor no caso de 500 tarefas. Vários motivos contribuem para estes resultados. O Condor emprega uma forma mais sofisticada de escolha de recursos utilizando um algoritmo denominado *matchmaking*. Esta pode ser uma das razões de porque o Condor demora mais tempo, pois o AppMan usa um algoritmo mais simples, que não se preocupa, por exemplo, em garantir que não existe nenhum usuário utilizando um recurso escolhido. Por outro lado, neste experimento o Condor utiliza o NFS para transferir seus arquivos de entrada e saída, enquanto o AppMan está buscando em um repositório FTP (que como o experimento da seção anterior indica, é bem mais eficiente que o NFS).

4. Trabalhos Relacionados

Uma das preocupações principais do GRAND é tratar aplicações com número muito grande de tarefas. Dutra *et al.* [9] apresenta experimentos de programação em lógica indutiva que geraram mais de 40 mil tarefas. Entretanto, a solução proposta era específica para o gerenciamento de tal aplicação, sendo concentradas em prover uma ferramenta de usuário para controlar e monitorar essas tarefas. Existem alguns trabalhos que tratam de dependências de tarefas tais como o DAGMan [28], o VDS [14] e o TrellisDAG [17]. Entretanto, o usuário deve indicar explicitamente as dependências, enquanto o GRAND permite inferir as de-

pendências automaticamente. Além disso, nesta tese foi proposta uma taxonomia para aplicações distribuídas [32] – *independent*, *loosely-coupled* e *tightly-coupled* – que permite aplicar um algoritmo de particionamento considerando as características da classe de aplicação.

Existem várias linguagens de descrição propostas na literatura tais como GEL [22] e PASWL[10]. Estas duas são as que possuem mais pontos em comum com nosso trabalho, conforme analisado na tese. GRID-ADL [32, 31] possui algumas similaridades com a linguagem de script GEL [22]. GEL possui alguns comandos adicionais, *e.g* comando de repetição condicional (*while*), não suportados por GRID-ADL. No entanto, GEL necessita que o usuário utilize construções paralelas clássicas como o *pfor* para permitir a inferência do DAG, enquanto no GRID-ADL a inferência é automática bastando a indicação dos arquivos de entrada e saída. De qualquer modo, além do fato de considerarmos a nossa proposta mais adequada para o perfil de usuário e aplicação alvo, este trabalhos foram publicados após a GRID-ADL ter sido proposta e implementada [30].

Existem vários *middlewares* disponíveis para grades computacionais tais como Globus [16], Condor [28], gLite [15], OurGrid [6] e EasyGrid [3]. Porém estes não são escaláveis, não fazem particionamento e nem permitem distribuição da submissão de tarefas de forma automática.

5. Conclusão

Foi projetado um modelo arquitetural denominado GRAND para ser implementado como um *middleware*. A principal proposta da tese é um mecanismo de gerenciamento hierárquico que pode, de forma completamente automática, controlar a execução de um número muito grande de tarefas distribuídas, preservando localidade de dados e ao mesmo tempo reduzindo a carga das máquinas onde ocorre a submissão. A idéia central é balancear a carga de controle da submissão de tarefas em várias máquinas. O modelo considera que pode contar com alguns componentes de software já disponíveis na infra-estrutura de grade utilizada, tais como RMS nos nodos da grade e controle de autenticação e autorização.

No modelo GRAND, considera-se que os recursos e tarefas são modelados como grafos orientados acíclicos (DAGs). Alguns aspectos importantes no contexto de aplicações que disparam um número muito grande de tarefas e que são tratados incluem: *(*1) particionamento das aplicações (*clustering*) de tal modo que, quando possível, tarefas dependentes sejam colocadas no mesmo nodo da grade para evitar migração desnecessária de arquivos de dados intermediários e/ou transientes; *(*2) particionamento das aplicações (*clustering*) de tal modo que tarefas sejam alocadas perto dos seus dados de entrada; *(*3) distribuição do processo de submissão de modo que uma máquina de submissão não fique sobrecarregada; e *(*4) garantia da integridade dos resultados.

Embora a contribuição principal da tese seja a definição do modelo GRAND como um todo, podemos esboçar em linhas gerais outras contribuições: *(*a) proposta e implementação da linguagem GRID-ADL; *(*b) definição de um novo XML-Schema para a descrição dos DAGs da aplicação; *(*c) definição de serviços de gerenciamento de dados e de monitoramento; *(*d) implementação e avaliação do protótipo AppMan.

