DIOGO CEZAR TEIXEIRA BATISTA

UM MODELO DE EXECUÇÃO DE FLUXOS DE TRABALHO CIENTÍFICO UTILIZANDO TÉCNICAS DE PLANEJAMENTO AUTOMÁTICO

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Fabiano Silva

CURITIBA

2012

DIOGO CEZAR TEIXEIRA BATISTA

UM MODELO DE EXECUÇÃO DE FLUXOS DE TRABALHO CIENTÍFICO UTILIZANDO TÉCNICAS DE PLANEJAMENTO AUTOMÁTICO

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Fabiano Silva

CURITIBA

2012

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE CÓDIGOS

LISTA DE SIGLAS E ABREVIAÇÕES

BPEL *The Business Process Execution Language for Web Services*

DPLL *Davis-Putnam-Logemann-Loveland*

FF *Fast-Forward*

HSP *Heuristic Search Planning*

IA Inteligência Artificial

IPC4 *The 4th International Planning Competition*

MDS *Globus Monitoring and Discovery Service*

MoML *Modeling Markup Language*

ms Milisegundos

P2P *Peer to Peer*

PDDL *Planning Domain Definition Language*

RLS *Globus Replica Location Service*

SAT Satisfabilidade

STRIPS *STranford Research Institute Problem Solver*

SWEP *Smart Workflow Execution by Planning*

TC *Transformation Catalog*

VDL *Virtual Data Language*

XML *eXtensible Markup Language*

RESUMO

Experimentos científicos produzem grande quantidade de informações que necessitam de processamento para uma posterior análise. Um cientista, que não é da área da computação, nem sempre possui as habilidades para desenvolver seu próprio ambiente de testes. Por isso a utilização de executores de fluxos de trabalhos científicos vêm sido largamente estudada. Uma das principais vantagens de se utilizar um processador de fluxo de trabalho científico é a transparência oferecida para o cientista em relação a maneira com que os experimentos serão organizados, distribuídos e processados. Este trabalho propõe um modelo para criação de um ambiente que seja capaz de processar esses fluxos de trabalho. A ênfase está em um escalonamento inteligente que utiliza técnicas para resolução de problemas de planejamento da área de inteligência artificial.

Palavras-chave: Planejamento, Inteligência Artificial, Fluxo de Trabalho Científico, Computação Distribuída

ABSTRACT

Scientific experiments produce a large amount of information that require processing for further analysis. A scientist, that not work in computing area, does not always have the skills to develop their own test environment. Therefore the usage of executors of scientific workflows have been widely studied. The main advantage in use a workflow scientific processor is the transparency offered for the scientist regarding the manner in which the experiments will be organized, distributed and processed. This work proposes a model for creating an environment that be able to process these workflows. The emphasis is on a intelligent schedule that uses techniques to solve planning problems in the area of artificial intelligence.

Key-words: Planning, Artificial Intelligence, Scientific Workflow, Distributed Computing

CAPÍTULO 1

INTRODUÇÃO

Experimentos científicos produzem terabytes de informações que precisam ser processadas e analisadas. A interação multidisciplinar entre a computação e outras áreas da ciência carece de ferramentas que ofereçam processamento massivo dos dados de seus experimentos científicos. Para que estes sejam processados, podem ser usados fluxos de trabalho científico que dividem uma consulta maior em partes menores, permitindo uma distribuição estratégica do processamento desses fluxos, visando minimizar o tempo de execução e o melhor aproveitamento dos recursos computacionais [32].

Os experimentos científicos, normalmente utilizam um ambiente distribuído que ofereça recursos computacionais de alto desempenho. Neste ambiente, utilizam-se técnicas de computação distribuída que possibilitem tal processamento. Um ambiente distribuído envolve a coordenação e o compartilhamento dos recursos computacionais, armazenamento de dados e recursos de rede.

Uma das dificuldades ao se trabalhar com fluxos de trabalhos científicos está no escalonamento das tarefas que os compõem. Além disso, é necessário planejar a alocação das tarefas de múltiplos fluxos de trabalho científico, a fim de otimizar a utilização dos recursos computacionais disponíveis. Por isto, é necessário um mecanismo para o escalonamento de tarefas. A técnica de resolução de problemas de planejamento em Inteligência Artificial (IA) pode ser explorada para a elaboração de um plano de execução mais refinado [34].

Um problema de planejamento pode ser definido como o problema de encontrar uma sequência de ações que quando executada em um contexto que satisfaça um estado inicial do problema, vai atingir o objetivo do mesmo [34]. Ao mapear computacionalmente situações do mundo real, nota-se a complexidade obtida por uma grande quantidade de escolhas na construção de um plano de ações, o que torna o problema intratável com

algoritmos de busca convencionais.

O problema de escalonamento pode ser visto como um subproblema do problema de planejamento, no qual já se sabe quais são as ações que compõem o plano, o objetivo é encontrar uma ordenação coordenada destas ações. Ao tratar problemas complexos recomenda-se a utilização de metadados específicos do domínio, que auxiliam na busca de uma solução para o problema de planejamento. Um metadado é um dado que possui informação sobre outros dados. Uma dentre várias abordagens que utiliza metadados é chamada de proveniência de dados, que prevê as ações e comportamentos baseando-se nos históricos fornecidos em execuções anteriores [32].

Com a aplicação de técnicas de proveniência é possível obter um escalonamento refinado. Analisando o ambiente de execução e armazenando seu histórico, é possível extrair informações que auxiliam no processo de escalonamento. Por exemplo, se uma tarefa s com determinadas características é executada com um tempo t em determinado conjunto de nós, uma tarefa que possui as mesmas características s' pode ser executada em um tempo próximo a t no mesmo conjunto de nós. Assim é possível prever o tempo de execução, o que pode ajudar no processo de escalonamento. Depois que as tarefas estão escalonadas em seus respectivos nós, torna-se necessária uma estrutura que possibilite interpretar o plano e disparar a execução das tarefas em um ambiente distribuído. O gerenciador de execução faz com que os nós do sistema distribuído sejam capazes de executar as tarefas que lhe são designadas, respeitando o plano de execução anteriormente traçado.

O objetivo deste trabalho é criar um sistema de distribuição de tarefas automático, capaz de aproveitar metadados dos históricos de execução para obter planos mais refinados. Para isso, propõe-se a utilização de técnicas para resolução de problemas de planejamento da área da IA com o objetivo de escalonar as tarefas dos fluxos de trabalho científico [14]. O trabalho proposto utiliza um modelo que atua como ligação entre o planejador e o gerenciador de execução. O planejador recebe como entrada os dados a serem processados, os operadores que atuarão sobre esses dados e as informações sobre os recursos que estão sendo utilizados na camada de execução. Como saída, gera-se um plano de execução. O

gerenciador de execução, por sua vez, utiliza o plano de execução como entrada, além dos dados a serem processados, e os operadores que atuarão sobre esses dados. Como saída, são gerados os dados processados e informações com o estado atual dos recursos utilizados.

No modelo proposto são explorados dois tipos de paralelismo: *paralelismo sobre os dados* e *paralelismo sobre a execução*. O paralelismo sobre os dados é dividido em dois submodelos: *direto* e *paralelo por plano*. O modelo *direto* escalona o processamento dos dados a partir de parâmetros inseridos na descrição do fluxo de trabalho. O modelo *paralelo por plano* utiliza o planejador para dividir os dados entre os nós disponíveis. O *paralelismo sobre a execução* é a execução paralela dos fluxos de trabalho. Seu escalonamento é feito pelo planejador, que a partir das informações provenientes dos recursos disponíveis, pondera quais fluxos podem ser executados em paralelo e quais os melhores nós para tal execução.

Cada fluxo de trabalho é dividido em tarefas. Na composição destes fluxos, são definidos dados de entrada e saída. Estes dados mapeiam relações de dependência, que serão analisadas posteriormente pelo planejador. Estas tarefas podem ser implementadas de diferentes maneiras, explorando ou não os tipos de paralelismo. Existe ainda um banco de metadados que armazena informações sobre execuções anteriores, bem como as características dos nós disponíveis para execução. O planejador utiliza estas informações para criar um plano de execução que define qual tarefa será executada em qual nó. Com o plano de execução pronto, uma camada de execução é acionada e dispara as tarefas para seus respectivos nós. Após a execução de uma tarefa, cada nó envia informações sobre a utilização de seus recursos, que são armazenados e utilizados para refinar uma próxima execução.

Um fluxo de trabalho científico pode conter ciclos em sua estrutura. Os ciclos aumentam a complexidade em sistemas que incluam o tratamento dessa classe de problemas [10]. Este modelo não trata fluxos de trabalho que necessitem de uma especificação cíclica. Entretanto, uma das funcionalidades desenvolvidas, permite ao usuário especificar múltiplas entrada e saídas de dados (seção 5.3). Este recurso permite um encapsulamento de

operadores que necessitem de ciclos, unindo-os em um outro operador capaz de executar as operações dos anteriotes.

Como resultados apresenta-se uma análise dos planos gerados e do comportamento do sistema em um cenário de execução. Os experimentos descrevem três implementações que utilizam as principais características do modelo proposto.

Este documento está organizado em capítulos. O capítulo 2 apresenta uma revisão bibliográfica dos assuntos relacionados com o modelo proposto. O capítulo 3 apresenta-se os principais trabalhos utilizados como referência. No capítulo 4 é apresentado o modelo proposto. O capítulo 5 descreve os experimentos. Apresenta-se um capítulo com a conclusão e os trabalhos futuros em 6. Ao final de cada capítulo, apresenta-se uma seção de discussões, que descreve a relação dos tópicos abordados com o modelo proposto. O apêndice A apresenta a primeira modelagem do problema utilizando a ferramenta JSHOP. O apêndice B apresenta exemplos de códigos que representam as configurações utilizadas para o gerenciamento do sistema. O apêndice C apresenta exemplos de códigos gerados em *Planning Domain Definition Language* (PDDL) . O apêndice D apresenta uma descrição dos componentes do sistema. No apêndice E apresenta-se um arquivo de configuração do sistema PeerUnit. E finalmente no apêndice F apresenta-se exemplos de códigos fonte utilizados para a implementação do sistema.

CAPÍTULO 2

REVISÃO BIBLIOGRÁFICA

Neste capítulo discute-se os assuntos relacionados ao trabalho proposto. Na seção 2.1 estão descritos os conceitos de Fluxo de Trabalho Científico. A seção 2.2 discorre sobre Proveniência de Dados. A seção 2.3 introduz os conceitos de Grades e Computação em Nuvens. A seção 2.4 descreve as características das aplicações *Peer to Peer* (P2P) . A seção 2.5 apresenta as considerações sobre Planejamento. E finalmente, na seção 2.6 estão as considerações sobre o capítulo.

2.1 Fluxo de Trabalho Científico

Os experimentos científicos necessitam de processamento e analise de seus dados, essa tarefa vem se tornando complexa e pode envolver muitas etapas que necessitam coordenação e integração. Além disso, é necessário tratar a distribuição de dados a fim de se obter um tempo de execução viável destes fluxos de trabalho.

Para tratar os desafios dos experimentos científicos, utiliza-se sistemas que tem como objetivo o foco na coordenação, integração e distribuição de dados. Desta forma o sistema é capaz de executar experimentos que sejam estruturados de acordo com um modelo. Os experimentos científicos são modelados em fluxos de trabalho (*workflow*) científico.

Um fluxo de trabalho científico é o encapsulamento de informações que pode ser processado automaticamente em um fluxo de execução, gerenciado por um sistema dito sistema gerenciador de fluxos de trabalho científico. Estes sistemas transmitem documentos, tarefas ou informações de um participante para outro através de ações. Estas ações trabalham de forma coordenada de acordo com as regras estabelecidas entre os participantes formando um plano de execução [14].

