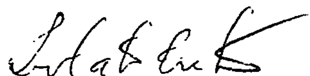
# GRAND: UM MODELO DE GERENCIAMENTO HIERÁRQUICO DE APLICAÇÕES EM AMBIENTE DE COMPUTAÇÃO EM GRADE

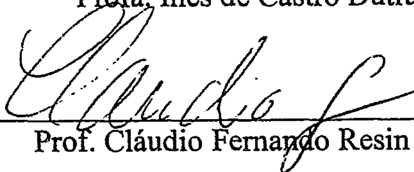
Patrícia Kayser Vargas Mangan

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE DOUTOR EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

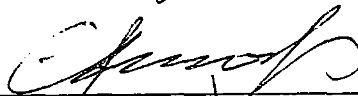
Aprovada por:



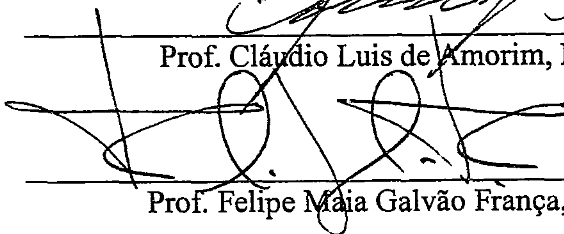
Profa. Inês de Castro Dutra, Ph.D.



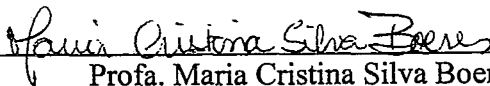
Prof. Cláudio Fernando Resin Geyer, Dr.



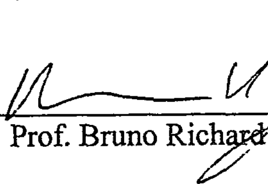
Prof. Cláudio Luis de Amorim, Ph.D.



Prof. Felipe Maia Galvão França, Ph.D.



Profa. Maria Cristina Silva Boeres, Ph.D.



Prof. Bruno Richard Schulze, D.Sc.

RIO DE JANEIRO, RJ - BRASIL

MARÇO DE 2006

MANGAN, PATRÍCIA KAYSER VARGAS

GRAND: Um Modelo de Gerenciamento Hierárquico de Aplicações em Ambiente de Computação em Grade [Rio de Janeiro] 2006

XIII, 150p. 29,7 cm (COPPE/UFRJ, D.Sc., Engenharia de Sistemas e Computação, 2006)

Tese – Universidade Federal do Rio de Janeiro, COPPE

1 - Processamento Distribuído

2 - Computação em Grade

3 - Gerenciamento de Recursos

I. COPPE/UFRJ II. Título (série)

# Agradecimentos

Gostaria de agradecer a todos os que direta ou indiretamente contribuíram para que eu concluísse essa tese.

Ao CNPq pelo suporte financeiro.

Ao UniLaSalle, na figura de colegas e administradores, por me liberar e me incentivar a realizar o curso de doutoramento.

A todos os colegas, professores e funcionários da COPPE/Sistemas da UFRJ.

Aos professores Vinod Rebelo e Cláudio Amorim fazerem parte da minha banca de exame de qualificação. Aos professoes Cláudio Amorim, Felipe França, Bruno Schulze e Cristina Boeres por terem aceitado participar da banca de avaliação de tese. A todos o meu agradecimento por suas críticas construtivas.

Aos meus orientadores Inês de Castro Dutra e Cláudio F. R. Geyer que além de possuirem um grande conhecimento técnico são seres humanos ímpares. Com certeza o apoio, o carinho e compreensão de ambos tornaram todo o estresse suportável e até agradável. Sem a orientação e a amizade deles esta tese não existiria.

Aos colegas do LabIA, LENS e LAND que tornaram esta jornada mais agradável. A Tatiana, Sérgio, José Afonso e Luciana que dividiram as incertezas dos primeiros anos. Ao Vinícius, meu fiel aliado nas brigas operacionais com o Monarc e AppMan. Ao Luis Otávio pela parceria no dia-a-dia do laboratório. A Carol por toda a ajuda, simpatia e carinho.

A Marluce que, além de colega e parceira de trabalho, se tornou uma amiga muito querida. Nenhuma palavra expressa minha admiração e meu carinho e meu agradecimento por tudo que partilhamos neste anos de doutorado.

Aos colegas da UFRGS que sempre me acolheram com carinho no “saloon”. Em especial ao Lucas e ao Luciano pela parceria no desenvolvimento do AppMan. Aos amigos de longa data Adenauer e Jorge e a minha irmã de coração Débora que sempre me passaram carinho e confiança.

Aos colegas do Rio2, que ajudaram a tornar a nossa estadia no Rio de Janeiro mais divertida, com as parcerias para churrasco e tênis.

A amiga Andressa e sua família que nos acolheram como amigos e parceiros de carteado e orgias gastrônomicas.

A toda a minha família, pelo suporte afetivo e pelo incentivo. Em particular, o apoio de todos, e em especial da minha mãe, nas últimas semanas foram fundamentais para a conclusão deste texto. Agradeço o carinho de minha mãe, Lourdes, que é um referencial pela seriedade com que assume seus compromissos. Ao meu avô, *in memorian*, devo principalmente a lição de valorizar o estudo.

Ao meu marido, colega de trabalho e incentivador há mais de doze anos, o meu agradecimento pela companhia carinhosa, pelas críticas duras e consistentes e pelo apoio e incentivo constantes para meu crescimento profissional e pessoal.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Doutor em Ciências (D.Sc.)

GRAND: UM MODELO DE GERENCIAMENTO HIERÁRQUICO DE APLICAÇÕES EM AMBIENTE DE COMPUTAÇÃO EM GRADE

Patrícia Kayser Vargas Mangan

Março/2006

Orientador: Inês de Castro Dutra
Cláudio Fernando Resin Geyer

Programa: Engenharia de Sistemas e Computação

Um ambiente computacional em grade apresenta desafios tais como o controle de um grande número de tarefas e a sua alocação nos nodos da grade. Muitos dos trabalhos apresentados na literatura não conseguem tratar adequadamente todos os aspectos relacionados com o gerenciamento de aplicações. Alguns problemas que não são tratados adequadamente incluem gerenciamento de dados e sobrecarga das máquinas de submissão (i.e. as máquinas onde as aplicações são submetidas). Esta tese trata destas limitações, focando em aplicações que disparam um grande número de tarefas. Neste contexto é proposto e avaliado um novo modelo 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; (2) uma nova linguagem de descrição chamada GRID-ADL; (3) uma hierarquia de gerenciadores que realizam a submissão das tarefas. Um protótipo foi implementado e avaliado mostrando bons resultados relacionados ao gerenciamento de recursos e de dados.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Doctor of Science (D.Sc.)

GRAND: A MODEL FOR HIERARCHICAL APPLICATION MANAGEMENT IN GRID COMPUTING ENVIRONMENT

Patrícia Kayser Vargas Mangan

March/2006

Advisors: Inês de Castro Dutra

Cláudio Fernando Resin Geyer

Department: Computing and Systems Engineering

A grid computing environment presents challenges such as the control of huge numbers of tasks and their allocation to the grid nodes. Many works presented in the literature can not deal properly with all issues related to application management. Some problems that are not properly solved include data management and overload of the submit machines (i.e. the machine where the applications are launched). This thesis deals with these limitations, focusing on applications that spread a very large number of tasks. In this context, we propose and evaluate a new application management system called GRAND (**G**rid **R**obust **A**pplicatio**n** **D**eployment). Some of the contributions of the proposed model are (1) a flexible partitioning ; (2) a new description language called GRID-ADL; (3) the hierarchy of managers that perform task submission. A prototype was implemented and evaluated showing good results related to resource and data management.

# Sumário

**1 Introdução** **1**

**2 O Modelo GRAND** **7**

2.1 Premissas . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
2.2 Caracterização das Aplicações . . . . . . . . . . . . . . . . . . . . 10
2.3 Gererenciamento de Dados . . . . . . . . . . . . . . . . . . . . . . 11
2.4 Particionamento . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
2.5 Modelo de Gerenciamento . . . . . . . . . . . . . . . . . . . . . . 12
2.6 AppMan: Protótipo GRAND . . . . . . . . . . . . . . . . . . . . . 14
2.7 Conclusão . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15

**3 Estrutura dos Apêndices** **16**

**A Overview** **18**

A.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
A.2 Goals and Contributions . . . . . . . . . . . . . . . . . . . . . . . 20

**B Grid Computing Concepts** **23**

B.1 Definitions of Grid Computing . . . . . . . . . . . . . . . . . . . . 23
B.2 Grid Applications Classifications . . . . . . . . . . . . . . . . . . . 25
B.3 Classifying Grid Systems . . . . . . . . . . . . . . . . . . . . . . . 28
B.4 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29

**C Application Management: Description and Clustering** **31**

C.1 Representing an Application as a Graph . . . . . . . . . . . . . . . 31
C.2 Application Description Languages . . . . . . . . . . . . . . . . . 32
C.2.1 DAGMan . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33

C.2.2 Chimera . . . 37
C.2.3 GXML . . . 39
C.2.4 AGWL . . . 40
C.2.5 XPWSL . . . 40
C.2.6 GEL . . . 43
C.3 Application Partitioning . . . 43
C.4 Conclusion . . . 48

**D Application Management: Resource and Task Allocation 50**
D.1 Resource Management Issues . . . 50
D.2 Scheduling Taxonomy . . . 52
D.3 Job Management Systems for Clusters . . . 55
D.4 Scheduling Mechanisms for Grid Environments . . . 56
D.4.1 Legion . . . 56
D.4.2 Globus . . . 57
D.4.3 Condor and Condor-G . . . 58
D.4.4 MyGrid and OurGrid . . . 59
D.4.5 MetaScheduler in GrADS Project . . . 60
D.4.6 EasyGrid . . . 61
D.4.7 ISAM . . . 61
D.5 Comparison . . . 61
D.6 Conclusion . . . 65

**E Experiments with Distributed Submission 66**
E.1 Experiments using Condor . . . 66
E.1.1 Methodology . . . 66
E.1.2 Results and Analysis . . . 67
E.2 Experiments using Monarc . . . 74
E.2.1 Methodology . . . 75
E.2.2 Results and Analysis . . . 76
E.3 Conclusion . . . 81

**F GRAND: An Integrated Application Management System for Grid Environments** **82**
F.1 Premises . . . . . 83
F.2 Architectural Model Overview . . . . . 86
F.2.1 Model Components . . . . . 89
F.2.2 Model Components Interaction . . . . . 91
F.3 Application and DAG Representation . . . . . 95
F.3.1 Application Description and GRID-ADL . . . . . 96
F.3.2 DAG and sub-DAGs Representation . . . . . 99
F.4 GRAND Steps to Execute a Distributed Application . . . . . 101
F.4.1 DAG Inference . . . . . 101
F.4.2 Clustering . . . . . 104
F.4.3 Mapping . . . . . 107
F.4.4 Submission Control . . . . . 109
F.5 Additional GRAND Services . . . . . 110
F.5.1 Data Management . . . . . 110
F.5.2 Monitoring . . . . . 113
F.6 Conclusion . . . . . 115

**G AppMan: Application Submission and Management Prototype** **116**
G.1 Implementation Details . . . . . 116
G.1.1 Monitoring Graphical Interface . . . . . 118
G.1.2 Data Management . . . . . 119
G.2 Experimental Results . . . . . 121
G.2.1 Resource Management Experiments . . . . . 121
G.2.2 Data Management Experiments . . . . . 123
G.3 Conclusion . . . . . 126

**H Conclusion** **127**
H.1 Main Contributions . . . . . 128
H.2 Future works . . . . . 130

**I GRID-ADL ABNF** **131**

**Referências Bibliográficas** **134**

# Lista de Figuras


2.1 Categorias de aplicações de grade: (a) *independent tasks*, (b) *loosely-coupled tasks* (*phase*), (c) *loosely-coupled tasks* (*pipeline*), e (d) *tightly-coupled tasks* . . . 10
2.2 Principais componentes do modelo hierárquico de gerenciamento de tarefas 13

B.1 Grid application kind: (a) independent tasks, (b) loosely-coupled tasks (phase), (c) loosely-coupled tasks (pipeline), and (d) tightly-coupled tasks 28

C.1 Diamond DAG example . . . 33
C.2 DAGMan input file . . . 35
C.3 Condor submit description files for DAG example . . . 35
C.4 Another solution for DAG example: DAGMan input file and Condor submit description file . . . 36
C.5 DAG example expressed in Chimera VDL . . . 38
C.6 DAG example expressed in GXML (skeleton) [14, 15] . . . 39
C.7 DAG example expressed in AGWL (skeleton) . . . 41
C.8 DAG example expressed in XPWSL . . . 42
C.9 DAG example expressed in GEL . . . 43
C.10 Kwok's partial taxonomy of the multiprocessor scheduling problem [87] . 47

D.1 Part of the Casavant and Kuhl's taxonomy [23] . . . 53

E.1 Experiment 1 – Load average of the submit machine . . . 68
E.2 Time measurements in the execution timeline . . . 70
E.3 Condor Experiment 1 – average task time for centralized and distributed submission . . . 72
E.4 Condor Experiment 1 – total execution time (sum) . . . 72
E.5 Condor Experiment 1 – response time in seconds . . . 73

E.6 Condor Experiment 1 – percentage of executed tasks per machine . . . . 73
E.7 Condor Experiment 2 – percentage of executed tasks per machine . . . . 74
E.8 Monarc Experiment – methodology: DAG of tasks . . . . . . . . . . . . 76
E.9 Monarc Experiment: simulated grid elements . . . . . . . . . . . . . . . . 77
E.10 Monarc - cluster 40 CPUs: (a) CPU load average and (b) logical execution time . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78
E.11 Monarc - cluster 80 CPUs: (a) CPU load average and (b) logical execution time . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 79
E.12 Monarc - grid, submit cluster with 5 CPUs: (a) CPU load average and (b) logical execution time . . . . . . . . . . . . . . . . . . . . . . . . . . . 80

F.1 The GRAND model: steps to execute a distributed application . . . . . . 83
F.2 The GRAND model: a possible scenario . . . . . . . . . . . . . . . . . . 89
F.3 The GRAND model: hierarchical architecture - main components . . . . 91
F.4 The GRAND model: *Application Manager* details . . . . . . . . . . . . . 92
F.5 The GRAND model: *Submission Manager* details . . . . . . . . . . . . . 93
F.6 The GRAND model: *Task Manager* details . . . . . . . . . . . . . . . . 95
F.7 DAG used in input file examples . . . . . . . . . . . . . . . . . . . . . . 97
F.8 Example of input file for the simple DAG (F.7(a)) . . . . . . . . . . . . . 98
F.9 Input file for DAG example F.7(b) . . . . . . . . . . . . . . . . . . . . . 99
F.10 Input file for DAG example F.7(c) . . . . . . . . . . . . . . . . . . . . . 99
F.11 Example of XML manifest for Figure F.7(b) . . . . . . . . . . . . . . . . 101
F.12 Fragment of extended JSDL file for Figure F.7(b) . . . . . . . . . . . . . 102
F.13 DAG description for Figure F.7(b) . . . . . . . . . . . . . . . . . . . . . 103
F.14 A possible example of clustering for Figure F.7(b) . . . . . . . . . . . . . 103
F.15 Clustering algorithm for independent tasks application . . . . . . . . . . 105
F.16 Clustering algorithm for phases-loosely tasks application . . . . . . . . . 106

G.1 AppMan executing main steps . . . . . . . . . . . . . . . . . . . . . . . 118
G.2 AppMan graphical interface for monitoring application execution: snapshots 120
G.3 GRID-ADL code for experiment 1 . . . . . . . . . . . . . . . . . . . . . 121
G.4 Condor code for experiment 1 . . . . . . . . . . . . . . . . . . . . . . . 121
G.5 Application execution time for AppMan and Condor . . . . . . . . . . . 123
G.6 Optimized stage in – scalability . . . . . . . . . . . . . . . . . . . . . . 125

# Lista de Tabelas

B.1 Comparison of conventional distributed environments and grids [103] . . 25

C.1 Grid description languages: comparison . . . . . . . . . . . . . . . . . . . 49

D.1 Comparison of presented schedulers – part I . . . . . . . . . . . . . . . . . 62
D.2 Comparison of presented schedulers – part II . . . . . . . . . . . . . . . . 63

E.1 Experiment 1: Statistical values for centralized (c) and distributed (d) submission . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
E.2 Experiment 2: Statistical values for centralized (c) and distributed (d) submission . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
E.3 Condor Experiment 1 – response time in seconds . . . . . . . . . . . . . 71

G.1 Execution Times: Condor and AppMan . . . . . . . . . . . . . . . . . . . 122
G.2 AppMan experimental values obtained from six nodes . . . . . . . . . . . 124

# Capítulo 1

# Introdução

O tema principal desta tese é o gerenciamento de aplicações em ambiente de computação em grade. Foi proposto um modelo de gerenciamento hierárquico de aplicações denominado GRAND (**G**rid **R**obust **A**pplicatio**n** **D**eployment). O texto da tese com o detalhamento deste modelo foi escrito em inglês e está apresentado em forma de apêndices. Este capítulo e os próximos dois capítulos apresentam um resumo estendido em português.

Um sistema distribuído pode ser definido como "uma coleção de computadores independentes que parecem ao usuário como um único computador" [130]. Na prática, poucos sistemas distribuídos possuem um grau de transparência capaz de dar esta visão de recurso único. Assim, embora algumas vezes questionável, a utilização de forma cooperativa e transparente de recursos distribuídos considerando-os como um único e poderoso computador é algo buscado há bastante tempo por diversos pesquisadores. Várias abordagens vêm sendo propostas, como por exemplo, sistemas operacionais distribuídos e ambientes de exploração de paralelismo implícito.

Uma das abordagens existentes é conhecida por diversos nomes tais como *metacomputing*, *seamless scalable computing*, *global computing* e, mais recentemente, *grid computing* ou computação de grade [9]. Esta abordagem vem se mostrando viável tecnologicamente, mesmo quando a distribuição envolve grandes distâncias ou grande heterogeneidade de recursos. De fato, a computação de grade ou *grid computing* é uma proposta promissora para resolver as crescentes demandas da computação paralela e distribuída por transparência, desempenho e capacidade computacional através do uso de recursos disponíveis em diferentes organizações.

Os ambientes de grade buscam utilizar de forma cooperativa e transparente recursos distribuídos geograficamente considerando-os como se pertencentes a um único e

poderoso computador. Estes recursos manipulados podem ser de diferentes tipos, havendo grande heterogeneidade dentro de uma mesma classe de recursos.

Segundo Foster *et al.* [54], o termo *grid computing* foi estabelecido no meio da década de 90 para denotar "uma proposta de infraestrutura computacional distribuída para engenharia e ciências avançadas". Baker *et al.* [9] consideram que a popularização da Internet e a disponibilidade de computadores poderosos e redes de alta velocidade a baixo custo fornecem a oportunidade tecnológica de se usar as redes de computadores como um recurso computacional unificado. Assim, devido ao seu foco em compartilhamentos inter-organizacionais e dinâmicos, as tecnologias de grade complementam ao invés de competir com as tecnologias de computação distribuídas existentes [54].

Uma definição um pouco mais precisa de grade é apresentada em Krauter *et al.* [84]: "uma grade é um sistema computacional de rede que pode escalar para ambientes do tamanho da Internet com máquinas distribuídas através de múltiplas organizações e domínios administrativos. Neste contexto, um sistema computacional de rede distribuído é um computador virtual formado por um conjunto de máquinas heterogêneas ligadas por uma rede que concordam em compartilhar seus recursos locais com os outros."

Assim, podemos dizer que a infraestrutura de grade deve prover de forma global e transparente os recursos requisitados por aplicações de grande demanda computacional e/ou de dados, como por exemplo aplicações em física de altas energias (HEP) ou em biologia. Esta infraestrutura pode ligar e unificar globalmente recursos diversos e remotos abrangendo de sensores meteorológicos a dados de estoque, de supercomputadores paralelos a organizadores digitais pessoais. Deste modo, é dito que ela deve prover "serviços pervasivos para todos os usuários que precisarem deles". Esta conclusão leva ao conceito de computação pervasiva (*pervasive computing* ou *ubiquitous computing*). Os termos "pervasivo" e "pervasiva" são utilizados em alguns textos sobre computação sem fio e de grade, embora a tradução mais correta para o português talvez fosse difundido(a) ou ubíquo(a), isto é, algo que está ao mesmo tempo em toda a parte. O objetivo dos pesquisadores neste contexto é criar um sistema que esteja integrado ao ambiente computacional, estando totalmente conectado e constantemente disponível, bem como sendo intuitivo e realmente portátil.

Existem três principais aspectos que caracterizam *grids* computacionais:

- *heterogeneidade (heterogeneity)*: uma grade envolve uma multiplicidade de recur-

sos que são heterogêneos por natureza e que podem estar dispersos por numerosos domínios administrativos através de grandes distâncias geográficas;

- *escalabilidade (scalability)*: uma grade pode crescer de poucos recursos para milhões. Isto levanta o problema da potencial degradação do desempenho à medida que o tamanho de uma grade cresce. Conseqüentemente, aplicações que requerem um grande número de recursos dispersos geograficamente devem ser projetados para serem extremamente tolerantes a latência;

- *dinamicidade ou adaptabilidade (dynamicity or adaptability)*: em uma grade, a falha de um recurso é a regra, e não a exceção. De fato, com tantos recursos, a probabilidade de que algum recurso falhe é naturalmente alta. Os gerenciadores de recursos ou aplicações devem adaptar o seu comportamento dinamicamente a fim de extrair o máximo de desempenho a partir dos recursos e serviços disponíveis.

Um ambiente de grade ideal irá prover acesso aos recursos disponíveis de forma homogênea de tal modo que descontinuidades físicas tais como diferenças entre plataformas, protocolos de rede e barreiras administrativas se tornem completamente transparentes. Deste modo, deseja-se que um *grid middleware* torne um ambiente radicalmente heterogêneo em um virtualmente homogêneo. Este ideal talvez seja algo impossível de ser obtido, mas com certeza algum grau de homogeneidade e abstração deve ser fornecido às aplicações.

Existem problemas ou sub-áreas com características de demandas computacionais particulares. Stockinger [125] afirma que é possível dividir os campos de pesquisa em grade em duas áreas: *Computational Grid* e *Data Grid*. No primeiro caso, temos de certa forma uma extensão natural da tecnologia de *cluster*, onde grandes tarefas computacionais devem ser computadas em recursos computacionais distribuídos. Já um *Data Grid* trata do gerenciamento, localização (*placement*) e replicação eficientes de quantidades bastante grandes de dados. Note que no primeiro caso existirão dados envolvidos, bem como no segundo caso, tarefas computacionais também irão executar neste ambiente após a disponibilização dos dados.

Pode-se identificar pelo menos três comunidades que precisam de acesso a fontes de dados distribuídas [126], não considerando apenas as aplicações de *data grid*: (1) bibliotecas digitais (e coleções de dados distribuídos): possuem serviços para manipulação,

procura e visualização de dados; (2) ambientes de grade para processamento de dados distribuídos: permitem a execução de diversos tipos de aplicações tais como visualização distribuída e descoberta de conhecimento; e (3) armazenamentos persistentes: disponibilizam dados independentes da tecnologia de armazenamento. O ideal é que embora com objetivos diferentes todas pudessem manipular seus dados em fontes distribuídas através de uma interface (API – *Application Programming Interface*) comum. Esta API deve ser igual seja qual for a forma de armazenamento: objetos em Bancos de Dados Orientados a Objetos (BDOO), BLOBs (*Binary Large OBject*) em Banco de Dados objeto-relacional, ou como arquivo.

Mesmo dentro de uma mesma comunidade ou classe de aplicação haverá características peculiares. Por exemplo, em ambientes de grade para processamento de dados distribuídos podemos verificar estas demandas peculiares pela caracterização de três classes de aplicações: (a) Física de altas energias (*High Energy Physics* ou HEP): uma fonte gerando grandes quantidades de dados (acelerador) que são utilizados por pesquisadores em diferentes países [21]; (b) Observação da Terra: várias fontes distribuídas (estações) gerando dados independentes que são posteriormente processados de forma integrada [120]; (c) Bioinformática: várias bases independentes, e freqüentemente heterogêneas, que precisam ser integradas para realizar uma determinada análise [86].

Na primeira aplicação, temos o problema de armazenar, em princípio em um único local, um volume muito grande de dados gerado a partir de uma fonte única e encontrar uma forma eficiente de difundir esses dados para locais cuja conexão possivelmente tenha restrições de largura e banda, além de restrições de espaço de armazenamento local. Nos dois outros casos, temos os dados globais formados a partir de informações armazenadas de forma distribuída. A grande diferença é que no segundo caso temos o fator da heterogeneidade de armazenamento como um complicador.

Vários trabalhos vêm sendo propostos para tratar o gerenciamento de recursos e aplicações no ambiente de grade (*computational grid*) [9, 13, 50, 51, 101, 113]. No entanto, escalonar e controlar a execução de aplicações compostas por centenas ou milhares de tarefas ainda apresenta-se como um desafio técnico e científico. Isso se deve principalmente a dois fatores. Primeiro, o grande número de tarefas pode causar uma sobrecarga na máquina de submissão. Segundo, o controle manual é proibitivo uma vez que estas aplicações (a) utilizam uma grande quantidade de recursos e (b) podem levar vários dias ou meses para serem concluídas com sucesso. Existem várias classes de aplicações que

podem ser caracterizadas como de alta demanda por recursos computacionais tais como ciclos de CPU e/ou armazenamento de dados. Por exemplo, pesquisas em física de altas energias (HEP) [21] e bioinformática [86] usualmente exigem processamento de grandes quantidades de dados utilizando algoritmos que fazem uso de muitos ciclos de CPU.

Geralmente, estas aplicações são compostas por tarefas e a maioria dos sistemas trata cada tarefa individualmente como se elas fossem aplicações *stand-alone*. Freqüentemente, as aplicações são compostas por tarefas hierárquicas que precisam ser tratadas em conjunto, ou porque elas precisam de alguma forma de interação com o usuário ou porque precisam se comunicar. Estas aplicações também podem apresentar uma característica de grande escala e disparar um grande número de tarefas exigindo a execução de centenas ou centenas de milhares de experimentos.

A maioria dos sistemas computacionais existentes falha ao tratar dois problemas: (1) o gerenciamento e o controle de um grande número de tarefas; e (2) a regulagem (ou balanceamento) da carga da máquina de submissão e do tráfego da rede. Um trabalho na direção do item (1) é o de Dutra *et al.* [39] que reporta experimentos de programação em lógica indutiva que geraram mais de quarenta mil tarefas, e cujo foco era prover uma ferramenta para o usuário para o controle e monitoração da execução de uma aplicação específica.

Acredita-se que uma hierarquia de gerenciadores, que distribua dinamicamente dados e tarefas, pode ajudar o gerenciamento de aplicações, tratando os problemas citados. Este texto apresenta um modelo de gerenciamento deste tipo de aplicações, baseado em submissão e controle particionados e hierárquicos, denominado GRAND ( **G**rid **R**obust **A**pplicatio**n** **D**eployment) [114, 142, 143, 144, 145, 146, 147].

Este trabalho pressupõe que aplicações formadas por tarefas distribuídas devam ser executadas em um ambiente heterogêneo de máquinas não necessariamente dedicadas, podendo ser formado por diversos domínios (i.e, uma grade formada tanto por *clusters* quanto por redes locais). Neste contexto, as tarefas podem ter dependências na sua ordem de execução, mas não realizam comunicação por troca de mensagens.

O objetivo é garantir que, além da realização do escalonamento das tarefas, o ambiente de execução realize o gerenciamento dos dados envolvidos na computação bem como a autenticação e autorização para execução das tarefas. Também busca-se garantir a integridade dos dados e controlar o fluxo no retorno dos arquivos de resultados.

O modelo de gerenciamento GRAND (**G***rid* **R***obust* **A***pplicatio***n** **D***eployment*) reali-

za a submissão e o controle de forma distribuída e hierárquica de aplicações compostas por uma grande quantidade de tarefas em um ambiente de computação em grade. Este modelo disponibiliza mecanismos para o gerenciamento hierárquico que pode controlar a execução das tarefas preservando a localidade dos dados ao mesmo tempo que reduz a carga das máquinas de submissão.

As principais contribuições desta tese são:

- a definição de um modelo arquitetural para realizar o gerenciamento de aplicações. Este modelo, denominado GRAND, é composto por várias etapas e permite a distribuição da submissão e do controle da aplicação;

- uma nova linguagem de descrição de aplicação denominada GRID-ADL (*GRID Application Description Language*), uma linguagem baseada em *script* com definição implícita do grafo da aplicação e facilidades para expressar um grande número de tarefas;

- um novo esquema XML para descrição de aplicações. Foi proposta uma extensão do padrão JSDL [79] proposto pelo Global Grid Forum [61] permitindo expressar dependências entre tarefas;

- escalonamento proposto dividido em etapas de agrupamento (*clustering*) e mapeamento (*mapping*). Foram propostos também uma taxonomia de aplicações e algoritmos distintos para o tratamento de cada classe de aplicação;

- utilização de técnicas simples para o gerenciamento de dados que se mostraram efetivas para um bom aproveitamento dos recursos computacionais;

- a implementação de um protótipo que permite a execução de aplicações no ambiente ISAM/EXEHDA com algumas das principais funcionalidades deste modelo. O protótipo foi avaliado com algumas aplicações.

No próximo capítulo apresenta-se os principais detalhes do modelo GRAND. No Capítulo 3, apresenta-se as considerações finais relacionadas a este resumo estendido, bem como a organização dos anexos contendo o detalhamento da tese.

# Capítulo 2

# O Modelo GRAND

Este capítulo apresenta uma visão geral do modelo GRAND bem como do protótipo AppMan. Ele resume alguns dos principais pontos apresentado nos apêndices. O restante deste capítulo está organizado da seguinte forma. Inicialmente apresentamos as premissas iniciais que nortearam a concepção do modelo GRAND (Seção 2.1). Depois, as aplicações mais importantes de ambiente de grade são caracterizadas (Seção 2.2).

Na Seção 2.3, alguns aspectos sobre gerenciamento de dados são apresentados. Uma discussão sobre o particionamento de aplicações para execução em um ambiente de grade é realizada na Seção 2.4. Depois, é apresentado o modelo de gerenciamento hierárquico de aplicações GRAND, que pode controlar a execução de um grande número de tarefas distribuídas preservando a localidade dos dados ao mesmo tempo que reduz a carga das máquinas de submissão (Seção 2.5). Finalmente, o protótipo AppMan, implementado com o *middleware* EXEHDA, é analisado (Seção 2.6) e as conclusões e trabalhos futuros são apresentados (Seção 2.7).

## 2.1 Premissas

Considerando que atualmente os ambientes de grade envolvem principalmente instituições de ensino e pesquisa, e partindo do pressuposto que nestas instituições estão sendo executadas aplicações usualmente classificadas como aplicações científicas, limita-se o escopo do nosso ambiente alvo conforme descrito nos próximos itens.

**Ambiente é heterogêneo.** Heterogeneidade é característica inerente ao ambiente de grade. Se a idéia é executar em ambiente de grade, nada mais natural que considerar heterogenei-

dade. Tratar heterogeneidade também afeta diretamente a política de escalonamento que precisa saber que as máquinas têm características distintas de hardware e software.

**Um grande número de tarefas pode ser submetido.** Esta premissa é também uma das motivações principais deste trabalho e é fundamental na definição de vários detalhes do modelo. Por um número grande de tarefas, refere-se a aplicações que geram centenas ou milhares de processos.

Um usuário pode submeter de sua máquina de submissão (*home machine* ou *submit machine*) uma aplicação com dezenas de milhares de tarefas. Se toda a submissão e controle forem realizadas por esta mesma máquina, provavelmente a máquina de submissão irá responder de forma muito lenta ao usuário impossibilitando que ele/ela continue trabalhando nesta máquina. Este problema já é conhecido na literatura. Por exemplo, o gerenciador de recursos Condor [136] permite que o usuário especifique um limite de tarefas que podem ser submetidas em uma máquina específica ao mesmo tempo. Acredita-se que embora resolva o problema, esta não seja a melhor solução, uma vez que exige que o usuário tenha alguma experiência prévia com submissão de tarefas no ambiente corrente para determinar o limite apropriado que evite travar a máquina e ao mesmo tempo que garanta um bom grau de concorrência.

Além disso, a monitoração da execução e o tratamento de erros de forma manual também não é factível uma vez que este tipo de aplicação (a) usa um grande número de recursos e (b) podem levar vários dias ou semanas para terminar com sucesso. Deste modo, nossa proposta precisa ser escalável e deve prover retorno ao usuário.

**Tarefas não se comunicam por troca de mensagem.** Vários trabalhos permitem troca de mensagens em ambiente de grade [81, 82]. Entretanto, troca de mensagens introduzem vários aspectos a serem considerados nas fases de agrupamento e mapeamento. Por isso, assumimos que as aplicações a serem gerenciadas pelo nosso modelo não se comunicam por troca de mensagens.

**Tarefas podem ter dependências com outras tarefas devido ao compartilhamento de arquivos.** Aplicações podem ser modeladas como um grafo de dependência de tarefas, onde as dependências ocorrem devido ao compartilhamento de arquivos. Por exemplo, se uma tarefa $a$ produz um arquivo de saída $f_a$, que uma tarefa $b$ utiliza como arquivo de

entrada, então a tarefa $b$ deve aguardar até que a tarefa $a$ termine a sua execução. Deste modo, as decisões de escalonamento devem considerar tanto tarefas independentes quanto com dependência.

**Um grande número de arquivos pode ser manipulado pelas tarefas.** Como as tarefas podem se comunicar e sincronizar através de arquivos, cada tarefa normalmente manipula pelo menos dois arquivos (uma entrada e uma saída). Como assumimos um número grande de tarefas, conseqüentemente teremos um número grande de arquivos. Algoritmos eficientes para manter a localidade dos dados e para transferir arquivos de forma eficiente são cruciais para o sucesso do modelo.

**Arquivos grandes podem ser usados na computação.** A transferência de arquivos muito grandes pode causar o congestionamento da rede, perda de pacotes e tornar o tempo de transmissão muito alto. Preservar localidade, e o uso de técnicas de *staging* e *caching* podem ajudar a minimizar a perda de desempenho devido a latência de transmissão de dados.

**A infraestrutura de grade utilizada é segura.** Assumimos que existe uma conexão segura entre os nodos da grade. Assumimos também que quaisquer tipos de autenticação e autorizações necessários para a execução da aplicação serão disponibilizado por uma infraestrutura de grade pré-existente. Por exemplo, o GSI [149] do Globus pode ser utilizado.

**A infraestrutura de grade utilizada permite a descoberta dinâmica de recursos.** Descoberta (*discovery*) é um serviço que permite que um sistema recupere a descrição de recursos. Existem vários serviços de descoberta para ambiente de grade reportados na literatura [64, 99, 117]. Consideramos que a infraestrutura de grade disponibiliza um ou mais serviços de descoberta.

**Cada nodo da grade possui o seu gerenciador de recursos local.** Cada nodo da grade possui o seu próprio sistema de gerenciamento de recursos (RMS) local.

**Uma tarefa submetida à um RMS local irá executar até a sua finalização.** Assumimos também que uma vez que uma tarefa seja alocada, ela não será mais escalonada, ou

pelo menos qualquer tipo de migração ou re-escalonamento ocorrerá de forma transparente.

## 2.2 Caracterização das Aplicações

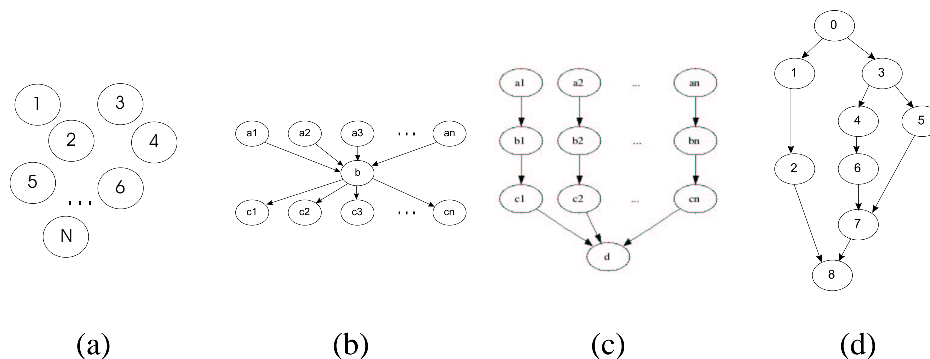
Considerando-se aplicações típicas de ambientes de grade, sem troca de mensagens, que podem se comunicar via troca de arquivos e que sabem previamente quantos processos precisam ser criados, propõe-se a seguinte taxonomia de aplicações [143, 145, 146]:

- **independent tasks** ou tarefas independentes caracterizam o tipo mais simples de aplicação onde as tarefas não possuem dependências entre si. Esse tipo de aplicação é muitas vezes chamado de *bag-of-tasks*. Simulações de Monte Carlo, tipicamente usadas em Física de Altas Energias (HEP) são exemplos deste tipo de aplicação;

- **loosely-coupled tasks** ou tarefas fracamente acopladas são caracterizadas por poucos pontos de compartilhamento, i.e., uma aplicação dividida em fases (*phases*) ou em seqüência (*pipeline*). Experimentos em programação em lógica indutiva são exemplos de fracamente acoplada em fases;

- **tightly-coupled tasks**: ou tarefas fortemente acopladas caracterizadas por grafos complexos. Aplicações de programação em lógica com restrições (CLP) normalmente se enquadram nesta categoria.

A Figura 2.1 ilustra de forma visual os grafos das aplicações que são típicos de cada uma das categorias da taxonomia proposta.