Os resultados experimentais são bastante promissores. Técnicas simples, mas ainda assim efetivas, baseadas em mecanismos de *caching* e localidade de dados para suportar o gerenciamento de dados das aplicações mostraram-se suficientes para garantir melhoras de desempenho da ordem de 50%. Este é um ótimo resultado, pois garante que o sistema de gerenciamento não sofre de alta sobrecarga. Além disso, o segundo conjunto de experimentos indicou que o Condor pode impor um custo adicional (*overhead*) alto quando executando com um número elevado de tarefas. Nesse cenário, o AppMan se mostrou uma excelente alternativa como RMS.

Como trabalhos futuros, pretende-se aprimorar tanto o modelo quanto o protótipo. O GRAND se preocupa em escalonar tarefas cujos requisitos são principalmente memória e CPU. A investigação de outros tipos de requisitos e restrições é necessária para permitir o uso deste modelo em outras classes de aplicações. Também houve avanços no sentido de integrar o AppMan com dois RMSs: PBS e SGE. No entanto, isso precisa ser melhor trabalhado. Embora do ponto de vista de modelo, qualquer tipo de RMS possa ser suportado, mudanças significativas precisariam ser feitas na implementação do protótipo AppMan, pois este, atualmente, somente suporta sua própria política de escalonamento e é preciso que se faça o mapeamento não mais para máquinas, mas para nodos da grade. Atualmente, existe um trabalho em andamento relacionado a monitoramento de recursos [21]. Uma ferramenta foi construída para obter tanto informações dinâmicas quanto estáticas bem como de hardware e software. O objetivo é integrar essas informações ao AppMan. As decisões de alocação do AppMan estão no momento baseadas em um algoritmo que não leva em conta o estado atual das máquinas. Utilizando as informações de monitoramento, é possível implementar no AppMan o algoritmo proposto pelo modelo GRAND conforme descrito na tese. Finalmente, dois tópicos relacionados a gerenciamento de dados devem ser estudados. Em primeiro lugar, a integração do procedimento automático de *staging* com um serviço de réplicas, isto é consultar um *replica catalog service* para obter arquivos. Em segundo lugar, como integrar o GRAND com trabalhos relacionados à integração de bases de dados com a infra-estrutura de grade.

Referências

[1] F. Berman, G. Fox, and T. Hey. *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley, 2003.

[2] H. S. Bhatt, D. Bhansali, S. N. Shah, P. Patel, and A. Dasgupta. GANGA: Grid application information gathering and accessing framework. *International Journal of Information Technology*, 11(4):58–73, 2005.

[3] C. Boeres, A. P. Nascimento, V. E. F. Rebello, and A. C. Sena. Efficient hierarchical self-scheduling for MPI applications executing in computational grids. In *Proc. of the 3rd Intl. Workshop on Middleware for grid computing (MGC05)*, pages 1–6, New York, NY, USA, 2005. ACM Press.

[4] F. Brisard and A. Ly. Job submission description language (jsdl) specification – version 1.0, November 7th 2005.

[5] R. Buyya and S. Venugopal. The gridbus toolkit for service oriented grid and utility computing: An overview and status report. In *Proc of the 1st IEEE Intl Workshop on Grid Economics and Business Models (GECON 2004)*, pages 19–36, Seoul, Korea, April 23 2004.

[6] W. Cirne *et al.* Labs of the world, unite!!! *Journal of Grid Computing*, 4(3):225–246, September 2006.

[7] Paradyn – Condor week 2007. `http://www.cs.wisc.edu/condor/PCW2007/`.

[8] Distributed resource management application api working group (drmaa-wg). `http://www.drmaa.org/`.

[9] I. C. Dutra *et al.* Toward automatic management of embarrassingly parallel applications. In *Proc of the 26th Intl Conference on Parallel and Distributed Computing (Europar 2003)*, pages 509–516, Klagenfurt, Austria, August 2003.

[10] C. Enomoto and M. A. A. Henriques. A flexible specification model based on XML for parallel applications. In *Proc. of the 17th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2005)*, pages 109–116, Rio de Janeiro, Brasil, October 24-27 2005.

[11] T. Fahringer, J. Qin, and S. Hainzer. Specification of grid workflow applications with AGWL: An abstract grid workflow language. In *Proceedings of Cluster Computing and Grid 2005 (CCGrid 2005)*, Cardiff, UK, May 9–12 2005.

[12] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, San Francisco, California, USA, 2 edition, 2004.