O processo de execução de um fluxo de trabalho científico pode ser dividido em etapas, são elas: *composição*, *orquestração*, *mapeamento* e *execução* [14]. A figura 2.1 ilustra as

etapas para execução de fluxos de trabalho cientifico.

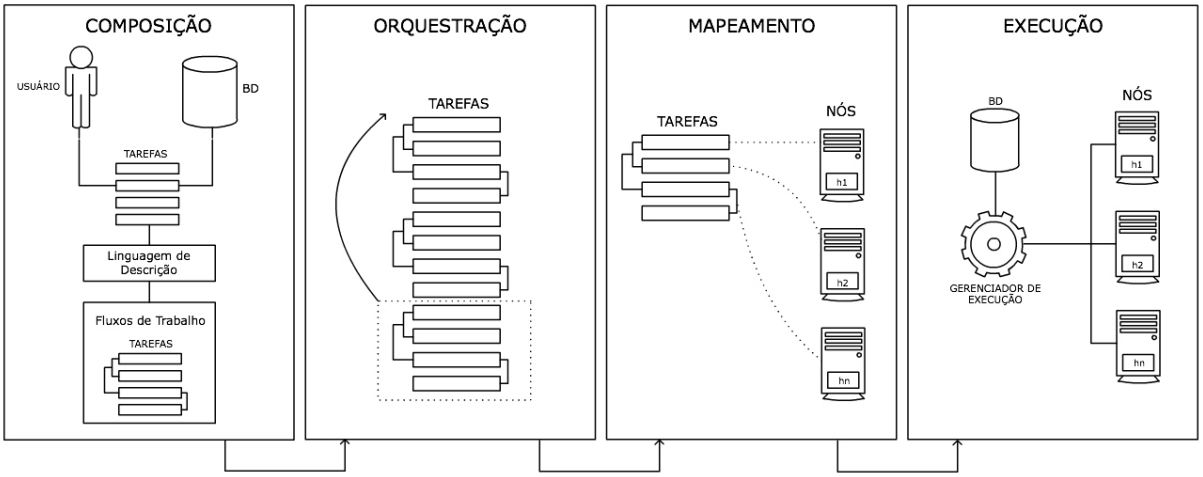

Figura 2.1: Etapas para execução de fluxos de trabalho científico

O processo de *composição* de um fluxo de trabalho científico está relacionado à descrição das tarefas que cada fluxo deverá executar, bem como a relação de dependência entre elas. Normalmente sua representação é feita através de grafos dirigidos, no qual os vértices representam as tarefas e as arestas suas dependências [14].

Existem diversas linguagens para especificação de fluxo de trabalho científico, cada qual com características distintas de semântica, tipo de aplicação, ambiente de execução ou representação *abstrata* ou *concreta*. Uma representação *abstrata* não descreve os detalhes do ambiente execução. Uma representação *concreta* descreve detalhes do ambiente de execução. Como exemplo de linguagens, apresenta-se: *Modeling Markup Language* (MoML) [3], *Scufl* [6], *BPEL* [1], *Virtual Data Language* (VDL) [13] e *DAX* [26]. MoML , *Scufl* e *DAX*, são linguagens especificamente desenvolvidas para o processamento de fluxos de trabalho. *BPEL* é uma linguagem para especificar ações em processos de negócios dentro de serviços *web*, adaptada para execução de fluxos de trabalho. *VDL* é uma linguagem utilizada para descrição de dados utilizada pelo *software* Chimera [18].

O processo de *orquestração* tem como objetivo a coordenação das tarefas que compõem um fluxo de trabalho científico. Outro objetivo é garantir que as dependências sejam atendidas.

A etapa de *mapeamento* tem como função o escalonamento entre tarefas e recursos,

para que um fluxo de trabalho seja executável no ambiente de execução. Essa etapa pode ser feita com a intervenção do usuário, ou automaticamente através de um componente específico. Em ambos os casos é necessária a obtenção de informações relativas aos recursos disponíveis.

A *execução* tem como função a alocação e execução das tarefas nos nós disponíveis, além de tratar a tolerância a falhas e adaptabilidade dos fluxos de trabalho. A tolerância a falhas deve prever a realocação das tarefas de forma automática caso um dos nós seja desconectado. A adaptabilidade diz respeito a modelagem do fluxo de trabalho durante a execução. Isso se deve pela necessidade da variação dos dados de entrada específica de cada aplicação.

O modelo de execução pode utilizar abordagens orientadas a dados, controle ou híbrido. Essas abordagens tratam as dependências entre tarefas. O modelo orientado a dados recebe os dados de entrada, aplica as operações e fornece o dado de saída como resultado ou como entrada para outra tarefa. Ao se trabalhar um sistema de controle, o gerenciador de fluxo de trabalho utiliza sinais que indicam o início ou término da execução de determinado conjunto de dados. Nesse caso os dados são transportados de alguma forma implícita, não especificada, fazendo com que o sistema controle apenas o fluxo no qual as tarefas são executadas. Os sistemas híbridos utilizam a junção das duas abordagens.

Algumas implementações estão detalhadas no capítulo 3.

2.2 Proveniência de Dados

A proveniência de dados é uma técnica de analise de dados que ajuda a determinar a derivação histórica de um dado ou processo, partindo de sua fonte original [32, 27]. O uso da proveniência pode auxiliar na análise da *qualidade de dados*, utilizado para estabelecer a qualidade e/ou confiabilidade dos dados baseado nos dados originais e suas transformações. Ainda é possível a utilização de rastreamento de dados pela *trilha de auditoria*, que busca determinar o uso de recursos e a detecção de erros na geração de dados. A *replicação* fornece informações detalhadas da proveniência possibilitando a

repetição da derivação de dados, para gerar uma possível replicação. Com a *atribuição* é possível estabelecer o direito do autor sobre a propriedade de um dado, e determinar sua responsabilidade em caso de dados incorretos. Com a técnica *informacional* possibilita-se a consulta baseada no histórico de metadados para a descoberta de novos dados [32].

No que diz respeito à representação da proveniência, diferentes técnicas podem ser usadas. As duas principais são *anotação* e *inversão*. A *anotação* diz respeito ao armazenamento de informações coletadas sobre os dados e os processos. A *inversão* faz um rastreamento de todo o processo de transformação dos dados, com intuito de traçar suas modificações [32].

A técnica de *anotação* gera uma grande quantidade de metadados. O volume de metadados pode ficar maior que o próprio armazenamento de dados caso o método de extração dos metadados seja detalhado. Por isso, a maneira de como a proveniência será armazenada é importante para a escalabilidade. Quanto a forma de armazenamento, muitos sistemas de proveniência utilizam a linguagem *eXtensible Markup Language* (XML) para representá-los [27].

Em [28], são abordados casos de uso que justificam a utilização de proveniência em um ambiente científico. Estes ilustram situações em que as técnicas de proveniência de dados resolveriam problemas recorrentes do ambiente científico, como por exemplo, um caso de uso que descreve uma situação na qual um cientista deseja registrar uma patente para os resultados de seu experimento, mas ao revisar o experimento um outro cientista percebe que o experimento possui restrições legais e não pode ser patenteado. De acordo com um histórico de experimentos referenciados o cientista que irá validar o experimento foca suas atenções as restrições legais, que passa desapercebido pelo cientista que deseja a patente.

2.3 Ambiente de Execução Distribuído

Um ambiente distribuído envolve a coordenação e o compartilhamento dos recursos computacionais, armazenamento de dados e recursos de rede.

Segundo [31], o modelo básico de uma grade computacional é geralmente composto

por um determinado número de nós, cada um contendo vários recursos computacionais. O agrupamento desses nós pode ser classificado como *homogêneos* ou *heterogêneos*. Quando uma grade é homogênea os recursos disponíveis apresentam as mesmas características, enquanto que em uma grade heterogênea os nós apresentam características diferentes.

Destacam-se quatro componentes básicos em um modelo de grade:

- *usuário*: responsável por alimentar a grade com tarefas a serem processadas;
- *gerenciador de recursos*: responsável por gerenciar a distribuição das tarefas para os nós, além de obter e utilizar informações do *serviço de informações da grade*;
- *serviço de informações da grade*: captura informações sobre a execução dos nós, que ajuda no processo de escalonamento;
- *recursos da grade*: reunião dos nós disponíveis para processamento.

Quando um usuário necessita executar determinada tarefa essa solicitação é enviada ao gerenciador de recursos, que com a ajuda do serviço de informação da grade, escalona e distribui as tarefas. Cada nó processa a parte que lhe foi designada e retorna a tarefa processada para o gerenciador de recursos. O gerenciador de recursos junta as tarefas processadas e devolve o resultado para o usuário.

A Computação em Nuvens (do inglês *Cloud Computing*) é um conceito de computação distribuída que surgiu no meio comercial para oferecer serviços e recursos na Internet específicos para a necessidade dos usuários [5]. Os aplicativos desenvolvidos para esse ambiente são desenvolvidos como um serviço. Isso é, para extrair a informação do recurso na nuvem, deve-se adaptar o ambiente a um modelo de requisição e resposta. Para executar o serviço, uma requisição com parâmetros específicos é enviada para a nuvem que processa as informações e retorna como resposta o processamento dos dados em questão.

A vantagem em utilizar os recursos como serviços está na instalação, manutenção e centralização do controle de versões. Dessa forma os usuários podem usar o serviço a qualquer hora, e de qualquer lugar.

Do ponto de vista dos recursos, a computação em nuvens pode oferecer três novos aspectos:

1. A possibilidade de recursos "infinitos". Na nuvem não existem limitações de *hardware*. Na teoria os recursos são flexíveis, limitados somente pelo provedor do recurso;

2. Usuários da nuvem podem requisitar processamento de acordo com sua necessidade, e expandir o poder de processamento caso seja necessário;

3. A capacidade de pagar pelos recursos por curtos prazos de utilização, uma ou duas horas, conforme o necessário.

2.4 Sistemas Ponto a Ponto

Sistemas P2P são sistemas distribuídos, que consistem em nós interconectados capazes de se auto-organizar em topologias de rede com o objetivo de compartilhar recursos, tais como: conteúdo, ciclos de processamento, armazenamento e largura de banda. Estes sistemas são capazes de se adaptar a falhas e a acomodação transitória de nós, sem a necessidade da intermediação ou suporte de um servidor centralizado [4]. Segundo [4], existem duas características que definem uma arquitetura P2P :

1. O compartilhamento de recursos de forma direta, sem a necessidade de um servidor centralizado. No entanto, sistemas que dependem da utilização de um servidor centralizado para seu funcionamento básico (por exemplo, para a manutenção de índices globais ou busca) se estendem a definição de P2P . Como os nós de uma rede P2P não podem confiar em um servidor central para a coordenação da troca de conteúdo, eles são obrigados a participar ativamente de forma independente e unilateral realizando tarefas como: procura de nós, localização ou cache de conteúdo, informações de roteamento, conexão ou desconexão de outros nós vizinhos, sistema de criptografia e verificação de conteúdo;

2. Sua capacidade de tratar a instabilidade e conectividade como regras, adaptando-se automaticamente às falhas de conexão. Bem como uma população transitória

de nós. Esta capacidade de auto-organização tolerante a falhas necessita de uma topologia de rede adaptativa, que irá mudar conforme a entrada ou saída de nós, a fim de manter a sua conectividade e desempenho.

Quanto à classificação proposta por [4], as aplicações podem ser divididas em diferentes categorias descritas a seguir.

As aplicações de *Comunicação e Colaboração* incluem sistemas que oferecem uma infra-estrutura para facilitar a comunicação direta e colaborativa entre computadores. Como exemplo estão as aplicações de troca de mensagens instantâneas e bate-papo.