(a) (b) (c) (d)

Figura 2.1: Categorias de aplicações de grade: (a) *independent tasks*, (b) *loosely-coupled tasks* (*phase*), (c) *loosely-coupled tasks* (*pipeline*), e (d) *tightly-coupled tasks*

## 2.3 Gererenciamento de Dados

Uma das tarefas a serem tratadas pela infraestrutura de grade é o gerenciamento de dados, estejam eles armazenados em arquivos ou em bases de dados. Como estas infraestruturas ainda estão em desenvolvimento, ainda há muito a ser feito nesta questão. Inicialmente, cada usuário resolvia o tratamento dos dados através de soluções específicas, muitas vezes recorrendo a *shell scripts* ou a transferências de arquivo manuais via protocolo *ftp*. Vários trabalhos vêm se preocupando principalmente com os dados em forma de arquivos, até por ser relativamente simples extrair dados armazenados em um banco de dados para o formato de arquivo texto. Existem propostas desde protocolos eficientes de transferência de dados como o GridFTP [1] do Projeto Globus até sistemas de arquivos como o LegionFS [150]. No caso específico do modelo GRAND, será tratado apenas o gerenciamento de arquivos, com relação a enviar dados para o local de processamento, bem como fazer o retorno dos resultados (respectivamente *stage in* e *stage out*).

Um gerenciador de aplicações precisa controlar o envio dos dados e tarefas e o recebimento dos resultados gerados das tarefas. A máquina na qual o usuário submete um número grande de tarefas é denominada máquina de submissão (*submit machine*). Temos uma *arquitetura hierárquica* onde na máquina de submissão fica um gerenciador que delega o encargo de submeter as tarefas a outros gerenciadores, e aguarda o resultado final da execução das tarefas sem ter que se preocupar com os detalhes. Esse gerenciador é denominado *Gerenciador de Aplicação* (*Application Manager* ou *AM*).

O gerenciador da máquina de submissão dispara ou utiliza gerenciadores já previamente inicializados do sistema que tem como função fazer o escalonamento das tarefas (achar máquinas com que atendam aos requisitos do usuário). Além disso, ele deve procurar privilegiar a localidade dos dados. Uma vez que os dados cheguem aos gerenciadores, o envio dos resultados à máquina *home* é feita de forma controlada evitando congestionamento.

Uma vantagem deste esquema é liberar a carga da máquina onde é feito o disparo da aplicação, já que normalmente é a máquina de trabalho do usuário. Além disso, caso haja falha em um dos gerentes, é mais barato fazer a recuperação (elimina o ponto único de falha).

Além disso, são propostas otimizações no envio de arquivos de dados compartilhados por mais de uma tarefa, permitindo a criação de uma espécie de cache.

## 2.4 Particionamento

O problema de particionamento a ser tratado neste contexto pode ser definido do seguinte modo. Pressupõe-se que as aplicações submetidas são formadas por tarefas que podem ter dependências na sua ordem de execução, mas não realizam comunicação por troca de mensagens. Estas dependências são determinadas pelos dados de entrada e saída (normalmente arquivos de dados), conforme explicitado em um arquivo de descrição da aplicação.

A descrição da aplicação deve ser feita utilizando a linguagem de descrição GRID-ADL (*Grid Application Description Language*) [114, 146] proposta nesta tese. Tal linguagem é simples porém poderosa. Ela permite que grafos sejam descritos de forma implícita.

Uma aplicação pode então ser representada como um grafo onde cada nodo representa uma tarefa e as arestas representam a ordem de precedência entre as tarefas. Este grafo é necessariamente um grafo direcionado acíclico (DAG ou *directed acyclic graph*). Para que este DAG possa ser executado, ele precisa primeiro ser particionado. Particionamento neste contexto é a divisão do DAG em sub-grafos de tal forma que eles possam ser alocados em diferentes processadores disponíveis de forma eficiente. Para realizar o particionamento do DAG, será utilizado o algoritmo DSC (*Dominant Sequence Clustering*) [158]. O algoritmo DSC possui complexidade $O((v + e) \log v)$ onde $v$ é o número de tarefas e $e$ o número de arestas.

## 2.5 Modelo de Gerenciamento

O modelo GRAND trata três aspectos do gerenciamento de dados: (a) os dados de entrada são transferidos automaticamente para o local onde o arquivo será necessário como ocorre em outros trabalhos (e.g. [135]); (b) como o volume de dados a ser tratado é potencialmente muito grande, o modelo contempla o envio dos resultados ao usuário de forma controlada para evitar congestionamento da rede; (c) o escalonamento prioriza a localidade no disparo de tarefas para evitar transferências desnecessárias de dados transientes e conseqüente degradação do desempenho.

O disparo e controle das aplicações é feito através de uma hierarquia de gerenciadores conforme ilustrado na Figura 2.2: (nível 0) um usuário submete uma aplicação em uma máquina através do *Application Manager*; (nível 1) os *Application Managers* enviam aos *Submission Manager* descrições de tarefas; (nível 2) os *Task Managers* são instanciados

sob demanda pelos *Submission Managers* a fim de controlar a submissão de tarefas a escalonadores de domínios específicos da grade; (nível 3) escalonadores nos domínios específicos recebem requisições dos *Task Managers* e fazer a execução de fato das tarefas.

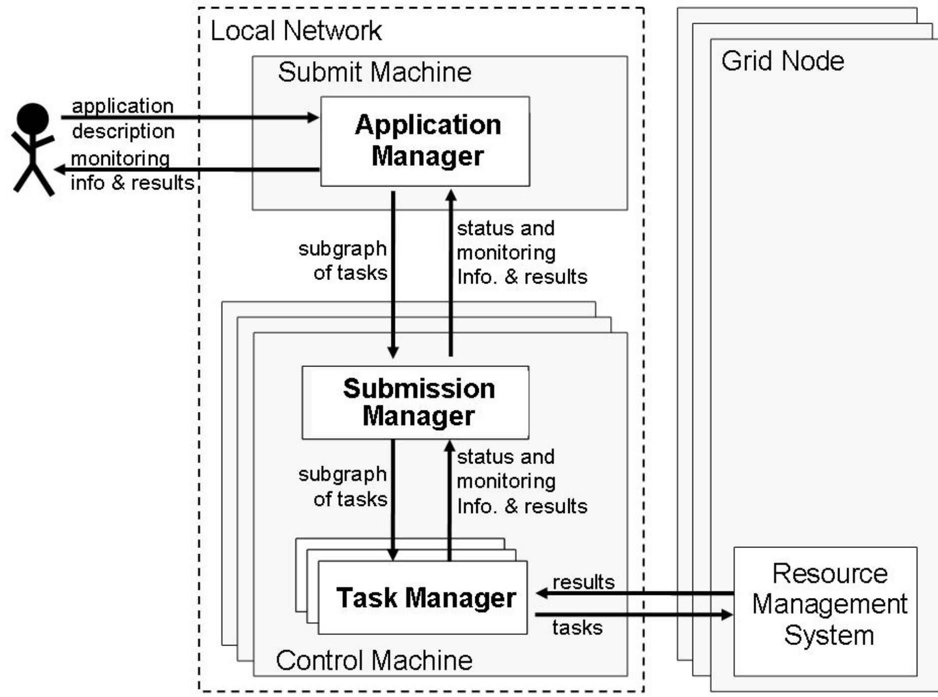



Figura 2.2: Principais componentes do modelo hierárquico de gerenciamento de tarefas

O *Application Manager* é encarregado de: (a) receber um arquivo de entrada descrevendo as tarefas; (b) particionar as tarefas em sub-grafos que são enviados para diferentes *Submission Managers*, buscando manter a localidade de dados e minimizar a comunicação entre os *Submission Managers*; (c) mostrar ao usuário, de forma amigável, informações do estado da execução da aplicação.

As principais funções do *Submission Manager* são: (a) decidir a alocação dos subgrafos com base em informações dinâmicas sobre os recursos computacionais; (b) indicar o estado da execução e falhas através de comunicação periódica com o *Submission Manager*; (c) acompanhar o andamento da aplicação e recuperar falhas através de um registro persistente (*log*); (d) criar e monitorar os *Task Managers*; (e) avaliar, durante a execução, se deve ou não continuar a execução de tarefas em máquinas que forem liberadas ou retornarem após uma falha.

Cada *Task Manager* é responsável por: (a) se comunicar com o escalonador de um determinado domínio a fim de garantir a execução remota de tarefas; (b) garantir a ordem

de execução de acordo com as dependências de dados; (c) controlar a transferência de dados garantindo a disponibilidade dos dados de entrada e o recebimento dos dados de saída.

O uso de uma hierarquia de escalonadores é considerado na literatura com uma boa alternativa para o ambiente de grade [140, 84]. O principal diferencial do modelo proposto é a hierarquia de gerenciadores de submissão acima do meta escalonador. Deste modo, pode-se notar que o modelo proposto realiza um balanceamento da carga computacional necessário para controlar a execução das tarefas, evitando que a máquina de submissão fique sobrecarregada e que ocorra perda de dados por congestionamento da rede. Além disso, permite que escalonadores já existentes sejam integrados em um único ambiente de escalonamento.

## 2.6 AppMan: Protótipo GRAND

A entrada do protótipo é a descrição da aplicação. O usuário deve especificar a aplicação através da linguagem de descrição GRID-ADL. Em GRID-ADL, o usuário descreve apenas as características das tarefas incluindo arquivos manipulados. Como considera-se que as dependências entre tarefas são programadas usando arquivos de dados, as dependências entre as tarefas são inferidas automaticamente através da análise do fluxo de dados. O *parser* para leitura do arquivo de descrição e a inferência do DAG foram implementados usando a ferramenta JavaCC [78].

Uma vez que o DAG tenha sido obtido, o protótipo pode iniciar a execução do mesmo. Para implementar as fases de submissão e controle da execução, utilizou-se a linguagem Java dentro do ambiente EXEHDA.

EXEHDA (**Ex***ecution* **E***nvironment for* **H***igh* **D***istributed* **A***pplications*) [155] é um modelo destinado à execução de aplicações distribuídas. As aplicações alvo são distribuídas e contemplam mobilidade de hardware e software, sendo baseadas no paradigma de programação empregado pelo projeto ISAM (**I**nfra-estrutura de **S**uporte às **A**plicações **M**óveis)

O protótipo implementado possui um *Application Manager* para disparar e monitorar a execução de cada aplicação em máquinas de uma rede local. Cada máquina possui um *Submission Manager*. No estado atual da implementação, os *Submission Managers* não se comunicam nem sabem a localização de outros *Submission Managers*. Deste modo, as

aplicações funcionam desde que se assuma que o particionamento gera grafos totalmente disjuntos, i.e. sem dependência entre os *Submission Managers*, e (2) cada sub-grafo executa até o final sem falhas. Uma interface gráfica simplificada apresenta para o usuário a representação do DAG a medida que as tarefas são executadas.

Como principais contribuições desta implementação, destaca-se a possibilidade de verificar (1) a viabilidade do modelo proposto e (2) o potencial do ambiente de programação ISAM/EXEHDA.

Como trabalhos futuros inclui-se o refinamento e a otimização do protótipo bem como a obtenção de mais dados experimentais.

## 2.7 Conclusão

Este capítulo apresentou a visão geral da tese que trata do gerenciamento de aplicações em ambientes de grade, focando em aplicações que disparam um grande número de tarefas e dados através da rede da grade. Foi projetado um modelo arquitetural para ser implementado como um *middleware* denominado GRAND. A fim de projetar tal arquitetura, foi considerado que os recursos e tarefas são modelados como grafos. Este texto apresentou a implementação do modelo GRAND utilizando o ambiente de programação ISAM/EXEHDA.

# Capítulo 3

# Estrutura dos Apêndices

O detalhamento deste trabalho encontra-se nos apêndices, que constituem a tese propriamente dita, e está organizado do seguinte modo:

**Apêndice A:** Apresenta uma introdução com uma motivação e uma visão geral sobre a tese.

**Apêndice B:** Apresenta os conceitos básicos de computação em grade (*grid computing*), incluindo classificações apresentadas na literatura, e nossa taxonomia proposta para aplicações distribuídas.

**Apêndice C:** Trata de alguns dos principais aspectos relacionados ao gerenciamento de aplicações.

**Apêndice D:** Analisa alguns dos gerenciadores de recursos (*resource management systems* ou RMSs) que são utilizados para gerenciar recursos, principalmente CPU, em nodos de uma grade.

**Apêndice E:** Reporta os experimentos realizados para verificar a influência da submissão distribuída em um ambiente distribuído (*cluster* e/ou *grid*). Os dados dos experimentos conduzidos em um *cluster* gerenciado pelo RMS Condor e em um ambiente de simulação construído com o Monarc também são analisados.

**Apêndice F:** Apresenta e analisa as principais características e contribuições do modelo de gerenciamento de aplicações proposto nesta tese. Este modelo é denominado GRAND

(**G**rid **R**obust **A**pplicatio**n** **D**eployment) e propõe principalmente a utilização de uma estrutura hierárquica para realização do gerenciamento da aplicação.

**Apêndice G:** O protótipo implementado, denominado AppMan (**App**lication **Man**ager), é apresentado, e alguns experimentos analisados.

**Apêndice H:** As considerações finais e os trabalhos futuros concluem a descrição da tese.

**Apêndice I:** Ao final tem-se uma descrição da linguagem GRID-ADL, proposta nesta tese, em notação ABNF.

# Apêndice A

# Overview

The subject of this thesis is the application management in a grid computing environment. We understand application management in a grid environment as the task of submitting and monitoring the execution progress of all tasks that compose the user application. Application management encompasses several activities, as we will analyze in this thesis, such as application description, mapping, submission, and monitoring .

We focus on applications that spread a huge number (thousands) of tasks and manipulate very large number of files across the grid. We propose the GRAND (**G**rid **R**obust **A**pplicatio**n** **D**eployment) model [114, 142, 143, 144, 145, 146, 147] to perform such application management.

## A.1 Motivation

The term *grid computing* [50, 54] was coined in the mid-1990s to denote a distributed computing infrastructure for scientific and engineering applications. The grid can federate systems into a supercomputer beyond the power of any current computing center [10]. Several works on grid computing have been proposed in the last years [9, 13, 50, 51, 101, 113].

A grid computing environment supports sharing and coordinated use of heterogeneous and geographically distributed resources. These computational resources are made available transparently to the application independent of its physical location as if they belong to a single and powerful logical computer. These resources can be CPUs cycles, storage systems, network connections, or any other resource made available due to hardware or software. In the last years, many works on resource management for grid computing environments have been proposed [11, 26, 43, 84, 107, 134, 140, 159]. However, no single

work presented in the literature can deal properly with all issues related to application management.

Grand Challenge applications are fundamental problems in science and engineering with broad economic and scientific impact [65]. They have a high demand for computational resources such as CPU cycles and/or data storage. For instance, research in High Energy Physics (HEP) and Bioinformatics usually requires processing of large amounts of data using processing intensive algorithms. The major HEP experiments for the next years aim to find the mechanism responsible for mass in the universe and the "Higgs" particles associated with mass generation [21]. These experiments will be conducted through collaborations that encompass around 2000 physicists from 150 institutions in more than 30 countries. One of the largest collaborations is the Compact Muon Solenoid (CMS) project. The CMS project estimates that 12-14 Petabytes of data will be generated each year [28].

In Bioinformatics, genomic sequencing is one of the hot topics. The genomic sequences are being made public on a lot of target organisms. A great amount of gene sequences are being stored in public and private databases. It is said that the quantity of stored genomic information should double every eight months. The more the quantity of information increases, the more computation power is required [86].

There are several applications that can run in a grid computing environment. We consider only applications composed by several tasks which can have dependencies through file sharing. We classify those applications in three types, as we present in more detail later on: independent tasks (bag-of-tasks), loosely-coupled tasks (few sharing points), and tightly-coupled tasks (more complex dependencies).

Usually, applications are composed of tasks and most systems deal with each individual task as if they are stand-alone applications. Very often, as for example in some applications of HEP, they are composed of hierarchical tasks that need to be dealt with altogether, either because they need some feedback from the user or because they need to communicate. These applications can also present a large-scale nature and spread a very large number of tasks requiring the execution of thousands or hundreds of thousands of experiments. Most current software systems fail to deal with these two problems: (1) manage and control large numbers of tasks; and (2) regulate the submit machine load and network traffic, when tasks need to communicate. One work in the direction of item (1) is that of Dutra *et al.* [39] that reported experiments of inductive logic programming that generated over 40 thousand jobs that required a high number of resources in parallel in

order to terminate in a feasible time. However, this work was concentrated on providing an user level tool to control and monitor that specific application execution, including automatic resubmission of failed tasks.

Dealing with huge amounts of data requires solving several problems to allow the execution of the tasks as well as to get some efficiency. One of them is data locality. Some applications are programmed using data file as a means of communication to avoid interprocess communication. Some of the generated data can be necessary only to get the final results, that is, they are intermediate or transient data which are discarded at the end of the computation. Transient data can exist in loosely- and tightly-coupled tasks. We consider that grouping dependent tasks and allocating them to the same grid node allows to keep data locality. The goal is to get a high data locality so that data transfer costs are minimized.

So, applications that spread a large number of tasks must receive a special treatment for submission and execution control. Submission of a large number of tasks can stall the machine. A good solution is to have some kind of distributed submission control. Besides, monitoring information to indicate application progress must be provided to make the system user friendly. Finally, automatic fault detection is crucial since handling errors manually is not feasible.

Our work deals with these application control limitations, focusing on applications that spread a very large number of tasks. These non trivial applications need a powerful distributed execution environment with many resources that currently are only available across different network sites. Grid computing is a good alternative to obtain access to the needed resources.

## A.2 Goals and Contributions

Two open research problems in grid environments are addressed in this work. The first one is the task submission process. Controlling the execution of an application which is embarrassingly parallel is usually not complex. However, if the application is composed by a huge number of tasks, the execution control is a problem. Tasks of an application can have dependencies. The user should not need to start such tasks manually. Besides, the number of tasks can be very large. The user should not need to control start and termination of each task. Moreover, tasks should not be started from the same machine to avoid

(1) overloading the submit machine, and (2) the submit machine becomes a bottleneck. In this work we employ a hierarchical application management organization for controlling applications with a huge number of tasks and distribute the task submission among several controllers. We believe that a hierarchy of adaptable and grid-aware controllers can provide efficient, robust, and resilient execution.

The second open problem is the data locality maintenance when tasks are partitioned and mapped to remote resources. In this work, the applications are distributed applications, i.e. applications composed by several tasks. Tasks can be independent or dependent due to file sharing. The application partitioning goal is to group tasks in blocks that are mapped to available processors. This process must keep data locality, which means to minimize data transfer through processors and to avoid unnecessary data transfers. This is fundamental to get good application performance.

The main contributions achieved in this thesis are the following:

- the definition of an architectural model to perform application management called GRAND (**G**rid **R**obust **A**pplicatio**n** **D**eployment) [114, 143, 144, 145, 146, 147]. This model is composed by several steps we formalized and allows a distributed submission of the application. GRAND is an application management system for grid environments, and differs from other approaches [22, 89, 134] in that it incorporates features to support application scalability, data locality, and network flow control;

- a new application description language, called GRID-ADL (**Grid** **A**pplication **D**escription **L**anguage) [114], a script like language with implicit DAG definition and facilities to express a large number of tasks;

- a new XML-based description of application DAGs. It was proposed as an extension of JSDL standard [79] proposed by the Global Grid Forum [61];

- the steps proposed includes clustering and mapping. Clustering is a kind of static partitioning that is performed without information about the grid nodes. Clusters are defined to be submitted altogether. An application taxonomy was proposed and we argue that for different kinds of applications, there is a clustering algorithm more appropriate;

- a prototype was implemented and evaluated [142, 144]. The obtained results showed good scalability for resource and data management.

The remaining of this text is organized as follows. First, we present basic concepts related to grid computing (Appendix B). Then, we analyze concepts and works related to application management (Appendix C). We focus mainly on application partitioning. We also analyze concepts and works in resource management for grid (Appendix D).

We introduce some experimental motivation to our model design in Appendix E. We present and analyze our hierarchical model in Appendix F. We discuss some of the problems that need to be solved to satisfy user needs, which are the motivation to our model. We also present our architecture.

Then, we analyze some results obtained using our prototype in Appendix G. Our results show that our proposal is promising. Finally, we conclude this text with our final remarks and future works (Appendix H).

# Apêndice B

# Grid Computing Concepts

Grid computing is closely related to the distributed computing and network research areas. It differs from conventional distributed computing in the focus on large-scale resource sharing, innovative applications, and, in some cases, high-performance and/or high-throughput orientation [54, 88]. It also needs a network infrastructure to support high-bandwidth consumption and/or high-throughput needs.

This chapter presents basic concepts related to grid computing. First, we present grid computing definitions (Section B.1). Then, we discuss about grid applications (Section B.2), and grid computational systems (Section B.3). Finally, some final considerations are presented (Section B.4).

## B.1 Definitions of Grid Computing

An increasing number of research groups have been working in the field of network wide-area distributed computing [12, 36, 37, 62, 71, 97, 121, 135]. They have been implementing middleware, libraries, and tools that allow cooperative use of geographically distributed resources. These initiatives have been known by several names [9, 113] such as metacomputing, global computing, and more recently grid computing.

The term grid computing was coined by Foster and Kesselman [48] in late 90's as a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities. Actual grid computing efforts can be seen as the third phase of metacomputing evolution as stated in the Metacomputing's paper of Smarr and Catlett in 1992 [123]: a transparent network that will increase the computational and information resources available to an application. It is also a synonym of metasystem [70] which "supports the illusion of a single machine by

transparently scheduling application components on processors; managing data migration, caching, transfer, and the masking of data-format differences between systems; detecting and managing faults; and ensuring that users' data and physical resources are protected, while scaling appropriately".

The Global Grid Forum (GGF) [61] is a community-initiated forum of thousands of individuals, representing over 400 organizations in more than 50 countries. The GGF creates and documents technical specifications, user experiences, and implementation guidelines. The GGF's Grid Scheduling Dictionary [112] defines grids as "persistent environments that enable software applications to integrate instruments, displays, computational and information in widespread locations". But this definition is adopted by few researchers.

Since there is not a unique and precise definition for the grid concept, we present two attempts to define and check if a distributed system is a grid system. First, Foster [47] proposes a three point checklist to define grid as a system that:

1. coordinates resources that are not subject to centralized control...
2. ... using standard, open, general-purpose protocols and interfaces...
3. ... to deliver nontrivial qualities of service.

Second, Németh and Sunderam [103] presented a formal definition of what a grid system should provide. They focused on the semantics of the grid and argue that a grid is not just a modification of "conventional" distributed systems but fundamentally differs in semantics. They present some criteria comparing distributed environments and grids that we transcribe (Table B.1) and analyze. A grid can present heterogeneous resources including, for example, sensors and detectors and not only computational nodes. Individual sites may belong to different administrative domains. Thus, the user has access to the grid (the pool) but not to the individual sites and access may be restricted. The user has limited knowledge about each site due to administrative boundaries, and even due to the large number of resources. Resources in the grid typically belong to different administrative domains and also to several trust domains. Finally, while conventional distributed environments tend to be static, except due to faults and maintenance, grids are dynamic by definition.

Németh and Sunderam [102] also made an informal comparison of distributed systems and computational grids.

Table B.1: Comparison of conventional distributed environments and grids [103]

| Conventional distributed environments | Grids |
|---|---|
| a virtual pool of computational nodes | a virtual pool of resources |
| a user has access (credential) to all the nodes in the pool | a user has access to the pool but not to individual sites |
| access to a node means access to all resources on the node | access to a resource may be restricted |
| the user is aware of the capabilities and features of the nodes | the user has limited knowledge about each site |
| nodes belong to a single trust domain | resources span multiple trust domains |
| elements in the pool 10-100, more or less static | elements in the pool 1000-10000, dynamic |

Besides, Foster *et al.* [54] presents the specific problem that underlies the grid concept as coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations. The sharing is, necessarily, highly controlled, with resource providers and consumers defining clearly and carefully what is shared, who is allowed to share, and the conditions under which sharing occurs. A set of individuals and/or institutions that agree with such sharing rules form what they call a virtual organization.

Though there might be a strong relation among the entities building a virtual organization, a grid still consists of resources owned by different, typically independent organizations. This leads naturally to heterogeneity of resources and policies [115].

There are several reasons for programming applications on a computational grid. Laforenza [88] presents some examples: to exploit the inherent distributed nature of an application, to decrease the turnaround/response time of a huge application, to allow the execution of an application which is outside the capabilities of a single (sequential or parallel) architecture, and to exploit affinities between an application component and grid resources with specific functionalities.

In the next sections we give some of the classifications found in the literature for grid applications and systems.

## B.2 Grid Applications Classifications

Foster and Kesselman [48] identify five major application classes of grid environments:

- *Distributed supercomputing* applications use grid to aggregate substantial computational resources in order to tackle problems that cannot be solved on a single system.

They are very large problems, such as simulation of complex physical processes, which need lots of resources like CPU cycles and memory;

- *High-Throughput Computing* uses grid resources to schedule a large number of loosely coupled or independent tasks, with the goal of putting unused processor cycles to work. The Condor system [135] has been dealing with this kind of application, as for example, in molecular simulations of liquid crystal and biostatistical problems solved with inductive logic programming;

- *On-Demand Computing* applications use grid capabilities to meet short-term requirements for resources that cannot be cost-effectively or conveniently located locally. For example, one user doesn't need to buy a supercomputer to run an application once a week. Another example, the processing of data from meteorological satellites can use dynamically acquired supercomputer resources to run a cloud detection algorithm;

- *Data-Intensive Computing* applications, where the focus is on synthesizing new information from data that is maintained in geographically distributed repositories, digital libraries, and databases. This process is often computational and communication intensive, as expected in HEP experiments;

- *Collaborative Computing* applications are concerned primarily with enabling and enhancing human-to-human interactions. Many collaborative applications are concerned with enabling the shared use of computational resources such as data archives and simulations. For example, the CAVE5D [24] system supports remote, collaborative exploration of large geophysical data sets and the models that generated them.

Some examples of applications that are representative of these main classes of applications are the following:

**Monte Carlo Simulation** Monte Carlo experiments are sampling experiments, performed on a computer, usually done to determine the distribution of a statistic under some set of probabilistic assumptions [69]. The name Monte Carlo comes from the use of random numbers.

In HEP, there are several applications to this technique. The results can usually be obtained running several independent instances, with different parameters, which can be easily executed in parallel.

**Biostatistic Problems using Inductive Logic Programming (ILP)** Dutra *et al.* [39] reported experiments of ILP to solve biostatistic problems. Since, it is a machine learning problem, it had two main phases: experimentation and evaluation. During the experimentation phase, the user wants to run a learner, adjusting the learner parameters, using several datasets, and sometimes, repeating the learning process several times. During the evaluation phase, the user is interested in knowing how accurate a given model is, and which one of the experiments gave the most accurate result.

Typically, these experiments need to be run in parallel in both phases. Due to the highly independent nature of each experiment, all machine learning process can be trivially parallelized. However, all experiments of the first phase must be finished before proceeding to the second phase.

**Finite Constraint Satisfaction Problems (Finite CSP)** Finite CSP [5] usually describe NP-complete search problems. Algorithms exist, such as arc-consistency algorithms, that help to eliminate inconsistent values from the solution space. They can be used to reduce the size of the search space, allowing to find solutions for large CSP.

Still, there are problems whose instance size make it impossible to find a solution in a feasible time with sequential algorithms. Concurrency and parallelism can help to minimize this problem because a constraint network generated by a constraint program can be split among processes in order to speed up the arc-consistency procedure. The dependency between the constraints can be represented usually as a complex and highly connected graph.

Considering the examples presented and other presented in the literature we certainly have a wide variety of applications that can profit from grid infrastructure. These distributed applications can be expressed as a graph, and usually as a Directed Acyclic Graph (DAG) as we present in more details in Section C.1. Because we intend to partition the application graphs to map their tasks to resources, we propose the following **taxonomy** for distributed applications in grid [143, 145, 146]:

- **independent tasks**: the simplest kind of distributed application is the one usually called bag-of-tasks. It characterizes applications where all tasks are independents. Monte Carlo simulations, typically used in HEP experiments, are examples of independent tasks applications;

- **loosely-coupled tasks**: this kind of graph is characterized by few sharing points, i.e, an application divided in phases or pipeline. The ILP experiments mentioned are examples of phase loosely-coupled tasks;

- **tightly-coupled tasks**: highly complex graphs are not so often, but are more difficult to be partitioned. Constraint logic programming applications fall in this kind.

We present in Figure B.1 visual representations of application graphs for each of the presented categories.

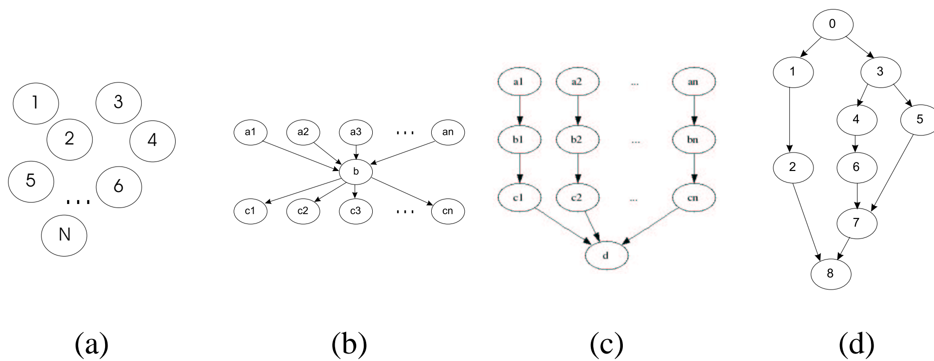



Figure B.1: Grid application kind: (a) independent tasks, (b) loosely-coupled tasks (phase), (c) loosely-coupled tasks (pipeline), and (d) tightly-coupled tasks

This classification proposal is important in the definition of the application partitioning approach of our model, as we will discuss later (Chapter F).

## B.3 Classifying Grid Systems

A current classification of grid computing systems is computational and data grid [107]. The *computational grid* focuses on reducing execution time of applications that require a great number of computer processing cycles or on execution of applications that can not be executed sequentially. The *data grid* provides the way to solve large scale data management problems. Data intensive applications such as HEP and Bio-informatics require both computational and data grid features.

Krauter *et al.* [84] presents a similar taxonomy for grid systems, which includes a third category, the service grid:

- The *computational grid* category denotes systems that have higher aggregate computational capacity available for single applications than the capacity of any constituent machine in the system. It is subdivided into *distributed supercomputing* (parallel application execution on multiple machines) and *high throughput* (stream of jobs). A computational grid can also be defined as a "large-scale high performance distributed computing environment that provide access to high-end computational resources" [112];

- The *data grid* is a terminology used for systems that provide an infrastructure for synthesizing new information from data repositories that are distributed in a wide area network;

- The *service grid* is the name used for systems that provide services that are not provided by any single local machine. This category is further divided as *on demand* (aggregate resources to enable new services), *collaborative* (connect users and applications via a virtual workspace), and *multimedia* (infrastructure for real-time multimedia applications).

Two publications emphasized the importance of middleware construction for a grid environment and presented the main guidelines and concepts for exploiting this kind of environment. One is the already referred "Grid Anatomy" [54], which starts using the term grid. The "Metasystems" paper [70] is another paper that, actually, was published first. In this paper, the authors state that the challenge to the computer science community is to provide a solid, integrated middleware foundation on which to build wide-area applications.

## B.4 Conclusion

We conclude this chapter by emphasizing that grid is the result of several years of research. There are several open research problems, and this work proposes solutions to some of them.

An important issue to allow the application execution in a grid environment is the application management. Application management, as we present in the next chapter, is concerned with questions such as how to partition and describe the application.

Once the application is partitioned, one of the main challenges is the mapping of the application to the large number of geographically distributed resources. Solutions to the problems related to these issues in the grid computing context will be discussed in the subsequent chapters and are the main theme of this thesis.

# Apêndice C

# Application Management: Description and Clustering

In this chapter, we discuss application management related issues. In the literature, usually, the user uses description languages such as VDL [55] and the DAGMan language [135] to describe the application. As previously discussed, an application can be represented as a graph, where each node represents a task and the edges represent the task precedence order. When the application management system receives the application graph, it needs to dispatch the tasks in the right order and control its execution. A partitioning algorithm is required to allow parallel execution. So, initially we present how an application can be represented as a graph and the meaning of application partitioning (C.1). In this chapter, partitioning is mostly used as a synonym to graph clustering. Note that we also assume the user or a preprocessor tool had identified the tasks before partitioning. Then, we discuss some examples of how to represent the application as a graph (Section C.2) and how to partition the application graph (Section C.3). Finally, we conclude with Section C.4.

## C.1 Representing an Application as a Graph

Usually, distributed applications can be expressed as a graph $G = (N, E)$, where $N$ is a set of weighted nodes representing tasks and $E$ is the set of weighted edges that represent dependencies between tasks [46, 75, 85]. This graph is usually undirected [85] and is sometimes a Directed Acyclic Graph (DAG) [135].

A distributed application will usually run in several processors. Before executing, the application must be partitioned.

Application partitioning can be defined as follows: tasks must be placed onto machines in such a way as to minimize communication and achieve best overall performance of the application [76]. Thus, application partitioning, in our work, is to divide the application task graph in subgraphs such as they can be allocated to different available processors efficiently. Efficiency can be measured considering the interprocessor communication and the execution time. If they are kept low, the efficiency is high. Thus, application partitioning can be done using graph partitioning techniques.

Graph Partitioning can be defined as the problem of dividing a set of nodes of a graph into disjoint subsets of approximately equal-weight such that the number of edges with end points in different subsets is minimized. Graph partitioning is an NP-complete problem [59], thus heuristics must be applied.

Partitioning is a more general term, but in this thesis we are interested in a specific class of grouping algorithms called *clustering*. As defined by Boeres and Rebelo [18], clustering algorithms refers to "the class of algorithms which initially consider each task as a cluster (allocated to a unique virtual processor) and then merge clusters (tasks) if the completion time of the parallel program can be reduced."

## C.2 Application Description Languages

Some grid environments offer a description language by which the user can describe the application. For example, VDL [55], in the context of the GriPhyN project[72], is being used to describe task dependencies that are converted to a graph represented in another language used by Condor DAGMan (Directed Acyclic Graph Manager) [135]. Condor DAGMan manages task dispatching based on this graph. Another example is JSDL (Job Submission Description Language) [20, 79, 80], which is a standard description language being proposed by the Global Grid Forum [61]. JSDL allows the mapping from one submission language to another depending on the submission manager, and thus can solve interoperability problems between resource managers. With JSDL, the user can express attributes of individual jobs, but he/she cannot express an application graph. Since it cannot express application DAGs, we will not present details about the JSDL syntax.

Three recent works on description languages are: (1) GXML, developed in the context of the GANGA framework [14, 15], (2) AGWL (Abstract Grid Workflow Language), developed in the context of the ASKALON Project [44], and (3) XPWSL (XML based Par-

allel Workflow Specification Language), proposed in the context of the JoiN platform [41]. Both are XML-based languages that can be used to express application characteristics and task dependencies. Finally, GEL (Grid Execution Language) [94] is a scripting language designed to express application for grid environments.

We use in this section the DAG example of Figure C.1 to illustrate how DAGs are expressed in these systems. This simple example, sometimes referred as diamond, is frequently used to illustrate languages functionalities. First, it is simple and small. Second, it presents one-to-many and many-to-one dependencies. Supporting both kinds of dependencies allows to support any phase task application.

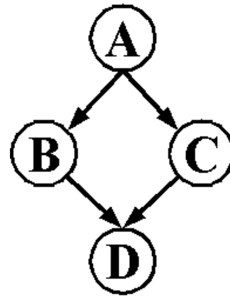



Figure C.1: Diamond DAG example

Note that the DAG representation of an application is commonly used by parallel programmers. But this is quite similar to workflow terminology and technologies adopted in information technology area. Nowadays, there is an increasing interest in exploiting and adapting workflow ideas and techniques to grid environments. There are many efforts towards the development of workflow management systems for grid computing, some of the main works are described in Yu and Buyya's survey [159]. Workflow is usually related to automation of a business process. In the grid applications context, it normally represents the computation sequence that needs to be performed to analyze scientific datasets. There are several works applying workflow techniques to grid problems, and some of the languages described in the following sections are closely related to workflow research for grid environments.

### C.2.1 DAGMan

The Directed Acyclic Graph Manager (DAGMan) [33, 132, 135] manages dependencies between tasks at a higher level invoking the Condor Scheduler to execute the tasks. Condor, as we will present in Subsection D.4.3, finds resources for the execution of tasks,

but it does not schedule tasks based on dependencies. DAGMan submits jobs to Condor in an order represented by a DAG and processes the results. Presently, DAGMan is a stand-alone command line application.

**DAG Representation** DAGMan receives a file defined prior to submission as input, which describes the DAG. Each node (task) of this DAG has a Condor submit description file associated to be used by Condor. As DAGMan submits jobs to Condor, it uses a single Condor log file to enforce the ordering required for the DAG. DAGMan is responsible for submitting, recovery, and reporting for the set of programs submitted to Condor.

The input file used by DAGMan specifies four items: (1) a list of the tasks in the DAG. This aims to name each program and specify each program's Condor submit description file; (2) processing that takes place before submission of any program in the DAG to Condor or after Condor has completed execution of any program in the DAG; (3) description of the dependencies in the DAG; and (4) number of times to retry if a node within the DAG fails. The items 2 and 4 are optional.

Figure C.2 presents the DAGMan input file that describes the DAG example of Figure C.1. The first four lines describe the tasks. Each task is described by a single line called a *Job Entry*: *job* is the keyword to indicate it is a job entry, the second element is the name of the task and the third is the name of the Condor submit file. Thus, a job entry maps a job name to a Condor submit description file.

A job entry can also have the keyword *DONE* at the end, which identifies a job as being already completed. This is useful in situations where the user wishes to verify results, but does not require that all jobs within the dependency graph to be executed. The *DONE* feature is also utilized when an error occurs causing the DAG to not be completed. DAGMan generates a Rescue DAG, a DAGMan input file that can be used to restart and complete a DAG without re-executing completed programs [33].