[13] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. Grid services for distributed system integration. *IEEE Computer*, 35(6):37–46, June 2002.

[14] I. Foster, J. Voeckler, M. Wilde, and Y. Zhao. The Virtual Data Grid: A new model and architecture for data-intensive collaboration. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR 2003)*, Asilomar, CA, USA, January 5-8 2003.

[15] gLite – Ligthweight Middleware for Grid Computing. `http://glite.web.cern.ch/glite/`.

[16] The Globus Toolkit, 2006. `http://www.globus.org/toolkit/`.

[17] M. Goldenberg, P. Lu, and J. Schaeffer. TrellisDAG: A system for structured DAG scheduling. In D. Feitelson, L. Rudolph, and U. Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 21–43. Springer, 2003. Lecture Notes in Computer Science, Volume 2862/2003.

[18] Grid Engine Project. `http://gridengine.sunsource.net/`.

[19] A. S. Grimshaw, A. Ferrari, F. Knabe, and M. Humphrey. Wide-area computing: Resource sharing on a large scale. *IEEE Computer*, 32(5):29–36, May 1999.

[20] Job Submission Description Language WG. `https://forge.gridforum.org/projects/jsdl-wg/`.

[21] D. d. T. Lemos and P. K. Vargas. Monitoramento de recursos em ambiente de grade. In *Sessão de Pôsteres de Iniciação Científica em PAD*, Porto Alegre, RS, Brasil, 16 a 19 de janeiro 2007. Evento da ERAD2007.

[22] C. C. Lian, F. Tang, P. Issac, and A. Krishnan. GEL: grid execution language. *Journal of Parallel and Distributed Computing*, 65(7):857–869, 2005.

[23] J. Nabrzyski, J. M. Schopf, and J. Weglarz. *Grid Resource Management: State of the Art and Future Trends*. Kluwer Academic Publishers, 2003.

[24] Open PBS (Portable Batch System). `http://www.openpbs.org/main.html`.

[25] J. A. L. Sanches *et al.* ReGS: user-level reliability in a grid environment. In *Cluster Computing and Grid 2005 (CCGRID05)*, Cardiff, UK, May 9–12 2005.

[26] U. Schwiegelshohn and R. Yahyapour. Attributes for communication between scheduling instances. In J. Nabrzyski, J. M. Schopf, and J. Weglarz, editors, *Grid Resource Management*, pages 41–52. Kluwer Academic Publishers, 2004.

[27] I. Taylor *et al.* Distributed computing with Triana on the grid. *Concurrency and Computation: Practice and Experience*, 17(1–18):1197–1214, 2005.

[28] D. Thain, T. Tammembaum, and M. Livny. Distributed computing in practice: the condor experience. *Concurrency and Computation: Practice and Experience*, 17(2–4):323–356, February – April 2005.

[29] P. K. Vargas. *GRAND: A Model for Hierarchical Application Management in Grid Computing Environment*. PhD thesis, COPPE/Systemas, UFRJ, Rio de Janeiro, RJ, Brasil, 2006.

[30] P. K. Vargas, I. C. Dutra, and C. F. Geyer. Hierarchical resource management and application control in grid environments. RT ES-608/03, COPPE/Sistemas - UFRJ, 2003.

[31] P. K. Vargas, I. C. Dutra, and C. F. Geyer. Application partitioning and hierarchical application management in grid environments. RT ES-661/04, COPPE/Sistemas - UFRJ, 2004.

[32] P. K. Vargas, I. C. Dutra, and C. F. Geyer. Application partitioning and hierarchical management in grid environments. In *1st Intl Middleware Doctoral Symposium*, pages 314–318, Toronto, Canadá, October 19th 2004.

[33] P. K. Vargas, L. A. S. Santos, I. C. Dutra, and C. F. Geyer. An implementation of the GRAND hierarchical application management model using the ISAM/EXEHDA system. In *III Workshop on Computational Grids and Applications*, Petrópolis, RJ, Brazil, Jan. 31 – Feb. 2 2005.

[34] P. K. Vargas *at al.* Hierarchical submission in a grid environment. In *3rd Intl Workshop on Middleware for Grid Computing*, Grenoble, France, Nov. 28 – Dec. 2 2005.

[35] P. K. Vargas *et al.* Grand: Toward scalability in a grid environment. *Concurrency and Computation: Practice and Experience*, 19, 2007. To appear.

[36] A. C. Yamin *et al.* Towards merging context-aware, mobile and grid computing. *International Journal of High Performance Computing Applications*, 17(2):191–203, June 2003.