Nas aplicações que envolvem *Computação Distribuída* estão os sistemas que utilizam o processamento distribuído dos nós. Essa técnica distribui entre os nós do sistema P2P pequenas tarefas para serem executadas, ao final do processamento executa-se um roteiro para coletar os resultados obtidos. Uma coordenação central é invariavelmente necessária, principalmente para dividir, distribuir as tarefas e coletar os resultados.

Com aplicações de *Suporte a Serviços de Internet* é possível utilizar o ambiente P2P para compor um serviço. Esse serviço é utilizado como um componente que trabalha no modelo de requisição-resposta.

Aplicações com *Sistemas de Banco de Dados* podem utilizar a arquitetura P2P para construção de bancos de dados distribuídos.

A *Distribuição de Conteúdo* é a classificação mais comum para aplicações P2P . Estas aplicações incluem sistemas e infra-estruturas feitos para o compartilhamento de mídia digital e outros dados entre os usuários.

O trabalho elaborado se enquadra na classificação *Computação Distribuída*, pois apresenta um sistema P2P como executor de tarefas de fluxos de trabalho científico.

2.5 Planejamento

Problemas mapeados do mundo real podem apresentar um alto nível de complexidade. É difícil representar algumas características, como por exemplo: representação temporal, eventos inesperados e principalmente como estará espaço de estados depois de determi-

nada modificação. Problemas que envolvam tais características estão em uma classe de complexidade que se tornam inviáveis com algoritmos convencionais [34].

O planejamento em IA é uma técnica que busca tratar esse nível de complexidade de forma independente do domínio do problema. Nessa abordagem, após o mapeamento detalhado do problema do mundo real, define-se um estado inicial conhecido e um estado final desejado. Para gerar um plano, elabora-se uma busca por um conjunto de ações já conhecidas, que modificam o cenário a cada execução. O desafio está em encontrar em todo o espaço de estados quais as ações e em que ordem devem ser invocadas para gerar um plano válido para o problema de planejamento [34].

O *STranford Research Institute Problem Solver* (STRIPS) [17] foi o precursor dos planejamentos modernos. Com essa abordagem definiu-se a forma com que um problema e um domínio de planejamento são representados, e que é mantida na maioria dos planejadores atuais. Um *estado* do problema é dado pelo conjunto de fatos verdadeiros em um determinado instante do plano. Um operador, ou ação, é representado por:

- *pré-condição*: são os requisitos necessários para que um determinado operador seja aplicado sobre um estado;
- *lista de adição*: são os fatos a serem adicionados ao estado depois que operador é executado;
- *lista de exclusões*: são os efeitos negativos, ou seja, o conjunto de fatos que será retirado do estado após a execução de um operador.

Um problema de planejamento é dado por dois conjuntos de fatos que representam o estado inicial e o estado final ou objetivo do problema, em um conjunto de operadores. Um plano, que é a solução para o problema de planejamento, é uma sequência de ações que quando aplicada sobre o estado inicial obtém o estado final do problema.

Quanto à classificação, os planejadores podem ser divididos em: *clássicos* ou *hierárquicos*. Os clássicos executam uma busca no espaço de estados com objetivo de encontrar um caminho (que representará um conjunto de ações) do estado inicial até o estado objetivo. Os hierárquicos utilizam uma técnica de decomposição de tarefas, dividindo uma tarefa

complexa (abstrata) em uma ou mais tarefas com menor grau de complexidade (primitiva). Esse mapeamento é feito através de métodos que descrevem possíveis reduções entre os operadores.

Outra classificação está relacionada ao encadeamento, o encadeamento define como se dará a busca no espaço de estados. Um planejador pode possuir *encadeamento para frente*, quando as ações retornadas partem do estado inicial, ou *encadeamento para traz*, quando as ações retornadas partem do estado final [21, 11].

2.5.1 Redes de Tarefas Hierárquicas

Os planejadores *hierárquicos* são uma evolução dos clássicos, e conseguem tratar uma gama de problemas com maior grau de complexidade pela inferência direta de informações que refinam o processo de busca. Basicamente, a principal característica da abordagem hierárquica está na utilização de ações capazes de resolver subobjetivos que formam um objetivo maior [29, 30]. Essa inferência de informações é feita através de *métodos*. Cada *método* é responsável por descrever quais ações ou subtarefas irão compor uma tarefa mais complexa. Se uma tarefa é primitiva, ela não pode ser subdividida em uma tarefa menor, pois é possível resolvê-la com as ações já descritas. O principal objetivo dessa abordagem é reduzir as tarefas complexas até que existam somente tarefas primitivas, obtendo assim o plano [30].

A maioria dos planejadores hierárquicos são independentes do domínio, mas os métodos, são específicos do domínio. Isso faz com que seja necessária a ação de um especialista para construir o domínio do problema, o que torna o planejador muito mais rápido.

JSHOP2 [23] é uma implementação de um planejador hierárquico desenvolvida na linguagem Java. Seu desenvolvimento foi inspirado no planejador SHOP2 [30]. O SHOP2 planeja as tarefas T na mesma ordem que serão executadas. Para isso, uma escolha não-determinística de uma tarefa $t \in T$ que não possui predecessores, resulta em dois casos:

1. se t é primitivo, então encontra-se uma ação a que resolve t de forma que as precondições são satisfeitas em um estado s, e aplica a em s. Se não existe essa ação,

então esse ramo do espaço de busca falha;

2. se a tarefa t é composta, então o planejador encontra de forma não-determinística um método m que irá decompor a tarefa t em subtarefas. Se não existe um método para tal decomposição, então esse ramo do espaço de busca falha.

Se existe uma solução que envolve m, então suas subtarefas farão parte de uma nova lista atualizada T' que é percorrida recursivamente até que todas as tarefas sejam primarias e resolvíveis com operadores.

As duas escolhas não-determinística presentes no algoritmo apresentado recaem inevitavelmente em um processo de busca exponencial [29].

No início da elaboração deste trabalho, os planejadores hierárquicos foram foco dos estudos por apresentarem uma estrutura de resolução bastante semelhante a um fluxo de trabalho, que também pode ser hierarquicamente dividido. No apêndice A apresenta-se um modelo inicial, construído com um planejador hieárquico. Entretanto notou-se que para extrair um plano paralelo do planejador em questão, seria necessário cálculos de pós-processamento, pois o planejador escolhido não era capaz de retornar planos paralelos. Por isso o *CRIKEY* foi o planejador escolhido, descrito na seção 2.5.4.

2.5.2 Planejadores Clássicos

Em 1998, Daniel S. Weld [34] fez um apanhado de duas principais técnicas que possibilitaram a retomada dos estudos em planejamento devido ao grande desempenho dos planejadores obtidos. Estes são exemplos de planejadores clássicos, e estão brevemente descritas a seguir.

A idéia do *Graphpan* [7] é a representação do espaço de estados através de um grafo. Nesse grafo existem dois tipos de vértices: *proposição* e *ação*. A construção do grafo é feita de modo que os vértices estejam separados por camadas. As camadas pares são formadas por vértices de proposição enquanto que as camadas ímpares são compostas por vértices de ações. A camada zero (inicial) é formada por proposições que representam os fatos do estado inicial do problema. Arestas ligam os vértices de proposições aos de ações,

cujas pré-condições mencionam tais proposições ou que representam os efeitos das ações.

Ações que estão na mesma camada, podem ter uma característica de exclusão mútua, se uma for escolhida a outra não pode ser, devido a conflitos entre as pre-condições e os efeitos destas ações. Nesse caso existem arestas que as conectam classificando-as como mutuamente exclusivos.

Para extrair a solução na estrutura definida, o planejador faz uma tentativa de gerar o plano com n passos. Logo, é necessária a representação do grafo até a camada $2n$, depois faz-se uma busca de encadeamento para traz, procurando um caminho que chegue ao estado inicial do problema. Se ao voltar pelo grafo, ações mutuamente exclusivas são escolhidas, aplica-se a técnica de *backtraking* até que todas as opções estejam esgotadas. Caso não se encontre um caminho com n passos, o algoritmo expande o grafo para a camada $2(n+1)$ e realiza o procedimento novamente.

Baseados no grafo de planos proposto por [7], alguns planejadores direcionam os estudos para buscas heurísticas. O planejador *Fast-Forward* (FF) [22], por exemplo, utiliza o grafo de planos e uma busca heurística baseada no *Heuristic Search Planning* (HSP) [8]. O FF realiza uma busca progressiva no espaço de estados utilizando a técnica de *subida de encosta reforçada*. Essa técnica pode ser definida como: partindo de um estado S, avalia-se todos os seus sucessores diretos S'. Se nenhum deles tem uma heurística melhor que S, procure pelos sucessores dos sucessores, e assim sucessivamente até encontrar um estado S' com heurística melhor que S. O planejador *CRIKEY* [21, 11] é inspirado no planejador FF e está detalhado na seção 2.5.4.

Satisfabilidade (SAT) é um problema da IA que procura uma valoração que torne uma fórmula lógica proposicional verdadeira. Inicialmente proposto por [24], um planejador baseado em SAT tem como entrada um problema de planejamento que gera uma fórmula lógica proposicional, que se for satisfeita por alguma valoração para suas variáveis proposicionais, implica na existência de um plano, que pode ser obtido a partir de tal valoração.

Ao traduzir o problema de planejamento para SAT , torna-se necessário um resolvedor SAT eficiente. Devido ao alto desempenho dos resolvedores SAT atuais, esta técnica de

planejamento ainda é a que apresenta um melhor desempenho [25].

Com o avanço dos estudos na área de planejamento foi proposta a unificação da descrição dos domínios e problemas em uma linguagem comum a vários planejadores, a linguagem PDDL .

2.5.3 *Planning Domain Definition Language*

A linguagem PDDL fornece uma base para a descrição de problemas de planejamento, permitindo que os modelos possam ser compartilhados de forma padronizada entre a comunidade científica. Em sua versão 2.1 a linguagem conta com definições para utilização de domínios que tratem ações com um tempo de duração, bem como a utilização de métricas que permitem procurar por um plano específico que busque maximizar ou minimizar o valor de uma variável, que pode ser incrementada ou decrementada dependendo da ação que é executada [19].

A linguagem PDDL é formada por:

- *objetos*: entidades representadas num problema de planejamento;
- *predicados*: propriedades dos objetos;
- *estado inicial*: é um conjunto de predicados sobre objetos que representam o estado do mundo quando o processo de planejamento é iniciado;
- *objetivo*: é o conjunto de predicados que deve ser verdadeiro no estado final gerado pelo plano;
- *ações/operadores*: meios de mudar o estado do mundo.

As especificações em PDDL são separadas em dois arquivos: *domínio* e *problema*. O *domínio* possui os predicados e as ações. O *problema* possui os objetos, o estado inicial e a especificação do objetivo.

Um arquivo de domínio tem a estrutura disposta no código 2.1 e um arquivo de problema tem sua estrutura exemplificada no código 2.2.

Código 2.1: Estrutura de um arquivo PDDL de domínio

```
(define (domain <domain name>)
    <PDDL code for predicates>
    <PDDL code for first action>
    [...]
    <PDDL code for last action>
)
```

Código 2.2: Estrutura de um arquivo PDDL de problema

```
(define (problem <problem name>)
    (:domain <domain name>)
    <PDDL code for objects>
    <PDDL code for initial state>
    <PDDL code for goal specification>
)
```

Para exemplificar a descrição de um problema na linguagem PDDL , apresenta-se um problema clássico da IA chamado *mundo dos blocos*. Esse problema consiste basicamente de blocos sobre uma mesa e um braço mecânico que movimenta os blocos. O objetivo é determinar um plano a partir de um estado inicial e um estado final. Os códigos 2.3 e 2.4 representam o domínio e um problema do problema do mundo dos blocos.