The last two lines in Figure C.2 describe the dependencies within the DAG. The *PARENT* keyword is followed by one or more job names. The *CHILD* keyword is followed by one or more job names. Each child job depends on every parent job on the line. A parent node must be completed successfully before any child node may be started. A child node is started once all its parents have successfully completed.

Figure C.3 presents four Condor job description files to describe the four tasks of the

```
#
# first_example.dag
#
Job  A  A.condor
Job  B  B.condor
Job  C  C.condor
Job  D  D.condor
PARENT A CHILD B C
PARENT B C CHILD D
```

Figure C.2: DAGMan input file

DAG example. In this example, each node in a DAG is a unique executable, each with a unique Condor submit description file.

```
#
# task A
#
executable = A.exe
input    = test.data
output   = A.out
log      = dag.log
Queue
```

```
#
# task B
#
executable = B.exe
input    = A.out
output   = B.out
log      = dag.log
Queue
```

```
#
# task C
#
executable = C.exe
input    = A.out
output   = C.out
log      = dag.log
Queue
```

```
#
# task D
#
executable = D.exe
input    = B.out C.out
output   = final.out
log      = dag.log
Queue
```

Figure C.3: Condor submit description files for DAG example

Figure C.4 is an alternative code to implement the DAG example. This example uses the same Condor submit description file for all the jobs in the DAG. Each node within the DAG runs the same program (*/path/dag.exe*).

The *$(cluster)* macro is used to produce unique file names for each program's output. *$(cluster)* is a macro, which supplies the number of the job cluster. A cluster is a set of jobs specified in the Condor submit description file through a queue command. In DAGMan, each job is submitted separately, into its own cluster, so this provides unique names for the output files. The number of the cluster is a sequential number associated with the number of submissions the user had done. For example, in the tenth Condor submission the cluster

number will be called “10”, and the first task of this cluster “1” or more precisely “10.1”. In our example, if task A receives a cluster number “10”, its output file will be called “dag.out.10”, and the output file for task B will be called “dag.out.11” due to the order of the task definitions in the DAGMan input file.

We could also use the *$(cluster)* macro to run a different program for each task. For example, the first line could be replaced by the following statement: *executable = /path/dag_${cluster}.exe*

This example is easier to code but less flexible. A problem is how to specify the inputs. For example, task B should receive as input the output of task A, whose name is dependent on the variable *cluster*. But, the Condor DAGMan description language does not support arithmetics to allow a code like *input=dag.out.{$(cluster)-1}*.

```
#
# second_example.dag
#
Job  A  dag_job.condor
Job  B  dag_job.condor
Job  C  dag_job.condor
Job  D  dag_job.condor
PARENT A CHILD B C
PARENT B C CHILD D
```

```
#
# dag_job.condor
#
executable = /path/dag.exe
output     = dag.out.$(cluster)
error      = dag.err.$(cluster)
log        = dag_condor.log
queue
```

Figure C.4: Another solution for DAG example: DAGMan input file and Condor submit description file

An alternative to allow the input description in this example is to use a *VARS* entry in the DAGMan input file. Each task would have a *VARS* entry to define an input file name (e.g. *VARS A inputfile=“test.data”* ). The Condor submit file would have an extra line: *input = ${inputfile}*.

Besides, two limitations exist in the definition of the Condor submit description file to be used by DAGMan. First, each Condor submit description file must submit only one job (only one *queue* statement with value one or no parameter). The second limitation is that the submit description file for all jobs within the DAG must specify the same log. DAGMan enforces the dependencies within a DAG, as mentioned, using the events recorded in the log file produced by the job submission to Condor.

**Management** The DAGMan literature does not present any algorithm for doing partitioning. A job is submitted using Condor as soon as it is detected that it does not have

parent jobs waiting to be submitted.

Before executing the DAG, DAGMan checks the graph for cycles. If it is acyclic it proceeds. The next step is to detect jobs that can be submitted.

The submission of a child node will not take place until the parent node has successfully completed. There is no ordering of siblings imposed by the DAG, and therefore DAGMan does not impose an ordering when submitting the jobs to Condor. For instance, in the previous example, jobs B and C will be submitted to Condor in parallel.

### C.2.2 Chimera

The Chimera Virtual Data System (VDS) [8, 35, 56, 55] has been developed in the context of the GriPhyN project [72]. The GriPhyN (Grid Physics Network) project has a team of information technology researchers and experimental physicists, which aims to provide infrastructure to enable Petabyte-scale data intensive computation.

Chimera handles the information of how data is generated by the computation. Chimera provides a catalog that can be used by application environments to describe a set of application programs ("transformations"), and then track all the data files produced by executing those applications ("derivations").

**DAG Representation** The Chimera input consists of transformations and derivations described in the Virtual Data Language (VDL). VDL comprises data definition and query statements. We will concentrate our description on data definition issues. In Chimera's terminology, a transformation is an executable program and a derivation represents an execution of a transformation (it is analogous, respectively, to program and process in the operating system terminology).

Figure C.5 presents a VDL example that expresses the diamond DAG example. The first two statements define transformations (*TR*). The first *TR* statement defines a transformation named *calculate* that reads one input file (*input a*) and produces one output file (*output b* ). The *app* statement specifies the executable name. The keyword *vanilla* indicates one of the execution modes of Condor. The *arg* statements usually describe how the command line arguments are constructed. But, in this example, the special argument *stdin* and *stdout* are used to specify a filename into which, respectively, the standard input and output of the application would be redirected.

```
TR calculate{ output b, input a} {
  app vanilla = "generator.exe";
  arg stdin = ${output:a};
  arg stdout = ${output:b};
}
TR analyze{ input a[], output c} {
  app vanilla = "analyze.exe";
  arg files  = ${:a};
  arg stdout = ${output:a2};
}
DV calculate { b=@{output:f.a},
               a=@{input:test.data} };
DV calculate { b=@{output:f.b},
               a=@{input:f.a} };
DV calculate { b=@{output:f.c},
               a=@{input:f.a} };
DV analyze{ a=[ @{input:f.b},
                @{input:f.c} ],
            c=@{output:f.d} };
```

Figure C.5: DAG example expressed in Chimera VDL

The *DV* statements define derivations. The string after a *DV* keyword names the transformation to be invoked. Actual parameters in the derivation and formal parameters in the transformation are associated. Note that these four *DV* statements define, respectively, tasks A, B, C, and D of the DAG example shown in Figure C.1. However, there is no explicit task dependency in this code. The DAG can be constructed using the data dependency chain expressed in the derivation statements.

**Management** Chimera uses a description of how to produce a given logical file. The description is stored as an abstract program execution graph. This abstract graph is turned into an executable DAG for DAGMan by the Pegasus planner which is included in the Chimera VDS distribution. Chimera does not control the DAG execution: DAGMan is used to perform this control.

Pegasus (Planning for Execution in Grids) [35, 109] is a system that can map and execute workflows. Pegasus receives an abstract workflow description from Chimera, produces a concrete workflow, and submits it to Condor's DAGMan for execution. The abstract workflow describes the transformations and data in terms of their logical names. The concrete workflow, which specifies the location of the data and the execution platforms, is optimized by Pegasus: if data described within an abstract workflow already exist, Pegasus reuses them and thus reduces the complexity of the concrete workflow.

### C.2.3 GXML

The GANGA (Grid Application iNformation Gathering and Accessing) framework [14, 15] addresses the problem of interoperability between resource managers. This framework provides some services related to application information such as discovery. The application information definition is done by the user using the GXML. GXML allows to describe jobs in a similar way to JSDL [79].

**DAG Representation** GXML is an XML language that allows to describe application information including jobs, DAGs and loops. Not much detail is presented in the literature. The GXML code for our DAG example is presented in [14, 15] and reproduced in Figure C.6 in order to show how dependencies between tasks are expressed. However, it does not present how the user can indicate details about the tasks such as executable, input, and output files. Note that dependencies between tasks are explicitly represented through the *flow:Depend* tag.

```
<flow:Workflow flow:start="A">
  <jsdl:Job jsdl:id="A">
    ...
  </jsdl:Job>
  <jsdl:Job jsdl:id="B">
   <flow:Depend flow:success="A"/>
    ...
  </jsdl:Job>
  <jsdl:Job jsdl:id="C">
   <flow:Depend flow:success="A"/>
    ...
  </jsdl:Job>
  <jsdl:Job jsdl:id="D">
   <flow:Depend flow:success="B & C"/>
    ...
  </jsdl:Job>
</flow:Workflow>
```

Figure C.6: DAG example expressed in GXML (skeleton) [14, 15]

It allows expressing loops using the *flow:LoopCount* tag.

**Management** The GANGA framework was developed using Java. It provides classes to store application information parsed from the GXML input file. It also has converters to Ganesh, an unpublished work on grid management, and to Condor's DAGMan [33]. Once

a GXML file is converted to a DAGMan file, the application execution management can be all done through DAGMan and the task execution through Condor.

### C.2.4 AGWL

The ASKALON project [43] proposes AGWL (Abstract Grid Workflow Language) [44], also an XML-based language. AGWL provides advanced workflow constructs to facilitate the parallel execution of workflows in a grid environment. It allows to express control flow information such as loops and branches and sub-workflows that can be invoked by other workflows.

**DAG Representation** An AGWL specification contains activities to describe a DAG or a more complex workflow. An activity can be either an *atomic activity*, which refers to a single computational task, or a *compound activity*, which encloses some atomic activities or other compound activities that are connected by control and data flows [44].

Figure C.7 presents the skeleton code of Diamond DAG example. The activity element is used to describe a task, including details such as input and output files.

**Management** AGWL is converted to CGWL (Concrete Grid Workflow Language) that combines the contents of the AGWL specification with information about the grid infrastructure. The CGWL specification is executed through an enactment engine provided by ASKALON [43]. ASKALON is a grid service oriented middleware. The provided services are built on top of the Globus toolkit.

### C.2.5 XPWSL

XPWSL (an XML-based Parallel Workflow Specification Language) [41, 42] is a parallel application specification language. XPWSL implements a specification model that allows to specify DAGs and more complex structures.

**DAG Representation** XPWSL supports the following control structures: (1) *sequential block*; (2) *iterative block (or loop) for a fixed number of iterations*; (3) *conditional iterative block*; (4) *switch/case block*. An application is specified in XPWSL using a file with three sections: header, assignment, and data link. The *header* section identifies the application. It is useful to the user. The *assignment* section defines the files with the executable code

```
<agwl-workflow>
   <activity name="A" type="teste:A">
     <dataIn name="test.dat" > </dataIn>
     ...
     <dataOut name="A.out">  </dataOut>
   </activity>
   <subWorkflow name="tasksBandC">
     <body>
       <activity name="B" type="teste:B">
         <dataIn name="A.out" > </dataIn>
         ...
         <dataOut name="B.out">  </dataOut>
       </activity>
       <activity name="C" type="teste:C">
         <dataIn name="A.out" > </dataIn>
         ...
         <dataOut name="C.out">  </dataOut>
       </activity>
     </body>
   </subWorkflow>
   <activity name="D" type="teste:D">
     <dataIn name="B.out" > </dataIn>
     <dataIn name="C.out" > </dataIn>
     ...
     <dataOut name="D.out">  </dataOut>
   </activity>
</agwl-workflow>
```

Figure C.7: DAG example expressed in AGWL (skeleton)

that will be used for each batch of tasks. The *data link* section defines the relationship between the execution blocks of the application. The execution order respects the order of the specification.

Figure C.8 presents the code for the diamond DAG example (Figure C.1). Note that a task is identified using the *id* attribute. The id must be the letter T followed by a unique numerical identifier. Thus, task ”A” was mapped to “T0”, “B” to “T1”, and so on. Note that the documentation does not explain how to provide more than one input file to a task, as needed by task “D”. We assume a syntax similar to AGWL.

**Management** The XPWSL is implemented in the JoiN platform. JoiN is a Java system that allows execution of massively parallel applications in a non dedicated system. In JoiN, the application is described using PASL (Parallel Application Specification Language), which is a text file based on tags and sections. Since, PASL does not support all control structures provided by XPWSL, JoiN was extended to support XPWSL [42].

```
<header>
  <name>DAG example</name>
  <description>same example previously used</description>
</header>

<assignment>
  <task id="T0" delay="delay_A">
    <path>/path</path>
    <code>A.exe</code>
  </task>
  <task id="T1" delay="delay_B">
    <path>/path</path>
    <code>B.exe</code>
  </task>
  <task id="T2" delay="delay_C">
    <path>/path</path>
    <code>C.exe</code>
  </task>
  <task id="T3" delay="delay_D">
    <path>/path</path>
    <code>D.exe</code>
  </task>
</assignment>

<datalink>
  <block type="sequential">
    <task_id>T0<task_id>
    <multi>1</multi>
    <input>test.dat</input>
  <block>
  <block type="sequential">
    <task_id>T1<task_id>
    <multi>1</multi>
    <input>A.out</input>
    <task_id>T2<task_id>
    <multi>1</multi>
    <input>A.out</input>
  <block>
  <block type="sequential">
    <task_id>T3<task_id>
    <multi>1</multi>
    <input>B.dat</input>
    <input>C.dat</input>
  <block>
</datalink>
```

Figure C.8: DAG example expressed in XPWSL

### C.2.6 GEL

GEL (Grid Execution Language) [94] is a script based language that allows a succinct representation of parallel programs. It was developed in the Bioinformatics Institute at Singapore. They aimed to be grid middleware independent. All middleware dependencies are inserted into specific interpreter instances.

**DAG Representation** GEL allows the user to express acyclic and cyclic graphs, such that dependencies between tasks are implicit in the syntactic structure of the script. GEL programs consist of a combination of tasks (jobs) using some parallel or sequential constructs. The sequential constructs are the following: sequential composition, loop iteration, and conditional if. Parallel execution is mainly parallel composition (tasks are independent and can be executed in parallel) and parallel iteration (pfor or pforeach).

```
taskA = {exec="A.exe"; dir="/path"; args="test.dat"}
taskB = {exec="B.exe"; dir="/path"; args="A.out"}
taskC = {exec="C.exe"; dir="/path"; args="A.out"}
taskD = {exec="D.exe"; dir="/path"; args="B.out","C.out"}
taskA; taskB | taskC; taskD
```

Figure C.9: DAG example expressed in GEL

**Management** The GEL implementation has two components: a DAG *builder* and a DAG *executor*. The current GEL implementation has five DAG executors: (1) SMP machines; (2) Sun GridEngine (SGE), (3) LSF, and (4) PBS for cluster environments; and (5) Globus for grid environments.

## C.3 Application Partitioning

One of the characteristics of grid applications is that they usually spread a very large number of tasks. Very often, these tasks present dependencies that enforce a precedence order. In a grid, as in any distributed environment, one important issue to increase performance is how to distribute the tasks among resources. Although research in scheduling is very active, efficient partitioning of applications to run in a grid environment is a hot research topic. A distributed application can be partitioned by grouping related tasks aiming to decrease communication costs. The partitioning should be such that dependencies between

tasks and data locality are maintained. Few partitioning works deal specifically with the high resource heterogeneity and dynamic nature of grid environments.

We will now present some related works on graph partitioning techniques applied to the application partitioning problem.

The conventional graph partitioning approach divides the nodes into $p$ equally weighted sets while minimizing interprocessor communication (crossing edges between partitions). In other words, traditional graph partitioning algorithms compute a $k$-way partitioning of a graph such that the number of edges that are cut by the partitioning is minimized and each partitioning has an equal number of nodes.

Hogstedt *et al.* [76] considers a graph $G = (N, E, M)$, with a set of nodes $N$, a set of edges $E$, and a set of distinguished nodes $M \subset N$, denoting machine nodes. Each edge has a weight, denoted $w_e$ for an edge $e \in E$. Each node $n \in N$ can be assigned to any of the machine nodes $m \in M$. The machine nodes $M$ cannot be assigned to each other. Any particular assignment of the nodes of $N$ to machines $M$ is called a cut, in which the weight of the cut is the sum of the weights of edges between nodes residing on two different machines. The minimal cut set of a graph minimizes the interprocessor communication.

Hogstedt *et al.* present five heuristics to derive a new graph, which is then partitioned aiming to obtain the minimal cut (*min-cut*) set for the original graph. The five heuristics are the following: (1) *dominant edge*: there is a min-cut not containing the heaviest edge $e$, so we can contract [1] $e$ to obtain a new graph G'; (2) *independent net*: if the communication graph can be broken into two or more independent nets, then the min-cut of the graph can be obtained by combining the min-cut of each net; (3) *machine cut*: let a machine cut $M_i$ be the set of all edges between a machine $m_i$ and non machine nodes N. Let $W_i$ be the sum of the weight of all edges in the machine cut $M_i$. Let the $W_i$'s be sorted so that $W_1 \geq W_2 \geq W_3 \geq \ldots$. Then any edge which has weight greater than $W_2$ cannot be present in the min-cut. (4) *zeroing*: if a node $n$ has edges to each of the $m$ machines with weights $w_1 \leq w_2 \leq \ldots \leq w_m$, we can reduce the weights of each of the $m$ edges from $n$ to the machines by $w_1$ without changing min-cut assignment; (5) *articulation point*: nodes

[1]The contraction of $(x, y) \in N$ corresponds to replacing the vertex $x$ and $y$ by a new vertex $z$, and for each vertex $v$ not in $(x, y)$ replacing any edge $(x, v)$ or $(y, v)$ by the edge $(z, v)$. The rest of the graph remains unchanged. If the contraction results in multiple edges from node $z$ to another node $v$ they are combined and their weights added. [76]

that would be disconnected from all machines if node $n$ was deleted cannot be in min-cut, thus they can be contracted.

Hendrickson *et al.* [75] also consider that minimizing the number of graph edge cut minimizes the volume of communication. But they argue that the partitioning and mapping should be generated together to also reduce message congestion. Thus, they adapt an idea of the circuit placement community called terminal propagation to the problem of partitioning data structures among processors of a parallel computer. The basic idea of terminal propagation is to associate with each vertex in the subgraph being partitioned a value which reflects its net preference to be in a given quadrant board. In parallel computing, the quadrants represent processors or sets of processors. Their major contribution is a framework for coupling recursive partitioning schemes to the mapping problem.

Karypis and Kumar [83] extend the standard partitioning problem by incorporating an arbitrary number of balancing constraints. A vector of weights is assigned to each vertex, and the goal is to produce a $k$-partitioning such that the partitioning satisfies a balancing constraint associated with each weight, while attempting to minimize the edge-cut.

Hendrickson and Kolda [74] argue that the standard graph partitioning approach minimizes the wrong metrics and lacks expressibility. One of the metrics is minimizing edge cuts. They present the following flaws of the edge cut metric: (1) edges cuts are not proportional to the total communication volume; (2) it tries to (approximately) minimize the total volume but not the total number of messages; (3) it does not minimize the maximum volume and/or number of messages handled by any single processor; (4) it does not consider distance between processors (number of switches the message passes through). To avoid message contention and improve the overall throughput of the message traffic, it is preferable to have communication restricted to processors which are near each other.

Despite the limitations of edge cut metric, the authors argue that the standard approach is appropriate to applications whose graph has locality and few neighbors.

Hendrickson and Kolda [74] also argue that the undirected graph model can only express symmetric data dependencies. An application can have the union of multiple phases, and this cannot generally be described via an undirected graph.

Kumar *et al.* [85] propose the MiniMax scheme. MiniMax is a multilevel graph partitioning scheme developed for distributed heterogeneous systems such as the grid, and differs from existing partitioners in that it takes into consideration heterogeneity in both the system and workload graphs. They consider two weighted undirected graphs: a work-

load graph (to model the problem domain) and a system graph (to model the heterogeneous system).

Another class of related partitioning algorithms are the static scheduling of task precedence graph. Kwok [87] presents a survey of the main static scheduling algorithms, where he proposes a taxonomy reproduced in Figure C.10. In this thesis, we are concerned with *Arbitrary Graph Structure*, since it allows to execute a larger number of applications. We also want to allow the user or the program to express *Arbitrary Computational Costs*. Task precedence graph *With Communication* is a more general case. It is important to consider communication costs since in a heterogeneous environment, message passing or file transfer through a slow link can cause application slowdown. We also are not interested in algorithms with *Duplication* since this would restrict to partition the application where a task can be executed more than once producing always the same results (idempotent tasks). Finally, for an initial partitioning of the tasks, we are interested in obtaining a clustering according to data locality without considering where tasks will be executing, thus, we are interested in algorithms with *Unlimited Number of Processors*. Sarkar *apud* Yang and Gerasoulis [158] have proposed a two step method for scheduling with communication: (1) schedule an unbounded number of completely connected processors (cluster of tasks); and (2) if the number of clusters is larger than the number of available processors, then merge the clusters until it gets the number of real processors, considering the network topology (merging step). Thus, we could use any of the algorithms of this class to implement the Sarkar's step one.

For this class of algorithms, there are some well known works. A well know example is the List Scheduling. Its basic mechanism is the following: "initially, priorities are assigned to all tasks; and then, until all tasks have been scheduled, a list of tasks ready for execution is formed from which the task with the highest priority is chosen to be scheduled on the processor upon which it can execute earliest" [18]. List scheduling algorithms only produce good results for coarse-grained applications.

An example of List Scheduling is the DSC (Dominant Sequence Clustering) algorithm proposed by Yang and Gerasoulis [158]. Yang and Gerasoulis defines that a *scheduled DAG* is a DAG with a given clustering (the task to processor assignment) and task execution ordering information. The critical path of the scheduled DAG is called the *dominant sequence* (DS). A *critical path* is the longest path in a graph, including both nonzero communication edge cost and task weights in that path. Reduction of the DS and consequently

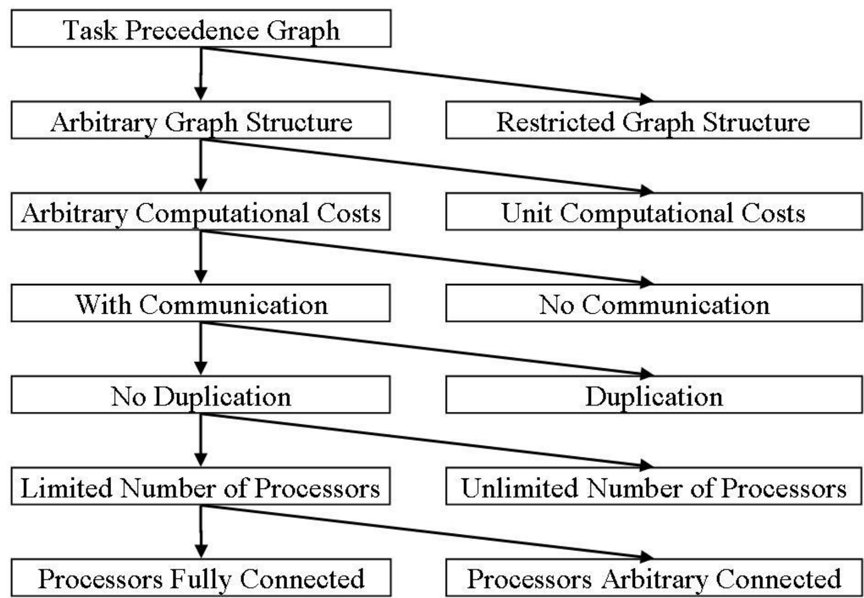



Figure C.10: Kwok's partial taxonomy of the multiprocessor scheduling problem [87]

of the parallel time is the key point of this algorithm. The basic mechanism is the following: initially each task belongs to a unit cluster; the DAG is analyzed bottom-up (or top-down); each node in the DAG is analyzed according to a priority list constructed dynamically; when a node under analysis can decrease the DS if merged with a predecessor (or successor) cluster, the merging is done and the communication between tasks in the same cluster is considered zero; when all nodes are visited, the algorithm finishes.

Finally, an interesting work is the ARMaDA framework proposed by Chandra and Parashar [25]. It does not perform DAG partitioning. ARMaDA is defined by its authors as an adaptive application-sensitive partitioning framework for structured adaptive mesh refinement (SAMR) applications. The goal is to adaptively manage dynamic applications on structured mesh based on the runtime state. The authors argue that no single partitioning scheme performs the best for all types of applications and systems:

> "Even for a single application, the most suitable partitioning technique and associated partitioning parameters depend on input parameters and the application's runtime state. This necessitates adaptive partitioning and run-

time management of these dynamic applications using an application-centric characterization of domain-based partitioners." [25]

We believe that this statement applies to DAG partitioning problems as well.

## C.4 Conclusion

In this chapter, some of the most well known application description languages for grid environments were analyzed. All the analyzed languages have interesting features, but none has all characteristics we think that would be necessary. So, we proposed a new language in this thesis called GRID-ADL (Grid Application Description Language). Table C.1 summarizes the main characteristics of description languages presented previously in this chapter. In this table, we also included our proposed language, which is presented in detail in Section F.3.1. GEL is the one that is closer to our needs and could have been adapted to our work. However, it was published after GRID-ADL was proposed and implemented. Actually, most of the presented languages published their work after we had proposed GRID-ADL.

Most languages are associated to a specific execution environment. For GRID-ADL, we used the information concerning the current prototype implementation, as will be presented in Chapter G. Concerning the GRAND model, GRID-ADL tasks can be mapped to any resource management system (RMS). Besides, most of the languages does not present automatic DAG inference. Systems like Chimera and DAGMan allow the user to specify, and, in the case of DAGMan, control, dependencies among tasks. In these systems, the user needs to specify dependencies among tasks using a description language. In DAGMan, the user explicitly specifies the task dependence graph, where the nodes are tasks and edges represent dependencies that can be through data files or simply control dependencies. In Chimera, the user only needs to specify the data files manipulation. GRID-ADL, which is presented in detail in Section F.3.1, is also included in this table. GRID-ADL automatically detects the DAG structure through data dependencies, while GEL detects through control dependencies expressed by the declaration order in the description file.

Although almost all works deal with task dependency, they do not handle data locality. Data locality should be preserved in order to avoid unnecessary data transfer, and consequently, reduce network traffic. As the available systems do not deal with this issue, transient files can be unnecessarily circulating in the network.

Table C.1: Grid description languages: comparison

| Language | Reference | Grid Middleware | RMS | Language Type | Workflow structure | DAG | DAG inference |
|---|---|---|---|---|---|---|---|
| DAGMan | 2002 [132, 135, 136] | DAGMan | Condor | plain | no | yes | manual |
| VDL | 2002 [8, 35, 55, 56] | DAGMan | Condor | plain | yes | yes | automatic |
| GXML | 2005 [14, 15] | Ganesh or DAGMan | Ganesh or Condor | XML | yes | yes | manual |
| AGWL | 2005 [44] | ASKALON | Globus | XML | yes | yes | manual |
| XPWSL | 2005 [41, 42] | JoiN | JoiN | XML | yes | yes | manual |
| GEL | 2005 [94] | GEL | SMP, SGE, LSF, PBS, or Globus | script | no | yes | automatic |
| GRID-ADL | 2004 [114, 143, 146] | AppMan | AppMan and/or PBS | script | no | yes | automatic |

None of these works mentions any partitioning algorithm. Tasks are submitted one by one when their dependencies are satisfied. We consider that an application partitioning should be applied to get a more efficient execution. In this chapter, we presented several alternatives that could be applied to solve this problem.

In the next chapter we discuss resource management issues that is another topic fundamental for computational grid environments. We consider that each domain in the grid has a local RMS which can be accessed remotely to allow application scheduling decisions. Using an RMS support, an application can be executed in the grid through a meta scheduler.

# Apêndice D

# Application Management: Resource and Task Allocation

A central part of a distributed system is the resource management system (RMS). There are several RMSs to deal with cluster and local network resources. Recently, some RMSs to deal with grid computing environment are being made available. A grid resource management system manages the pool of resources that are available and that can include resources from different providers. Managed resources are mainly processors, network bandwidth, and disk storage. Applications are executed using RMS information and control.

In this chapter we discuss the RMS subject. We present some initial background (Sections D.1 and D.2), and then some of the most important scheduling systems (Sections D.3 and D.4). We conclude with an analysis of the presented concepts and systems (Section D.5).

## D.1 Resource Management Issues

Scheduling is one of the most important and interesting research topics in the distributed computing area. Casavant and Kuhl [23] consider scheduling as a resource management problem. This management is basically a mechanism or policy used to efficiently and effectively manage the access to and use of a resource by its various consumers. On the other hand, according to the GGF's Grid Scheduling Dictionary [112], scheduling is the "process of ordering tasks on computer resources and ordering communication between tasks". Thus, both applications and system components must be scheduled.

Two key concepts related to scheduling and resource management are task and job. They can be defined as follows according to Roehrig *et al.* [112]:

- **Task** is a specific piece of work required to be done as part of a job or application;
- **Job** is an application performed on high performance computer resources. A job may be composed of steps/sections of individual schedulable entities.

Thus, to the rest of this text, we will be referring to applications as a set of tasks to be executed.

The Resource Management System (RMS) is a central part of a distributed system. Note that resource management is traditionally an operating system problem. Resource management in a distributed system differs from that in a centralized system in a fundamental way [131]: centralized systems always have tables that give complete and up-to-date status information about all the resources being managed; distributed do not. The problem of managing resources without having accurate global state information is very difficult. Managing large-scale collections of resources poses new challenges. The grid environment introduces five resource management problems [32]: (1) site autonomy: resources are typically owned and operated by different organizations, in different administrative domains; (2) heterogeneous substrate: sites may use different local RMS or the same systems with different configurations; (3) policy extensibility: the RMS must support development of new application domain-specific management mechanisms, without requiring changes to code installed at participating sites; (4) co-allocation: some applications have resource requirements that can be satisfied only by using resources simultaneously at several sites; and (5) online control: the RMS must support negotiation to adapt application requirements to resource availability.

An RMS for grids can be implemented in different ways. But, it may not be possible for the same scheduler to optimize application and system performance. Thus, Krauter *et al.* [84] states that a grid RMS is most likely to be an interconnection of RMSs that are cooperating with one another within an accepted framework. To make the interconnections possible, some interfaces and components are defined in an abstract model.

Krauter *et al.* [84] also present a taxonomy that classifies RMSs by characterizing different attributes. We used some of these criteria and others to compare the systems we present in the next sections.

## D.2 Scheduling Taxonomy

In this section we consider resource management concerning only job allocation to processors. Several solutions have been proposed to this problem, which is usually called the scheduling problem. Some authors studied solutions to this problem and presented classifications. We present in this section some of the most well known classifications or taxonomies.

Casavant and Kuhl [23] is one of the most referred scheduling taxonomies. They propose a hierarchical taxonomy that is partially presented in Figure D.1, since we selected some of the classifications that are relevant to this study. Besides, Casavant and Kuhl also propose a flat classification from which we also selected just some issues.

At the highest level, they distinguish between *local* and *global*. Local Scheduling is related with the assignment of a process to the time-slices of a single processor. Global scheduling is the problem of deciding where (in which processor) to execute a process.

In the next level, beneath global scheduling, the time at which the scheduling or assignment decisions are made define *static* and *dynamic* scheduling.

Static scheduling can be *optimal* or *suboptimal*. In case that all information regarding the state of the system as well as the resource needs of a process are known, an optimal assignment can be made based on some criterion function (e.g. minimizing total process completion time). When computing an optimal assignment is computationally infeasible, suboptimal solutions may be tried. Within the realm of suboptimal solutions, there are two general categories: *approximate* and *heuristic*.

The next issue, beneath dynamic scheduling, involves whether the responsibility for the task of global dynamic scheduling should physically reside in a single processor (*physically non-distributed*) or whether the work involved in making decisions should be *physically distributed* among the processors. In this text, we use for simplicity non-distributed (or centralized) and distributed scheduling instead of physically non-distributed and physically distributed. Within the range of distributed dynamic global scheduling, there are mechanisms which involve cooperation between the distributed components (*cooperative*) and those in which the individual processors make decisions independent of the actions of the other processors (*non-cooperative*).

In addition to the hierarchical portion of the taxonomy, Casavant and Kuhl [23] present other distinguishing characteristics which scheduling systems may have:

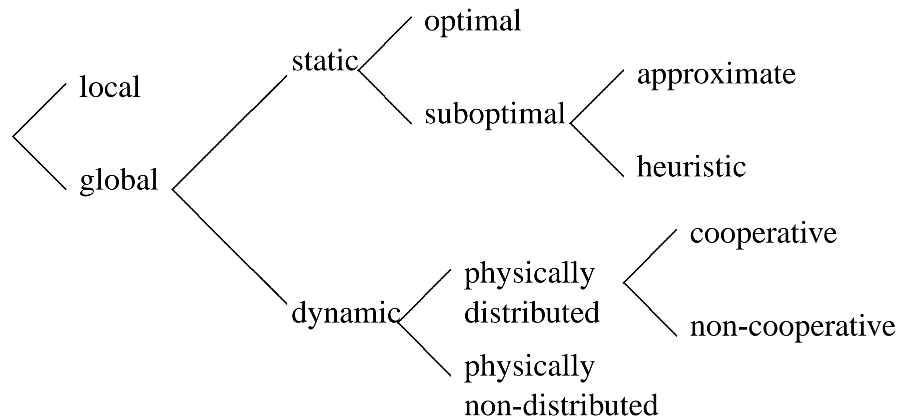



Figure D.1: Part of the Casavant and Kuhl's taxonomy [23]

- *Adaptive versus Nonadaptive*: an adaptive solution to the scheduling problem is one in which the algorithms and parameters used to implement the scheduling policy change dynamically according to the previous and current behavior of the system in response to previous decisions made by the scheduling system. A nonadaptive scheduler does not necessarily modify its basic control mechanisms on the basis of the history of system activity;

- *One-Time Assignment versus Dynamic Reassignment*: One-time assignment can technically correspond to a dynamic approach, however it is static in the sense that once a decision is made to place and execute a job, no further decisions are made concerning the job. In contrast, solutions in the dynamic reassignment class try to improve on earlier decisions. This category represents the set of systems that (1) do not trust their users to provide accurate descriptive information, and (2) use dynamically created information to adapt to changing demands of user processes. This adaptation is related to migration of processes.

The scheduling algorithm has four components [122]: (1) transfer policy: *when* a node can take part of a task transfer; (2) selection policy: *which task* must be transferred; (3) location policy: *which node* to transfer to; and (4) information policy: *when to collect* system state information. Considering the location policy component, a scheduling algorithm can be classified as *receiver-initiated* (started by the task importer), *sender-initiated* (started by the task exporter), or *symmetrically initiated*. Their performance are closely re-

lated to system workloads. Sender-initiated gives better performance if workers are often idle and receiver-initiated performs better when the load is high.

Another important classification is *dedicated* and *opportunistic* scheduling [111, 152]. Opportunistic scheduling involves placing jobs on non-dedicated resources under the assumption that the resources might not be available for the entire duration of the jobs. Using opportunistic scheduling, resources are used as soon as they become available and applications are migrated when resources are no longer available. Dedicated scheduling algorithms assume the constant availability of resources to compute fixed schedules. Most software for controlling clusters relies on dedicated scheduling algorithms. The applications that most benefit from opportunistic scheduling are those that require high throughput rather than high performance. Traditional high-performance applications measure their performance in instantaneous metrics like floating point operations per second (FLOPS), while high throughput applications usually use application-specific metrics, and the performance might be measured in TIPYs (trillions of instructions per year).

In 1998, Berman [11] classified the scheduling mechanisms for grid in three groups. Nowadays, a current concept is the Metascheduler as presented by the GGF [112]. Thus, we consider that the scheduling mechanisms can be classified in four groups:

- *task schedulers*[1] (high-throughput schedulers) promote the performance of the system (as measured by aggregate job performance) by optimizing throughput. Throughput is measured by the number of jobs executed by the system;

- *resource schedulers* coordinate multiple requests for access to a given resource by optimizing fairness criteria (to ensure that all requests are satisfied) or resource utilization (to measure the amount of resource used);

- *application schedulers* (high-performance schedulers) promote the performance of individual applications by optimizing performance measures such as minimal execution time, speedup, or other application-centric cost measures;

- *meta-scheduler* is a scheduler that allows to request resources of more than one machine for a single job. May perform load balancing of workloads across multiple systems. Each system would then have its own local scheduler to determine how its job queue is processed. Requires advance reservation capability of local schedulers.

[1]Berman called this kind of scheduler as *job scheduler*. We changed it to be consistent with the terminology adopted in this text.

## D.3 Job Management Systems for Clusters

We consider that a cluster may be made of a set of workstations, multiple CPU systems, or a set of nodes in a parallel computer. Usually, this set of execution nodes or hosts have a single batch server that manages batch jobs. The batch processing is related to the following concepts [112]:

- **Queue** is a collection of schedulable entities, e.g. jobs (or job-related tasks) within the (batch) queuing system. Each queue has a set of associated attributes that determine which actions are to be performed upon each job within the queue. Typical attributes include queue name, queue priority, resource limits, destination(s), and job count limits. Selection and scheduling of jobs are implementation-defined. The use of the term "queue" does not imply the ordering is "first in, first out";

- **Batch** is a group of jobs which are submitted for processing on a computer and the results of which are obtained at a later time;

- **Batch Processing** is the capability of running jobs outside the interactive login session and providing for additional control over job scheduling and resource contention;

- **Batch Queue** is an execution queue where the request actually is started from;

- **Batch Server** is a persistent subsystem (daemon) upon a single host that provides batch processing capability;

- **Batch System** is a set of batch servers that are configured for processing. The system may consist of multiple hosts, each with multiple servers.

The single batch system or centralized job management system lets users execute jobs on a cluster. This system must perform at least the following tasks [133]: (a) monitor all available resources; (b) accept jobs submitted by users together with resource requirements for each job; (c) perform centralized job scheduling that matches all available resources with all submitted jobs according to the predefined policies; (d) allocate resources and initiate job execution; (e) monitor all jobs and collect accounting information.

Some of the most well know centralized job management systems are **LSF** (Load Sharing Facility) [95, 96] from Platform Computing Corporation, **SGE** (Sun Grid Engine) [129] from Sun Microsystems, and **PBS** (Portable Batch System) [16]. The original version of PBS is a flexible batch queuing system developed for NASA in the early to mid-1990s. Nowadays, the Altair Grid Technologies offer two versions: OpenPBS [105], the unsupported older original version and PBS Pro [108], the commercial version.