Código 2.3: Domínio exemplo: mundo de blocos

```
(define (domain BLOCKS)
(:requirements :strips)
(:predicates    (on ?x ?y)
                (ontable ?x)
                (clear ?x)
                (handempty)
                (holding ?x))

(:action    pick-up
            :parameters (?x)
            :precondition (and (block ?x) (clear ?x) (ontable ?x) (handempty))
            :effect
            (and (not (ontable ?x))
            (not (clear ?x))
            (not (handempty))
            (holding ?x)))

(:action    put-down
            :parameters (?x)
            :precondition (and (block ?x) (holding ?x))
            :effect
            (and (not (holding ?x))
            (clear ?x)
            (handempty)
            (ontable ?x)))

(:action    stack
            :parameters (?x ?y)
            :precondition (and (block ?x) (block ?y) (holding ?x) (clear ?y))
            :effect
            (and (not (holding ?x))
            (not (clear ?y))
            (clear ?x)
            (handempty)
            (on ?x ?y)))

(:action    unstack
            :parameters (?x ?y)
            :precondition (and (block ?x) (block ?y) (on ?x ?y) (clear ?x)
            (handempty))
            :effect
            (and (holding ?x)
            (clear ?y)
            (not (clear ?x))
            (not (handempty))
            (not (on ?x ?y)))))
```

Código 2.4: Problema exemplo: mundo de blocos

```
(define (problem BLOCKS-4-0)
(:domain BLOCKS)
(:objects D B A C )
(:INIT
(BLOCK D) (BLOCK B) (BLOCK A) (BLOCK C)
(CLEAR C) (CLEAR A) (CLEAR B) (CLEAR D)
(ONTABLE C) (ONTABLE A) (ONTABLE B) (ONTABLE D)
(HANDEMPTY))
(:goal (AND (ON D C) (ON C B) (ON B A))))
```

Na definição do domínio, são criados os predicados que serão utilizados para vali-

dar as possíveis ações. São elas: pegar um bloco, soltar um bloco, empilhar um bloco, desempilhar um bloco.

O problema é composto por 4 blocos A, B, C e D o estado inicial diz que todos os blocos estão desempilhados, o braço mecânico está vazio e os 4 blocos estão na mesa. O objetivo é ter C sobre D, B sobre C e A sobre B.

O plano gerado está transcrito no código 2.5.

Código 2.5: Plano gerado: mundo de blocos

```
1 0.01: (pick-up A)
2 0.02: (stack B A)
3 0.03: (pick-up C)
4 0.04: (stack C B)
5 0.05: (pick-up D)
6 0.06: (stack D C)
```

No apêndice C encontram-se exemplos de especificações de problemas na linguagem PDDL utilizados neste trabalho.

2.5.4 Crikey

O *CRIKEY* [21, 11] é um planejador que utiliza uma heurística de busca com encadeamento para frente, baseada na métrica FF [22], implementado na linguagem Java, em sua versão 1.4. Ele é completo mas não é ótimo (no tempo ou na métrica especificada). Ele vai, entretanto, fazer uma tentativa para minimizar o número de ações em um plano.

O planejador foi escrito para trabalhar com os modelos de métricas e tempo da linguagem PDDL 2.1 [19]. Ele é capaz de tratar ambos os aspectos temporais: ações durativas e recursos de métricas. Nos domínios PDDL : `:typing`, `:fluents`, e `:durative-actions`.

CRIKEY separa o planejamento do escalonamento das partes temporais do problema (no caso do planejamento temporal). A técnica utilizada busca detectar onde esses dois sub-problemas estão fortemente acoplados, para separá-los completamente. Depois que o planejador obtém um plano totalmente ordenado, aplica-se a rede temporal, que gera como saída o plano temporal.

Durante o planejamento a informação temporal é ignorada. A estratégia de pesquisa

é aplicada em subidas de encosta, ou seja, uma vez que o melhor estado é encontrado, a pesquisa continua daquele estado sem retrocesso. Se a busca por subida de encosta falha, retorna-se para o estado inicial, o que tecnicamente gera um novo plano. Assim como na técnica FF, somente as ações úteis são consideradas. Ações úteis são ações que aparecem na primeira camada do grafo de planejamento. [21].

A utilização do planejador é feita por um arquivo do tipo *.jar* que recebe como parâmetro três arquivos: *domínio*, *problema* e *saída*. Os arquivos de *domínio* e *problema* são construções definidas pela linguagem PDDL , e o arquivo de *saída* possui um padrão que dispõe as informações sobre a ordem de execução, a ação utilizada e as proposições que compõem essa ação. De acordo com o modelo proposto, a integração com o planejador foi feita através da chamada externa de um comando que gera o arquivo de saída, que é utilizado para extrair as devidas informações necessárias para dar seguimento ao fluxo de execução.

A utilização de métrica utilizada no modelo, diz respeito a procura por um plano que minimize o incremento de uma variável. Esse incremento é feito com valores diferentes para cada uma das ações do planejador, dessa forma, procura-se por um caminho que busque o menor custo, gerando assim um plano mais refinado e próximo da solução ótima.

O *CRIKEY* participou da competição *The 4th International Planning Competition* (IPC4) realizada em 2004, onde obteve bons resultados.

2.6 Considerações

Ao utilizar fluxos de trabalhos científicos, os usuários podem focar suas atenções em seus experimentos e deixar a complexidade de gerenciamento para o ambiente de execução. Atualmente, pela grande vantagem em se utilizar ambientes de nuvem, os projetos que utilizam esse ambiente tem destaque.

Como quase sempre o objetivo principal de experimento científico é a minimização do tempo de execução, a utilização de um coletor de informações sobre os recursos pode ser utilizado como um mecanismo de coleta de metadados. Com estas informações é possível traçar um histórico da taxa de ocupação dos recursos e utilizá-las para a obtenção de um

plano refinado.

Independente do ambiente de execução (uma grade ou nuvem) ao se utilizar uma aplicação baseada na estrutura P2P cada nó deve possuir as características para se manter de forma autônoma na rede. Para a adaptação de um modelo que utilize *Computação Distribuída* é necessária a modificação do ambiente para a utilização de um nó central responsável por coletar os resultados processados. Quanto ao armazenamento de informações sobre a ocupação dos nós, é necessário que cada nó seja indexado com um identificador. A utilização de um sistema P2P e os detalhes modificados são apresentados na seção 4.2.1.

Quanto às técnicas de planejamento, inicialmente optou-se pela utilização de uma planejador hierárquico. Isso pela similaridade da decomposição de um fluxo de trabalho e a decomposição de tarefas oferecida pelo planejador hierárquico. Por exemplo, um fluxo de trabalho pode ser definido em n subtarefas e cada subtarefa dividida em n_i subsubtarefas. Entretanto, depois de um modelo inicial testado (apêndice A), notou-se que para extrair um plano paralelo do planejador em questão, seria necessário cálculos de pós-processamento. Por permitir a execução de planos paralelos, ter sua implementação em Java e mostrar-se um planejador rápido, o *CRIKEY* foi adotado como planejador no modelo proposto. O *CRIKEY* ainda utiliza a linguagem PDDL como entrada para resolução de seus problemas.

A vantagem ao se utilizar a linguagem PDDL colaborou diretamente para o desacoplamento do planejador. Se a implementação seguisse com a utilização do *JSHOP*, todo o tradutor teria que ser re-escrito caso fosse necessária a utilização de outro planejador. Ao optar pela tradução para a linguagem PDDL , qualquer planejador que suporte a linguagem em sua versão 2.1 pode ser facilmente acoplado ao sistema. Aumentando assim a modularidade do sistema.

A escolha do planejamento para compor um plano de execução, tem potencialidade para fornecer tipos refinados de planos. A medida que se incrementa a possibilidade de parametrização na geração de um plano, planos mais específicos podem ser gerados. Como por exemplo, tentar traçar um plano que minimize ao invés do tempo de execução,

o consumo de energia. Isso é possível através da inferência de informações coletadas do próprio histórico de execuções do modelo. Por esse motivo, a criação de um modelo que utilize técnicas de planejamento tem potencialidade para não só escalonar de forma eficiente, mas também alterar o esquema de execução de acordo com a necessidade do usuário. A potencialidade da geração de planos de execução é discutida na seção 4.7.

CAPÍTULO 3

TRABALHOS RELACIONADOS

As seções seguintes apresentam as principais referências para a implementação do modelo. A seção 3.7, ao final do capítulo, apresenta a relação de cada trabalho com o modelo proposto, bem como uma comparação entre alguns trabalhos apresentados.

3.1 Ptolemy

Ambientes distribuídos tendem a ser heterogêneos, por serem compostos por subsistemas de diferentes características. Quando se desenvolvem aplicações para execução nesses ambientes, a linguagem deve se adaptar a essa heterogeneidade [16]. A proposta do *Ptolemy* é oferecer um modelo e um *framework* capaz de suportar tais características.

O *Ptolemy* implementa um modelo orientado a atores, no qual, cada ator é um bloco do sistema. Os atores são componentes concorrentes que se comunicam através de interfaces nomeadas de *ports*. Um ator é atômico, e deve estar na base da hierarquia do modelo. O ator pode ser composto e nesse caso ele contém outros atores. Uma porta pode ser de entrada, saída ou ambos. Os canais de comunicação são estabelecidos pela conexão dessas portas. Uma porta para um ator composto pode ter duas conexões, uma de entrada e outra de saída.

Uma implementação do modelo associada com uma composição de atores é chamada de domínio. Um domínio define a comunicação semântica e a execução da aplicação através dos atores. Isso é feito por dois componentes: receptor e diretor. Os receptores implementam o mecanismo de comunicação que estão nas portas de entrada, onde existe um receptor para cada canal de comunicação. Os atores quando em um domínio, adquirem um receptor específico do domínio, que controla a entrada específica de um fluxo corrente.

O diretor define como os atores serão executados, podendo escolher uma execução serial ou paralela. Em um fluxo de trabalho for composto por tarefas que podem ser

executadas em paralelo, se o diretor escolhido executa tarefas serialmente, as tarefas do fluxo serão executadas de forma serial.

O diretor e os receptores devem trabalhar de forma conjunta. O diretor também é o responsável por criar os receptores. Quando um ator é executado, o diretor dentro da composição dispara os subatores dentro do modelo de forma recursiva. Dessa maneira, o modelo criado pelo projeto *Ptolemy* é capaz de executar tarefas de maneira coordenada, utilizando os componentes descritos para executar os operadores do fluxo de trabalho descritos pelos atores sobre os dados necessários.

O *Ptolemy* é um modelo para execução de tarefas em um ambiente distribuído, por isso não é restrito a um ambiente específico, podendo ser adaptado para execução em grades ou nuvens.

3.2 Kepler

O *Kepler* é uma ferramenta que processa fluxos de trabalho científicos, baseado na estrutura definida pelo *Ptolemy*. Seu foco está na simplificação da criação e execução dos fluxos de trabalho, assim os cientistas podem desenvolver, executar, monitorar, e analisar os seus experimentos sem a necessidade de focar nos detalhes de implementação [3].

Com o *Kepler* os cientistas podem capturar os fluxos de trabalho em um formato que permite migração, arquivamento, versionamento e execução. Seus fluxos de trabalho podem ser descritos em XML utilizando o MoML [3]. Esse formato, representa um grafo, no qual os vértices representam atores e as arestas representam as dependências de dados entre os atores. O tipo de composição utilizada pelo sistema é concreta, pois o usuário precisa definir quais os recursos serão utilizados.

A composição dos fluxos de trabalho no *Kepler* é feita através de uma interface gráfica, que utiliza como base o software Virgil. Com essa interface o usuário é capaz de definir os recursos a serem utilizados, bem como a estrutura do fluxo de trabalho, que gera um arquivo no formato MoML .