These three systems are general purpose distributed queuing systems that unite a cluster of computers into a single virtual system to make better use of the resources on the network. Most of them perform only dedicated scheduling, but SGE is also capable of performing opportunistic scheduling.

Although they can automatically select hosts in a heterogeneous environment based on the current load conditions and the resource requirements of the applications, they are not suitable for grid environments. Actually, these systems are very important in our study since they will take part in a grid environment as a "grid node". However, they cannot be used to manage a grid environment since they have scalability problems, have a single point of failure, and do not provide security mechanisms to schedule across administrative domains.

## D.4 Scheduling Mechanisms for Grid Environments

Grid environments are composed by several trust domains. Usually, each domain has its private scheduler, which works isolated from each other. A resource manager grid aware is necessary to allow all these isolated schedulers to work together taking advantage of all grid potential. Several works in the literature present scheduling mechanisms for grids. In this section we present some of the most well know works: Legion (Subsection D.4.1), Globus (Subsection D.4.2), Condor-G (Subsection D.4.3), OurGrid (Subsection D.4.4), the GrADS's Metascheduler (Subsection D.4.5), and ISAM (Subsection D.4.7).

### D.4.1 Legion

The Legion Project of University of Virginia started in 1993. The main result is the Legion system [26, 71, 93], which is nowadays an Avaki commercial product. Legion is an object oriented infrastructure for grid environments layered on top of existing software services. It uses the existing operating systems, resource management tools, and security mecha-

nisms at host sites to implement higher level system-wide services. The Legion design is based on a set of core objects.

In Legion, resource management is a negotiation between resources and active objects that represent the distributed application. In the allocation of resources for a specific task there are three steps [137]: decision (considers task's characteristics and requirements, the resource's properties and policies, and users' preferences), enactment (the class object receives an activation request; if the placement is acceptable, start the task), and monitoring (ensures that the task is operating correctly).

### D.4.2 Globus

Globus [49, 52, 62, 116] is one of the most well know projects on grid computing. The most important result of the Globus Project is the Globus Toolkit. The toolkit consists of a set of components that implement basic services, such as security, resource location, resource management, data management, resource reservation, and communication. From version 1.0 in 1998 to the 2.0 release in 2002 and the latest 3.0, the emphasis is to provide a set of components that can be used either independently or together to develop applications. The Globus Toolkit version 2 (GT2) design is highly related to the architecture proposed by Foster *et al.* [54]. The Globus Toolkit version 3 (GT3) design is based on grid services, which are quite similar to web services. GT3 implements the *Open Grid Service Infrastructure* (OGSI) [53]. The current version, GT4, is also based on grid services, but with some changes in the standard [124].

The Globus Resource Allocation Manager (GRAM) [32] is one of the available services in Globus. Each GRAM is responsible for a set of resources operating under the same site-specific allocation policy, often implemented by a local resource management system, such as LSF or Condor. GRAM provides an abstraction for remote process queuing and execution with several powerful features such as strong security and file transfer.

Thus GRAM does not provide scheduling or resource brokering capabilities but it can be used to start programs on remote resources, despite local heterogeneity due to the standard API and protocol. The Resource Specification Language (RSL) is used to communicate requirements.

To take advantage of GRAM, a user still needs a system that can remember what jobs have been submitted, where they are, and what they are doing. To track large numbers

of jobs, the user needs queuing, prioritization, logging, and accounting. These services cannot be found in GRAM alone, but is provided by systems such as Condor-G [135].

The Globus research group and IBM provide in GT4 the WS-Resource Framework [153]. They released an initial architecture and specification documents with co-authors from HP, SAP, Akamai, TIBCO and Sonic on January 20, 2004. The WS-Resource Framework (WSRF) is an extension of OGSI. WSRF is a set of six Web services specifications. To date, drafts of three of these specifications have been released, along with an architecture document that motivates and describes the WS-Resource approach to modeling stateful resources with Web services.

### D.4.3 Condor and Condor-G

Condor High Throughput Computing System [132, 135, 152], or simply Condor, is a specialized workload/resource management system for compute-intensive jobs. It can be used to manage a cluster of dedicated nodes as well as to harness wasted CPU power from otherwise idle desktop workstations. Using idle desktop workstation computational power is the main difference between Condor and the traditional RMS. The Condor scheduling policy can be classified as *opportunistic*.

Condor is composed of a collection of different daemons [152]. Condor exercises administrative control over a Condor pool. A pool is a set of resources that report to a single daemon called the *collector*. The *collector* is the central repository of information in the Condor system. Almost all Condor daemons send periodic updates to it. Each update is in the form of a *ClassAd*, a data structure consisting of a set of attributes describing a specific entity in the system. The machine where the *collector* runs is referred to as the *central manager*.

Condor has a mechanism called *flocking* that allows jobs to be scheduled across multiple Condor pools [135]. Nowadays, it is implemented as direct flocking that only requires agreement between one individual and another organization, but accordingly only benefits the user who takes the initiative. A particular job will only flock to another pool when it cannot currently run in the pool of submission. It is a useful feature, but is not enough to enable jobs to run in a grid environment, mainly due to security issues.

Condor-G [57, 135] is the job management part of the Condor project to allow users to access grid resources. Instead of using the Condor-developed protocols to start running a job on a remote machine, Condor-G uses the Globus Toolkit to start the job on the remote

machine. Thus, applications can be submitted to a resource accessible through a Globus interface.

Condor-G uses the protocols for secure inter-domain communications and standardized access to a variety of remote batch systems from Globus. The user concerns of job submission, job allocation, error recovery, and creation of a friendly execution environment comes from Condor.

The major difference between Condor flocking and Condor-G is that Condor-G allows inter-domain operation on remote resources that require authentication, and uses Globus standard protocols that provide access to resources controlled by other resource management systems, rather than the special-purpose sharing mechanisms of Condor [57].

### D.4.4 MyGrid and OurGrid

MyGrid [27, 106] enables the execution of bag-of-tasks parallel applications on all machines the user has access to. A bag-of-tasks application has completely independent tasks. The authors [106] argue that scheduling independent tasks, although simpler than tightly coupled parallel applications, is still difficult due to the dynamic behavior and the intrinsic resource heterogeneity exhibited by most grids. The authors also indicate that it is usually difficult to obtain good information about the entire grid as well as about the tasks to make the scheduling plan. Thus, they propose a solution that does not require almost any kind of information: MyGrid uses a dynamic algorithm called Workqueue with Replication (WQR).

The WQR is a dynamic scheduling algorithm that is not based on performance information. It is an extension of the Workqueue algorithm.

The WQR algorithm uses task replication to cope with the heterogeneity of hosts and tasks, and also with the dynamic variation of resource availability due to the load generated by other users in the grid. Note that this strategy allows to deal only with the heterogeneity related to computational capacity. It works like the Workqueue algorithm until all tasks are assigned ("the bag-of-tasks becomes empty"). At this time, hosts that finished their tasks are assigned to execute replicas of tasks that are still running. Tasks are replicated until a predefined maximum number of replicas is achieved (in MyGrid, the default is one). This replication leads to wasted CPU cycles. Note that the replication assumes that the tasks do not cause side effects. If a task does not have side effects, it can be executed more than once producing always the same final results.

MyGrid executes at the user level. There is a machine called *home machine* that coordinates the execution of the applications through MyGrid. It is assumed that the user has good access to the home machine (often it will be the user's desktop). The home machine schedules tasks to run on the *grid machines* using the WQR algorithm. Grid machines do not necessarily share file systems with the home machine.

OurGrid [6, 7] extends the MyGrid efforts, including the utilization of grid technology on commercial settings and the creation of large-scale community grids. OurGrid is a resource sharing system based on peer-to-peer technologies. The resources are shared according to a "network of favors model", in which each peer prioritizes those who have credit in their past history of interactions.

### D.4.5 MetaScheduler in GrADS Project

The GrADS system [12] is an application scheduler. Its execution model can be described as follows. The user invokes the Grid Routine component to execute his/her application. The Grid Routine invokes the component Resource Selector. The Resource Selector accesses the Globus MetaDirectory Service (MDS) to get a list of machines that are alive and then contact the Network Weather Service (NWS) to get system information for the machines. The Grid Routine then invokes a component called Performance Modeler with the problem parameters, machines and machine information. The Performance Modeler builds the final list of machines and sends it to the Contract Developer for approval. The Grid Routine then passes the problem, its parameters, and the final list of machines to the Application Launcher. The Application Launcher spawns the job using the Globus management mechanism (GRAM) and also spawns the Contract Monitor. The Contract Monitor monitors the application, displays the actual and predicted times, and can report contract violations to a rescheduler.

Although the execution model is efficient from the application perspective, it does not take into account the existence of other applications in the system. Thus, Vadhiyar and Dongarra [140] proposed a metascheduling architecture in the context of the GrADS Project. The metascheduler receives candidate schedules of different application level schedulers and implements scheduling policies for balancing the interests of different applications.

### D.4.6 EasyGrid

EasyGrid [17, 19, 40] is a framework for the automatic grid enabling of MPI parallel applications. It provides services oriented towards the individual application in such a way that each application appears to have exclusive access to a virtual grid. EasyGrid middleware provides application level scheduling.It employs a distributed hierarchy of management processes to control de execution of MPI applications on a computational grid [17].

### D.4.7 ISAM

ISAM (*Infra-estrutura de Suporte às Aplicações Móveis* – Support Infrastructure to Mobile Applications) [154, 155, 156] is a proposal of an integrated solution, from development to execution, for general purpose pervasive applications. These applications are distributed, mobile, adaptive an reactive to the context. Aiming at supporting the follow-me semantics (the application follows the user) for the pervasive applications, the ISAM middleware concerns with resource management in heterogeneous, multi-institutional, networks.

## D.5 Comparison

We presented some of the most representative RMSs used nowadays in grid research. We now present a brief comparison of them, summarized in Tables D.1 and D.2. We built this table using some of the characteristics we considered more important to our work.

In Table D.1, the first column presents our classification presented in Section D.1 which includes Berman's classification and the GGF metascheduler definition. PBS, LSF, and SGE are task schedulers. Condor and Legion are resource schedulers and both support at some extent scheduling over multiple domains. MyGrid and EasyGrid are classified as application scheduler. Globus, Condor-G, the GrADS' Metascheduler, and ISAM can be classified as metaschedulers or high-level schedulers.

PBS, LSF, and Legion can also be classified as dedicated schedulers, SGE can work as a dedicate scheduler as well as an opportunistic scheduler, while Condor is an opportunistic scheduler. In our table, the symbol "–" means that the classification cannot be applied to the corresponding system.

Table D.1: Comparison of presented schedulers – part I

| System | Classification | | | Source Available |
|---|---|---|---|---|
| PBS | task scheduler | dedicated | centralized | OpenPBS – yes PBSPro – no |
| LSF | task scheduler | dedicated | centralized | no |
| SGE | task scheduler | dedicated and opportunistic | centralized | no |
| Condor | resource scheduler | opportunistic | centralized | yes |
| Legion | resource scheduler | dedicated | hierarchical | no |
| MyGrid | application scheduler | – | centralized (per application) | yes |
| EasyGrid | application scheduler | – | hierarchical | no* |
| Globus | metascheduler | – | hierarchical | yes |
| Condor-G | metascheduler | – | hierarchical | yes |
| GrADS' Metascheduler | metascheduler | – | hierarchical | no |
| ISAM | metascheduler | – | hierarchical | no* |

The third column indicates that PBS, LSF, SGE, and Condor have centralized schedulers while Legion, Globus, Condor-G, GrADS' Metascheduler, and ISAM are hierarchical. MyGrid was classified as centralized, but actually there is one instance of scheduler for each running application.

Other criterion shown in the table (last column) is if there is source code available for research purposes in a public repository. Some listed as no, could release source code on demand and are marked with a "*". Most of the RMS' source code are available as our table shows.

Table D.2 presents other aspects of these systems and is a continuation of the previous table. All RMS use some kind of static information to take scheduling decisions. The first column indicates if the RMS also needs some dynamic information (usually related to resource utilization and availability). MyGrid is the only one that does not need any dynamic information to make its decisions. MyGrid and ISAM use the task replication technique to obtain fault tolerance and better performance at the cost of wasting some CPU cycles.

Since computational grids are dynamic, jobs should adapt themselves according to characteristics such as availability and load in order to obtain application performance. Execution must be flexible and adaptive to achieve either robust or even good performance due to heterogeneity of configuration, performance, and reliability in grid environments.

Table D.2: Comparison of presented schedulers – part II

| **System** | **Dynamic information** | **Migration** | **Replication** |
|---|---|---|---|
| PBS | yes | no | no |
| LSF | yes | no | no |
| SGE | yes | no | no |
| Condor | yes | yes | no |
| Legion | yes | yes | no |
| MyGrid | no | no | yes |
| EasyGrid | yes | no | yes |
| Globus | yes | no | no |
| Condor-G | yes | no | no |
| GrADS' Metascheduler | yes | yes | no |
| ISAM | yes | yes | no |

So, one of the approaches is that jobs are able to migrate. Migration [112] describes the rearrangement of allocated resources within a resource pool. Several works deal with the so called "opportunistic" migration of jobs when a "better" resource is discovered [140, 141, 100, 3]. The Migration's column indicates if the RMS can migrate a job after its allocation. Condor, Legion, GrADS' metascheduler, and ISAM can migrate a running job for performance reasons. Condor can also migrate to avoid disturbing a user due to opportunistic scheduling. Note that in Globus and Condor-G migration in the local RMS can happen, but both cannot directly control this kind of mechanism.

The two most popular grid-aware systems are Condor and Globus. As mentioned before, Condor is a specialized workload management system for compute-intensive jobs. Like other full-featured batch systems presented, Condor provides a job queueing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. Users submit their serial or parallel jobs to Condor, Condor places them into a queue, chooses when and where to run the jobs based upon a policy, carefully monitors their progress, and ultimately informs the user upon completion. Among other things, Condor allows transparent migration of jobs from overloaded machines to idle machines and checkpointing, which permits that jobs can restart in another machine without the need to start from the beginning. These are typical tasks of a resource manager.

Globus, by its turn, is a whole framework that includes a set of services used, for example, to securely transfer files from one grid node to another (GridFTP), to manage meta data (MDS), and to allocate remote resources (GRAM). Condor-G [57] puts together Con-

dor facilities to manage jobs with the Globus grid facilities such as secure authentication (GSI) and data transfer.

Condor applies an opportunistic scheduling policy that concentrates on allocating idle resources to take advantage of idle CPU cycles. Globus focuses on providing several different services to execute secure code on authorized machines.

Application management and control, load balancing and data locality are not their main focus.

The Condor scheduling system and the MetaScheduler in the GrADS Project present some similarities. Both support preemption of executing jobs to either accommodate other jobs or to transfer the control of the resources to the resource owners. However, the first is more concerned to free resources reclaimed by the owners whereas the second one tries to get performance. Besides, the Condor's Negotiator component has similar functionality to the Metascheduler's Contract Negotiator.

An aspect not covered in our tables is related to resource description. The resources must be represented in a somehow independent way to manage the inherent heterogeneity of grid environments. Legion has adopted the object model as a way to get a uniform way to access the resources. Globus and Condor-G have decided to adopt a description language which is translated to an internal representation.

All systems present some way to deal with faults. One important issue that RMS must deal with is data loss. Most available software can not handle network traffic properly. For example, Condor, one of the software that we had experience with, can either loose jobs that were in the job queue, or generate corrupt data files, because of lack of network flow control. The user is responsible to manually control the number of jobs that will be simultaneously submitted in order to avoid network congestion. As a consequence of the little attention given to flow control and data management, data loss can occur due to overflow when too much traffic is generated on data and code transfers. Some experiments reported on [39] illustrate this problem. From 45,000 tasks launched, around 20% failed for several reasons, including data corrupted due to packet loss, and had to be re-submitted.

Other problem concerns application submission. In some systems, applications are launched in the user machine. Because most grid aware software create one connection or a new process to each launched job, the submit machine can become overloaded and have a very low response time.

## D.6 Conclusion

On the light of this discussion about grid aware systems and their limitations, we performed some experiments to evaluate some of the problems, which are described in Appendix E, and discuss a promising solution to these limitations in the Appendix F.

# Apêndice E

# Experiments with Distributed Submission

This chapter presents some computational experiments to evaluate how the number of tasks affects submit machine load, and how centralized and distributed submissions affects submit machine(s). Our main goals are: (1) to show that a hierarchy of submit machines, i.e. distributed submission, is an important alternative in a grid environment; (2) to check how much we can profit from a distributed submission schema compared to a centralized one.

Two kinds of experiments are performed in two kinds of computational systems: application execution in a local cluster with Condor system [136] and simulations of cluster and grid environments in a single machine running the Monarc simulator [90]. Section E.1 presents the results from experiments executed using Condor. Section E.2 presents the results and analysis of the second experiment.

## E.1 Experiments using Condor

We present in this section results from experiments that were executed using a cluster with machines Pentium IV 1.8 GHz with 1 GB RAM. For our experiments, only seven nodes were available, with Red Hat 7.3 (kernel 2.4.18-3) and Condor version 6.4.7, connected with a 3Com gigabit Ethernet switch.

### E.1.1 Methodology

We assume that there is a graph of independent tasks. The graph is described using a Condor submission file. Instead of using a real application, we decide to run synthetic tasks.

Our tasks can be considered as synthetic tasks, since they do not represent a real application. We created some deterministic jobs to be executed, because we were concerned in having control of how much CPU would be used and how much data would be generated.

We implemented a C program that writes an amount of data and then performs a calculation loop. This way we could simulate a CPU intensive application with deterministic tasks and make it possible to vary the task execution times. We ran two experiments, each one with tasks of different execution times: a few seconds ($\sim$ 3 seconds) and some minutes ($\sim$ 20 minutes). All tasks write around 10 kbytes of data, which is the average output file size of ILP experiments reported in Dutra *et al* [39].

For both experiments, we defined two cases: (1) all tasks are submitted in a single submit machine (centralized submission), and (2) the tasks are divided in two subgraphs, each subgraph is submitted in a different submit machine creating two queues at the central manager (distributed submission).

In both cases, the size of the application graph is variable. For the first experiment: 50, 100, 250, 500, 1000, 2000, 3000, 5000 or 7500 tasks. For the second: 50, 100, 250, and 500. During execution, we get information from the */proc/loadavg* directory, which provides information about load averages for the Linux system.

### E.1.2 Results and Analysis

In the first experiment, we monitored the submit machine during all execution time. At each 5s, the load information was obtained from */proc/loadavg* and stored to be analyzed at the end of the execution. Figure E.1 presents the load average for the centralized submission. The dashed line represents the increase in the average load as the number of submitted tasks increases. The vertical lines represent the standard deviation which is quite high in all cases. It means that there were some load peaks during the experiment. Note that the standandard deviation shows the load variation during one single execution.

At first, we thought this monitoring methodology would be sufficient to check how much the machine was overloaded by the submission. However, this methodology was not effective to detect machine saturation, because of the imprecision of the measurements (the monitoring script was started and terminated by hand). We then tried to measure the load during all execution time to detect load saturation caused both by submission and by receiving results. However, for a large number of tasks, the submit machine also executed some jobs producing a misleading load figure.

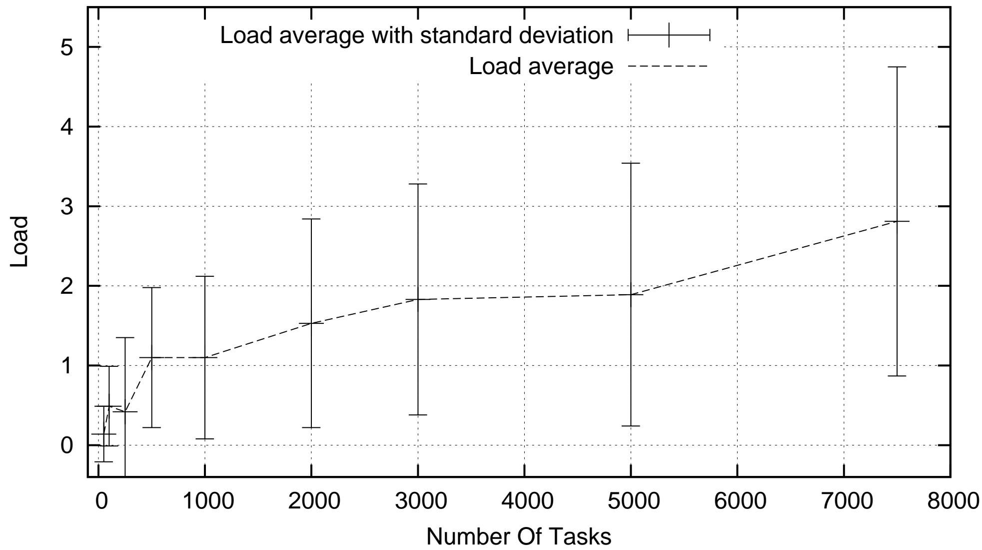



Figure E.1: Experiment 1 – Load average of the submit machine

Thus, we had two main problems with this approach: (1) since Condor does not present a direct way to detect termination, we had to do some *condor_q* during the execution, and this could cause changes in the machine load; (2) after submission, the submit machine can be used as an execution machine. Thus, some of the load peaks can be caused either by task execution or results gathering.

For Experiment 2, another load measurement methodology was adopted. A script file was written to perform submission. Load information was collected in two moments: before and after the condor_submit command.

Table E.1 presents statistical values related to tasks submission and execution times for both centralized and distributed submissions for Experiment 1, and Table E.2 for Experiment 2.

Different workloads were executed for Experiment 1 and Experiment 2 and are represented in column Tasks. Table E.2 presents the same information for Experiment 2. Table E.1 shows for all workloads of Experiment 1, values related to time (column Time) of submission and of execution, and total time for centralized (c) and distributed (d) submission (column Sub.). Figure E.2 presents graphically the time measurement.

The statistical values presented for both tables are: (a) Mean: the time average; (b) Std. Dev.: standard deviation i.e. a measure of the dispersion of the frequency distribution; (c) Median: the middle value in the distribution; (d) Min.: minimum value; (e) Max.: maximum value; (f) Sum: the amount obtained as a result of adding the time for each task.

Table E.1: Experiment 1: Statistical values for centralized (c) and distributed (d) submission

| Tasks | Time | Sub. | Mean | Std. Dev. | Median | Min. | Max. | Sum |
|---|---|---|---:|---:|---:|---:|---:|---:|
| 50 | Submission | c | 107.9400 | 61.1036 | 110.00 | 5.00 | 210.00 | 5397.00 |
| | | d | 196.6600 | 156.3943 | 200.00 | 8.00 | 376.00 | 9833.00 |
| | Execution | c | 3.9200 | 0.8769 | 4.00 | 3.00 | 5.00 | 196.00 |
| | | d | 6.0200 | 2.5354 | 6.00 | 3.00 | 10.00 | 301.00 |
| | Total | c | 111.8600 | 61.1328 | 113.00 | 8.00 | 215.00 | 5593.00 |
| | | d | 202.6800 | 156.8943 | 205.50 | 12.00 | 379.00 | 10134.00 |
| 100 | Submission | c | 90.8900 | 43.2516 | 90.5000 | 6.00 | 173.00 | 9089.00 |
| | | d | 214.3700 | 165.3999 | 215.5000 | 8.00 | 422.00 | 21437.00 |
| | Execution | c | 4.9300 | 1.9083 | 5.0000 | 3.00 | 8.00 | 493.00 |
| | | d | 4.7900 | 1.6654 | 4.0000 | 3.00 | 8.00 | 479.00 |
| | Total | c | 95.8200 | 43.5953 | 95.5000 | 9.00 | 176.00 | 9582.00 |
| | | d | 219.1600 | 165.4037 | 218.5000 | 11.00 | 425.00 | 21916.00 |
| 250 | Submission | c | 212.6720 | 117.9474 | 214.5000 | 7.00 | 409.00 | 53168.00 |
| | | d | 301.1800 | 191.7456 | 282.0000 | 9.00 | 633.00 | 75295.00 |
| | Execution | c | 4.2840 | 1.1280 | 4.0000 | 3.00 | 8.00 | 1071.00 |
| | | d | 4.2360 | 1.0700 | 4.0000 | 3.00 | 7.00 | 1059.00 |
| | Total | c | 216.9560 | 118.1403 | 218.5000 | 10.00 | 412.00 | 54239.00 |
| | | d | 305.4160 | 191.7565 | 285.0000 | 12.00 | 637.00 | 76354.00 |
| 500 | Submission | c | 405.5380 | 227.3701 | 409.0000 | 7.00 | 801.00 | 202769.00 |
| | | d | 574.3280 | 363.5963 | 471.0000 | 9.00 | 1158.00 | 287164.00 |
| | Execution | c | 4.5520 | 1.7290 | 4.0000 | 3.00 | 9.00 | 2276.00 |
| | | d | 4.5120 | 1.4567 | 4.0000 | 3.00 | 9.00 | 2256.00 |
| | Total | c | 410.0900 | 227.3254 | 412.0000 | 10.00 | 804.00 | 205045.00 |
| | | d | 578.8400 | 363.5972 | 476.5000 | 12.00 | 1164.00 | 289420.00 |
| 1000 | Submission | c | 1068.8520 | 590.1029 | 1105.5000 | 8.00 | 2006.00 | 1068852.00 |
| | | d | 971.0920 | 585.6023 | 867.0000 | 9.00 | 1918.00 | 971092.00 |
| | Execution | c | 4.3470 | 1.2570 | 4.0000 | 3.00 | 9.00 | 4347.00 |
| | | d | 4.5170 | 1.5595 | 4.0000 | 3.00 | 11.00 | 4517.00 |
| | Total | c | 1073.1990 | 590.1135 | 1109.5000 | 11.00 | 2010.00 | 1073199.00 |
| | | d | 975.6090 | 585.7662 | 871.0000 | 15.00 | 1921.00 | 975609.00 |
| 2000 | Submission | c | 2306.6130 | 1299.7641 | 2352.0000 | 10.00 | 4757.00 | 4613226.00 |
| | | d | 1964.4805 | 1146.1761 | 1817.5000 | 10.00 | 3956.00 | 3928961.00 |
| | Execution | c | 4.3335 | 1.4827 | 4.0000 | 3.00 | 10.00 | 8667.00 |
| | | d | 4.9545 | 2.0242 | 4.0000 | 3.00 | 11.00 | 9909.00 |
| | Total | c | 2310.9465 | 1299.7020 | 2356.0000 | 13.00 | 4761.00 | 4621893.00 |
| | | d | 1969.4350 | 1146.1113 | 1822.5000 | 13.00 | 3961.00 | 3938870.00 |
| 3000 | Submission | c | 3725.9780 | 2005.8243 | 3869.5000 | 33.00 | 7014.00 | 11177934.00 |
| | | d | 3285.6307 | 1822.1958 | 3186.0000 | 9.00 | 6321.00 | 9856892.00 |
| | Execution | c | 4.3133 | 1.4852 | 4.0000 | .00 | 11.00 | 12940.00 |
| | | d | 4.7413 | 1.7801 | 4.0000 | 3.00 | 11.00 | 14224.00 |
| | Total | c | 3730.2913 | 2005.8161 | 3873.5000 | 36.00 | 7018.00 | 11190874.00 |
| | | d | 3290.3720 | 1822.1690 | 3191.0000 | 12.00 | 6324.00 | 9871116.00 |
| 5000 | Submission | c | 7253.7438 | 3815.4792 | 7498.0000 | 19.00 | 13075.00 | 36268719.00 |
| | | d | 5508.9044 | 3134.6970 | 5358.0000 | 15.00 | 10963.00 | 27544522.00 |
| | Execution | c | 6.0598 | 1.9842 | 6.0000 | 3.00 | 26.00 | 30299.00 |
| | | d | 6.9050 | 6.9565 | 7.0000 | 3.00 | 356.00 | 34525.00 |
| | Total | c | 7259.8036 | 3815.6316 | 7504.0000 | 22.00 | 13078.00 | 36299018.00 |
| | | d | 5515.8094 | 3134.6459 | 5367.5000 | 20.00 | 10972.00 | 27579047.00 |
| 7500 | Submission | c | 12006.3413 | 5957.6143 | 12673.0000 | 361.00 | 21131.00 | 90047560.00 |
| | | d | 9375.0604 | 5119.7247 | 9391.0000 | 34.00 | 17790.00 | 70312953.00 |
| | Execution | c | 6.4372 | 2.2508 | 7.0000 | 3.00 | 22.00 | 48279.00 |
| | | d | 4.7716 | 1.8624 | 4.0000 | 3.00 | 23.00 | 35787.00 |
| | Total | c | 12012.7785 | 5957.1864 | 12681.0000 | 368.00 | 21135.00 | 90095839.00 |
| | | d | 9379.8320 | 5119.9184 | 9395.5000 | 38.00 | 17793.00 | 70348740.00 |

Table E.2: Experiment 2: Statistical values for centralized (c) and distributed (d) submission

| Tasks | Time | Sub. | Mean | Std. Dev. | Median | Min. | Max. | Sum |
|---|---|---|---|---|---|---|---|---|
| 50 | Submission | c | 4280.2000 | 2736.2375 | 4139.0000 | 4.00 | 8706.00 | 214010.00 |
| | | d | 4706.5000 | 2893.7614 | 4898.0000 | 5.00 | 9347.00 | 235325.00 |
| | Execution | c | 1269.0600 | 85.7318 | 1233.0000 | 1209.00 | 1615.00 | 63453.00 |
| | | d | 1265.6800 | 61.9817 | 1227.5000 | 1205.00 | 1412.00 | 63284.00 |
| | Total | c | 5549.2600 | 2749.4419 | 5584.5000 | 1217.00 | 9944.00 | 277463.00 |
| | | d | 5972.1800 | 2895.6075 | 6120.5000 | 1213.00 | 10644.00 | 298609.00 |
| 100 | Submission | c | 8907.7100 | 5366.8455 | 9004.5000 | 5.00 | 17778.00 | 890771.00 |
| | | d | 11005.7900 | 3832.6245 | 10799.0000 | 4214.00 | 18755.00 | 1100579.00 |
| | Execution | c | 1262.9600 | 66.8299 | 1234.0000 | 1215.00 | 1587.00 | 126296.00 |
| | | d | 1246.9300 | 42.5192 | 1228.5000 | 1209.00 | 1381.00 | 124693.00 |
| | Total | c | 10170.6700 | 5349.3733 | 10228.0000 | 1226.00 | 18998.00 | 1017067.00 |
| | | d | 12252.7200 | 3825.2304 | 12057.0000 | 5435.00 | 19985.00 | 1225272.00 |
| 250 | Submission | c | 22125.4560 | 13305.0442 | 21918.5000 | 3.00 | 45920.00 | 5531364.00 |
| | | d | 25442.0160 | 9162.2504 | 25578.0000 | 318.00 | 42885.00 | 6360504.00 |
| | Execution | c | 1240.5560 | 31.7401 | 1224.0000 | 1209.00 | 1343.00 | 310139.00 |
| | | d | 1249.6920 | 43.7549 | 1229.5000 | 1207.00 | 1380.00 | 312423.00 |
| | Total | c | 23366.0120 | 13306.5348 | 23156.5000 | 1225.00 | 47136.00 | 5841503.00 |
| | | d | 26691.7080 | 9159.8378 | 26824.5000 | 1625.00 | 44248.00 | 6672927.00 |
| 500 | Submission | c | 49828.0240 | 29943.8155 | 48463.5000 | 6.00 | 102940.00 | 24914012.00 |
| | | d | 74674.1060 | 42339.9788 | 66607.5000 | 343.00 | 169463.00 | 37337053.00 |
| | Execution | c | 1270.7400 | 75.4304 | 1252.5000 | 781.00 | 2086.00 | 635370.00 |
| | | d | 1260.6660 | 53.8226 | 1234.5000 | 1206.00 | 1521.00 | 630333.00 |
| | Total | c | 51098.7640 | 29959.1608 | 49693.5000 | 1220.00 | 104157.00 | 25549382.00 |
| | | d | 75934.7720 | 42334.8140 | 67828.5000 | 1663.00 | 170726.00 | 37967386.00 |

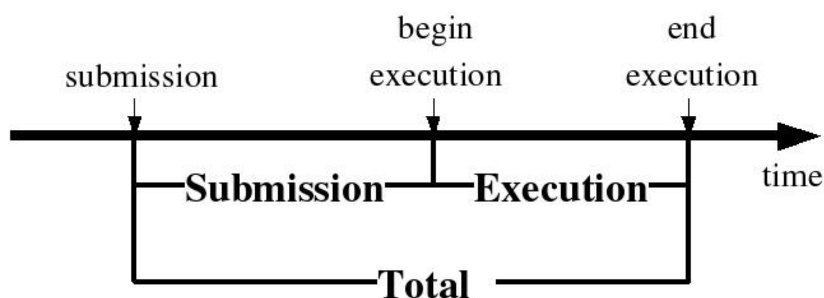



Figure E.2: Time measurements in the execution timeline

All these data were obtained as follows: first, scripts are used to get information about individual tasks in the Condor log files, then, these data is submitted to the SPPS statistic program.

Figure E.3 presents the average total execution time for each workload in both cases for Experiment 1. The total execution time is the sum of submission and execution times. The total execution time linearly increases since the submission time increases as increase the number of tasks to be executed.

Considering all workloads, with 95% of confidence and eight degrees of freedom, the $t$ value is 1.614. Since it is lower than the $t^t$=1.860, the difference in performance for centralized and distributed cases is not significant. However, if we consider only the five last workloads, which have at least 1000 tasks, with 95% of confidence and four degrees of freedom, $t$ is 2.153. Since $t^t$ is equal to 2.132, for instances bigger than 1000 tasks, the distributed submission is significantly different of centralized submission with 95% of confidence.

Therefore, distributed submission is better for workloads bigger than 1000 tasks. This is an interesting result, since we are concerned with applications with huge number of tasks, in the order of thousands. This is an indication that having a distributed submission can help decreasing the total execution time.

Figure E.4 shows the total application execution time, i.e., the sum of all tasks execution. This time is higher than the user response time.

The response time for Experiment 1 is presented in Table E.3 and Figure E.5. In most cases, the response time is better for centralized submission, but for the higher instances (5000 and 7500 tasks) the response time is better for distributed submission.

Table E.3: Condor Experiment 1 – response time in seconds

| Number of Tasks | Centralized Submission | Distributed Submission |
|---:|---:|---:|
| 50 | 210 | 5605 |
| 100 | 173 | 5321 |
| 250 | 409 | 5870 |
| 500 | 801 | 5724 |
| 1000 | 2006 | 6100 |
| 2000 | 4757 | 7082 |
| 3000 | 7014 | 11587 |
| 5000 | 13075 | 11370 |
| 7500 | 21131 | 17790 |

Another interesting aspect to be analyzed is the load balancing of the execution machines. Figures E.6 and E.7 present load balancing information, respectively for Experi-

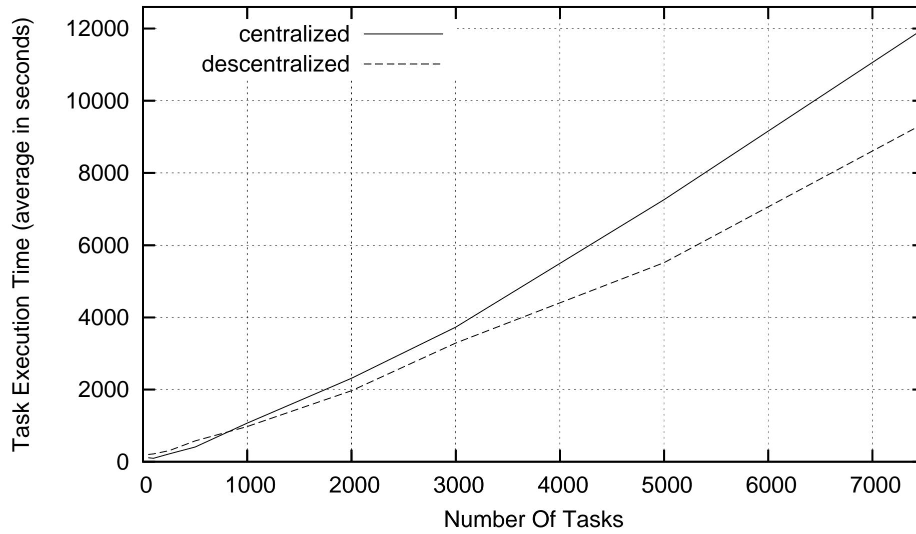



Figure E.3: Condor Experiment 1 – average task time for centralized and distributed submission

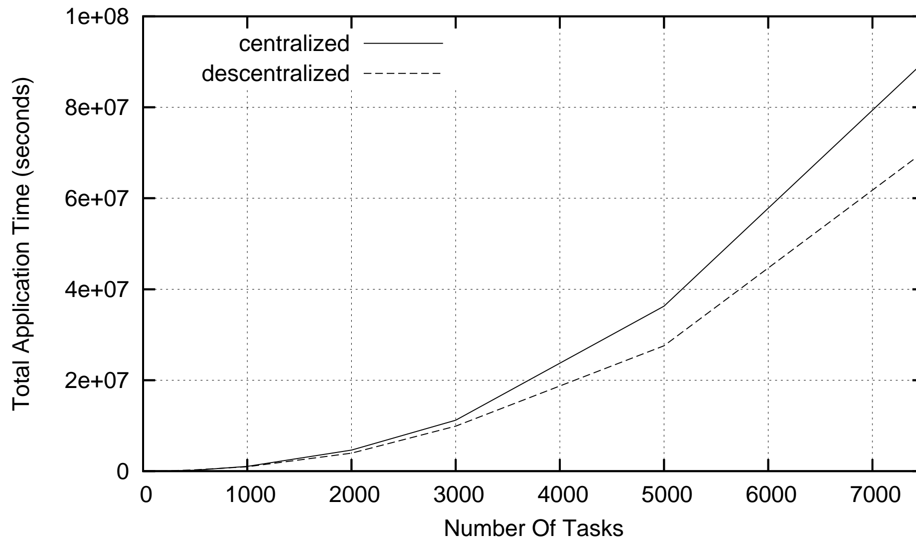



Figure E.4: Condor Experiment 1 – total execution time (sum)

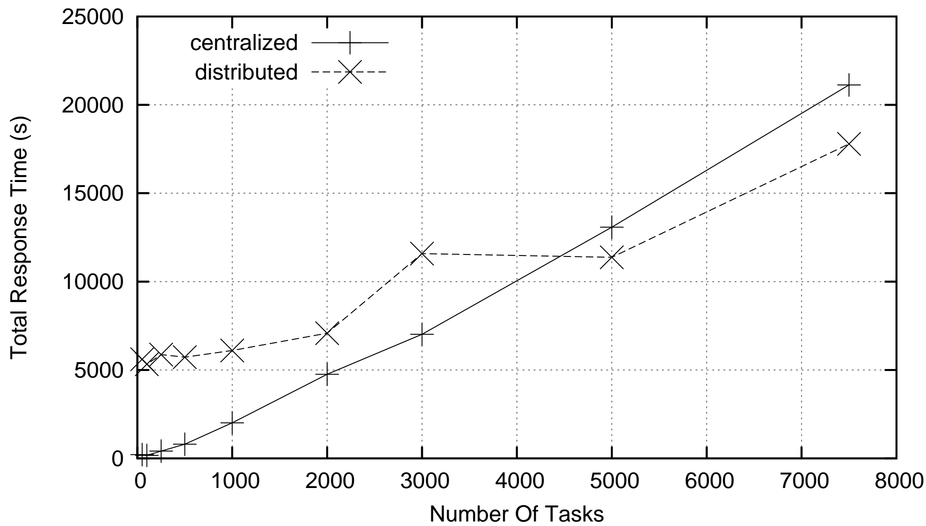



Figure E.5: Condor Experiment 1 – response time in seconds

ment 1 and 2. For each execution, the bar represents the percentage of tasks each machine executed. Each color represents a different machine in the cluster. We can observe that Experiment 2 did not obtain a good load balancing. For instance, for 50 tasks submitted from one machine, all tasks were executed in the same machine, and for bigger instances, most used four or five machines. The distributed submission presented the best results considering load balancing. Experiment 2 allocated the seven machines in almost all cases.