Como características o *Kepler* apresenta:

- *prototipação de fluxos*: permite que os usuários definam um protótipo do fluxo de trabalho antes mesmo de uma implementação pronta para execução;
- *execução distribuída*: permite a utilização dos recursos oferecidos por ambientes distribuídos;
- *acesso a dados e consultas*: inclui atores que controlam os dados em uma base;

A execução pode ser distribuída de duas formas: por *threads* ou em um ambiente distribuído. Ao utilizar a paralelização por *threads*, um recurso específico da linguagem Java é acionado para distribuir as tarefas localmente. Ao utilizar um ambiente distribuído o sistema permite uma configuração para proveito dos recursos disponíveis no ambiente.

3.3 Pegasus

O sistema *Pegasus* consegue mapear fluxos de trabalhos complexos em um ambiente de grade. O sistema proposto utiliza uma composição abstrata de fluxo de trabalho. A interface de composição abstrada é baseada no software *Chimera* [18], que descreve os fluxos de trabalho na linguagem VDL [13]. A partir da composição abstrata o sistema faz um mapeamento para um modelo executável [13]. Os fluxos de trabalho abstratos descrevem os dados e os operadores aplicáveis. Ainda indicam as possíveis dependências entre os componentes do fluxo de trabalho.

Para mapear um fluxo de trabalho abstrato em concreto, deve-se buscar por:

- recursos disponíveis para executar as tarefas;
- os dados utilizados no fluxo de trabalho em questão;
- o componente adequado para realizar o processamento dos dados.

Para recuperar essas informações, uma consulta é feita em um banco de metadados que mantém informações sobre a grade. O *Pegasus* utiliza os nomes de arquivos referenciados na definição do fluxo de trabalho para consultar o *Globus Replica Location Service* (RLS)

que localiza réplicas dos dados necessários para o processamento. Para localizar o componente que irá processar os dados é feita uma consulta ao *Transformation Catalog* (TC) que retorna a localização física dos componentes de transformação, bem como as variáveis de ambiente necessárias para a execução do aplicativo. Logo após, sistema consulta *Globus Monitoring and Discovery Service* (MDS) para encontrar a informação necessária para o escalonamento das tarefas na grade.

Na etapa de mapeamento, busca-se a melhor forma de alocar os recursos para as tarefas. Para isso, o sistema utiliza metadados com informações sobre os recursos disponíveis, redução de fluxos de trabalho (quando resultados iguais já estejam computados), agrupamento de tarefas e registro de proveniência sobre os dados de entrada e saída. Com essas informações é gerado um fluxo de trabalho concreto, pronto para ser executado.

O processo de execução é feito sobre o *middleware Globus Toolkit*, que utiliza um escalonador de tarefas chamado *Condor-G*. Nesse sistema existe um gerenciador de fluxo de trabalho chamado *DAGMan*, que interpreta o fluxo de trabalho concreto gerado pelo *Pegasus* e faz a submissão das tarefas para os respectivos nós da grade.

3.4 SciCumulus

O *SciCumulus* é um ambiente mediador para distribuir, controlar e monitorar execuções paralelas de fluxos de trabalho científicos em um ambiente de nuvens. O sistema conta com a execução de fluxos de trabalho em um conjunto de máquinas virtuais [12]. O modelo é composto por três camadas:

- *área de trabalho*: despacha as atividades dos fluxos de trabalho para serem executadas no ambiente de nuvem, usando um sistema local para gerenciamento de fluxos de trabalho científico, tal como *VisTrails* [9], por exemplo;
- *distribuição*: gerencia a execução das atividades em um ambiente de nuvem;
- *execução*: executa os programas a partir dos fluxos de trabalho.

É possível a utilização de dois tipos de paralelismo, por *dados* e por *troca de parâmetros*. Para exemplificar os tipos de paralelismo adotados, supõe-se um fluxo de trabalho

científico é definido por F que é composto por um conjunto de atividades. F é uma quádrupla (A, P, D, S), onde:

- A é um conjunto de ações $\{a_1, a_2, \ldots, a_n\}$ que compõem um fluxo de trabalho;
- P é um conjunto de parâmetros $\{p_1, p_2, \ldots, p_n\}$ de F;
- D é um conjunto de dados $\{d_1, d_2, \ldots, d_n\}$ a serem consumidos;
- S é um conjunto de saída de dados $\{s_1, s_2, \ldots, s_n\}$.

O paralelismo de dados pode ser caracterizado pela execução simultânea de mais de uma atividade a_i que consome um subconjunto específico de D.

O paralelismo por troca de parâmetros pode ser definido como uma execução simultânea das atividades de a_i onde cada execução consome um subconjunto específico dos valores possíveis para os parâmetros P.

O *SciCumulus* foca seu desenvolvimento em um ambiente de computação em nuvens. O trabalho trata além das políticas de escalonamento e troca de informações entre os nós, a paralelização de dados sob diferentes abordagens.

Quanto a execução, utilizou-se um ambiente realístico de nuvem, proporcionando pelo sistema *CloudSim*.

3.5 Escalonador BPEL

Esta abordagem relaciona as dependências de dados entre as etapas dos fluxos de trabalhos e a sua utilização de recursos no momento da execução. O algoritmo de escalonamento simula uma combinação de modo a minimizar os recursos utilizados pelos fluxos de trabalho. Se essa combinação é possível, os recursos da nuvem são configurados automaticamente para maximizar o rendimento [15].

The Business Process Execution Language for Web Services (BPEL) é um serviço de composição para aplicações de negócios, mas é comumente modificado para tratar aplicações científicas. Ao contrário do cenário de negócios, ao se trabalhar com fluxos de trabalho científicos tem-se a necessidade de processamento massivo de dados. Por isso o

BPEL precisa prover uma série de recursos tais como: tolerância a falha, adaptação em tempo de execução e escalonamento de fluxos de dados.

O sistema seleciona automaticamente as máquinas com baixo processamento, disponibilizando-as em tempo de execução. Para lidar com cargas de pico, o sistema automaticamente inicia máquinas virtuais e implanta a pilha de serviço *web* sob demanda.

Para executar uma atribuição das etapas dos fluxos de trabalho para os nós disponíveis, o sistema precisa de informações sobre a quantidade de dados que serão transferidos entre as etapas dos fluxos de trabalhos e o tempo de execução esperado para cada execução. Existem duas formas de se adquirir tais informações:

1. monitorar a execução dos fluxos de trabalho e anotar os resultados;

2. deixar o desenvolvedor do fluxo de trabalho definir essas informações na descrição do fluxo de trabalho;

O sistema BEPEL possui uma representação de fluxo de dados por um grafo no qual os vértices correspondem as chamadas das atividades e as arestas representam a atribuição das operações do processo. Todo o processo de execução é mapeado nesse grafo.

Depois que o fluxo de trabalho está desenvolvido, deve ser executado pelo motor BEPEL. Em seguida é exposto como um serviço *web*. Quando o fluxo de trabalho é executado pela primeira vez, a máquina BEPEL constrói um grafo interno com a representação dos fluxos de dados e com isso, faz o escalonamento.

O algoritmo de escalonamento pode operar em todo o grafo que representa o fluxo de trabalho, ou somente em uma única tarefa do fluxo de trabalho. Para contornar a complexidade do grafo gerado pela expressividade do BEPEL, um algoritmo genético é aplicado. Esse algoritmo opera numa população aleatória e usa estratégias para seleção, *crossover* e mutações para gerar um escalonamento subótimo que é o melhor dentre a mutação de todas as gerações.

3.6 Inteligência Artificial e Fluxo de Execução

O trabalho [20] utiliza um ambiente de grades para criar um sistema gerenciador de fluxo de trabalhos baseado no sistema *Pegasus* [13] (seção 3.3). Além disso, apresenta um sistema de IA integrado com o ambiente distribuído, que aproveita as informações sobre as execuções. O sistema de IA gera fluxos de trabalho válidos e submete para execução no ambiente distribuído.

A geração do fluxo de trabalho e o sistema de mapeamento (*Pegasus*) integra um planejador no ambiente da grade. Na configuração do *Pegasus* o usuário informa os detalhes do resultado desejado. O sistema gera automaticamente os fluxos de trabalho selecionando as aplicações e componentes apropriados. Além disso é feita a atribuição dos recursos necessários para a execução. O fluxo de trabalho pode ser otimizado com base no tempo estimado de execução.

O planejador utiliza dados de saída mapeados como objetivos e os operadores como componentes da aplicação. Cada operador possui como parâmetro o nó em que a tarefa será executada. As pré-condições contém as restrições sobre os nós, as dependências de dados e os arquivos de entrada necessários. Com isso é possível a obtenção de um plano que também representa um fluxo de trabalho [20].

No trabalho utiliza-se o conceito de fluxos de trabalhos colaborativos, supondo que vários cientistas utilizem o mesmo cenário, as estratégias de execução podem ser reaproveitadas, minimizando a necessidade de execuções redundantes, e maximizando o aproveitamento dos metadados gerados por cada experimento.

3.7 Considerações

Um sistema de gerenciamento e execução de fluxo de trabalho científico, deve possuir um método para a descrição destes fluxos. Nesta descrição, são listadas informações relacionadas a execução de tarefas, por exemplo: quais os dados a serem trabalhados, quais operadores serão aplicados a estes dados, e qual a técnica de paralelização será aplicada no ambiente.

É importante que uma linguagem de descrição de fluxos de trabalho científico armazene parâmetros específicos das etapas de escalonamento de tarefas e execução. É indispensável, no processo de descrição dos fluxos, o acompanhamento de um usuário avançado destinado de desenvolver os operadores que atuarão sobre os dados. Isto porque é ele quem deve inserir algumas informações específicas sobre paralelismo de dados, coleta de metadados e resultados.

Os executores de fluxos de trabalho científico, em sua maioria, precisam da interferência direta do usuário, que ordena manualmente os fluxos de trabalho. É desejável que este sistema seja capaz de escalonar automaticamente as tarefas de um fluxo de trabalho científico, levando em consideração as dependências de dados. Ainda é importante que o sistema possa escalonar as tarefas levando em consideração além de um histórico de execuções de tarefas semelhantes, o estado atual dos recursos. Com isso é possível inferir critérios de otimização como por exemplo: tempo de execução, gasto de energia e minimização ou maximização de recursos computacionais.

Ao utilizar um sistema que execute fluxos de trabalho científicos a principal dificuldade está relacionada a como serão escalonadas as subtarefas que o compõem. Diante do conjunto de recursos disponíveis, escolher o nó mais adequado para a execução pode se tornar complexo quando esse conjunto de recursos é dinâmico. Isso acontece em ambientes de nuvem, no qual ocasionalmente o recurso se adapta à tarefa a ser executada e não ao contrário. No trabalho BEPEL [15] (seção 3.5), propõe-se um sistema que monitora a flexibilidade dos recursos em tempo de execução. Com estas informações são consideradas novas políticas de escalonamento, que ajudam a obter um plano mais adequado no momento da execução. É desejável que um coletor de informações monitore a camada de execução, dessa forma, mesmo que o ambiente seja flexível, garante-se que a política de escalonamento levará em consideração o estado atual dos recursos.

O *Pegasus* [13] (seção 3.3) propõe um sistema que utiliza uma busca nos históricos de execuções para refinar o plano de execução. Por isso é importante que um coletor de informações popule um banco de metadados. Esses metadados estão disponíveis para aplicação de técnicas de proveniência, que podem traçar custos de execuções anteriores

para refinar o processo de escalonamento.

Nos trabalhos *Ptolemy* [16] (seção 3.1) e *Kepler* [3] (seção 3.2), nota-se a utilização de diferentes atores, que são implementados de forma a executar ações diretas sobre os dados. É importante que um sistema gerenciador de fluxos de trabalho científico suporte a implementação da mesma operação sobre os dados de diferentes formas. Isto é, um operador pode transformar um determinado conjunto de dados de forma sequencial, paralela, ou até mesmo de forma específica para uma quantidade n de nós. Desta forma, é desejável que o sistema seja capaz de selecionar qual a melhor implementação do operador, analisando a situação dos recursos no momento da execução.