For the second experiment, the cluster was isolated avoiding users from logging into the machines, allowing a more stable testing environment. Besides, tasks take more time to complete. Probably, these are the reasons for the general better load balancing.

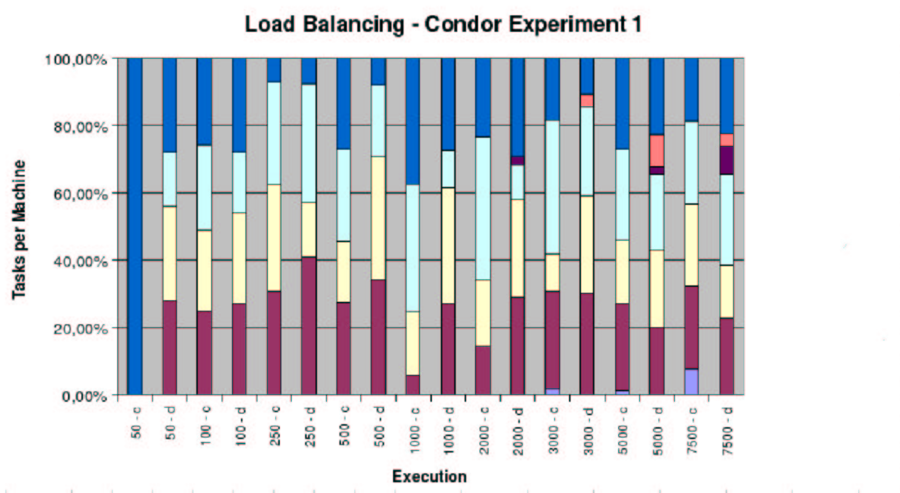



Figure E.6: Condor Experiment 1 – percentage of executed tasks per machine

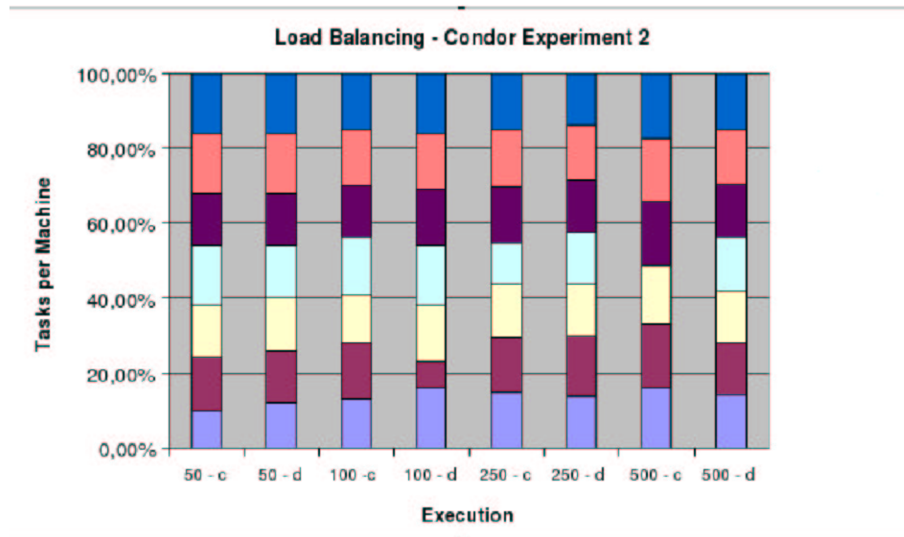



Figure E.7: Condor Experiment 2 – percentage of executed tasks per machine

Some possible drawbacks of the first experiment are: (1) results are specific to the Condor environment, so some results may not be extrapolated to other RMSs; (2) we used an old version of Condor, and maybe newer versions have better performance; (3) our experimental environment was a small cluster and not a grid. Thus, we decided to perform the simulations presented in the next section.

## E.2 Experiments using Monarc

As a further step to analyze the distribution of the submission we decided to use simulation. Simulation is a powerful tool to test distributed models. It is cheaper, simpler, faster, and more flexible than running in a real environment, avoiding operational problems and implementation dependent issues. Besides, it is possible to reproduce experiments that otherwise would be non deterministic. It is easier to setup different parameters, simplifying debugging and monitoring of events.

Several tools exist for simulation of distributed systems that allow grid environment simulation [128]. Some are extensions of already available simulation tools while others were implemented from scratch.

We analyzed four tools in a previous work [146], and we chose to use Monarc (Models of Networked Analysis at Regional Center) [90, 91] as our simulation tool. Monarc is a popular tool among physicists. It is written in Java and thus is portable. Besides, Monarc presents high level abstractions suitable to our purposes.

Monarc is a discrete-event simulation tool that allows distributed system simulation. It is part of the Monarc Project and it was developed by a team with members from CERN, Caltech, and Politehnica University of Bucharest. Monarc is now in its second version [138]. Legrand and Newman [90] report some experiments used to validate Monarc. They compare simulation results with theoretical expected results based on queuing theory, and they conclude that the simulation is quite close to the theoretical model. They also show that the simulation tool reproduce results (job execution time) obtained with a real testbed.

Some characteristics of Monarc version 2.1.7 must be outlined, since they are relevant for data analysis:

- Monarc aims at providing high level abstractions, thus some lower level aspects cannot be analyzed, e.g. message package traffic. It is an advantage because irrelevant details are omitted allowing a macro vision of the system. But, it can be a disadvantage, because some low level details should be monitored, e.g. limit of socket ports;
- The simulator allows to allocate tasks into a CPU while there is available memory. With this restriction, there is no simulation of swap;
- It is possible to set bandwidth limits and speeds of links between grid nodes (regional centers). However, in the used version, the bandwidth limit was not respected. Thus, network saturation tests could not be done;
- Monarc allows to create a task to be executed in a specific CPU. But, there is a termination detection problem when the first CPU is manually scheduled;
- the standard scheduling algorithm is round-robin: each task is allocated for the first free CPU, if there is no more free CPU, then for the first CPU with available memory, otherwise it goes to the queue. There is no cost for scheduling (time for scheduling equals zero).

The next sections will describe our experiments and results using Monarc.

### E.2.1 Methodology

Two experiments were simulated: a cluster with a high speed network connection and a small and heterogeneous grid. Both experiments were conducted with different workloads,

and different number of submit machines. All simulated tasks have the same characteristics (same execution time and same memory consumption). The number of submitted tasks was the same for both experiments and the number of submit machines was 1, 2, 3, 4, 16, and 40 or 80 for the experiment 1 and 1 to 5 to experiment 2. In the experiment 2, the submission was restricted to a single grid node thus limiting to 5 machines.

As for the Condor experiment, we used an application composed by a set of independent tasks. To simulate this application we constructed a DAG as represented in Figure E.8. Initially, a set of small tasks is executed to simulate the cost of putting the tasks in a central queue. To synchronize the termination of all small tasks indicating that the application can start, a *Barrier* task is executed. This task has zero cost of execution and communication. Then, a set of big tasks is executed. This set of big tasks represents the real user task, and implements the cost of executing the real application. At the end, a task *GetStatistics* is called to print execution information about the simulation.

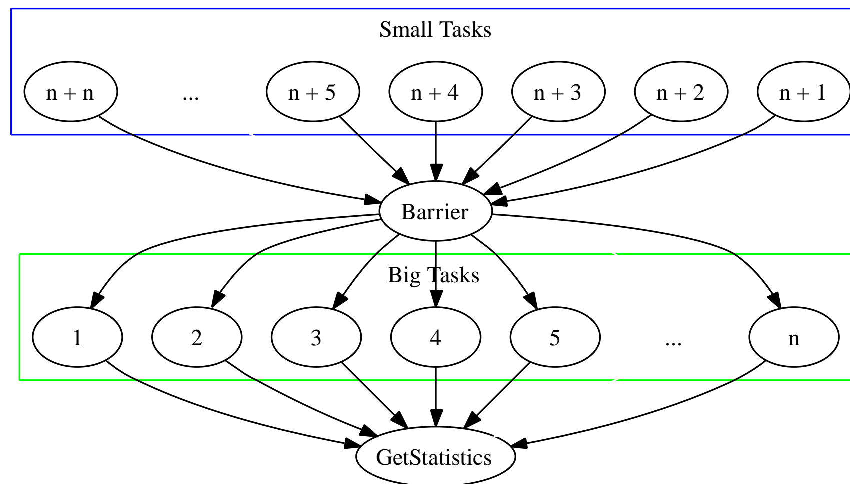



Figure E.8: Monarc Experiment – methodology: DAG of tasks

### E.2.2 Results and Analysis

The Monarc simulator is deterministic, so the results we will present in this section are the result of one single execution for each case. We simulated two medium clusters with, respectively 40 and 80 machines and a small grid. The cluster machines are set as the machines of our laboratory: each machine has the power equivalent of an AMD Athlon XP 2GHz and 512MB of memory. The grid represents a real testbed set between some

Brazilian Universities. Figure E.9 represents the institutions, the number of machines each one has, and the interconnection bandwidth.

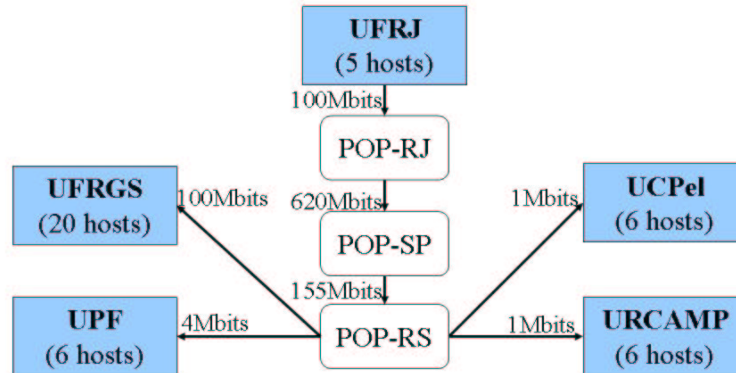
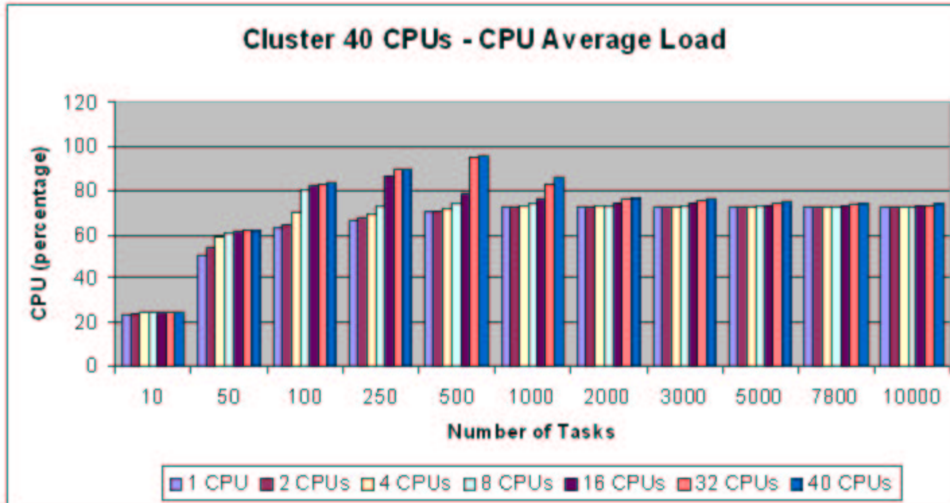


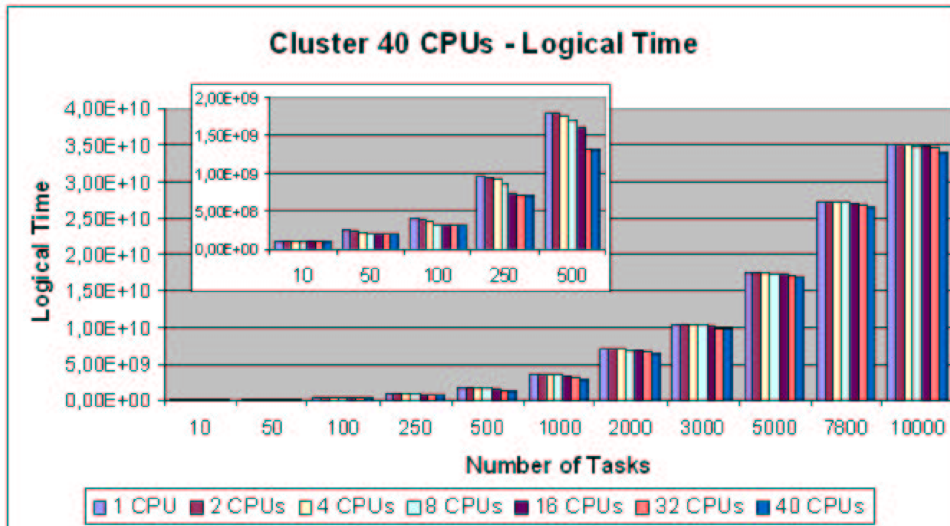

Figure E.9: Monarc Experiment: simulated grid elements

The main results are presented in Figures E.10, E.11, and E.12. We can see in the graphs of the three cases, that when the number of submit machines increases, the logical time decreases and the CPU usage is better. For a small number of tasks, increasing the number of submit machines can lead to a significant decrease in execution time.

These results are an indication that for a real environment would be possible to get better performance if more than one submission queue is used. The results in Figures E.10(a) and E.11(b) indicate that it is good to use as many machines as submit machines as possible. However, we must remember that there are some simplifications in our simulation model that need to be considered before applying the results: (a) the simulation environment is dedicated (only tasks from the application were running); (b) it did not consider file transfer costs; (c) it did not consider all network cost. So, probably there is a limit for the number of submit machines that could be used in practice. This number will depend on the load of the environment, the application characteristics and the number of tasks.

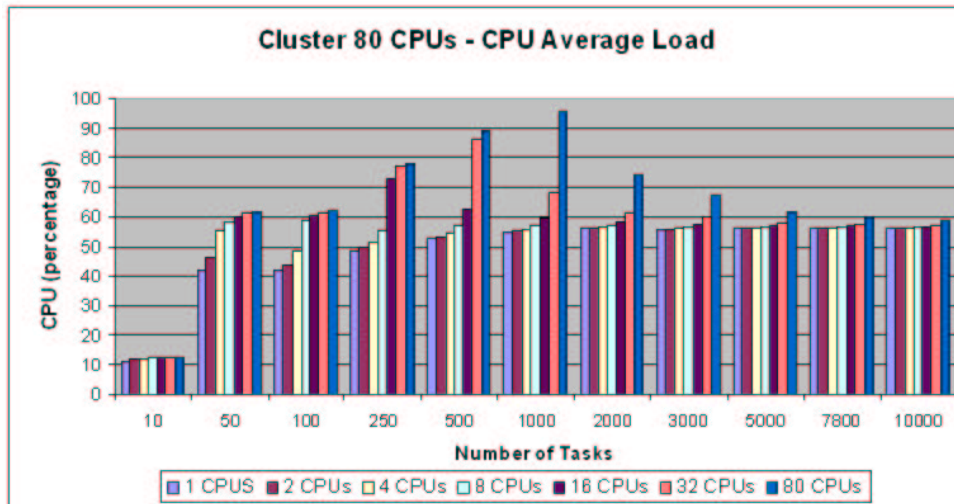



(a)

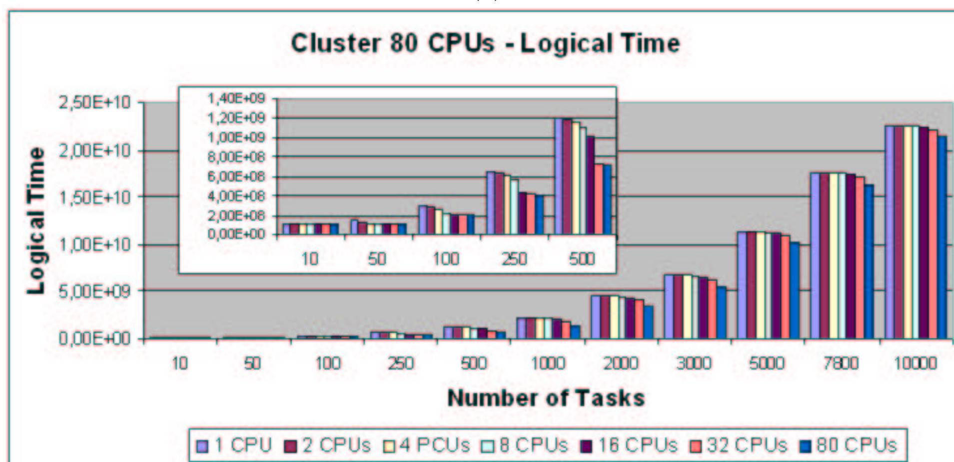



(b)

Figure E.10: Monarc - cluster 40 CPUs: (a) CPU load average and (b) logical execution time

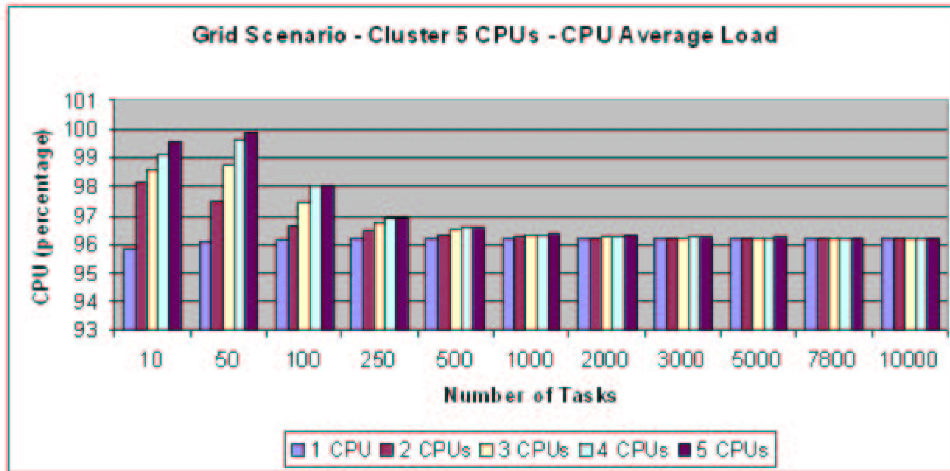



(a)

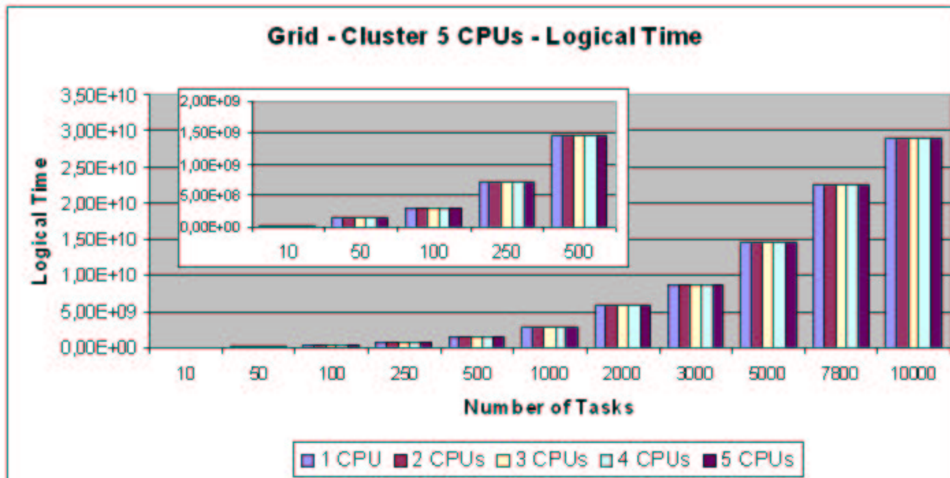



(b)

Figure E.11: Monarc - cluster 80 CPUs: (a) CPU load average and (b) logical execution time




(a)




(b)

Figure E.12: Monarc - grid, submit cluster with 5 CPUs: (a) CPU load average and (b) logical execution time

## E.3 Conclusion

The experiments reported in this chapter are performed so we could get some real data as basis for our hypothesis that a centralized submission is not a good option for large number of jobs. The main conclusion we can get from our initial experiments is that a distributed submission can be a good alternative.

In the next chapter we present a new application submission and management model for grid environments that relies in the distributed control idea.

# Apêndice F

# GRAND: An Integrated Application Management System for Grid Environments

This section aims to address some of the issues related to the construction of an integrated system to manage application in grid environments. Our proposed model is called GRAND (**G**rid **R**obust **A**pplicatio**n** **D**eployment) [114, 143, 144, 145, 146, 147]. More specifically, we are concerned with the management of applications that spread a high number of tasks (e.g. thousands) in a grid environment.

Application management consists in preparing, submitting, and monitoring the execution progress of all tasks that compose the user application. Application management encompasses several activities. From application description to final results gathering, there are several actions the system must perform including application deployment in grid nodes. The GRAND model, mainly due to its hierarchical organization, has some steps as presented in Figure F.1. This figure presents a schematic view of the main steps necessary to execute a distributed application in GRAND.

The first step is called *DAG inference*: given an application description, the system detects dependencies between tasks obtaining a directed acyclic graph (DAG). In the *clustering* step, an application composed by several tasks is divided in subgraphs. Clustering aims to get together tasks that are dependent or to obtain a group of independent tasks that can be submitted together. Then, the obtained subgraphs can be assigned to resources during the *mapping* step according to the available resources. Finally, the *submission* step is responsible for allocating the chosen resources, ensuring (a) data staging, (b) executable deployment, and (c) subgraphs execution on selected resources.

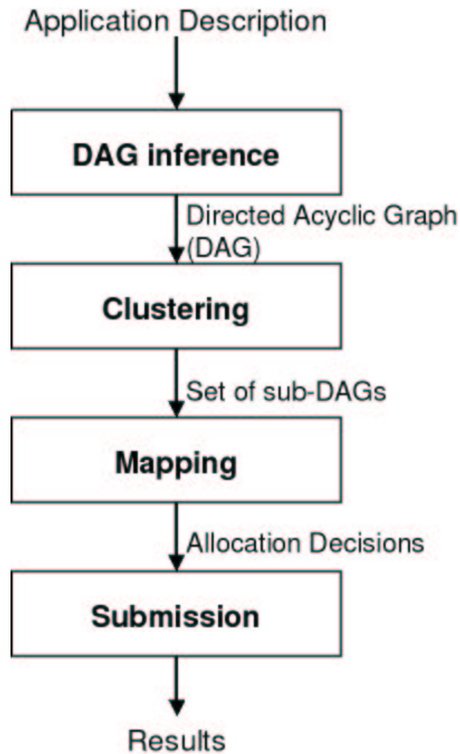



Figure F.1: The GRAND model: steps to execute a distributed application

This chapter discusses all these steps in detail. Before presenting our model to perform such steps, we present our premises (Section F.1). We advocate that a hierarchy of managers that can dynamically distribute data and tasks can aid the task of application management for applications that spread a high number of tasks. Then, we first present an overview of the proposed hierarchy (Section F.2). Next, we introduce our application description language and the XML based format to store DAG and sub-DAGs information (Section F.3). Then all steps are analyzed (Section F.4) and services provided in GRAND are presented (Section F.5). Finally, we conclude this chapter presenting some final considerations (Section F.6).

## F.1 Premises

The main premises assumed to the conception of our model are the following:

**The execution environment is heterogeneous** A grid environment is heterogeneous by definition. We assume that grid nodes have machines that could have different software

and hardware configurations and this must be taken into consideration in our allocation decisions.

**A huge number of tasks can be submitted** This assumption is also one of our motivations and is fundamental in the definition of several details in our model. By a huge number of tasks we mean applications that will generate hundreds or thousands of processes.

Suppose the user submits in his/her home machine (i.e. a submit machine) an application with, for example, ten thousand tasks. If all the submission and the control were done in the same machine, probably this machine would stall and the user would not continue to work there. This problem is already known in the literature. For example, Condor [135] allows the user to specify a limit of jobs that can be submitted in a specific machine at the same time. This is not the best solution, since users must have some previous experience with job submission in this environment to infer the appropriate limit avoiding stall his/her machine and getting a good degree of concurrency. It is also a difficult task to be accomplished manually.

Besides, monitoring execution and handling errors manually is not feasible since such kinds of applications (a) use a great amount of resources, and (b) can take several days or weeks to successfully terminate. Thus, our model needs to be scalable. It also must provide execution feedback to users since execution will probably take several hours, days, or even weeks.

**Tasks do not communicate by message passing** Several works allow message passing in grid environments, such as MPICH-G2 [82] and Phoenix [81]. However, message passing introduces many aspects to be considered in the clustering and allocation phases. In the context of this work, we decided to assume that our applications do not communicate through message passing.

**Tasks can have dependencies with other tasks due to file sharing** Applications can be modeled as a dependency *graph* of tasks due to file sharing. For example, if a task $a$ produces an output file $f_a$ that task $b$ uses as its input file, then $b$ must wait until task $a$ finishes. In this example, $a$ and $b$ are nodes and there is an edge from $a$ to $b$, therefore $b$ can only be launched after $a$ finishes its execution. This is a common assumption as

presented in Condor's DAGMan [135] and Globus' Chimera [55], and is presented in many applications.

We consider that we always have a graph of tasks. A *bag-of-tasks* is the simplest way an application can be organized: there are no dependencies between tasks, so they can be executed in any order. Some Monte Carlo simulations can be classified in this group. The grid system OurGrid [6, 7, 106] is an example of a system that deals properly with this kind of application, and has proved its usefulness. Thus, the simplest graph has only nodes and no edge but more complex graphs must also be supported. Some examples of applications with dependent tasks were discussed in Section B.2. So, our scheduling decisions consider independent tasks but also task precedence.

**Huge number of files can be manipulated by tasks** Tasks can communicate and synchronize through files, so, each task usually will manipulate at least two files (one input and one output). Since we assume a huge number of tasks, a very high number of files must be managed. Efficient algorithms to keep data locality and to efficiently transfer files are crucial to the success of the model under this assumption.

**Huge files can be used in computation** Huge file transfers could cause network congestion, package loss, and can make transfer times unbearable. So, some kind of action must be taken to control file transfer avoiding package loss and network saturation. Preserving data locality, and staging and caching techniques could help minimizing performance losses due to data transfer latency.

**Underlying grid environment is secure** We assume that a secure connection is available between grid nodes. We also assume that user authentication and authorization are available in the grid environment. For example, the Globus Security Infrastructure (GSI) [149] can be used to satisfy our requirements, as well as the security issues provided by the EXEHDA environment [157].

**Underlying grid environment allows dynamic resource discovery** Discovery is a service that enables the system to retrieve resource descriptions. There are several resource discovery proposals for grid environment [64, 99, 117]. We consider that the grid environment has available one or more discovery services that can be used to maintain a resource

directory service in our model. We consider a directory as a service to maintain and access an information repository related to resources.

**Each grid node has its local resource manager** Local resource management systems (lower-level schedulers or RMSs) can have several attributes such as presented in [118] and [119]. We assume the following attributes will be available at each grid node:

- exclusive control: this attribute indicates if the local manager is in exclusive control of the resources. This information is useful to determine the reliability of the scheduling information;
- consideration of job dependencies: the lower-level scheduler takes dependencies between allocations into account if they are provided by the higher-level scheduling instance. For instance, in case of a complex job request the lower-level scheduling will not start an allocation if the completion of another allocation is required and still pending.

If a scheduler checks these attributes, it can take an appropriate action to ensure a good scheduling and a proper application execution. For instance, if a local resource manager does not support job dependencies, the high-level scheduler must ensure the correct submission order.

Another important aspect of this assumption, is that GRAND respects local scheduling policies as well as specific administrative domain choices. We consider that most academic cluster or local networks, that are already operating and can be integrated in a research grid, have already local users, local job submission, and system administrators. GRAND aims to be less intrusive as possible allowing different RMSs to exist in the same grid.

**Task submitted to a local resource manager will run until completion** We also assume that once a task is allocated it will not be scheduled again or at least this will be transparent to the higher level resource managers.

## F.2 Architectural Model Overview

We designed an architectural model to be implemented at a middleware level. Our proposal is a hierarchical management mechanism that can control the execution of a huge

number of distributed tasks preserving data locality while reducing the load of the submit machines. The main idea is to balance the submission task control load into several machines. Our architectural model relies on already available software components to solve issues such as allocation of tasks within a grid node, and authorization control.

We assume users submit applications to GRAND using our description language GRID-ADL (Grid Application Description Language), that is described in detail next (Section F.3.1). Each application has several tasks and can be represented as a Directed Acyclic Graph (DAG). In our DAGs, the nodes represent tasks and edges represent dependencies between tasks through data files access. Our system can infer automatically the DAG through the analysis of the data flow.

The weight of a node denotes the required amount of computation. The user can specify these values, to allow a better clustering. Otherwise, a same default value is set to all nodes, i.e., tasks are considered as homogeneous. An expert user could be able to make a good guess of task complexity, or at least to know if they are homogeneous or not, based in his/her previous experience. Another possibility is to use some automatic tool to infer the node weight. This tool can use complexity or granularity analysis (e.g. [34]).

Since we have a limited number of resources, nodes must be grouped to be executed in the same execution unit (cluster or local network). We refer to this grouping as an application clustering. We consider that a good clustering is the one that minimizes data transfer (i.e. maximizing data locality) and maximizes application performance. Note that clustering the DAG is a compromise between two conflicting forces: keeping nodes separate increases parallelism at the cost of communication, whereas clustering them causes serialization but saves communication [45]. We describe how applications are clustered, considering our application taxonomy (defined in Section B.2), in Section F.4.2. The user can also store the results of this initial phase, i.e. task description, DAG structure and clusters, as a set of XML files as described in Section F.3.2.

Then, the submission and execution control will be controlled by three components:

- ***Application Manager*** (*AM*): it is the high level controller that takes care of one specific application. GRAND instantiates a different *AM* for each application launched. It runs on the user machine, also called submit machine. This component is responsible for DAG inference, clustering, creation of submission managers and giving feedback to the user;

- ***Submission Manager*** (*SM*): is responsible for mapping sub-DAGs to grid nodes. GRAND creates one or several *SM* in the same local network where the *AM* is instantiated, each one in a different machine. It can run sub-DAGs from different applications;

- ***Task Manager*** (*TM*): is a wrapper capable of submitting tasks to a specific grid node. Once an *SM* has made the allocation decisions, it chooses the appropriate *TM* to submit task(s) to a grid node. Each *SM* communicates with one or more *TM*. Each *TM* can communicate with one grid node.

Thus, GRAND instantiates one *Application Manager* per application and can use one or several *Submission Managers* available in the local network. There is at most one *Submission Manager* per machine. There is an *AM* per application to decrease the complexity of this component and because there is no need for synchronization between applications. *SM* component can map clusters from different applications, as illustrated in Figure F.2.

Figure F.2 presents an example execution scenario. Note that in each machine with one *SM*, there is one or more *TMs*. A machine could have more than one *SM*, but the main idea behind fixing one *SM* per machine is to have only one subcomponent getting information about grid nodes per machine. Besides, one *SM* per machine simplifies the communication protocol with *AM*, *SMs*, and *TMs*.

Each *TM* can communicate with the RMS of a specific grid node to submit tasks. If an *SM* wants to allocate tasks in two different grid nodes, it will use two different *TMs*. Having different instances of *TM* for different kinds of RMSs allow to construct a lightweight component, since it only needs to know how to communicate with an specific RMS protocol. If two *SMs* want to communicate with the same RMS, they will use their local instance of the appropriate *TM*. This minimizes communications between control machines, i.e., machines of the local network that has an *SM* running.

Our task submission model can be classified as a high-level scheduler [118] since it queries other schedulers for possible allocations. It can also be considered as a Super-Scheduler, since it is a "process that will (1) discover available resources for a job, (2) select the appropriate system(s), and (3) submit the job. Each system would then have its own local scheduler to determine how its job queue is processed." [112]

Thus, in our design we control the application through a hierarchical organization of controllers that is transparent to the user. We believe a distributed hierarchical control

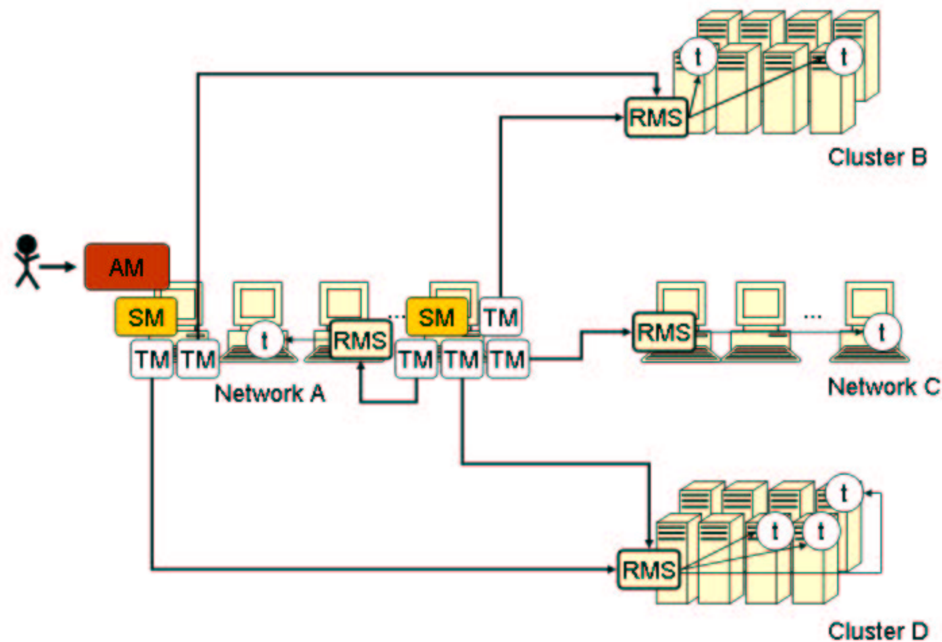



Figure F.2: The GRAND model: a possible scenario

organization meets the applications needs, is scalable and can alleviate the load of the submit machine avoiding stalling the user working machine. In the next subsections, we will present some details about these controllers.

### F.2.1 Model Components

The user submits his/her application through a *submit machine* that is a machine that has the *Application Manager* (*AM*) component installed, which is our higher level controller. The higher level of the application control must infer the DAG and make the clustering before starting submission process.

The *AM* is in charge of:

- processing the user submit file describing the tasks to be executed in order to infer the application DAG (*DAG inference step*);
- clustering the tasks into subgraphs (*Clustering step*);
- choosing *SMs* to send the clusters; and

- showing, in a user friendly way, the status and monitoring information about the application.

Note that the *AM* does not have information about individual resources available in the system. It only keeps track of the *SM* status to avoid communicating with a failure or overloaded node.

Subgraphs defined by the clustering algorithm are assigned to the second level controllers, called *Submission Managers* (*SMs*), which will instantiate the *Task Manager* (*TM*) processes to deal with the actual submission of the tasks to the nodes of the grid. This third level is necessary to isolate implementation details related to specific local resource managers.

The *SM* main functions are:

- to schedule clusters to grid nodes (*Mapping step*);
- to create daemons called *TM* to control actual task execution. Each daemon keeps control of a subgraph of tasks defined by the clustering;
- to keep information about computational resources; and
- to supply monitoring and status information useful to the user. It stores the information in a synthetic way. This information is sent to the *AM* that has the responsibility to present data to the user. This periodic information flow is also used to detect failures.

The *Task Manager* implements the *Submission step* and is responsible for

- communicating with a remote machine, at a grid node, which typically has a RMS;
- translating GRAND internal task description to a specific RMS submission format;
- launching tasks to remote nodes, typically through a local RMS request; and
- detecting faults with the grid node.

It works similarly to a wrapper being able to communicate with a specific local resource manager. For example, a *TM* is instantiated to communicate with a grid node that uses PBS while another *TM* can be instantiated to communicate with another grid node that

uses Condor. If none RMS is available, a direct way of instantiating remote tasks, in a specific machine at the grid node, can be used.