No trabalho *SciCumulus* [12] (seção 3.4) apresentam-se técnicas para aplicação de paralelismo sobre os dados ou parâmetros. É importante que um sistema que execute fluxos de trabalho científicos seja capaz de aplicar técnicas para paralelização de dados, bem como paralelização dos próprio fluxos de trabalho. Isso por que uma das principais características de um experimento científico é o grande volume de dados.

Uma característica desejável em sistemas que executam fluxo de trabalho científico é a modularização. Isso porque tais sistemas estão em constante adaptação para necessidades específicas. Assim é possível a substituição não somente do escalonador de tarefas, mas também do gerenciador de execução ou de outros componentes que compõem o sistema gerenciador de fluxos de trabalho. A modularização pode realizada através de componentes que exportam e importam arquivos XML, tornando-os uma aresta de comunicação entre os componentes.

O trabalho que envolve IA e fluxo de execução [20] (seção 3.6) é o principal trabalho relacionado. Utiliza como base o projeto *Pegasus* [13] (seção 3.3) e aprimora o escalonamento utilizando técnicas de planejamento para a distribuição das tarefas no ambiente de execução. O trabalho não descreve com exatidão a metodologia utilizada para a extração do plano da máquina de planejamento, focando em um modelo colaborativo genérico que funciona com o compartilhamento de metadados pela Internet.

Na tabela 3.1 apresenta-se um quadro comparativo entre os sistemas apresentados nesse capítulo.

Tabela 3.1: Comparação entre trabalhos

	Ambiente	Descrição Fluxos	Metadados	Escalonamento
Ptolemy	-	-	Não	Estrutura
Kepler	-	MoML	Não	Estrutura
Pegasus	Grade	VDL	Sim	Condor-G
SciCumulus	Nuvem	VisTrails	Não	VisTrails
BPEL	Nuvem	WS-BPEL	Não	Algoritmo Genético
IA e Fluxo de Trabalho	Grade	VDL	Sim	Planejamento
SWEP	-	Linguagem Própria	Sim	Planejamento

O projeto *Ptolemy* (seção 3.1), que deu origem ao *Kepler* (seção 3.2) não apresenta um ambiente distribuído definido, e pode ser adaptado para execução tanto em grades ou em um ambiente de nuvem. Como o *Ptolemy* descreve apenas um modelo de implementação, não existe uma linguagem para definição de fluxos de trabalho. O sistema *Smart Workflow Execution by Planning* (SWEP) é o modelo proposto neste trabalho, e utiliza as principais características positivas dos trabalhos apresentados. O SWEP não tem seu desenvolvimento direcionado para grades ou nuvens, pois utiliza uma aplicação baseada na estrutura P2P , assim, cada nó deve possuir as características para se manter de forma autônoma na rede, coordenado por um nó central. O escalonamento das tarefas dos fluxos de trabalho é dado por planos de execução construídos automaticamente a partir de metadados dos históricos de execução e do estado de disponibilidade dos recursos do ambiente. Utiliza-se ainda, o conceito de implementação modular, na qual cada componente pode ser substituído de acordo com a necessidade do experimento.

O modelo proposto é apresentado no capítulo 4.

CAPÍTULO 4

MODELO PROPOSTO - SWEP

Nesse capítulo é apresentado o modelo do SWEP (*Smart Workflow Execution by Planning*), um sistema capaz de utilizar técnicas de planejamento para a obtenção de um plano de execução em um ambiente distribuído que execute fluxos de trabalho científico. A seção 4.1 mostra uma visão geral do modelo, bem como a descrição do fluxo de dados que a compõe. A seção 4.2 descreve as necessidades da camada de execução. A seção 4.3 descreve em detalhes cada componente do sistema. A seção 4.4 mostra o fluxo de execução no modelo proposto. Na seção 4.5 descrevem-se os tipos de paralelismo sobre dados abordados no modelo. Na seção 4.6 discute-se o paralelismo sobre execução. A seção 4.7 descreve técnicas para definição de custos e mostra a potencialidade do modelo. A seção 4.8 descreve o método de tradução da linguagem de descrição dos fluxos de trabalho para um modelo PDDL que será executado pelo planejador. Finalmente, em 4.9 apresentam-se as considerações sobre o capítulo.

4.1 Visão Geral

Nesta seção são abordados os componentes que integram o modelo a partir de uma perspectiva de fluxo de processamento. Inicialmente, um usuário deve criar um arquivo de descrição dos fluxos de trabalho científicos. Um outro usuário chamado desenvolvedor, é responsável por escrever os operadores destes fluxos, bem como auxiliar em sua descrição. Cada fluxo de trabalho desse arquivo descreve, além de outras informações detalhadas a seguir, quais serão os dados a serem trabalhados e quais as operações sobre estes dados. Essas informações servem de entrada para para dois gerenciadores o *planejador* e o *executor*. O *planejador* tem como função a geração de um plano de execução que será utilizado pelo *executor*, que por sua vez, distribui as tarefas entre os recursos disponíveis. A figura 4.1 ilustra o fluxo de informações.

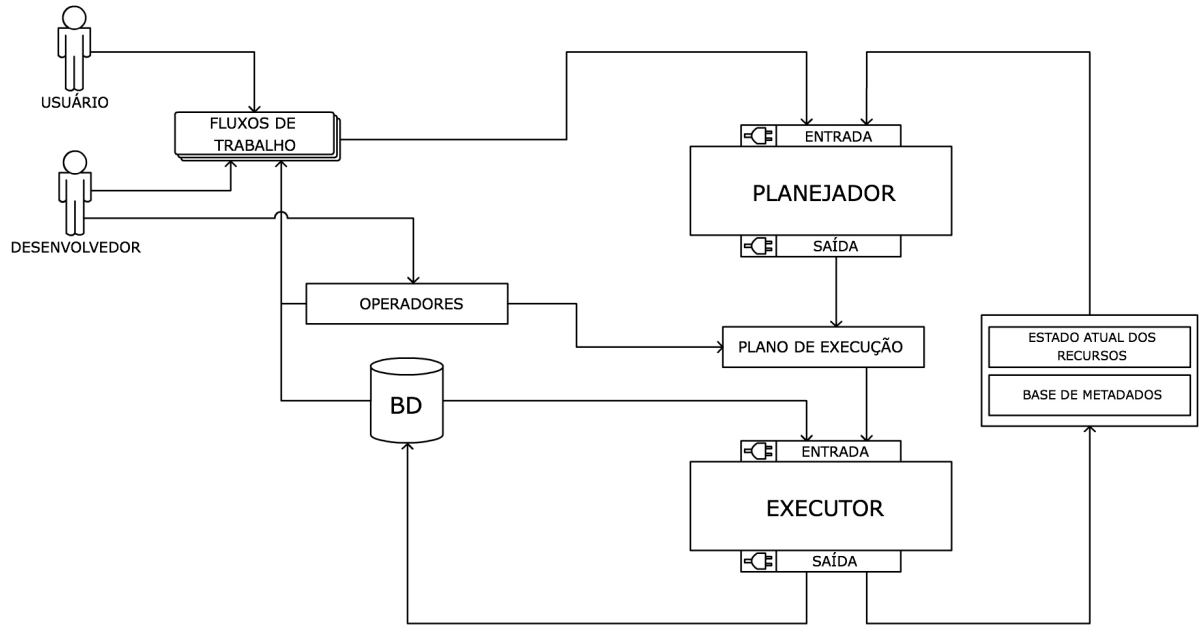

Figura 4.1: Visão geral do modelo

Para cada um dos gerenciadores existem *plugins* de entrada e saída. O planejador recebe como entrada três parâmetros: as *descrições dos fluxos de trabalho*, o *estado atual dos recursos* e a *base de metadados*. As *descrições dos fluxos de trabalho* armazenam as informações referentes aos *dados*, *operadores* entre outros detalhados a seguir. Os *dados* são as fontes a serem transformadas. Essas fontes devem possuir uma formatação específica para o reconhecimento no modelo e podem ser dados propriamente ditos, como por exemplo um vetor, uma matriz ou informações extraídas de um banco de dados. O apêndice B.3 exemplifica a estrutura de dados utilizada no modelo. Os *operadores* definem a regra de transformação de dados. Essa regra é definida na implementação de um método em Java, que segue um modelo previamente definido. O operador utiliza como fonte direta os dados do sistema e gera novos dados.

Nos apêndices F.3.1 e F.3.2, estão transcritos exemplos de códigos que representam a classe de implementação e um operador de soma. São apresentadas duas implementações da operação soma. A primeira, sequencial, pode ser observada entre as linhas 86 e 104. Uma implementação paralela pode ser analisada entre as linhas 129 e 144.

O *estado atual dos recursos* obtém informações sobre os recursos. Por exemplo a sua taxa de ocupação, ou ainda a quantidade de memória utilizada. Estas informações são

consideradas na geração do plano de execução.

Ainda como entrada do planejador, utiliza-se uma *base de metadados*, que armazena um histórico de execuções. Essas informações também auxiliam no processo de escalonamento, feito pelo planejador.

Com todas essas entradas, o planejador gera um plano com regras que irão definir qual tarefa será executada, em qual nó e qual a ordem dessa execução. Esse plano é chamado de *plano de execução*. No apêndice F.2 transcreve-se um plano de execução gerado automaticamente pelo sistema.

O *gerenciador de execução* recebe como entrada um *plano de execução* que irá definir o roteiro de execução, bem como os *dados* para a aplicação dos *operadores*, já definidos no plano de execução. Durante a execução, existe um mecanismo de coleta de metadados. Este permite repopular o banco de metadados com o *estado atual dos recursos*. O *gerenciador de execução* gera ainda os dados de saída, que são os dados depois da aplicação dos operadores.

Os trabalhos [20, 33, 15, 12, 13, 3, 16] influenciaram para a separação do modelo em módulos. Propõe-se uma modularização que desacople os ambientes de escalonamento e de execução. Desta forma, é possível a substituição não somente do planejador, mas de toda a máquina de escalonamento ou do gerenciador de execução. A modularização foi realizada através de componentes que exportam e importam arquivos XML.

No modelo, utilizou-se como planejador o *Crikey* [21, 11] (detalhado na seção 2.5.4). Como gerenciador de execução apresenta-se uma adaptação do trabalho *PeerUnit* [2] apresentado na seção 4.2.

Os *plugins* utilizados para compor o modelo estão detalhados na seção 4.3.

4.2 Gerenciador de Execução

O modelo proposto utiliza uma camada para distribuir as tarefas de forma coordenada. O *gerenciador de execução* recebe como entrada um *plano de execução* gerado pelo planejador. Esse *plano de execução* é dado por um conjunto de tarefas: $P = \{s_1, s_2, \ldots, s_n\}$. Cada tarefa é uma tripla formada por: $s_i = \{O, L, H\}$, onde:

- O é um número inteiro que define a ordem que a tarefa será executada. Assim, é possível definir que a tarefa s_i seja executada em uma ordem específica na linha de tempo da execução. Nessa definição é possível alocar mais de uma tarefa para uma determinada ordem. Dessa forma, a execução poderá ocorrer paralelamente;
- L define o tempo de execução estimado da tarefa em segundos;
- H é a identificação do nó que executará a tarefa.

Cada nó, identificado por H, deve ser único para o todas as execuções no ambiente. Essa característica é importante porque um sistema coletor de metadados deve armazenar uma base as informações referentes as execuções de um nó. Esta base de metadados possui informações que ajudam na política de escalonamento feita pelo planejador. Estas características serão detalhadas na seção 4.3.