We assume that our hierarchy of managers is running in the local network to (1) avoid forcing that other sites run our daemons, and (2) to minimize communication time between managers.

### F.2.2 Model Components Interaction

Figure F.3 illustrates the three main components of our model and their relationship.

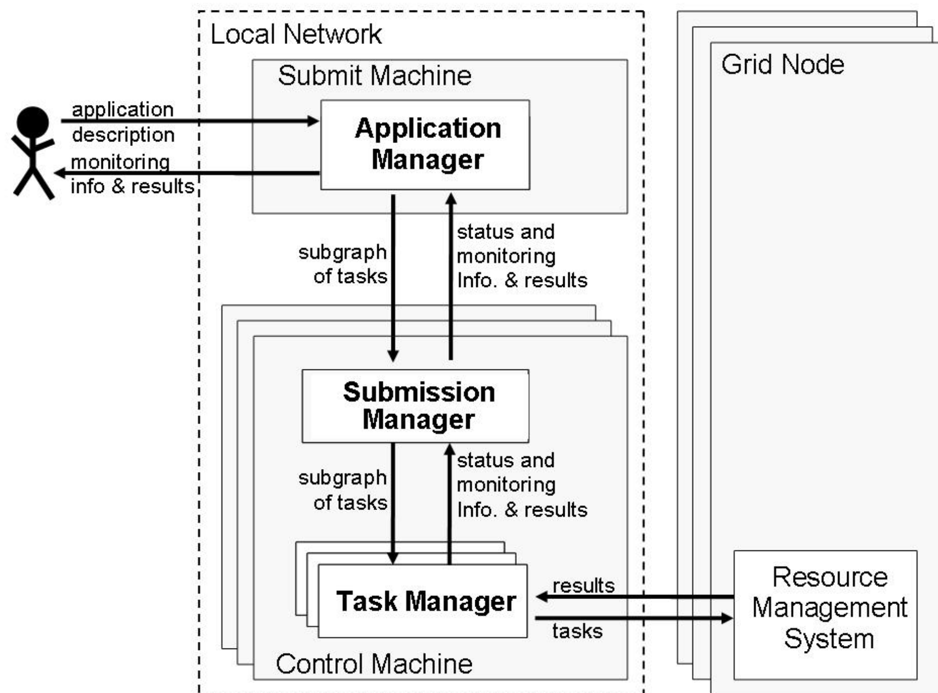



Figure F.3: The GRAND model: hierarchical architecture - main components

When the user submits his/her application in the submit machine, the *AM* is started due to the current request. When an *AM* becomes active, it broadcasts a message to its local network. All *SMs* reply to this message to inform their location and status. With these replies information, the *AM* builds a private database of *SMs*. Thus, an *SM* can deal with requests from several *AM*.

Figure F.4 presents some details about *AM*. First, the application description can be a GRID-ADL specification or an XML file. If it is a GRID-ADL file, three procedures must be executed: *GRID-ADL parser*, *DAG inferrer*, and *clustering*. If it is an XML file, it

already describes the DAG explicitly and the clusters. So, only an *XML parser* procedure must be executed. In both cases, the application description will be provided to the *Application manager executor*. The *Application manager executor* is the main component, responsible for communicating with the *SMs* and for sending information to the user. It keeps *SM* info in the *Active SMs* repository.

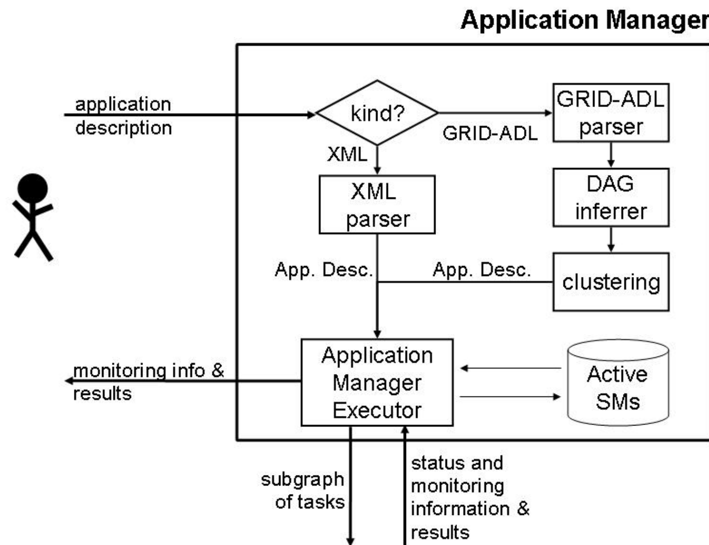



Figure F.4: The GRAND model: *Application Manager* details

The *Application Manager Executor* uses its local information and chooses one or more *SMs* to accomplish the required tasks. The choice is done based on the following criteria based on heuristics:

- the *SMs* that have recently communicated with the *Application Manager* and reported that are not overloaded have preference to receive subgraphs. The periodical communication can detect when a *Submission Manager* is faulty or overloaded and thus not able to help in the task submission and control;
- the computational power of the machine, considering CPU and memory, determines the upper bound on the number of subgraphs an *SM* can receive. The more memory and CPU power a machine has, the more subgraphs its *Submission Manager* can handle. This aims at avoiding to overload the machine;

- the *AM* keeps a weight for each *SM*. Greater values indicate powerful *SMs*. This value is based on previous execution data and indicates how well the *SM* accomplished the tasks it received.

If there is no available *SM* or all *SMs* are overloaded, *AM* tries to instantiate a new *SM*. A new *SM* can be instantiated when there is at least one available machine in the local network. If there are more than one machine capable of running a new *SM*, the more powerful machine with lower load is used.

The chosen *SMs* will receive subgraphs in an internal representation. At this moment, there is no transfer of executables or input data files.

Then, periodically the *SMs* will communicate to the *AM* the execution progress. This communication allows online monitoring information to the user and also fault detection. Notice that we are assuming that all *AM* and *SMs* will run in machines that belong to the local network to reduce latency and network traffic when submitting tasks.

Figure F.5 presents some details of this component. The *Mapping service* is the element responsible for *SM* orchestration. It receives clusters from *AM*, and can communicate with *TM* and other *SMs*, as well as contact the *Directory service*.

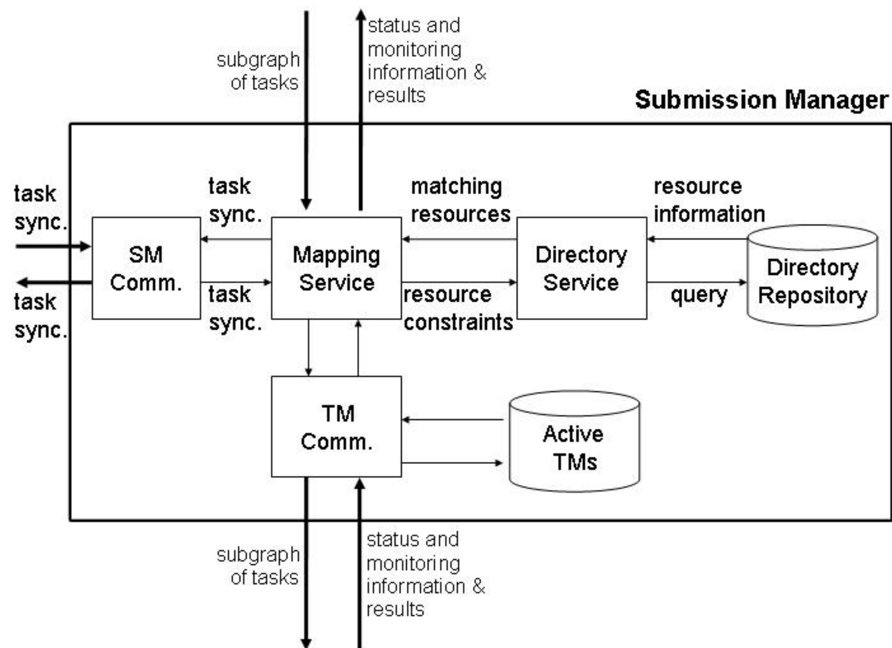



Figure F.5: The GRAND model: *Submission Manager* details

Communication between the *SMs* can happen, since some tasks in different subgraphs can have dependencies. Therefore some synchronization points must be established. The *AM* must send, included in the subgraph description, the identification of each manager that is related to this subgraph. For example, suppose a *Submission Manager* $sm_2$ has a task B which must be executed after task A assigned to *Submission Manager* $sm_1$. In this case, $sm_1$ must send a message to $sm_2$ when task A finishes.

Each *SM* must find the most suitable resources to run its subgraphs. A *SM* chooses a grid node using the following criteria:

- the *SMs* keep a list of available grid nodes. Some subgraphs will have requirements that just some grid nodes can match. Thus, grid nodes must match tasks requirements to be selected;

- for each grid node, an upper bound on the number of subgraphs it can receive will be estimated. Ongoing submissions are taken into consideration;

- the *SM* keeps information about previous executions. It uses this information to calculate a weight based on the application characteristics. Greater values for the weight indicate "better" grid node candidates.

For making this choice, the *SM* must have information about the available grid nodes. The *Directory service* element is responsible for getting information and storing it in the *Directory repository*. When the mapping algorithm is executed, the Directory service is consulted so as to obtain the matching grid nodes with some dynamic information that gives an estimation about the current grid node capacity and load. Note that the *SM* is the only component that has access to the grid discovery service and, therefore, access to grid resource information. It must get information such as the grid node address, the available RMS (if any), the physical resource characterization. It also gets dynamic information that gives information about the load of the grid node. The needed information and the way this information is manipulated by the *SM* is presented in Section F.4.3.

It is required a *Task Manager*, in the same machine of the *SM*, for each grid node an *SM* can access. *TMs* can be dynamically activated and deactivated according to the *SM* demands. The *SM* sends the subgraph to a *TM* according to the grid node chosen. In the *SM* side, it is the *TM Comm* element that is in charge of activating and keeping track of active *TMs* storing information in the *Active TMs* repository.

Figure F.6 presents the *Task Manager* details.

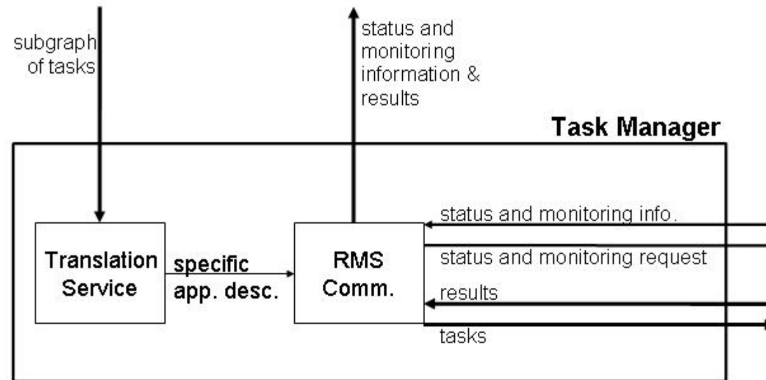



Figure F.6: The GRAND model: *Task Manager* details

The *Task Manager* is responsible for translating the internal subgraph description to the appropriate format for task submission. For example, a *Task Manager* that communicates with a Condor pool must prepare a Condor submit file and send the appropriate command to start tasks. This translation is performed by the *Translation service* element. The *RMS Comm* element is responsible for communicating with the RMS of a specific grid node, sending tasks and receiving results, as well as getting status and monitoring information.

## F.3 Application and DAG Representation

The steps of the GRAND model, represented in Figure F.1, require as initial input an application description. Also, the clustering and the mapping steps need, respectively, a DAG and a set of sub-DAG information. In this section, we explain how this information can be represented to submit to GRAND. A user can present an application through a GRID-ADL input file. GRID-ADL is a new language we proposed and Section F.3.1 presents it in detail. Also, a user or a program can provide a DAG and/or a set of sub-DAGs description, skipping at least the DAG inference step. Actually, the GRAND model

also allows to export this information. In both cases, it must follow an XML format. Section F.3.2 presents its DAG and sub-DAGs representation schema.

### F.3.1 Application Description and GRID-ADL

Generally, users run their jobs using some kind of description file that contains characteristics such as the tasks to be executed, the computational power required or the full path to the executable. As most users are acquainted with this kind of routine, we opted to maintain this classical approach, and provide to the user a simple description language that can quickly represent the user applications and needs.

Having an application represented as a graph in some description language is useful not only to allow the user to specify dependencies, but also to be able to distribute the tasks among the grid resources. The management system can use the graph to control the dispatch of tasks among grid resources. Therefore, our second motivation to have a powerful description language was also to be able to do application clustering. Application clustering, in our work, means to divide the application task graph in subgraphs such as they can be allocated to different available processors efficiently.

To allow clustering, the application must be represented and the dependencies expressed. Dependencies between tasks can be handled in three different ways:

1. Source code analysis: If the system has access to the user source files, it is not difficult to parse read and write accesses to files. If the file names are hard wired in the source code, it is sufficient to build a graph with all input/output dependencies.

   If the file is not hard wired in the code, the system can build graph node stubs, and fill the edges at execution time when the user submits the tasks, indicating the input and output filenames in a description file or in the command line.

2. Description language: Another alternative is to provide the user with a description language to explicitly express the DAG of tasks.

3. Runtime control: To launch all tasks at once and suspend whenever anyone accesses a file for reading that is not available in the local disk or shared file system. This last option does not need to handle an explicit graph, but makes the clustering process more complex.

In this work, we focus on the second approach: the user must describe the application in our high level description language and the system generates the DAG.

Besides, our language also allows for exploring compilation vectorization techniques [151] to represent and manipulate sets of tasks, as we will discuss later on Section F.4.2.

Our description language is called GRID-ADL (*Grid Application Description Language*). GRID-ADL has the legibility and simplicity of shell scripts and DAGMan [135] language while presents data relations that allow to infer automatically the DAG in the same way Chimera [55] does. The user submits a file describing only its tasks indicating input and output files. Optionally, he/she can also include the kind of application (independent, loosely-coupled, or tightly-coupled tasks) and application requirements.

Our language syntax is represented in Augmented Backus Naur Form (ABNF) [31] in Appendix I. Since it is self explanatory, we will not explain all details of our description language. We will only highlight the main aspects using the three DAG examples presented in Figure F.7.

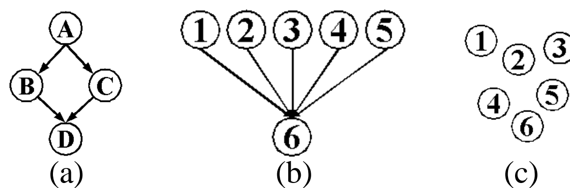



Figure F.7: DAG used in input file examples

Our language has some similarities with the DAGMan description language. Some main differences are the following:

- the user can give a hint on how the task graph can be classified (“independent", “loosely-coupled”, or “tightly-coupled”). This is useful to speed up the clustering phase, since there are different algorithms for each type of graph as we present in the next section;
- the task description presents, besides a name and a submission file, input and output file names;
- some shell script like constructions are available to facilitate the description of tasks.

The first two differences can be seen in Figure F.8. It presents an example where the user first indicates that his/her application has a loosely-coupled graph (first statement).

The *graph* keyword is a hint the user gives to our system. It is an optional directive that should be used to get a better performance when trying to infer the DAG. For instance, if the user defines that the DAG represents independent tasks, it is not necessary to run the algorithm to infer the DAG since there is no precedence order between tasks.

The next four subsequent lines describe four tasks using the statement *task*. For each one the user needs to define a name, a submission file, one or more input files, and one or more output files.

```
1 graph loosely-coupled
2 task A -e A.sub -i data.in -o a.out
3 task B -e B.sub -i a.out -o b.out
4 task C -e C.sub -i a.out -o c.out
5 task D -e D.sub -i b.out c.out -o data.out
```

Figure F.8: Example of input file for the simple DAG (F.7(a))

This example is simpler than the transformations and derivations used by the Chimera description language while being more powerful than the DAGMan specification language.

GRID-ADL also has some similarities to GEL scripting language [94]. The main difference is the DAG inference: GEL inference is based on control dependencies through classical parallel constructs such as *pfor* and declaration order, while GRID-ADL allows to detect data dependencies. GEL also provides some additional commands such as conditional iterator (*while*), that GRID-ADL does not support.

Besides these direct and simple statements, we added to GRID-ADL some shell script like constructions as illustrated in Figure F.9. In this example, some data is manipulated by gnuplot to generate a graphic. The final task combines all result graphics in a single postscript file.

A string variable assignment appears in the second and third statements. Then, the first task is declared. Next, an iteration command is used to declare four tasks as well as to store in the OUTPUT string variable the output file names. Then, the string variable is used to indicate the input file of task 6. The final command defines that the files stored in the string variable would not be copied back to the user workspace, i.e., they are transient or temporary files. Thus, the *transient* statement is useful to avoid copying files generated in the grid nodes that would be discarded because they do not belong to the final result.

This can minimize data transmission over the network and waste of data storage space in the user machine.

```
1  graph loosely-coupled
2  OUTPUT = "out1.png"
3  gnuplot= "/usr/local/bin/gnuplot"
4  task 1 -e ${gnuplot} -a "1.txt" -i in1.dat
5       -o out1.png
6  foreach TASK in 2..5 {
7     task ${TASK} -e ${gnuplot} -a ${TASK}".txt"
8          -i in${TASK}.dat -o out${TASK}.png
9     OUTPUT = ${OUTPUT} + ";out"+${TASK}+".png"
10 }
11 task 6 -e prepare_print -i ${OUTPUT} -o data.ps
12 transient ${OUTPUT}
```

Figure F.9: Input file for DAG example F.7(b)

The third example, in Figure F.10, illustrates with only three lines how to define an arbitrary number of tasks. In this case, we define an independent graph (a bag-of-tasks application) with six tasks. The application run six Monte Carlo simulations, each one receiving a different input file with a distinct seed. Note that in this example the task name is omitted, and the system will assign names as an integer number according to the creation order.

```
1  graph independent
2  foreach TASK in 1..6 {
3    task -e mcarlo -i ${TASK}.in -o ${TASK}.out
4  }
```

Figure F.10: Input file for DAG example F.7(c)

Our language presents some additional features that were not exemplified in our sample code: (a) GRID-ADL has a conditional statement *if*; and (b) GRID-ADL allows to express application and/or task requirements. The syntax of both features are presented in the ABNF of Appendix I.

### F.3.2 DAG and sub-DAGs Representation

GRAND also offers an XML output describing explicitly task characteristics, task dependencies, and clustering information. We believe XML is the appropriate language for programs exchange and data processing, but is not the most appropriate for human users.

We include such kind of output to simplify future interactions with other Grid RMSs that use XML for exchange data.

Being able to generate an XML output is a nice feature if the user wishes to execute the same application several times. Thus, the preprocessing phase is executed only once, saving time, and the stored XML files are used as input to the GRAND prototype, through the *AM*'s *XML parser* component. It is also a good choice to have XML files if different clustering algorithms are to be tested. Only the clustering information part must be changed. Our XML schemes are available from `http://www.cos.ufrj.br/~grand/2005/06/`. To illustrate the XML generation and its use, we present an example for the DAG of Figure F.7 (b) in Figures F.11, F.12, F.13, and F.14.

As JSDL (Job Submission Description Language) [20, 79] is being proposed as a standard language to describe grid applications, we base our task description on the JSDL's XML schema [80].

We extend the JSDL schema with a set of additional elements and attributes to describe DAG and clusters. We could have generated all information inside a single XML file. We opted for dividing it in several files because this organization is more flexible and also reduces complexity for understanding the XML files. Besides, it would be easier to get JSDL files generated by other programs.

We generate a zip file with four types of XML documents: a manifest, tasks, edges, and clusters. The manifest file indicates the remaining XML documents inside the zip file. This explicit indication of the files helps the automatic processing by XML tools. Figure F.11 is an example of manifest file. Note that in this example there is only one file of each type. For each file entry there is the specification of the file type. It is possible to have some missing files. For example, for independent tasks application there is no need of an edges file. Besides, it is possible to have more than one file of the same type. For instance, we can have two different clustering alternatives or more than one job file to compose an application. So, the file type helps to coordinate the processing order. Using the manifest, the processing tool can check if the application description is complete.

The first XML file describing the application is the task description (*task.xml*). It uses the JSDL schema from May 2005. Figure F.12 presents only a part of the required JSDL file. It describes only one of the tasks (jobs), since the other tasks are described in a similar way.

Task weight is an attribute commonly considered by clustering algorithms, but there

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <m:Manifest xmlns:m="http://www.cos.ufrj.br/~grand/2005/06/manifest"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="http://www.cos.ufrj.br/~grand/2005/06/manifest
5      /users/SO/kayser/jsdl/arquivos-artigo/manifest.xsd">
6          <m:FileEntry type="tasks" path="tasks.xml"/>
7          <m:FileEntry type="edges" path="edges.xml"/>
8          <m:FileEntry type="clusters" path="clusters.xml"/>
9  </m:Manifest>
```

Figure F.11: Example of XML manifest for Figure F.7(b)

is no such attribute in JSDL. Another common attribute is a unique job identifier. The *jsdl:ApplicationName* element is not ensured as unique in JSDL. There is a need for an unique identification because we need to refer to jobs when describing dependencies and clustering as well as for scheduling purposes. On the other hand, JSDL is an extensible language. For example, *jsdl-posix* is a JSDL extension which proposes additional attributes to describe posix-like job parameters.

Thus, we added an unique identifier (*grand:jobID*) and a task weight (*grand:jobWeight*) as can be observed in Figure F.12 at lines 17 and 18. We also need to add a reference to a schema defining both elements as we can see in lines 6, 12, and 13.

The file *edges.xml* describes the dependencies among tasks describing the DAG. For independent tasks applications, no file is generated since there is no edge between nodes. Figure F.13 describes the DAG for our example.

Finally, the file *clusters.xml* describes the clusters of the application, as for example we can observe in Figure F.14. Note that in these XML documents we refer to *JobName* instead of TaskName, since we adopted JSDL standard which refers to jobs.

## F.4 GRAND Steps to Execute a Distributed Application

This section presents the steps of the GRAND model, represented in Figure F.1: DAG inference (Section F.4.1), clustering (Section F.4.2), mapping (Section F.4.3), and submission (Section F.4.4).

### F.4.1 DAG Inference

As an initial approach, a quite simple algorithm can be used to infer a DAG with all tasks to be executed. The GRID-ADL description language is parsed and interpreted in such a way that for each *task* statement, a node in the DAG is created. Each node in the DAG stores

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <jsdl:JobDefinition
3    xmlns:jsdl-posix="http://schemas.ggf.org/jsdl/2005/04/jsdl-posix"
4    xmlns:jsdl="http://schemas.ggf.org/jsdl/2005/04/jsdl"
5    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6    xmlns:jsdl-grand="http://www.cos.ufrj.br/~grand/2005/06/jsdl-grand"
7    xsi:schemaLocation="http://schemas.ggf.org/jsdl/2005/04/jsdl
8    /users/SO/kayser/jsdl/arquivos-artigo/ggf/jsdl.xsd
9    http://schemas.ggf.org/jsdl/2005/04/jsdl-posix
10   /users/SO/kayser/jsdl/arquivos-artigo/ggf/jsdl-posix.xsd
11   http://www.cos.ufrj.br/~grand/2005/06/jsdl-grand
12   /users/SO/kayser/jsdl/arquivos-artigo/jsdl-grand.xsd">
13      <!-- task 1 -->
14      <jsdl:JobDescription>
15           <jsdl:JobIdentification>
16                <jsdl:JobName>gnuplot invocation</jsdl:JobName>
17                <jsdl-grand:JobUID>1</jsdl-grand:JobUID>
18                <jsdl-grand:JobWeight>10.0</jsdl-grand:JobWeight>
19           </jsdl:JobIdentification>
20           <jsdl:Application>
21                <jsdl:ApplicationName>gnuplot</jsdl:ApplicationName>
22                <jsdl-posix:POSIXApplication>
23                     <jsdl-posix:Executable>
24                           /usr/local/bin/gnuplot
25                     </jsdl-posix:Executable>
26                     <jsdl-posix:Argument>1.txt</jsdl-posix:Argument>
27                     <jsdl-posix:Input>in1.dat</jsdl-posix:Input>
28                     <jsdl-posix:Output>out1.png</jsdl-posix:Output>
29                </jsdl-posix:POSIXApplication>
30           </jsdl:Application>
31      </jsdl:JobDescription>
32 ...
33 </jsdl:JobDefinition>
```

Figure F.12: Fragment of extended JSDL file for Figure F.7(b)

the main information about the task: name, command, and input and output files. Since name is optional, if user has not specified a task name, an unique identifier is automatically assigned at node creation. Once this structure is filled, for each task, it checks if any of its input files is output of another task. When such input-output pair is found, and edge between tasks is inserted. This is not an efficient algorithm, since it is $O(n^2)$, where $n$ is the number of tasks (DAG nodes).

A more desirable approach would minimize memory consumption and have a better complexity for dependency inference. In GRAND, each node does not need to be explicitly defined, because tasks of an application can be described through an iterator command, provided by GRID-ADL. This allows for a compact representation of the application graph. It also allows for exploring compilation vectorization techniques [151] to represent and manipulate sets of tasks. Thus, instead of storing the descriptions for each task, we can store the description of how to obtain such tasks.

A proposal is to use a simple approach: the DAG inference is done in “two steps”:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <ge:DAGDefinition xmlns:ge="http://www.cos.ufrj.br/~grand/2005/06/grand-edges"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://www.cos.ufrj.br/~grand/2005/06/grand-edges
5   /users/SO/kayser/jsdl/arquivos-artigo/grand-edges.xsd">
6          <ge:Edge weight="1.0" sourceJobID="1" targetJobID="6"/>
7          <ge:Edge weight="1.0" sourceJobID="2" targetJobID="6"/>
8          <ge:Edge weight="1.0" sourceJobID="3" targetJobID="6"/>
9          <ge:Edge weight="1.0" sourceJobID="4" targetJobID="6"/>
10         <ge:Edge weight="1.0" sourceJobID="5" targetJobID="6"/>
11 </ge:DAGDefinition>
```

Figure F.13: DAG description for Figure F.7(b)

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <gc:ClusterSet xmlns:gc="http://www.cos.ufrj.br/~grand/2005/06/grand-clusters"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://www.cos.ufrj.br/~grand/2005/06/grand-clusters
5   /users/SO/kayser/jsdl/arquivos-artigo/grand-clusters.xsd">
6          <gc:ClusterDefinition clusterID="c0">
7                  <gc:JobID> 1 </gc:JobID>
8                  <gc:JobID> 2 </gc:JobID>
9                  <gc:JobID> 3 </gc:JobID>
10         </gc:ClusterDefinition>
11         <gc:ClusterDefinition clusterID="c1">
12                 <gc:JobID> 4 </gc:JobID>
13                 <gc:JobID> 5 </gc:JobID>
14                 <gc:JobID> 6 </gc:JobID>
15         </gc:ClusterDefinition>
16 </gc:ClusterSet>
```

Figure F.14: A possible example of clustering for Figure F.7(b)

(1) working with the loop command, then (2) expanding the loop command. This "expanding" or "execution" corresponds to expand task description to a data structure stored in memory to describe each task. Since it could require to consume more memory than available in the submit machine, it could cause swap and thus overload and/or slowdown the machine. Thus, the expand operation could be postponed until necessary. The first step do a dependency analysis based on language syntax, and breaks the tasks into clusters (or set of tasks) storing it as a loop command that must be expanded during mapping step.

For instance, a loop that creates one thousand tasks, represented by a *foreach* statement with index from 1 to 1000, can be split into 10 cluster: from 1 to 100, from 101 to 200 and so on. Thus, GRAND need to store for each of this clusters only the *foreach*'s statements and the index range. This is less memory consuming than storing the description of all these tasks.

A set of tasks (cluster or subDAG) can be expanded if some of its dependencies are satisfied and there is an *SM* available to manage a cluster, i.e. the set of tasks are only expanded when it will be manage by some *SM*.

### F.4.2 Clustering

The clustering step aims at obtaining a set of clusters of tasks. These clusters can be submitted almost independently. Since we are concerned with the submission of a huge number of tasks in a highly dynamic environment, the actual scheduling (mapping) step must be done dynamically. The clustering step should be done as fast as possible so as not to delay submission and to minimize overhead. Clustering should not consider a resource graph to produce clusters, because (1) only *SMs* need to get information about grid resources, and (2) since we target long term running applications, grid node may change during application execution, at least concerning CPU load, and also grid nodes can be included or excluded from the grid. Since the application probably will take a long time running, there is no advantage in getting an optimal mapping for a set of resources that will change during the execution. A simple and quick algorithm would be preferable. Thus, we consider an unbound number of homogeneous resources. Besides, the component that is responsible for clustering does not need to have access to the directory service. Note that the hierarchical submission is not enough to avoid overloading machines. The granularity of the cluster must be set in order to avoid *SM* overloading.

So, we consider application clustering without associating it to resources. Our objective with this step is to obtain groups of tasks to be submitted together to maintain data locality. For this step, we propose the use of different clustering algorithms according to the application taxonomy presented previously in Section B.2.

Therefore in the case of independent tasks, clustering becomes simple, because tasks are not dependent on each other. We need to cluster in blocks considering the computational power required for each task, but there is no restriction of which task will belong to each block regarding to dependencies and data locality issues. Figure F.15 presents the outline of this algorithm.

For this algorithm, as for the next two, we must define the size or granularity of the cluster. The right size is a compromise of being set small enough to avoid *SM* overloading, and large enough to get better performance due to efficient submission. We consider that a small granularity must be set, and if during execution this becomes too small, two or more clusters can be merged. Determining an actual number of tasks or complexity to define the granularity is implementation dependent.

To determine the granularity of a cluster set, we must know how many tasks an *SM* can

**Data**: graph of tasks
**Result**: set of clusters

```
1  begin
2      initialize set of clusters as empty;
3      initialize current cluster as empty;
4      while not at end of the graph do
5          get next task;
6          if current cluster reached granularity limit then
7              add current cluster to set of clusters;
8              initialize a new current cluster as empty;
9          end
10         add next task to current cluster;
11     end
12     add current cluster to set of clusters;
13     return set of clusters
14 end
```

Figure F.15: Clustering algorithm for independent tasks application

deal without overloading the control machine. We argue that this is an implementation and architecture dependent issue. We must consider the cost of individual task submission and control machine power. The individual task complexity is *a priori* irrelevant, since it is expected the same computational cost for submitting small or big tasks. The cost of task submission is dependent on the implementation data structures and the number of threads and/or processes used. To determine the control machine power requires that we have dynamic information about the available machines at the local network.

Thus, we propose the following analytical model to calculate, dynamically and according to the GRAND implementation, the maximum cluster granularity an *SM* can handle at a given moment (*MCG* or maximum cluster granularity).

First, the maximum number of tasks an *SM* can handle (*SM_MAX_TASKS*) must be calculated:

$$\text{CPU_LIMIT} = (\text{SM_CPU} \times \text{CPU_MAX_ALLOWED})/\text{CPU_PER_TASK}$$
$$\text{MEM_LIMIT} = (\text{SM_MEM} \times \text{MEM_MAX_ALLOWED})/\text{MEM_PER_TASK}$$
$$\text{SM_MAX_TASKS} = \min(\text{CPU_LIMIT}, \text{MEM_LIMIT})$$

Where, *SM_CPU* is CPU power in MFlops in the *SM* machine; *SM_MEM* is memory in MBytes in the *SM* machine; *CPU_PER_TASK* is the amount of MFlops consumed by the

*SM* implementation to manage one task; *MEM_PER_TASK* is the amount of MBytes consumed by the *SM* implementation to manage one task. The *CPU_MAX_ALLOWED* and *MEM_MAX_ALLOWED* variables indicates, respectively, the percentage of CPU power and memory that can be used by each *SM*, which is defined by the user or grid node administrator.

Then, considering that the GRAND system knows the current number of tasks an *SM* is managing at a given moment (*SM_CURRENT_TASKS*), it is possible to calculate the *MCG*:

$$\text{MCG} = \text{SM_MAX_TASKS} - \text{SM_CURRENT_TASKS}$$

For loosely-coupled graphs, where tasks can have a low degree of dependency, we considered that the application graph normally fits in some known patterns, as for instance "pipelines" or "phases". Figure F.16 presents an algorithm to cluster an phases graph. It uses the bread-first search (BFS), which is a classical graph transversal algorithm [92]. While visiting the nodes a list is constructed. This list is used to construct the clusters using the same grouping criteria of independent task algorithm.

```
   Data: graph of tasks
   Result: set of clusters
 1 begin
 2     initialize set of clusters as empty;
 3     initialize current cluster as empty;
 4     traverse the graph using a breadth-first search, and for each new visited vertex,
       put it into the vertex list;
 5     while not at end of the vertex list do
 6         get next task;
 7         if current cluster reached granularity limit then
 8             add current cluster to set of clusters;
 9             initialize a new current cluster as empty;
10         end
11         add next task to current cluster;
12     end
13     add current cluster to set of clusters;
14     return set of clusters
15 end
```

Figure F.16: Clustering algorithm for phases-loosely tasks application

For tightly-coupled graphs, we consider that classical graph clustering can be used. We analyzed several algorithms and we considered that the DSC algorithm [158], already outlined in Section C.3, is a good choice. DSC has low complexity and performs better than list scheduling algorithms because it is based on a more global view of the state of the DAG [158].

### F.4.3 Mapping

Once the application is conveniently clustered, we need to solve a second problem that is how to map the clusters to the available grid nodes. As explained in Section F.1, we assume that each grid node will have a local RMS to perform the scheduling of tasks that belong to a cluster. The mapping step is presented in this section.

The assumption of a huge number of tasks has important consequences to the scheduling architecture design. We cannot just submit tasks without controlling system parameters and flow control as some systems do. For example, MyGrid [106] considers that making a fast scheduling decision is more important. It is true for many applications, but for our target applications it is necessary to keep track of system information. For example, if the application takes several days to finish, one second to find a suitable cluster is not a problem.

The *Submission Managers* are the components responsible for performing the mapping step. In this step, an *SM* must send a query to the *Directory Service* to retrieve the matching available resources.

The directory service can be implemented in two ways: (1) as a centralized service in the local network; or (2) as a decentralized service. The centralized approach is easier to implement. However, the server becomes a potential performance bottleneck and a single point of failure. Thus the alternative is to implement a set of servers to provide the directory service. To facilitate their implementation and the information consistency between them, we propose the use of a peer organization.

Thus, the *SM* will access its local *Directory Service*, giving application's constraint information, such as operating system platform required to run the application, or minimum requirements such as computational power, memory, and disk space.

Then, when the *SM* receives the list of available grid nodes that match the requirements from its *Directory Service*, it calculates four properties for each grid node:

- **distance**: it is a integer number that indicates the relative distance between submit machine network and the grid node. Since latency should be minimized to get a quicker data transfer, closer grid nodes are preferred, since normally the latency will be lower. We assume three possible values: 0 for the submit node; 1 for regional connections; and 2 for international connections.

- **capacity**: it indicates the link capacity. Since bandwidth should be maximized to get a quicker data transfer, higher link capacities are preferred. *Capacity* must assume a number indicating the capacity for each grid node using the same unity. For instance, if there are two links, one of 1Gb and another with 100Mb, their *capacity* value must be assigned to 1024 and 100 respectively;

- **cpu**: indicates the computing power of a grid node. It is the value of the following formula:

$$\text{cpu} = (\sum_{k=1}^{n} \text{number_of_cpus}_i * \text{individual_power}_i)/\text{node_load}$$

  Where, number_of_cpus is the number of machines or cpus available at the grid node, and individual_power is the power represented in a standard measure: MFlops (mega floating point operations per second) or MIPS (million instructions per second). Note that we allow heterogeneous grid nodes. The node_load value is a percentage that indicates how busy is the grid node, and is used to not stimulate *SM* to choose powerful grid nodes that are overloaded.

- **memory**: integer value to indicate average grid node memory capacity. It is measured in MBytes. For instance, if a grid node is a homogeneous cluster with all machines with 1GB of RAM, the memory will be set to 1024.

Then, each grid node receives a value called *preference*, which is calculated according to the following equation:

$$\text{preference} = [(\text{distance} * 0.25) + (\text{capacity} * 0.25)] + [(\text{cpu} * 0.25) + (\text{memory} * 0.25)]$$

So, the *SM* sorts the available grid nodes list according to the following criteria: bigger *preference* comes first.

Note that all elements contribute with the same weight to the preference value. This was chosen, because there is no indication in the literature that one of these aspects would be prioritized for general cases. For data intensive applications, the first part of the equation should receive a bigger weight, since moving data can be the dominant part of the execution. For compute intensive the second part of the equation should be priories, since faster processors would lead to better performance. So, to keep it general, we did not give priority to any factor. However, optionally, the user can modify the weight of these factors according to his/her specific application. For instance, if the user knows its application is a highly memory consuming application, he/she can put more priority to *memory*.

### F.4.4 Submission Control

Each *TM* is attached to a specific grid node. A *TM* has the algorithm to map a task description to the submission format of a specific RMS. This mapping can be done directly to a RMS or using a standard interface. Using RMS through command line interface is a general way that work for any RMS available. However, since site policies differ widely across grid node administrators, using an interface that encapsulate site-specific details is desirable [38]. The GGF's DRMAA (Distributed Resource Management Application API) [38, 110] proposes an application programming interface for communicating with RMS (also called distributed resource management systems). It has bindings for different languages such as C, Java, and Perl that simplifies a DRMAA conformance implementation.

DRMAA proposes a common RMS API, that allows to submit, monitoring, and control of tasks as well as retrieval of the finished job status. DRMAA provide access methods such as *drmaa_init* and *drmaa_exit* for starting and stopping a DRMAA session with a specific RMS. It also provides a task controlling routine, called *drmaa_control*, which allows to start, stop, restart, or kill a task. These three methods are the base for *TM* accessing RMS.

Nowadays, there is support for DRMAA API for Condor [29] (C bindings) and SGE [68] (C and Java bindings). Support for other RMS are expected to be developed. Implementing GRAND support for other RMSs, that does not directly support for DRMAA, is simplified using this standard API. The *TM* can prepare task submission according to DRMAA methods, but instead of using a vendor provided API, it converts to a command line inter-

face. This command line, which is normally dependent of grid node configurations, can be replaced as soon as a DRMAA implementation became available.

## F.5 Additional GRAND Services

Data management for computing intensive applications is an important issue but it has received little attention from grid-aware software developers. Such applications usually consume and produce small files (e.g. few KBytes) that, in principle, should be easy to manage and fit in any storage. However, computing intensive applications that spread thousands of tasks across the grid nodes may consume and generate thousands of files, that can deteriorate system's performance, if they need to be moved across machines. If the tasks present dependencies due to file sharing, another problem that is raised is data locality. We present some solutions to these problems in Section F.5.1.

Another GRAND service is application monitoring, which is presented in Section F.5.2.

### F.5.1 Data Management

Data management is a broad research area that encompasses several issues and has several works reported in the literature. The grid infrastructure imposes some new challenges related to scalability, locality, and fault tolerance. We deal with some data management issues in the context of the GRAND middleware: data locality, automatic staging of data, and optimization of file transfers.

During submission step, both *SM* and *TM* must cooperate to perform an efficient data management, as we outline in this section.

The GRAND model is mainly concerned with applications that use data stored as plain files. All data manipulated by the application is available as files, which can be available in arbitrary locations. In its current version, the GRAND model is not concerned with database access, although we believe it is an important issue. It also does not support message passing applications.

Files can be already available from some repository or can be generated during application processing. In the first case, some of the files are *input* files, considered as immutable (read-only) files that contain data used by the tasks. Other files are *executable* files, which must be available and installed in the grid node or must be downloaded by GRAND to be executed by a task. Since both kinds of files, input and executable, are considered as

immutable, they will be staged in the grid node that executes the set of tasks that needs them. Therefore these files do not need to be copied back to the repository or any other disk area. The assumption of immutable files can be restrictive but is appropriate for applications that use data from shared repositories, as for instance some HEP collaborations that make available files that are read-only after creation [127].

In the second case, files are generated during application processing. Some of them are *transient* and considered temporary files used by dependent tasks. Transient files do not need to be staged out, and therefore can be discarded at the end of the application execution. They can not be discarded immediately after the dependent task uses them, because other future tasks that may happen to run on the same grid node may need them. Other files generated by tasks are the *result* files, which contain relevant data for the user and must be staged out to some repository. The user must indicate which files are transient otherwise GRAND assumes they are result files and stages out automatically.

From the several data management issues that an application management system should deal with, we focus on (1) data locality, (2) automatic staging of data, and (3) optimization of file transfer.

**Data Locality** An application management system should ensure data locality as much as possible. In the context of this paper, data locality means dispatching to the same grid node tasks that access the same input files, as well as producer and consumer(s) tasks of a given file. Trying to keep data locality aims at reducing file transfers over the network. This is desirable to avoid (1) slowing down the application due to transfer costs, and (2) waste of resources (mainly network bandwidth).

In the GRAND model, data locality is a priority during the *clustering* step, when an application composed by several dependent tasks is divided into subgraphs. Clustering is performed without being associated to resources. The actual scheduling (mapping) step must be done dynamically and on demand due to the dynamic nature of grid systems and to the long term running nature of the applications. Our objective with the clustering step is to obtain groups of tasks to be submitted together to maintain data locality. For this step, we propose the use of different clustering algorithms according to the kind of application [143]. Data locality is an issue for clustering *loosely-coupled* and *tightly-coupled* tasks, since *independent* tasks have no file dependencies. A loosely-coupled graph

is characterized by a few sharing points and a regular shape like phases or pipeline, while tightly-coupled has a complex graph.

**Automatic Staging of Files** In its current implementation, GRAND allows a limited form of automatic staging of input files. The user must place the input files on an accessible repository and from this repository, the input files are transferred automatically to the executing grid node, or the files must be accessible through NFS. There is no support for logical file names or access through a file catalog or service.

If tasks that belong to applications of different users or tasks that belong to the same application share the same input files and are selected to run on the same grid node, then the input files are copied only once to that grid node to avoid increasing the number of file transfers unnecessarily. A cache scheme is implemented by each *TM* to support this feature. Once the application is submitted, and the subgraphs are dispatched, the task manager writes a record of which files are being transferred. We used a hash to locate files in the cache. If later, other tasks need to access those files, no transfer needs to be performed.

A second case where automatic staging is performed is on application termination. Immediately after all tasks of an application finish execution, all result files are transferred back to the user's working space. Notice that if all tasks terminate execution at the same time, all files will be transferred at the same time, which may cause network traffic congestion, if the number of tasks is very large.

Experiments were performed to evaluate these strategies and are presented in the next section.

**Optimization of File Transfer** The GRAND hierarchy helps to automatically promote parallel transfer of results, and to some extent prevents network traffic congestion. However, other approaches such as network control flow algorithms should be used [58].

Our current implementation of GRAND does not prevent this situation, but we have been working on a network control flow algorithm to handle this problem. At the moment, we implemented a simplified, while effective, solution to prevent file losses. While staging in or out a file, if any exception occurs, a fault tolerance mechanism will be started. When trying to stage a file, if there is an exception, for instance a connect exception, our prototype assumes it is a temporary failure and will wait a random amount of time (some

milliseconds) before trying to download it again. This procedure will be repeated a limited number of times (by default five times). This random waiting is similar to the CSMA/CD protocol [77].

The Chimera system [55] has some similarities with our work: it deals with DAGs of tasks and can infer the DAG automatically. However, it handles the information of how data is generated by the computation but it does not control the DAG execution. It has a planner that maps the stored abstract graph to a DAGMan description file, and the execution is done in a Condor pool under the control of DAGMan [33]. Thus, files are managed only at the meta data level. DAGMan controls the submission of tasks to Condor RMS [30, 136]. Condor automatically stages in and out files using ftp. However, files must be available in the submit machine, locally or through a shared file system. Besides, there is no optimization in these operations, such as a data cache, or data locality, reported in the literature.

More recent works such as the software used by LCG (Large Hadron Collider Computational Grid) [89] supports some features needed for application management on top of Globus [52], Condor [136], and gLite [60]. However, the packages used by LCG do not support distributed submission and data caching as provided by our system. Moreover, data staging must be performed explicitly by the user.

### F.5.2 Monitoring

Two informational GGF documents were used to guide the monitoring terminology and decisions presented in this section: (1) the outline of the Grid Monitoring Architecture (GMA) described in [139], and (2) the main events available in monitoring tools listed in [73].

As a design decision, GRAND always relies in available software components when it means avoid interfering with local administrators decisions and avoid to oblige installing a specific software. For monitoring resources, we believe there are several good options that are already being used in practice (e.g. [98, 104]). Thus, the GRAND model provides a monitoring service related to application execution. It is not intended to make available full details regarding resources available at grid nodes. All resource information will be provided through querying a directory service provided by local grid node administrators.

Before presenting the proposed monitoring service, we list which events from existing resource monitoring services GRAND uses for allocation decisions.

**Using available resource monitoring tools** Most grid node administrators will provide restrict access to resources. Thus, our monitoring tool uses available resource monitoring tools available at grid nodes, instead of consulting each resource directly.

*Submission Managers* subscribe as consumers of directory services to receive event publication information produced by grid nodes. From the usual information provided by monitoring tools [73], GRAND queries for the following:

- Host architecture: name of host architecture;
- Host OS: name and version of host operating system;
- Physical memory: total memory on a host;
- CPU load: fraction of the CPU that is no idle;
- Available memory: memory not allocated.

The first three event types are used to apply user constraints when selecting possible nodes to execute. The CPU load and available memory event types are used to choose the less overload option, as explained previously (Section F.4.3).

**Providing an application monitoring service** GRAND provides a monitoring service through a directory service called GRID-AMDS (Grid Application Monitoring Directory Service). Following the GMA specification, GRID-AMDS must provide four functions: add, update, remove and search.

The *add* function adds an entry to the directory. The entry consists of an application description with the following items:

- **ID**: application unique identifier, generated using submit machine IP number, submission date, and submission time information;
- **user**: the user name of the user that started this application;
- **inputFile** GRID-ADL description file name;
- **total**: total number of tasks;
- **finished**: number of already executed tasks;

- **tasks**: list describing information of each individual task. Each element describes the following attributes:
  - **taskID**;
  - **predecessors**: taskID of all predecessor tasks;
  - **status**: indicates if the tasks is (1) *waiting* for some task to finish; (2) *ready* to execute, but waiting for resources); (3) *executing* in some grid node; or (4) *finished* its execution;
  - **node**: for status waiting and ready it is empty, otherwise, it indicates in which grid node it is executing or had executed. executed.

The *update* function updates some set of *tasks*' elements to indicate *status* changes and update *node* information.

The *remove* function removes the entire application entry. It can be called when application has finished and the user is no longer interested in monitoring information.

Finally, the *search* function can be invoked by a consumer to return information about the application status. The search can be used to return the ID, given user name and/or submit machine IP and/or GRID-ADL description file name. Having an application ID, the user can search task information: search can retrieve the entire list of tasks, or a partial list according to the status desired (waiting, ready, executing, and/or finished).

## F.6 Conclusion

This chapter presented the main aspects of the GRAND application management model.

To evaluate the main ideas of the GRAND model, we implemented a prototype. In the next chapter, we present some implementation issues as well as we analyze some experimental results.

# Apêndice G

# AppMan: Application Submission and Management Prototype

In this chapter we present an overview of the GRAND implementation. We have implemented a prototype of the GRAND model called AppMan (**App**lication **Man**ager) [142, 144]. AppMan implements the basic main features of GRAND, including distributed task submission and application monitoring, while giving feedback to the user. Initial results show that our approach can be more effective than other approaches in the literature.

The remaining of this chapter is organized as follows. First, in Section G.1 we outline the prototype implementation that implements some of the GRAND functionalities. In Section G.2, we present and analyze results of our prototype. Finally, in Section G.3, we conclude this chapter with our final remarks and future work.

## G.1 Implementation Details

AppMan is implemented in Java to allow portability. It uses the JavaCC [78] tool to implement the GRID-ADL parsing and interpretation. Our parser puts the complete application DAG into a matrix and use this matrix to infer dependencies between tasks. The AppMan runtime environment uses the services provided by EXEHDA [155] middleware that allows remote execution and monitoring. EXEHDA, which stands for "Execution Environment for Highly Distributed Applications", has facilities to instantiate remote objects and to coordinate their operation, as well as an integrated monitoring architecture.

Figure G.1 shows a possible scenario of AppMan running over a grid environment. One instance of AppMan and EXEHDA needs to be present in every machine of the grid. Any machine can submit or run tasks. However, inside each grid node, at most one ma-

chine will run EXEHDA daemon as a base. Users can provide input files in a web server, as illustrated in the example scenario, or in its local file system.

Step 1 in Figure G.1 represents the user submitting a description file in GRID-ADL. The GRID-ADL description file is parsed, and the application graph is built in memory. A clustering algorithm is executed, and the sub-DAGs of the application graph are obtained. The *Application Manager* (AM) is started. Then, the *Application Manager* instantiates *Submission Managers* (SM in step 2), and distributes some subgraphs to the SMs. Input files and the executable are fetched from a web server (step 3). In its current version, AppMan requires that the user indicates the machines where the *Submission Managers* (*SM*) will run. With this simplification, a new *SM* is instantiated for each application in each machine specified. After creating the SMs, the *Application Manager* (*AM*) assigns subgraphs to each SM. The SMs report to the AM the tasks progress. Each *SM*, independently, checks the list of available machines and chooses where to run using a round-robin approach. Round-robin [130] is one of the simplest scheduling algorithms, which is starvation free and threats all schedulable elements as having the same priority.

Then, a *Task Manager* (*TM*) is instantiated, which creates the remote task and monitors it until it successfully completes.

Before starting the execution of a task, AppMan transfers all input files specified in the GRID-ADL description to a temporary directory in the remote machine where the task will be executed, a kind of sand-box. The file transfer is performed automatically. Some optimization takes place during staging as we discuss later (Section G.1.2).

Then, each *SM* gets updated information, through the EXEHDA information service, about available nodes. Each *SM* chooses where to run its tasks using a round-robin approach and instantiates *TMs* to allocate the remote tasks (step 4).

AppMan implements some fault tolerance features. If input files cannot be transferred, it waits and then tries again, assuming that a temporary communication failure occurred. If a task finishes but the expected output file is not generated, the task is resubmitted. In both cases, this process is repeated up to a determined number of times. If the limit is reached, a permanent fault if assumed, and the application is aborted. The user is informed of a possible task or application bug or problems with grid resources.

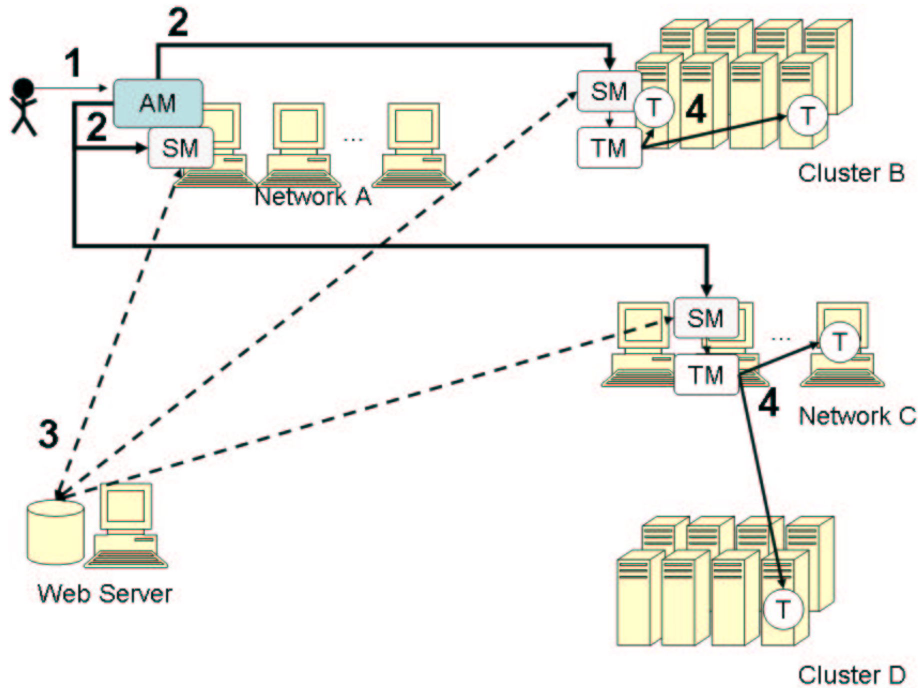



Figure G.1: AppMan executing main steps

### G.1.1 Monitoring Graphical Interface

An online monitoring graphical interface provides visual execution feedback to users. It runs independent from execution, reading the log files to get some information and communicating directly with the *AM* to refresh progress information. *AM* implements a simplified version of the GRID-AMDS (Grid Application Monitoring Directory Service) presented in Section F.5.2. Instead of a GRID-AMDS that stores information about all applications running in the local network, each *AM* provides the directory service concerning the application it is monitoring.

By using this graphical tool, the user can have an idea of the application execution progress. As the user will probably be running experiments for days or weeks, this becomes a very important and essential tool.

The interface shows the task graph (tasks, dependencies, and clustering) and a progress bar. Each node has four possible colors: (a) red: it depends on data not yet available; (b) yellow: task ready to execute, waiting for a free resource; (c) green: represents a running task; (d) blue: a completed task.

Figure G.2 shows two snapshots of two small applications to illustrate the color feedback and the bar progress indicator.

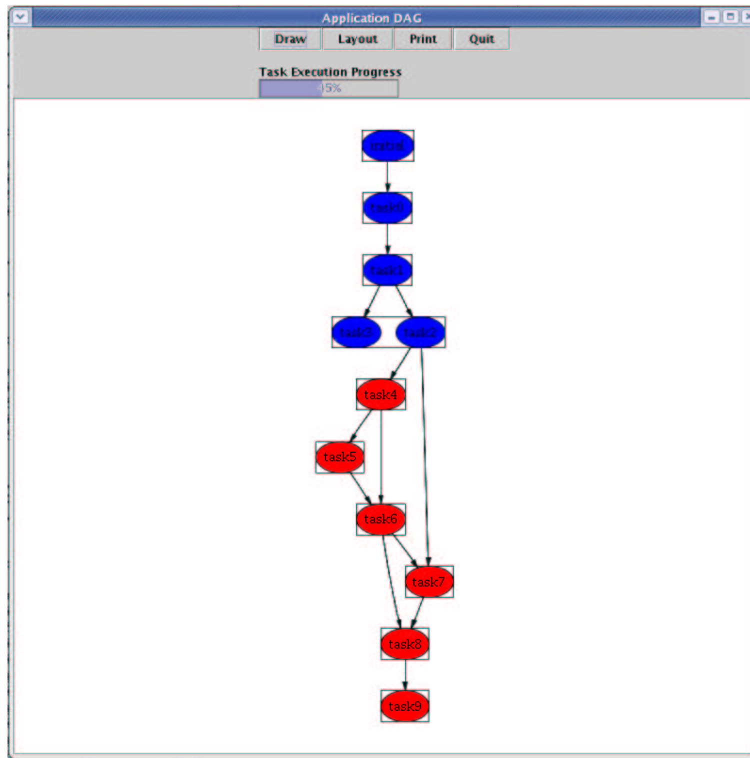
### G.1.2 Data Management

In our current implementation of the GRAND model, all files need to be explicitly defined by the user using GRID-ADL, with exception of the executable file that can be already available and installed at the grid nodes. Besides, input files need to be available through a shared file system or in a public repository. However this is only to simplify the implementation, because files can be automatically located in the grid data space if one uses a grid-aware distributed file system [66, 150], and/or the user compiles the application with a special compiler that translates file operations at source level to a code that handles remote file accesses [136]. That way, instead of yielding an error message, a task may suspend execution if it tries to access an input file that is not locally available.

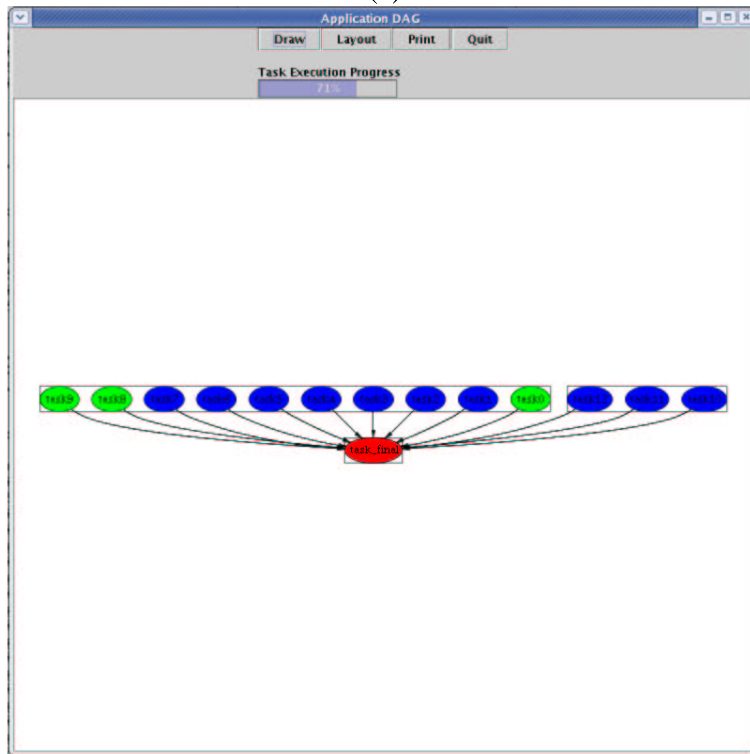
In fact, a mechanism like that is already used for transient files, when handling dependent tasks. GRAND adopts this approach to allow user applications to execute without any modification or intervention.

For the experiments reported at this work, we are using a *Submission Manager* (*SM*) that can select individual machines, working as a simplified RMS. When a *SM* receives a sub-DAG (cluster), it must select a machine available in the same grid node to execute the task. The *TM* component instantiates the task process directly into a machine inside the grid node. We implemented a simple scheduling policy, since GRAND does not aim to be an RMS. This *SM* implementation in the AppMan prototype can be used as an alternative RMS in a local network that does not have any one installed. Initial evaluation showed good results for small tasks compared to traditional RMSs [142].

The *SM* component in AppMan is responsible for downloading task files, before sending a request to the *TM* to instantiate the task. In a cluster or local network with NFS, the *SM* creates a directory in a temporary area, which is a kind of sandbox, since tasks can only write in this area. Then, all input and executable files required by the task are copied to this directory. Since some tasks can share input files, or use the same executables, the *SM* checks if the required file is already in the sandbox area, before downloading it. This sandbox works as a data cache for tasks that run on the same grid node.




(a)




(b)

Figure G.2: AppMan graphical interface for monitoring application execution: snapshots

## G.2 Experimental Results

In this section we present results produced by AppMan, implementing the main functionalities of GRAND, including the distributed submission of tasks. Two experiments are reported, respectively, in Section G.2.1 and G.2.2.

### G.2.1 Resource Management Experiments

We ran a program that performs mathematics calculation in a local network with 4 available machines with different configurations. The machines are Atlhon XP 2 GHz, 512 MB, with Linux kernel 2.4.21-20.EL. This cluster was not dedicated to our experiments, but almost none user was accessing it during our experiments, since there were few user accounts and all users know the experiments were running. This application was chosen as an exerciser to AppMan. The application is composed of tasks, where each task takes few seconds to run.

In order to show how effective are the design features of GRAND, we compare the AppMan results with the same application running using only Condor. When submitting the application to Condor, we queued jobs in a ClassAd submission file.

```
1  graph independent
2  foreach ${n} in 0..100
3  do
4    task ${n} -e "chmod a+x teste; ./teste > t.out."${n}
5         -i /home/SO/kayser/tmp/teste -o "t.out."${n}
6  done
```

Figure G.3: GRID-ADL code for experiment 1

Figure G.2.1 shows the factorial application description written in GRID-ADL. According to this description, 100 tasks are created, whose executable is the file `teste`, with each one consuming one input file (indicated by the `-i` syntax) and producing one output file (indicated by the `-o` syntax). Note that in this example, the executable is provided as an input file for implementation simplification.

```
1  output = t.out.$(Process)
2  error = t.error.$(Process)
3  log = t.log.$(Process)
4  executable = /home/SO/kayser/tmp/teste
5  queue 100
```

Figure G.4: Condor code for experiment 1

Table G.1: Execution Times: Condor and AppMan

| tasks | SM | task sub. time | | task exec. time | | app. exec. time |
|---|---|---|---|---|---|---|
| 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 |

This example does not present dependent tasks, but AppMan is capable of handling dependencies.

Table G.1 shows the average task execution and application execution time running the application factorial using AppMan, and using only Condor. Column SM indicates the number of *Submission Managers* created by AppMan to distribute the task submission (a1 corresponds to 1 SM, a2 corresponds to 2 SMs and a3 corresponds to 3 SMs) as well as the Condor (c) execution times. The submission and execution time columns present average and standard deviation between parenthesis. All times are presented in seconds. The “task sub. time” column for AppMan corresponds to the time to instantiate the *SM* and actually select a node to dispatch the task. The “task sub. time” in Condor includes the time the task spend on the task queue.

It becomes very clear from this table that AppMan consistently completes an application in much less time than Condor. As the number of tasks to be submitted increases, this difference becomes even higher, being almost 5 times better when running AppMan with 500 tasks. Condor employs a more sophisticated form of matching resources to tasks by using the matchmaking algorithm. This may be one of the reasons why it takes longer to complete the application. On the other hand, in this particular experiment, Condor relied only on NFS to transfer the application input and output files to and from the user’s home

directory. AppMan actually transferred all input and output files to and from a web site using ftp. As AppMan can dispatch several tasks to the same machine, it is possible that several processes compete for the same processor. It is more efficient for small tasks, since tasks can be submitted in parallel. Condor always dispatches only one task per processor, and this is the reason for the lower average execution times for Condor. On the other hand, tasks take longer to be dispatched and to start running, delaying the whole application progress. Figure G.5 plots the application execution time, allowing to better visualize the AppMan performance.

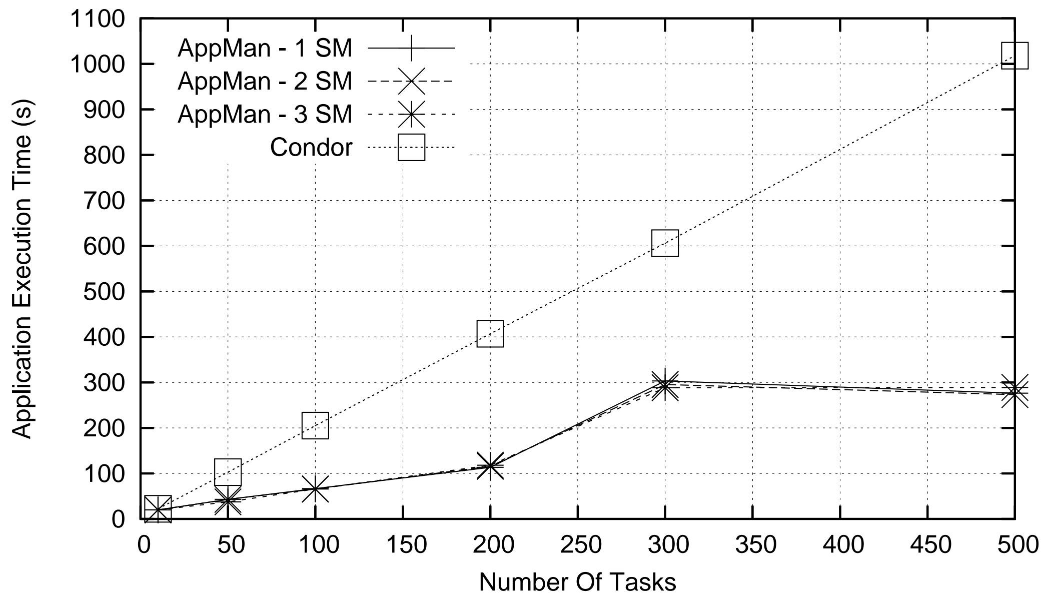



Figure G.5: Application execution time for AppMan and Condor

As this experiment ran using only 4 machines, the best average task execution is obtained with smaller numbers of *Submission Managers* (1 or 2). We expect this scenario to change when using more resources. In any case, this experiment shows that the time taken to execute the 100 tasks of this application by Condor is too high compared to the time taken to execute the same 100 tasks by AppMan. This may indicate that Condor may impose a too high overhead when running with a greater number of tasks, and that AppMan could be an alternative solution as a resource manager.

### G.2.2 Data Management Experiments

For this experiment, our workload is composed of experiments that run for a short period of time (approximately 3 seconds per each task). We vary the number of tasks per appli-

cation and the sizes of output files (results) produced by the tasks. We ran the experiments using two versions of AppMan: (1) without optimization, where input files are always transferred to the executing nodes, and (2) with optimization, where input files are cached at nodes with a *SM*. We also use two ways to obtain input files: via NFS and via ftp from a public site. All experiments were performed with the AppMan prototype over a local 100 Mbps Ethernet network with four machines Pentium IV 1.8 GHz with 1 GByte of RAM and two Atlhon XP 2GHz with 512 MByte of RAM. This experimental environment was a lab local area network with frequent users. To minimize interference, most experiments run at holidays and weekends, or at least at night.

AppMan used two types of protocol to obtain input files without optimizing file transfers (NFS and FTP) – and one with optimization (FTP). Using NFS, we copied files from directories on the NFS server of the local area network. FTP was used to get files from a web repository located in the same institution, but in a different network.

Table G.2 shows experimental data for AppMan, varying number of tasks from 50 to 200. The first column of this table shows the number of tasks for each run. The second column shows the input file size to be used by the tasks.

The third, fourth, and fifth columns show the total execution time for all tasks, respectively, for NFS and FTP transfers and FTP transfers with optimizations.

One curious aspect of this table is that the ftp protocol outperforms the NFS protocol as the size of the input files increases. This highlights the overhead of the NFS protocol when compared with the ftp protocol in a local network.

Table G.2: AppMan experimental values obtained from six nodes

| number of tasks | input file size | NFS | FTP | optimized FTP |
|---|---|---|---|---|
| 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 |

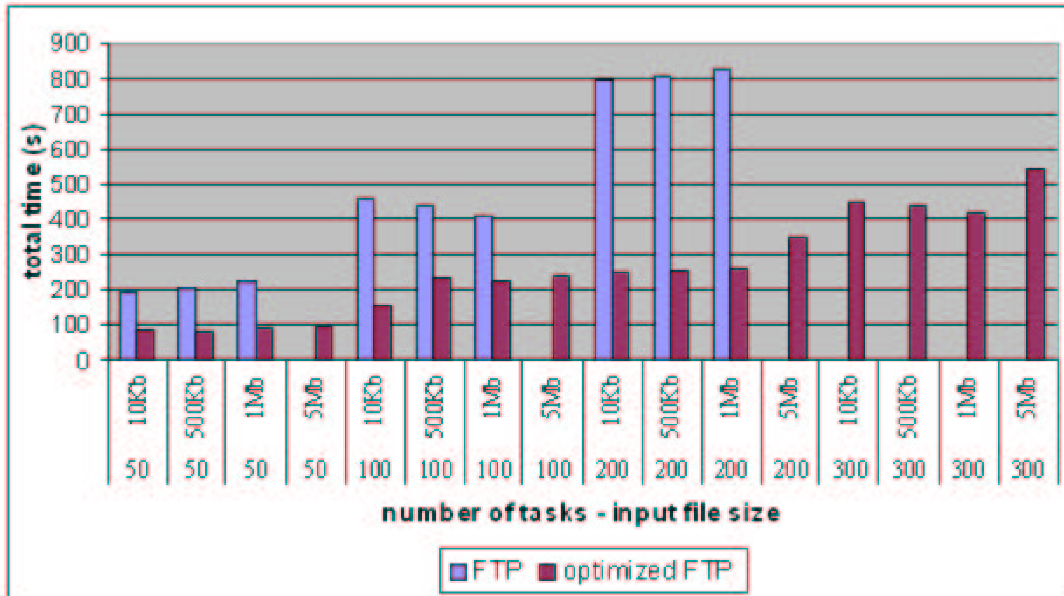



Figure G.6: Optimized stage in – scalability

The file transfer optimization uses a cache per task manager to keep track of files already available locally. We can clearly observe that the use of a cache per *TM* causes a positive effect in performance as we increase the number of tasks and the size of the input files. With the optimized version using FTP we have gains of more than 50% when transferring files of 1 Mbyte of size, running an application with 200 tasks.

These results show the need for data management for small to medium files and relatively small number of tasks. Data management is even more important when we increase the number of tasks and file sizes. We have been working on a medium to large scale experiment to improve our AppMan prototype and include new features that can facilitate data management. Figure G.6 outlines our initial results for bigger files (in the graph, up to 5Mb per task) and bigger number of tasks (up to 300). This graph indicates the scalability of the approach.

We presented simple, but effective techniques based on caching and data locality mechanisms to support data management of applications. Our aim was to propose a simple approach, i.e., with a low computational complexity, to avoid overloading the management system. The presented experimental evaluation shows promising results.

## G.3 Conclusion

We presented some preliminary evaluation of our prototype. Our initial results show that it is worthwhile distributing submission.

As a future work, we intend to complete the AppMan *Task Manager* in order to support interface with resource managers such as PBS [105] and Condor [135]. Besides, we intend to implement on demand file transfer (not all input files need to be specified).

In the current implementation, AppMan allocation decisions are based in the round-robin algorithm. We only get machines' addresses. We intend to use EXEHDA monitoring information to get dynamic information and implement the algorithm proposed in GRAND model.

Our main longer term step will be to have all the GRAND functionalities implemented and tested with real applications on a real grid environment.

# Apêndice H

# Conclusion

The assumption of submitting a huge number of tasks has become more important as new grid systems and environments are deployed worldwide. With more available resources, the user has the possibility of performing many more experiments that can easily saturate a working machine, if all tasks are launched simultaneously. The increase in the number of tasks that a user can now submit has a great impact to the scheduling architecture design.

In this thesis, we proposed and evaluated the GRAND model, that deals with application management in grid environments, focusing on applications that spread a very large number of tasks. We presented a general framework for grid environments, whose central idea is to have a flexible partitioning and a hierarchical organization where the load of the submit machine is shared with other machines. Our proposal takes advantage of hierarchical structures, because this is one of the most appropriate organization for grid environments.

We designed and implemented an architectural model at a middleware level. We consider that the resources and tasks are modeled as graphs. Our architectural model handles some important issues in the context of applications that spread a huge number of tasks: (1) partitioning applications such that, when possible, dependent tasks will be placed in the same grid node to avoid unnecessary migration of intermediate and/or transient data files, (2) partitioning applications such that tasks are allocated close to their required input data, (3) distributing the submission process such that the submit machine do not get overloaded, and (4) ensuring result files integrity.

This architectural model relies on already available software components to solve issues that otherwise would interfere in grid node administrative policies. The main services we would expect to be available at the grid infrastructure are: (1) RMS to provide alloca-

tion of tasks within a grid node, (2) resource monitoring tool inside the grid nodes; and (3) authorization control mechanisms.

We assume users submit applications to the system using our description language GRID-ADL (Grid Application Description Language) [114]. A GRID-ADL *description file* contains tasks description, which includes files that need to be staged. Each application has several tasks and can be represented as a Directed Acyclic Graph (DAG). In our DAGs, the nodes represent tasks and edges represent dependencies between tasks through data files access. Our system can infer automatically the DAG through the analysis of the data flow. A clustering algorithm is used to produce sets of tasks that can be submitted at the same time.

In order to support application scalability, GRAND relies on a hierarchical model where the tasks that compose the user's application are submitted from different machines, i.e., the application submission procedure is distributed among managers that are hierarchically organized. This contrasts to other grid systems where the submission is centralized on just one machine which can deteriorate response time [22, 52, 136, 150] and delay the execution of all tasks [142].

The submission and execution control are managed by three hierarchical components: (a) *Application Manager*(*AM*): is the high level controller that takes care of one specific application, and is responsible for DAG inference, clustering, and giving feedback to the user; (b) *Submission Manager*(*SM*): is responsible for mapping clusters to grid nodes; (c) *Task Manager*(*TM*): is a wrapper capable of submitting tasks to a specific grid node.

The rest of this section presents our main contributions (Section H.1) and some future works (Section H.2).

## H.1 Main Contributions

The main contribution of this thesis it is the definition of the GRAND model. This model was presented in detail in this thesis and allows to handle a huge number of tasks in a grid environment. Besides, we can outline some other contributions:

**Proposal of the GRID-ADL language.** GRID-ADL is a script-like language that allows to easily describe a huge number of tasks, with implicit DAG definition. GRID-ADL was proposed as a compact way to describe tasks, decreasing time to read application

description files. This representation also helps scalability: instead of storing all task descriptions in memory, GRAND unfolds tasks descriptions on demand.

**Definition of a new XML-based description of application DAGs.** This thesis presented some XML schemas to allow description of tasks, dependencies of tasks, and clusters of tasks. This schemas were proposed as an extension of JSDL standard [79] proposed by the Global Grid Forum [61].

**Application management in several steps. Task management in two steps: clustering and mapping.** The proposed steps are useful to understand the GRAND model as well as to guide new application management system design. The clustering and mapping steps are needed to submit the application tasks. Clustering is a kind of static partitioning that is performed without information about the grid nodes. Clusters are defined to be submitted altogether. An application taxonomy was proposed and we argue that for different kinds of applications, there is a clustering algorithm more appropriate. Each cluster is assigned to a *SM*, which is responsible for performing the mapping step.

The mapping step maps the clusters to the available grid nodes. We proposed a scheduling function to guide mapping decisions. This function considers four dynamic information: distance, capacity, cpu, and memory.

**Data management and monitoring services were defined.** As part of the application management system, data management and monitoring are essential. We presented simple and effective ways to perform data management as well how to get monitoring information from available monitoring tools.

**Implementation and evaluation of AppMan prototype.** We implemented a prototype called AppMan, and performed some experiments. Our experimental results are promising. For instance, an application can conclude execution in far less time (5 times faster) than other solutions that maintain a centralized task queue for submission, as we increase the number of tasks. We also showed the need for careful data management in grid environments, even with a small number of tasks and relatively small data files.

## H.2 Future works

As future work, we intend both to improve model and prototype.

Our model is concerned with scheduling tasks whose requirements are mainly memory and CPU. We are not considering special cases where tasks must have access to a specific resource such as a detector, a local database, or a special supercomputer. This kind of situation can occur in many real applications and must be considered in future works.

From the model point of view, any RMS can be used in the grid nodes. However, our prototype AppMan is only performing its own resource management. An integration with PBS [105], Condor [30] and SGE [67] resource management systems (RMSs) is under development.

We presented our model and implementation of the GRAND data management model. Automatic file transfer, without user intervention, is one of the goals of GRAND that was successfully implemented at the AppMan prototype. But we believe there is room for improvements. For instance, the GridFTP service [2] could be transparently used by AppMan to profit of some of its features, mainly stripped file transfer.

The GRAND model does not propose a replication model, but we intend to study the integration of our automatic data stage procedure to some replica catalog service, such as GRESS [160] or RLS [63].

Finally, there are some works aiming at integrating databases into the grid infrastructure [4, 148]. This is an important research topic which was not covered in the GRAND model and remains for a future work.