O *gerenciador de execução* deve estar preparado para executar em um ambiente paralelo e escalável, pois a execução de fluxos de trabalho científicos exige grande quantidade de processamento. Portanto, é proposta uma adaptação do sistema *PeerUnit* [2].

4.2.1 PeerUnit

O sistema *PeerUnit* [2] foi concebido inicialmente para realizar testes em sistemas *peer to peer*, a fim de garantir as seguintes propriedades:

- *funcionalidade*: garante que o sistema irá responder como o esperado;
- *escalabilidade*: garante que a funcionalidade desenvolvida poderá ser expandida para um ambiente com vários *peers*;
- *volatilidade*: garante que mesmo com a entrada ou a saída de *peers* o sistema continuará funcionando como o esperado.

A estrutura *PeerUnit* basicamente acopla a todos os *peers* um código escrito na linguagem Java. Nele são definidas instruções de como tais elementos devem se comportar. Ainda é possível (através de anotações especiais) indicar qual método será executado por

qual *peer* e qual será a ordem dessa execução. Além disso, é definido um tempo limite que se superado, retorna que o *peer* não foi capaz de executar o método com sucesso.

O código 4.1 representa um trecho de um caso de uso, no qual exemplifica-se a utilização do sistema *PeerUnit*.

Código 4.1: Caso de uso de exemplo

```
1  @Test(range = "*", timeout = 100, order = 1)
2  public void join(){
3      peer.join();
4  }
5  @Test(range = 0, timeout = 100, order = 2)
6  public void put(){
7      peer.put(expectedKey,expected);
8  }
9  @Test(range = "1-3", timeout = 100, order = 3)
10 public void retrieve(){
11     actual = peer.get(expectedKey);
12 }
13 @Test(range = "1-3", timeout = 100, order = 4)
14 public void assertRetrieve(){
15     assert expected.equals(actual);
16 }
```

Nota-se que, antes da declaração do método, é definida uma linha de anotação iniciada pelo caractere @. A anotação é composta dos seguintes elementos:

- *range*: especifica quais serão os *peers* atingidos. Pode ser uma samblagem que representa somente um *peer*, como por exemplo: “3”, uma faixa de atuação “1-3”, elementos específicos “1,3,4” ou todos os *peers* “*”;
- *order*: controla qual será a ordem de execução do método após a anotação;
- *timeout*: estabelece um tempo limite para execução.

No código, 4.1 o método *join()* inicia todos os *peers*. O método *put()* faz com que o *peer* de identificação 0 insira um dado em uma variável global, visível para todos os outros *peers*. O método *retrive()* captura o dado inserido pelo *peer* 0. Por fim, o método *assertRetrive()* faz uma verificação se o valor inserido foi recuperado de forma correta.

As configurações para a execução do *PeerUnit*, como por exemplo a quantidade de *peers*, o ip do *peer* servidor, a porta de comunicação ou o diretório de *logs* estão definidas em um arquivo padrão chamado *peerunit.properties*. No apêndice E está transcrito um exemplo de arquivo de configuração utilizado neste trabalho.

Algumas adaptações foram necessárias para transformar o sistema *PeerUnit* em um sistema executor de tarefas. Para isso, cada caso de uso acopla dois componentes: um *coletor de informações* e um *compilado de operadores*. O *coletor de informações* tem a função de monitorar os detalhes das execuções de um *peer*. Ao final da execução são armazenadas informações como: quais tarefas foram executas e qual o tempo de execução de cada tarefa. Este componente também tem a função de armazenar o estado atual do recurso, que engloba por exemplo, a taxa de ocupação do nó ou a quantidade de memória utilizada. O *compilado de operadores* representa o conjunto dos operadores a serem utilizados nos fluxos de trabalho científicos. Estes operadores devem ser acoplados aos *peer* permitindo que qualquer um deles possa executar determinada operação de forma autônoma. Maiores detalhes destes componentes estão descritos na seção 4.3.

Além disso, é necessário que o sistema de execução esteja adaptado para garantir que os nós possuam sempre a mesma identificação. Por exemplo, se um recurso r possui a identificação 1, deve-se garantir que em todas as próximas execuções a identificação de r será 1. Para que isso seja feito, assume-se que a ordem com que os *peers* sejam inseridos seja sempre a mesma.

4.3 Visão Detalhada do Modelo

Nesta seção apresenta-se um detalhamento dos componentes do modelo proposto. Com base no diagrama apresentado pela figura 4.2, nota-se a relação dos usuários com os módulos e componentes do modelo. Dentre os módulos estão: planejador, operadores, execução e coletor de informações. Os módulos, componentes, usuários e suas respectivas relações estão detalhados nas próximas seções.

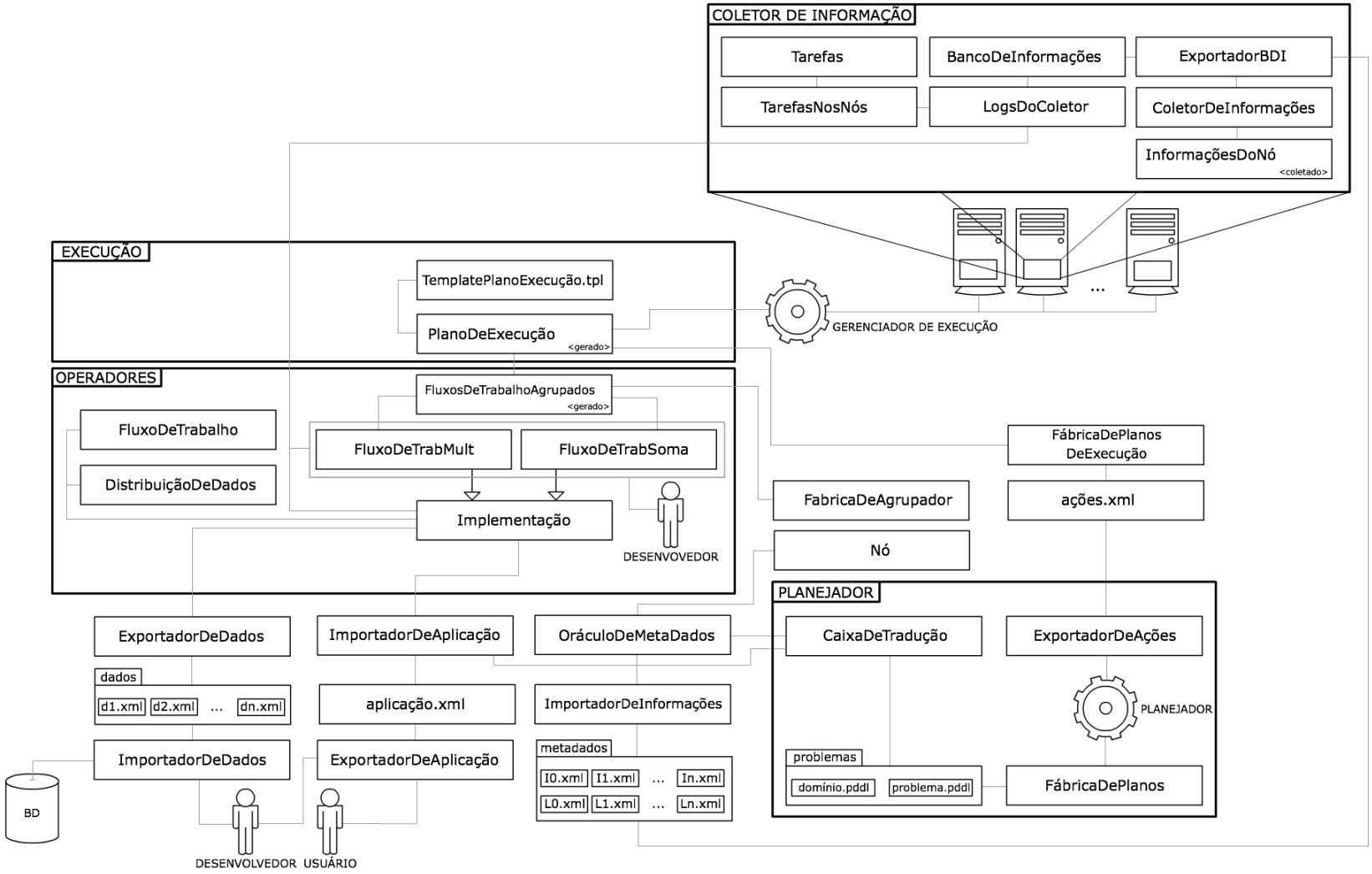

Figura 4.2: Visão detalhada do modelo

4.3.1 Usuários

São dois usuários que atuam no modelo: *usuário* e *desenvolvedor*. O *usuário* é o agente que utilizará o sistema depois que sua estrutura estiver devidamente configurada. Ele deve inserir informações de entrada para a composição dos fluxos de trabalho. Essas informações são referentes aos dados de entrada e saída, bem como o tipo de processamento de dados (sequencial ou paralelo), descrição do operador, entre outros. Todos os atributos estão listados no arquivo XML de exemplo no apêndice B.1. O *desenvolvedor* tem como função a escrita dos operadores que atuarão sobre os dados. Ele ainda deve inserir informações mais específicas na geração da aplicação, tais como: nome do método que executará a operação, a distribuição dos dados e o tempo limite para que a ação seja executada. Outra função designada ao *desenvolvedor* é o controle da importação de dados. Para isso, utiliza-se o componente *ImportadorDeDados* que conta com uma estrutura para a importação dos dados no formato aceito pelo modelo.

4.3.2 Planejador

No modelo, a subdivisão *planejador* compreende os componentes: *CaixaDeTradução*, *domínio.pddl* e *problema.pddl*, *FábricaDePlanos*, *ExportadorDeAções* e o próprio planejador.

As informações que os usuários inserem sobre os fluxos de trabalho, são capturadas e exportadas pelo componente *ExportadorDeAplicação*, que gera um arquivo no formato XML , que está transcrito no apêndice B.1. Após a interferência do *desenvolvedor* na importação dos dados, é criada uma coleção de arquivos XML que representam os dados. Estes dados são formatados de acordo com o padrão transcrito no apêndice B.3.

A próxima etapa está relacionada a transformação das informações capturadas em arquivos no formato PDDL que darão origem ao *plano de execução*. Para isso utiliza-se o componente *CaixaDeTradução* que tem a função de capturar as informações da aplicação, bem como a extração das informações sobre os nós. Esta informação é oferecida pelo componente *OráculoDeMetaDados* que tem como função analisar em um banco de metadados a situação corrente dos recursos disponíveis. No apêndice F.4 transcreve-se

o código do componente *CaixaDeTradução*. Em nossa implementação a informação que o *OráculoDeMetaDados* está preparado para responder é se um nó está disponível ou parcialmente disponível (quando esse nó já está executando alguma tarefa).

Com a geração dos arquivos de domínio e problema, é possível invocar o planejador para a obtenção do plano. Por trabalhar com a linguagem PDDL , torna-se possível a modularização da fase de planejamento com a utilização de outros planejadores. O componente responsável por invocar o planejador e gerar o plano é a *FábricaDePlanos*.

Após a geração do plano, o componente *ExportadorDeAções* é acionado para gerar um arquivo XML que contém as informações necessárias para a geração do plano de execução. Tais como: nome, nó que executará a ação, ordem de execução, tempo limite e operador. Outra modularidade estabelecida nesse momento é a alteração do componente de escalonamento. Com o *ExportadorDeAções*, qualquer escalonador pode substituir o planejador, desde que exporte o plano de execução de acordo com o padrão estabelecido. O apêndice B.2 transcreve um arquivo XML com a descrição de um conjunto de ações.