# Appendix I

# GRID-ADL ABNF

```
 1  ;
 2  ; The input_file non-terminal is the starting point of this grammar
 3  <input_file> = [ <graph_definition> ]
 4                 [ <application_requirements> ]
 5                 <set_of_task_definition>
 6                 [ <transient_file_definition> ]
 7  ;
 8  ; Optional graph definition
 9  <graph_definition> = "graph" <graph_type>
10  <graph_type> = "independent"
11               / "loosely-coupled"
12               / "tightly-coupled"
13
14  ;
15  ; Optional application requirements definition
16  ; This application constraints are based on a small subset of
17  ; Condor's ClassAd machine attributes
18  <application_requirements> = "requirement" "="  "(" <list_of_requirements'> ")"
19  <list_of_requirements'> =  <constraint>
20                          /  "(" <constraint>")"
21                             1*( <cond_operation> "(" <constraint> ")" )  ")"
22
23  <constraint> = ( "SysOp" ( "==" / "!=") <sys_op> )
24                          ; the operating system running on the machine
25               / ( "Memory" <logical_operation> <number> )
26                          ; memory available in the machine, expressed in megabytes
27               / ( "Arch" ( "==" / "!=") <arch>)
28                          ; the architecture of the machine
29               / ( "CPUs" <logical_operation> <number> )
30                          ; number of CPUs in this machine,  i.e.
31                          ; 1 = single CPU machine,
32                          ; 2 = dual CPUs, etc.
33  <sys_op> = "LINUX"   ; for LINUX 2.x.x kernel systems
34           / "WINDOWS" ; for Windows system
35  <arch> = "INTEL"  ; Intel x86 CPU (Pentium, Xeon, etc)
36         / "IA64"   ; Intel 64-bit CPU
37         / "SGI"    ; Silicon Graphics MIPS CPU
38         / "SUN4u"  ; Sun UltraSparc CPU
39
40  ;
41  ; Required task definition section
42  <set_of_task_definition>  = <task_definition>
43                            / <loop>
44                            / <if>
45                            / <assignment>
46                            / ( <task_definition> <set_of_task_definition> )
47                            / ( <loop> <set_of_task_definition> )
48                            / ( <assignment> <set_of_task_definition> )
49                            / ( <if> <set_of_task_definition> )
```

```
 50
 51
 52  <task_definition> = "task" [ <task_name> ] "-e" <executable>
 53                      "-i" <filenames>  "-o" <filenames>
 54                      [ "-c" <number> ]  [ "done" ]
 55  <task_name>   = <string>  /  <var>  /  <noum>
 56  <executable>  = <string>
 57
 58  <loop> = "foreach" ( <var> / <noum> ) "in" <range>
 59             "{" <set_of_task_definition> "}"
 60  <range> = <number> .. <number>
 61          /  "{" <symbols> "}"
 62  <symbols>  = <string>
 63             /  <string> ";" <symbols>
 64
 65  <if> = "if" <conditional_list> "{" <set_of_task_definition> "}"
 66         [  "else"  "{" <set_of_task_definition>  "}" ]
 67  <conditional_list>  = <conditional>
 68                      /  "("  <conditional> 1*( <cond_operation> <conditional> )  ")"
 69  <conditional> = "(" <operator> <logical_operation> <operator> ")"
 70  <cond_operation>  = "&&"  /  "||"
 71  <logical_operation>  = "=="  /  "!="  /  ">" /  ">=" /  "<" /  "<="
 72
 73
 74  <assignment> = <noum> "=" <assignment'>
 75  <assignment'>  =  <operator>
 76                 / <operator> <operation> <operator>
 77                 /  ( <var>/<number> ) <math_operation> ( <var>/<number> )
 78
 79  ;
 80  ; Optional transient file definition
 81  <transient_file_definition> = "transient" <filenames>
 82
 83
 84  ;
 85  ; Core definitions
 86  <operator> = <var>
 87             /  <string>
 88             /  <number>
 89  <operation> = "+"  /  "-"
 90  <math_operation> = "*"  /  "/"  /  "^"
 91
 92  <filenames> = <filename_unix>
 93              / <filename_windows>
 94              / <filename_unix> ";" <filenames>
 95              / <filename_windows> ";" <filenames>
 96  <filename_unix> = <noum>
 97                  / <noum> <filename_unix>
 98                  / <noum> "." <string>
 99                  / "/" <noum> <filename_unix>
100  <filename_windows> = <char> ":" <filename_windows2>
101                     / "\\" <noum> "\" <filename_windows2>
102  <filename_windows2> = <noum>
103                      / <noum> "." <noum>
104                      / "\" <noum> <filename_windows2>
105
106  <var> = "${" <noum> "}"
107  <string> =  <dquote> <noum'> <dquote>
108  <noum> = <char> <noum'>
109  <noum'> = <char>
110          /  <special_char>
111          /  <digit>
112          /  <char> <noum'>
113          /  <special_char> <noum'>
114          /  <digit> <noum'>
115  <char>  = %x01-1F      ; a-z /  A-Z
116  <special_char> = "-"  /  "_"
117
```

```
118  <number> = <digit>
119            /  <number> <digit>
120  <digit>  = %x30-39     ; 0-9
121  <dquote> = %x22        ; " (double quote)
```

# Referências Bibliográficas


[1] ALLCOCK, W., BRESNAHAN, J., FOSTER, I., *et al.*, "GridFTP Protocol Specification", September 2002. Technical Report. Available at `http://www.globus.org/research/papers/GridftpSpec02.doc`.

[2] ALLCOCK, W., BRESNAHAN, J., KETTIMUTHU, R., *et al.*, "The Globus striped GridFTP framework and server", In: *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, (Washington, DC, USA), p. 54, IEEE Computer Society, 2005.

[3] ALLEN, G., ANGULO, D., FOSTER, I., *et al.*, "The Cactus Worm: Experiments with dynamic resource discovery and allocation in a grid environment", *The International Journal of High Performance Computing Applications*, v. 15, n. 4, pp. 345–358, 2001.

[4] ALOISIO, G., CAFARO, M., FIORE, S., *et al.*, "The Grid-DBMS: Towards dynamic data management in grid environments", In: *Proceedings of IEEE International Conference on Information Technology: Coding and Computing (ITCC 2005)*, v. II, pp. 199–204, 4-6 April 2005.

[5] ANDINO, A. R., ARAÚJO, L., SÁENZ, F., *et al.*, "Parallel execution models for constraint programming over finite domains", In: *Proceedings of the International Conference Principles and Practice of Declarative Programming (PPDP'99)*, (Paris, France), pp. 134–151, September 29 – October 1 1999.

[6] ANDRADE, N., BRASILEIRO, F. V., CIRNE, W., *et al.*, "Discouraging free riding in a peer-to-peer cpu-sharing grid", In: *Proceedings of the 13th International Symposium on High-Performance Distributed Computing (HPDC-13 2004)*, (Honolulu, HW, USA), pp. 129–137, IEEE Computer Society, 2004.

[7] ANDRADE, N., CIRNE, W., BRASILEIRO, F. V., *et al.*, "OurGrid: An approach to easily assemble grids with equitable resource sharing", In: *Proceedings of the 9th Workshop on Job Scheduling Strategies for Parallel Processing*, (Seattle, WA, USA), pp. 61–86, June 2003.

[8] ANNIS, J., ZHAO, Y., VOECKLER, J., *et al.*, "Applying Chimera Virtual Data concepts to cluster finding in the sloan sky survey", In: *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, (Baltimore, Maryland, USA), pp. 1–14, November 16-22 2002.

[9] BAKER, M., BUYYA, R., LAFORENZA, D., "The Grid: International efforts in global computing", In: *Proceedings of the International Conference on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet (SSGRR 2000)*, (Rome, Italy), pp. ?–?, July 31 – August 6 2000.

[10] BELL, G., GRAY, J., "What's next in high-performance computing?", *Communications of The ACM*, v. 45, n. 2, pp. 91–95, February 2002.

[11] BERMAN, F., "High-Performance Schedulers", In: *The Grid: Blueprint for a New Computing Infrastructure* (FOSTER, I., KESSELMAN, C., eds.), pp. 279–309, Morgan Kaufmann, 1998.

[12] BERMAN, F., CHIEN, A., COOPER, K., *et al.*, "The GrADS Project: Software Support for High-Level Grid Application Development", *The International Journal of High Performance Computing Applications*, v. 15, n. 4, pp. 327–344, 2001.

[13] BERMAN, F., FOX, G., HEY, T., *Grid Computing: Making the Global Infrastructure a Reality*. 1 ed. John Wiley & Sons Inc., April 2003.

[14] BHATT, H. S., BHANSALI, D., SHAH, S. N., *et al.*, "GANGA: Grid Application iNformation Gathering and Accessing framework", In: *Proceedings of the 2nd Workshop on Grid Computing and Applications*, (Biopolis, Singapore), pp. ?–?, May 05 2005.

[15] BHATT, H. S., BHANSALI, D., SHAH, S. N., *et al.*, "GANGA: Grid Application iNformation Gathering and Accessing framework", *International Journal of Information Technology*, v. 11, n. 4, pp. 58–73, 2005.

[16] BODE, B., HALSTEAD, D. M., KENDALL, R., *et al.*, “The Portable Batch Scheduler and the Maui scheduler on linux clusters”, In: *Proceedings of the Usenix Conference*, (Atlanta, GA, USA), October 12–14 2000.

[17] BOERES, C., NASCIMENTO, A. P., REBELLO, V. E. F., *et al.*, “Efficient hierarchical self-scheduling for MPI applications executing in computational grids”, In: *MGC ’05: Proceedings of the 3rd international workshop on Middleware for grid computing*, (New York, NY, USA), pp. 1–6, ACM Press, 2005.

[18] BOERES, C., REBELLO, V. E. F., “On Solving the Static Task Scheduling Problem for Real Machines”, In: *Models for Parallel and Distributed Computation: Theory, Algorithmic Techniques and Applications* (CORRÊA, R., DUTRA, I. D. C., FIALLOS, M., *et al.*, eds.), Kluwer Academic Publishers, 2001.

[19] BOERES, C. B., REBELLO, V. E. F., “EasyGrid: Towards a framework for the automatic grid enabling of MPI applications”, In: *Proceedings of the 1st International Workshop on Middleware for Grid Computing (Middleware Workshops 2003)*, (Rio de Janeiro, Brazil), pp. 256–260, June 16–20 2003.

[20] BRISARD, F., LY, A., “Job Submission Description Language (JSDL) Specification – version 1.0”, November 7th 2005.

[21] BUNN, J. J., NEWMAN, H. B., “Data Intensive Grids for High Energy Physics”, In: *Grid Computing: Making the Global Infrastructure a Reality* (BERMAN, F., FOX, G., HEY, T., eds.), pp. 859–906, Wiley & Sons, 2003.

[22] BUYYA, R., ABRAMSON, D., GIDDY, J., “ Nimrod/G: An architecture of a resource management and scheduling system in a global computational grid”, In: *HPC Asia 2000*, (Beijing, China), pp. 283–289, May 14-17 2000.

[23] CASAVANT, T. L., KUHL, J. G., “A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems”, *IEEE Transactions on Software Engineering*, v. 14, n. 2, pp. 141–154, February 1988.

[24] “Cave5D Release 1.4”. `http://www.ccpo.odu.edu/~cave5d/`.

[25] CHANDRA, S., PARASHAR, M., “ARMaDA: An adaptive application-sensitive partitioning framework for SAMR applications”, In: *Proceedings of the 14th*

*IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2002)*, (Cambridge, MA, USA), pp. 446–451, ACTA Press, November 2002.

[26] CHAPIN, S. J., KATRAMATOS, D., KARPOVICH, J., *et al.*, "Resource management in Legion", In: *Proceedings of the IPDPS '99 5th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP '99)*, (San Juan, Puerto Rico, USA), pp. 162–178, April 16 1999.

[27] CIRNE, W., MARZULLO, K., "Open Grid: a user-centric approach for grid computing", In: *Proceedings of the 13th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2001)*, (Pirenópolis, GO, Brazil), pp. 106–111, September 10th–12th 2001.

[28] "The Compact Muon Solenoid (CMS) Project", 2005. `http://lcg.web.cern.ch/`.

[29] CONDOR EXTENSIONS, 2006. `http://sourceforge.net/projects/condor-ext`.

[30] Condor Project Homepage, 2006. `http://www.cs.wisc.edu/condor/`.

[31] CROCKER, D., OVERELL, P., "Augmented BNF for Syntax Specifications: ABNF", October 2005. IETF-RFC 4234. Available at `http://www.ietf.org/rfc/rfc4234.txt`.

[32] CZAJKOWSKI, K., FOSTER, I., KARONIS, N., *et al.*, "A resource management architecture for metacomputing systems", In: *Proceedings of the IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing*, pp. 62–82, 1998.

[33] "Directed Acyclic Graph Manager". `http://www.cs.wisc.edu/condor/dagman/`.

[34] DEBRAY, S., LIN, N.-W., HERMENEGILDO, M., "Task granularity analysis in logic programs", In: *Proceedings of the 1990 ACM Conf. on Programming Language Design and Implementation*, pp. 174–188, ACM Press, June 1990.

[35] DEELMAN, E., BLYTHE, J., GIL, Y., *et al.*, "Workflow Management in GriPhyN", In: *Grid Resource Management: State of the Art and Future Trends* (NABRZYSKI, J., SCHOPF, J. M., WEGLARZ, J., eds.), pp. 99–116, Kluwer Academic Publishers, 2003.

[36] DEFANTI, T. A., FOSTER, I., PAPKA, M. E., *et al.*, "Overview of the I-WAY: Wide-Area Visual Supercomputing", *The International Journal of Supercomputer Applications and High Performance Computing*, v. 10, n. 2/3, pp. 123–131, Summer/Fall 1996.

[37] "Distributed.Net", 2004. `http://www.distributed.net/`.

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

[39] DUTRA, I. D. C., PAGE, D., SANTOS COSTA, V., *et al.*, "Toward automatic management of embarrassingly parallel applications", In: *Proceedings of the 26th International Conference on Parallel and Distributed Computing (Europar 2003)*, (Klagenfurt, Austria), pp. 509–516, August 2003.

[40] "The EasyGrid project's research, reference and resource library". `http://easygrid.ic.uff.br/`.

[41] ENOMOTO, C., HENRIQUES, M. A. A., "A flexible specification model based on XML for parallel applications", In: *Proceedings of the 17th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2005)*, (Rio de Janeiro, RJ, Brasil), pp. 109–116, October 24-27 2005.

[42] ENOMOTO, C., HENRIQUES, M. A. A., "Implementação de uma linguagem de especificação de aplicações paralelas baseada em XML para o sistema join", In: *Anais do VI Workshop em Sistemas Computacionais de Alto Desempenho (WSCAD 2005)*, (Rio de Janeiro, RJ, Brasil), pp. 81–88, October 24-27 2005.

[43] FAHRINGER, T., JUGRAVU, A., PLLANA, S., *et al.*, "ASKALON: A Tool Set for Cluster and Grid Computing", *Concurrency and Computation: Practice & Experience*, v. 17, n. 2–4, pp. 143–169, 2005.

[44] FAHRINGER, T., QIN, J., HAINZER, S., "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.

[45] FEITELSON, D. G., RUDOLPH, L., "Parallel job scheduling: issues and approaches", In: *Job Scheduling Strategies for Parallel Processing* (FEITELSON, D. G., RUDOLPH, L., eds.), pp. 1–18, Springer-Verlag, 1995. Lecture Notes in Computer Science Vol. 949.

[46] FJÄLLSTRÖM, P.-O., "Algorithms for Graph Partitioning: A Survey", *Linköping Electronic Articles in Computer and Information Science*, v. 3, n. 010, 1998.

[47] FOSTER, I., "What is the Grid? A Three Point Checklist", *Grid Today*, v. 1, n. 6, July 22 2002. Available at `http://www.gridtoday.com/02/0722/100136.html`.

[48] FOSTER, I., KESSELMAN, C., "Computational Grids", In: *The Grid: Blueprint for a New Computing Infrastructure* (FOSTER, I., KESSELMAN, C., eds.), pp. 15–52, San Francisco, California, USA, Morgan Kaufmann, 1 ed., 1998.

[49] FOSTER, I., KESSELMAN, C., "The Globus project: A status report", In: *Proc. IPPS/SPDP '98 Heterogeneous Computing Workshop*, pp. 4–18, 1998.

[50] FOSTER, I., KESSELMAN, C., *The Grid: Blueprint for a New Computing Infrastructure*. 1 ed. San Francisco, California, USA, Morgan Kaufmann, 1998.

[51] FOSTER, I., KESSELMAN, C., *The Grid: Blueprint for a New Computing Infrastructure*. 2 ed. San Francisco, California, USA, Morgan Kaufmann, 2004.

[52] FOSTER, I., KESSELMAN, C., NICK, J., *et al.*, "Grid Services for Distributed System Integration", *IEEE Computer*, v. 35, n. 6, pp. 37–46, June 2002.

[53] FOSTER, I., KESSELMAN, C., NICK, J., *et al.*, "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration", June 2002. Available at `http://www.globus.org/research/papers/ogsa.pdf`.

[54] FOSTER, I., KESSELMAN, C., TUECKE, S., "The Anatomy of the Grid: Enabling Scalable Virtual Organizations", *The International Journal of High Performance Computing Applications*, v. 15, n. 3, pp. 200–222, Fall 2001.

[55] FOSTER, I., VOECKLER, J., WILDE, M., *et al.*, "Chimera: A virtual data system for representing, querying and automating data derivation", In: *Proceedings of the 14th Conference on Scientific and Statistical Database Management*, (Edinburgh, Scotland), July 2002.

[56] FOSTER, I., VOECKLER, J., WILDE, M., *et al.*, "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.

[57] FREY, J., TANNENBAUM, T., FOSTER, I., *et al.*, "Condor-G: A Computation Management Agent for Multi-Institutional Grids", *Cluster Computing*, v. 5, pp. 237–246, 2002.

[58] FULP, E. W., REEVES, D. S., "Distributed network flow control based on dynamic competive markets", In: *Proceedings International Conference on Network Protocol (ICNP'98)*, (Austin Texas), October 13-16 1998. Available at `http://citeseer.nj.nec.com/fulp98distributed.html`.

[59] GAREY, M. R., JOHNSON, D. S., *Computers and intractability: a guide to the theory of NP-Completeness*. New York, Freeman, 1979.

[60] gLite – Ligthweight Middleware for Grid Computing, 2006. `http://glite.web.cern.ch/glite/`.

[61] "Global Grid Forum". `http://www.ggf.org/`.

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

[63] Globus Alliance, "GT 4.0 Replica Location Service (RLS)", 2006. `http://www.globus.org/toolkit/docs/4.0/data/rls/`.

[64] GONG, Y., DONG, F., LI, W., *et al.*, "VEGA infrastructure for resource discovery in grids", *J. Comput. Sci. Technol.*, v. 18, n. 4, pp. 413–422, 2003.

[65] "Grand challenge applications". `http://www-fp.mcs.anl.gov/grand-challenges/`.

[66] GRID-SEQUENTIAL ACCESS VIA METADATA (GRID-SAM) PROJECT, T., 2006. http://venus.cs.ttu.edu/SAM/.

[67] "Grid Engine Project Home". http://gridengine.sunsource.net/.

[68] "gridengine: Developing applications with the DRMAA java language bindings". http://gridengine.sunsource.net/project/gridengine/howto/drmaa_java.htm.

[69] GRIER, D. A., "Systems for monte carlo work", In: *Proceedings of the 19th Conference on Winter Simulation*, pp. 428–433, ACM Press, 1987.

[70] GRIMSHAW, A., FERRARI, A., LINDAHL, G., *et al.*, "Metasystems", *Communications of the ACM*, v. 41, n. 11, pp. 46–55, 1998.

[71] GRIMSHAW, A. S., FERRARI, A., KNABE, F., *et al.*, "Wide-Area Computing: Resource Sharing on a Large Scale", *IEEE Computer*, v. 32, n. 5, pp. 29–36, May 1999.

[72] "GriPhyN – grid physics network". http://www.griphyn.org/.

[73] GUNTER, D., MAGOWAN, J., "An analysis of "Top N" Event Descriptions", January 2004. GGF Discovery and Monitoring Event Descriptions Working Group. http://www.gridforum.org/documents/GFD.25.pdf.

[74] HENDRICKSON, B., KOLDA, T. G., "Graph partitioning models for parallel computing", *Parallel Computing*, v. 26, n. 12, pp. 1519–1534, 2000.

[75] HENDRICKSON, B., LELAND, R., VAN DRIESSCHE, R., "Enhancing data locality by using terminal propagation", In: *29th Hawaii International Conference on System Sciences (HICSS'96) Volume 1: Software Technology and Architecture*, (Maui, Hawaii, USA), pp. 565–584, January 03–06 1996.

[76] HOGSTEDT, K., KIMELMAN, D., RAJAN, V. T., *et al.*, "Graph cutting algorithms for distributed applications partitioning", *ACM SIGMETRICS Performance Evaluation Review*, v. 28, n. 4, pp. 27–29, 2001.

[77] Institute of Electrical and Electronics Engineers, "Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications", 1985. ANSI/IEEE Std 802.3-1985.

[78] "javacc: Javacc project home". `https://javacc.dev.java.net/`.

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

[80] "JSDL WG – project documentation". `http://forge.gridforum.org/docman2/ViewCategory.php?group_id=122&category_id=814`.

[81] KANEDA, K., TAURA, K., YONEZAWA, A., "Routing and resource discovery in phoenix grid-enabled message passing library", In: *Proceedings of the 4th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCgrid 2004)*, (Chicago, Illinois, USA), April 19–22 2004.

[82] KARONIS, N., TOONEN, B., FOSTER, I., "MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface", *Journal of Parallel and Distributed Computing*, v. 63, n. 5, pp. 551–563, May 2003.

[83] KARYPIS, G., KUMAR, V., "Multilevel algorithms for multi-constraint graph partitioning", In: *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, pp. 1–13, IEEE Computer Society, 1998.

[84] KRAUTER, K., BUYYA, R., MAHESWARAN, M., "A Taxonomy and Survey of Grid Resource Management Systems for Distributed Computing", *Software – Practice and Experience*, v. 32, n. 2, pp. 135–164, 2002.

[85] KUMAR, S., DAS, S. K., BISWAS, R., "Graph partitioning for parallel applications in heterogeneous grid environments", In: *International Parallel and Distributed Processing Symposium (IPDPS'02)*, (Fort Lauderdale, CA, USA), April 15–19 2002.

[86] KURATA, K.-I., SAGUEZ, C., DINE, G., *et al.*, "Evaluation of unique sequences on the european data grid", In: *Proceedings of the First Asia-Pacific bioinformatics conference on Bioinformatics 2003*, pp. 43–52, Australian Computer Society, Inc., 2003.

[87] KWOK, Y.-K., AHMAD, I., “Static scheduling algorithms for allocating directed task graphs to multiprocessors”, *ACM Computing Survey*, v. 31, n. 4, pp. 406–471, 1999.

[88] LAFORENZA, D., “Grid Programming: some indications where we are headed”, *Parallel Computing*, v. 28, n. 12, pp. 1733–1752, December 2002.

[89] LCG Middleware, 2006. `http://lcg.web.cern.ch/lcg/activities/middleware.html`.

[90] LEGRAND, I., NEWMAN, H. B., “The MONARC toolset for simulating large network-distributed processing systems”, In: *Winter Simulation Conference 2000*, pp. 1794–1801, 2000.

[91] LEGRAND, I. C., DOBRE, C. M., STRATAN, C., “MONARC 2 (Models of Networked Analysis at Regional Centers) – distributed systems simulation”. `http://monalisa.cacr.caltech.edu/MONARC/Papers/MONARC_Implementation.zip`.

[92] LEISERSON, C. E., CORMEN, T. H., *An Introduction to Algorithms*. The MIT Press, 1990.

[93] LEWIS, M. J., FERRARI, A. J., HUMPHREY, M., *et al.*, “Support for Extensibility and site autonomy in the Legion Grid System Object Model”, *Journal of Parallel and Distributed Computing*, n. 63, pp. 525–538, May 2003.

[94] LIAN, C. C., TANG, F., ISSAC, P., *et al.*, “GEL: grid execution language”, *Journal of Parallel and Distributed Computing*, v. 65, n. 7, pp. 857–869, 2005.

[95] “Load Sharing Facility (LSF)”. `http://accl.grc.nasa.gov/lsf/`.

[96] “Plataform LSF Family”. `http://www.platform.com/products/LSFfamily/`.

[97] LU, P., “The Trellis project”. `http://www.cs.ualberta.ca/~paullu/Trellis/`.

[98] MASSIE, M. L., CHUN, B. N., CULLER, D. E., "The ganglia distributed monitoring system: design, implementation, and experience", *Parallel Computing*, v. 30, n. 7, pp. 817–840, July 2004.

[99] "GT Information Services: Monitoring & Discovery System (MDS)". `http://www.globus.org/toolkit/mds/`.

[100] MONTERO, R. S., HUEDO, E., LLORENTE, I. M., "Grid resource selection for opportunistic job migration", In: *26th International Conference on Parallel and Distributed Computing (Europar 2003)*, (Klagenfurt, Austria), pp. 366–373, August 2003.

[101] NABRZYSKI, J., SCHOPF, J. M., WEGLARZ, J., *Grid Resource Management: State of the Art and Future Trends*. Kluwer Academic Publishers, 2003.

[102] NEMETH, Z., SUNDERAM, V., "A comparison of conventional distributed computing environments and computational grids", In: *Proceedings of the International Conference on Computational Science (ICCS2002)*, (Amsterdam, Netherlands), pp. 729–738, April 2002. LNCS 2329.

[103] NEMETH, Z., SUNDERAM, V., "A formal framework for defining grid systems", In: *Proceedings of the Second IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID2002)*, (Berlin, Germany), pp. 202–211, May 21–24 2002.

[104] NEWMAN, H. B., LEGRAND, I. C., GALVEZ, P., *et al.*, "MonALisa: A distributed monitoring service architecture", In: *Computing in High Energy and Nuclear Physics (CHEP03)*, (La Jolla, California, USA), March 24-28 2003.

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

[106] PARANHOS, D. D. S., CIRNE, W., BRASILEIRO, F. V., "Trading cycles for information: Using replication to schedule bag-of-tasks applications on computational grids", In: *Proceedings of the 26th International Conference on Parallel and Distributed Computing (Euro-Par 2003)*, pp. 169–180, August 2003.

[107] PARK, S.-M., KIM, J.-H., "Chameleon: A resource scheduler in a data grid environment", In: *Proc of 3st International Symposium on Cluster Computing and the Grid*, (Tokyo, Japan), pp. 258–, May 12 - 15 2003.

[108] "PBS Pro Home". http://www.pbspro.com/.

[109] "Planning for execution in grids". http://www.isi.edu/~deelman/pegasus.htm.

[110] RAJIC, H., BROBST, R., CHAN, W., *et al.*, "Distributed Resource Management Application API 1.0 Specification", June 2004. http://www.ggf.org/documents/GWD-R/GFD-R.022.pdf.

[111] RAMAN, R., LIVNY, M., SOLOMON, M., "Matchmaking: Distributed resource management for high throughput computing", In: *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, (Chicago, USA), pp. 140–147, July 28-31 1998.

[112] ROEHRIG, M., ZIEGLER, W., WIEDER, P., "Grid Scheduling Dictionary of Terms and Keywords", document gfd-i.11, Global Grid Forum, Nov 2002. Available at http://forge.gridforum.org/projects/ggf-editor/document/GFD-I.11/en/1.

[113] ROURE, D. D., BAKER, M. A., JENNINGS, N. R., *et al.*, "The Evolution of the Grid", In: *Grid Computing: Making the Global Infrastructure a Reality* (BERMAN, F., FOX, G., HEY, T., eds.), pp. 65–100, Wiley & Sons, 2003.

[114] SANCHES, J. A. L., VARGAS, P. K., DUTRA, I. C., *et al.*, "ReGS: user-level reliability in a grid environment", In: *Cluster Computing and Grid 2005 (CCGRID 2005)*, (Cardiff, UK), May 9–12 2005.

[115] SANDER, V., ALLCOCK, W., CONGDUC, P., *et al.*, "Networking Issues of Grid Infrastructures", document draft-ggf-ghpn-netissues-0, version 1, Global Grid Forum, June 2003. Available at http://forge.gridforum.org/projects/ggf-editor/document/Networking_Issues_of_Grid_Infrastructures/en/1/Networking_Issues_of_Grid_Infrastructures.pdf.

[116] SANDHOLM, T., GAWOR, J., "Globus Toolkit 3 Core – A Grid Service Container Framework", May 2003. White paper. Available at `http://www-unix.globus.org/toolkit/3.0beta/ogsa/docs/gt3_core.pdf`.

[117] SCHAEFFER FILHO, A. E., SILVA, L. C., YAMIN, A. C., *et al.*, "PerDiS: A scalable resource discovery service for the ISAM pervasive environment", In: *Proceedings of the 1st International Workshop on Hot Topics in Peer-to-Peer Systems (Hot-P2P)*, pp. 80–85, IEEE Computer Society, 2004.

[118] SCHWIEGELSHOHN, U., YAHYAPOUR, R., "Attributes for Communication between Scheduling Instances", December 2001. Available at `http://ds.e-technik.uni-dortmund.de/~yahya/ggf-sched/WG/sched_attr/SchedWD.10.6.pdf`.

[119] SCHWIEGELSHOHN, U., YAHYAPOUR, R., "Attributes for Communication between Scheduling Instances", In: *Grid Resource Management* (NABRZYSKI, J., SCHOPF, J. M., WEGLARZ, J., eds.), pp. 41–52, Kluwer Academic Publishers, 2004.

[120] "SDSS - Sloan Digital Sky Survey". `http://www.sdss.org/`.

[121] "SETI@home – The Search for Extraterrestrial Intelligence at Home", 2004. `http://setiathome.ssl.berkeley.edu/`.

[122] SINGHAL, M., SHIVARATRI, N. G., *Advanced Concepts in Operating Systems: Distributed, Database, and Multiprocessor Operating Systems*. New York, MIT Press, 1994.

[123] SMARR, L., CATLETT, C. E., "Metacomputing", *Communications of the ACM*, v. 35, n. 6, June 1992.

[124] SOTOMAYOR, B., CHILDERS, L., *Globus Toolkit 4: Programming Java Services*. Morgan Kaufmann, 2006.

[125] STOCKINGER, H., "Distributed database management systems and the data grid", In: *Eighteenth IEEE Symposium on Mass Storage Systems*, (Hyatt Regency Islandia, San Diego, USA), April 17-20 2001.

[126] STOCKINGER, H., RANA, O. F., MOORE, R., *et al.*, "Data management for grid environments", In: *European High Performance Computing & Networking (HPCN 2001)*, (Amsterdam, The Netherlands), pp. 151–160, June 2001.

[127] STOCKINGER, H., SAMAR, A., HOLTMAN, K., *et al.*, "File and object replication in data grids", In: *10th IEEE International Symposium on High Performance Distributed Computing*, pp. 76–86, August 7–9 2001.

[128] SULISTIO, A., YEO, C. S., BUYYA, R., "A Taxonomy of Computer-based Simulations and its Mapping to Parallel and Distributed Systems Simulation Tools", *International Journal of Software: Practice and Experience*, v. 34, n. 7, pp. 653–673, 2004.

[129] Sun Microsystems, "Sun Cluster Grid Architecture", sun one grid engine white papers, Sun Microsystems, 2002. Available at `http://wwws.sun.com/software/grid/SunClusterGridArchitecture.pdf`.

[130] TANENBAUM, A. S., *Distributed Operating Systems*. Prentice Hall, 1995.

[131] TANENBAUM, A. S., VAN RENESSE, R., "Distributed Operating Systems", *ACM Computing Surveys (CSUR)*, v. 17, n. 4, pp. 419–470, December 1985.

[132] TANNENBAUM, T., WRIGHT, D., MILLER, K., *et al.*, "Condor – A Distributed Job Scheduler", In: *Beowulf Cluster Computing with Linux* (STERLING, T., ed.), pp. 307–350, MIT Press, October 2002.

[133] TAREK EL-GHAZAWI, P. I., GAJ, K., ALEXANDRIDIS, N., *et al.*, "Conceptual Comparative Study of Job Management Systems", technical report, George Mason University, USA, February 21 2001. Available at `http://ece.gmu.edu/lucite/reports/Conceptual_study.PDF`.

[134] TAYLOR, I., WANG, I., SHIELDS, M., *et al.*, "Distributed computing with Triana on the Grid", *Concurrency and Computation: Practice and Experience*, v. 17, n. 1–18, pp. 1197–1214, 2005.

[135] THAIN, D., TANNENBAUM, T., LIVNY, M., "Condor and the Gri d", In: *Grid Computing: Making The Global Infrastructure a Reality* (BERMAN, F., FOX, G., HEY, T., eds.), John Wiley, 2003.

[136] THAIN, D., TANNENBAUM, T., LIVNY, M., "Distributed computing in practice: the Condor experience", *Concurrency and Computation: Practice and Experience*, v. 17, n. 2–4, pp. 323–356, February – April 2005.

[137] The Legion Group, "Legion 1.8 – Developer Manual", 2001.

[138] The MONARC Project – Models of Networked Analysis at Regional Centres for LHC Experiments, "Distributed computing simulation". `http://monarc.web.cern.ch/MONARC/sim_tool/`.

[139] TIERNEY, B., AYDT, R., GUNTER, D., *et al.*, "A Grid Monitoring Architecture", January 2002. GGF Performance Working Group. `http://www.gridforum.org/documents/GFD.7.pdf`.

[140] VADHIYAR, S. S., DONGARRA, J. J., "A metascheduler for the grid", In: *11th IEEE International Symposium on High Performance Distributed Computing (HPDC-11)*, (Edinburgh, Scotland), pp. 343–354, July 24 – 26 2002.

[141] VADHIYAR, S. S., DONGARRA, J. J., "A performance oriented migration framework for the grid", In: *3rd International Symposium on Cluster Computing and the Grid (CCGRID 2003)*, (Tokyo, Japan), pp. 366–373, May 12 – 15 2003.

[142] VARGAS, P. K., DE CASTRO DUTRA, I., DO NASCIMENTO, V. D., *et al.*, "Hierarchical submission in a grid environment", In: *3rd International Workshop on Middleware for Grid Computing*, (Grenoble, France), November 28 – December 2 2005.

[143] VARGAS, P. K., DUTRA, I. C., GEYER, C. F., "Application partitioning and hierarchical management in grid environments", In: *1st International Middleware Doctoral Symposium 2004*, (Toronto, Canadá), pp. 314–318, October 19th 2004.

[144] VARGAS, P. K., SANTOS, L. A. S., DUTRA, I. C., *et al.*, "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), January 31 – February 2 2005. Available at `http://virtual.lncc.br/wcga05/text/7483.pdf`.

[145] VARGAS, P. K., DUTRA, I. D. C., GEYER, C. F., "Hierarchical Resource Management and Application Control in Grid Environments", Tech. Rep. ES-608/03, COPPE/Sistemas - UFRJ, 2003. Relatório Técnico.

[146] VARGAS, P. K., DUTRA, I. D. C., GEYER, C. F., "Application Partitioning and Hierarchical Application Management in Grid Environments", Tech. Rep. ES-661/04, COPPE/Sistemas - UFRJ, 2004. Relatório Técnico.

[147] VARGAS, P. K., DUTRA, I. D. C., GEYER, C. F., "Gerenciamento hierárquico de aplicações em ambientes de computação em grade", In: *Escola Regional de Alto Desempenho (ERAD 2004)*, (Pelotas, RS), 13 a 17 de janeiro 2004.

[148] WATSON, P., "Databases and the Grid", UK e-Science Programme Technical Report Series UKeS-2002-01, National e-Science Centre, February 2002. Document produced by "Database Access and Integration Services Working Group"of Global Grid Forum. Available at `http://www.cs.man.ac.uk/grid-db/papers/dbg.pdf`.

[149] WELCH, V., SIEBENLIST, F., FOSTER, I., *et al.*, "Security for grid services", In: *12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)*, (Seattle, Washington), pp. 48–57, June 22–24 2003.

[150] WHITE, B. S., WALKER, M., HUMPHREY, M., *et al.*, "LegionFS: A secure and scalable file system supporting cross-domain high-performance applications", In: *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (Supercomputing '01)*, (Denver, USA), November 2001.

[151] WOLFE, M., *High-Performance Compilers for Parallel Computing*. Addison Wesley, 1996.

[152] WRIGHT, D., "Cheap cycles from the desktop to the dedicated cluster: combining opportunistic and dedicated scheduling with condor", In: *Proceedings of the Conference on Linux Clusters: The HPC Revolution*, (Champaign - Urbana, IL, USA), June 2001.

[153] "The WS-Resource Framework", 2005. `http://www.globus.org/wsrf/`.

[154] YAMIN, A., AUGUSTIN, I., BARBOSA, J., *et al.*, "ISAM: a pervasive view in distributed mobile computing", In: *Proceedings of the IFIP TC6 / WG6.2 & WG6.7 Conference on Network Control and Engineering for QoS, Security and Mobility (NET-CON 2002)*, pp. 431–436, October 23–25 2002.

[155] YAMIN, A., AUGUSTIN, I., BARBOSA, J., *et al.*, "Towards Merging Context-aware, Mobile and Grid Computing", *International Journal of High Performance Computing Applications*, v. 17, n. 2, pp. 191–203, June 2003.

[156] YAMIN, A., BARBOSA, J., SILVA, L. C. D., *et al.*, "A framework for exploiting adaptation in high heterogeneous distributed processing", In: *Proceedings of the XIV Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, (Vitória, ES, Brasil), October 28–30 2002.

[157] YAMIN, A. C., *Arquitetura para um Ambiente de Grade Computacional Direcionado às Aplicações Distribuídas, Móveis e Conscientes do Contexto da Computação Pervasiva*. PhD thesis, II/UFRGS, Porto Alegre, RS, Brasil, 2004.

[158] YANG, T., GERASOULIS, A., "DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors", *IEEE Transactions on Parallel and Distributed Systems*, v. 5, n. 9, pp. 951–967, 1994.

[159] YU, J., BUYYA, R., "A Taxonomy of Workflow Management Systems for Grid Computing", Technical Report GRIDS-TR-2005-1, Grid Computing and Distributed Systems Laboratory, University of Melbourne, Australia, March 2005.

[160] ZHAO, Y., HU, Y., "GRESS – a grid replica selection service", In: *ISCA 16th International Conference on Parallel and Distributed Computing Systems PDCS-2003*, August 2003.