O próximo passo é a geração do plano. Isso é feito pelo componente *FábricaDePlanosDeExecução*. Este recebe como entrada o arquivo gerado pelo *ExportadorDeAções* e gera um *plano de execução* com base em um *template* pré-definido e as ações importadas. Esse plano está pronto para ser executado pelo *gerenciador de execução*.

4.3.3 Operadores

A subdivisão de *operadores* compreende os componentes: *FluxoDeTrabalho*, *DistribuiçãoDeDados*, *Implementação*, uma coleção de implementações de fluxos de trabalho e um arquivo com invocações para os fluxos de trabalhos desenvolvidos.

O componente *Implementação* é a base para a implementação de qualquer operador. Ele define regras de inserção e recuperação de variáveis globais (visíveis para todos os nós no ambiente), bem como a distribuição dos dados no modelo *direto*. Define ainda, os métodos de agrupamento e distribuição dos resultados obtidos. O componente de *Implementação* está transcrito no apêndice F.3.1.

No componente de *Implementação* são agregados dois componentes para controle das

informações extraídas do sistema: *FluxoDeTrabalho* e *DistribuiçãoDeDados*. O *FluxoDeTrabalho* é uma classe que representa um fluxo de trabalho no modelo. O componente *DistribuiçãoDeDados* contém as regras para a paralelização direta. Os modelos de paralelização serão detalhados na seção 4.5.

Cada componente deve implementar as funcionalidades para a transformação dos dados em questão. Antes da execução do sistema, um componente *FábricaDeAgrupador* é invocado, esse componente gera automaticamente uma compilação das funcionalidades (métodos) de transformação de dados e os reúne em um único arquivo chamado *FluxosDeTrabalhoAgrupados*. Este é utilizado diretamente no plano de execução para chamada dos operadores.

Os operadores devem estar preparados para suportar múltiplos conjuntos de dados de entrada e saída. Para isso é necessário que o método de tradução esteja preparado para tratar tarefas que recebem ou devolvem 1 ou mais conjuntos de dados. O método utilizado para tratar essa característica está descrito na seção 4.8.

A utilização dos operadores no modelo foi baseada nos trabalhos *Ptolemy* [16] (seção 3.1) e *Kepler* [3] (seção 3.2). Desta forma é possível implementações diferentes para um mesmo operador. Na implementação atual ainda é necessária a intervenção do desenvolvedor para estabelecer qual a versão do operador deverá ser utilizada.

4.3.4 Execução

A subdivisão de execução é formada pelos componentes: *TemplatePlanoExecução.tpl*, *PlanoDeExecução* e a invocação do sistema de gerenciamento de execução.

O arquivo *TemplatePlanoExecução.tpl* contém um formato pré-estabelecido para a criação dinâmica dos planos de execução. Nesse formato estão definidas as importações necessárias da linguagem Java, a invocação do coletor de informações ao fim do plano de execução e alguns métodos que se repetem em todas as execuções.

O *PlanoDeExecução* é um arquivo gerado automaticamente pelo sistema, de acordo com o componente *FábricaDePlanosDeExecução*. Esse arquivo utiliza o o *TemplatePlanoExecução.tpl* e preenche a área customizável do plano com as ações geradas pelo pla-

nejador.

4.3.5 Coletor de Informações

A subdivisão *Coletor de Informações* é formada pelos componentes: *Tarefas*, *TarefasNosNós*, *BancoDeInformações*, *LogsDoColetor*, *ExportadorBDI* e *ColetorDeInformações*.

O *Coletor de Informações* é um componente agregado a todos os nós do ambiente distribuído. Baseado nas técnicas apresentadas pelos trabalhos *BPEL* [15] (seção 3.5) e *Pegasus* [13] (seção 3.3), este tem como função a captura de informações referentes a execução. Essas informações são armazenada no formato XML em um banco de metadados. O apêndice B.4 mostra exemplos das informações armazenadas.

O componente *ColetorDeInformações* obtém as informações sobre o estado atual do nó. Além disso, informa qual foi a tarefa executada. Para isso, utiliza os componentes: *LogsDoColetor*, *TarefasNosNós* e *Tarefas*. Para que seja possível obter a informação de qual tarefa é executada em qual nó. Ao acionar cada operador o componente *LogsDoColetor* grava um *log* que armazena o início e o fim da operação, além do tempo de execução. O componente *BancoDeInformações* é responsável por manter as informações organizadas em um banco de metadados. Esse componente permite consultas específicas aos arquivos XML armazenados. O *ExportadorBDI* é o componente usado como aresta para exportar as informações em um formato específico utilizado no modelo.

Uma base de metadados específica é designada para armazenar os *logs* de execução de tarefa dos nós. Um exemplo desse arquivo está transcrito no código B.6 no apêndice B.4. Esse é o arquivo consultado pelo componente *OráculoDeMetaDados* para a extração do estado dos nós.

4.4 Fluxo de Execução

A figura 4.3 mostra o fluxo de execução do modelo proposto.

O componente principal *PreparaçãoDoAmbiente* tem como função coordenar a ordem de chamada dos componentes do modelo. O primeiro componente a ser invocado é o

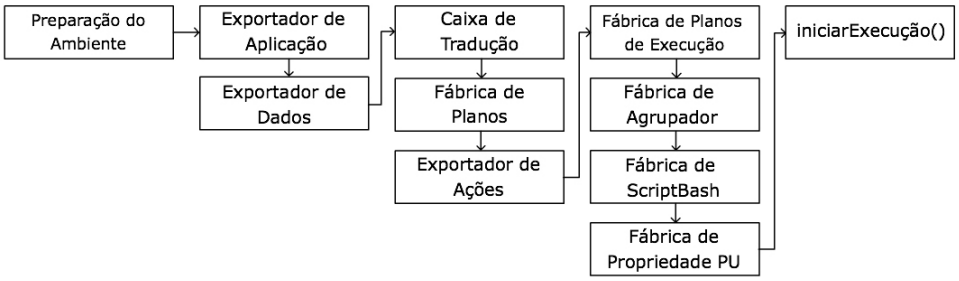

Figura 4.3: Fluxo de execução do modelo

ExportadorDeAplicação que exporta as informações principais do aplicativo. Em seguida, o componente *ImportadorDeDados* é invocado com o objetivo de se obter as fontes de dados do sistema. Em seguida, dá-se início a tradução das informações do sistema para os arquivos PDDL. Com os arquivos gerados, o plano é gerado pelo componente *FábricaDePlanos* que imediatamente aciona o componente *ExportadorDeAções*, que converte o plano em um formato XML aceito para geração do plano de execução. O componente *FábricaDePlanosDeExecução* utiliza as ações exportadas para geração do *plano de execução*.

Com o plano de execução pronto, o componente *FluxosDeTrabalhoAgrupados* é invocado para gerar o *FábricaDeAgrupador* que reúne todas as funcionalidades dos operadores. Em seguida o componente *FábricaDeScriptBash* prepara um roteiro para execução dos comandos para a execução do ambiente. O componente *FábricaDePropriedadesPU* altera as propriedades do *gerenciador de execução* (essencialmente o número de nós). Um método *iniciarExecução()* é invocado, este é responsável por compilar os arquivos gerando uma versão atualizada com o novo *plano de execução*. Por fim, o roteiro é invocado para a execução do ambiente.

Baseado no trabalho *SciCumulus* [12] (seção 3.4) são implementados dois tipos de paralelismo: o *paralelismo sobre dados* (seção 4.5) e o *paralelismo sobre execução* (seção 4.6).

4.5 Paralelismo sobre Dados

O paralelismo sobre os dados diz respeito a divisão do processamento de um único fluxo de trabalho. Dessa maneira, dois ou mais nós podem ser designados para aplicar os operadores sobre o conjunto de dados que compõe um fluxo de trabalho. A primeira implementação abordada no modelo diz respeito a uma paralelização direta, que necessita da intervenção do *usuário*. O segundo método de paralelização utiliza a técnica de planejamento para estipular a divisão dos dados entre os nós disponíveis.

4.5.1 Paralelização Direta

Ao definir um fluxo de trabalho, o *usuário* tem a opção de marcar o processamento como paralelo. Ao escolher essa opção é necessário que se informe uma das opções: tamanho dos blocos de dados (d) ou número de nós disponíveis para o processamento (n). Com uma dessas informações, é possível calcular a segunda:

$$n = \lceil \frac{t}{d} \rceil \tag{4.1}$$

Onde:

- t representa um número inteiro com o tamanho dos dados a serem executados.

Caso a divisão não seja exata, o último nó se encarrega de executar os dados restantes.

Nesse tipo de paralelização é necessária a intervenção direta do *desenvolvedor*, que deve adequar a distribuição dos dados. Por exemplo, se a operação sobre um fluxo de trabalho é a soma de uma matriz, fica a cargo do desenvolvedor estipular se a divisão pelos nós será feita por linhas ou colunas.

A operação de junção dos dados é implementada de acordo com o padrão estipulado em um *template* de fluxo de trabalho.

A definição do paralelismo direto pode ser analisado na linha 48 do código B.1 no apêndice B.1. Um exemplo da implementação paralela, pode ser observado entre as

linhas 129 e 145 do código F.4, bem como o método para junção de dados descrito entre as linhas 150 e 174 do mesmo código, que está transcrito no apêndice F.3.

4.5.2 Paralelização por Plano

Nesse tipo de distribuição, o número de nós que irá atuar sobre os dados de um fluxo de trabalho não é fixo. De acordo com as informações obtidas no *OráculoDeMetadados*, o planejador deve ser capaz de definir um número adequado para a execução paralela de um conjunto de dados. Para que isso seja possível, é necessária a criação de um operador flexível, que aceite como parâmetro quais serão os nós utilizados.

Antes de iniciar o processo de planejamento, uma consulta ao *OráculoDeMetadados* retorna uma lista dos recursos e suas respectivas informações quanto a sua *taxa de ocupação*. Essa taxa de ocupação é representada por um número percentual, que indica o quanto o nó está ocupado. Além disso outra consulta ao banco de metadados retorna uma *lista de custos* das tarefas baseadas nos históricos de execuções.

Para popular a lista de custos, sugere-se a seguinte estratégia. Seja T uma nova tarefa a ser executada. Se existe um conjunto de tarefas $T_1, T_2, \ldots, T_n$ no histórico de execução exatamente igual a T, assume-se que o custo de T é uma média dos custos de $T_1, T_2, \ldots, T_n$. Caso não se encontre nenhuma tarefa igual, assume-se certa prioridade na execução dessa tarefa. Desta forma, popula-se o banco de metadados com seu custo, auxiliando em futuras execuções.

Com a *taxa de ocupação* e a *lista de custos*, o planejador é capaz de expandir um grafo com todas possibilidades, utilizando o cruzamento das combinações para selecionar a quantidade de recursos, bem como quais serão utilizados para a realização de uma tarefa em específico.

Na paralelização por planos é necessária uma adaptação na tradução para a geração dos arquivos PDDL. O problema deve levar em consideração duas listas extraídas do banco de metadados pelo *OráculoDeMetadados*. A primeira com a taxa de ocupação de cada nó e a segunda com os respectivos custos de uma tarefa. Os operadores devem receber como parâmetro os nós a serem alocados, suas respectivas taxas de ocupação e o custo da

tarefa. Com uma métrica de redução de custos dos operadores, garante-se que uma tarefa com um custo mais elevado vai ser alocada em um conjunto nós com maior capacidade de processamento.

4.6 Paralelismo sobre Execução

Além do paralelismo sobre os dados de um fluxo de trabalho, também é explorado o paralelismo com relação a execução de múltiplos operadores de fluxos. Este nível de paralelismo está implícito ao gerar um plano paralelo que pondera a dependência dos dados que compõem as tarefas. Desta forma garante-se que além de uma ordem correta de dependência também serão gerados planos com execuções em paralelo.

Na figura 4.4 ilustra-se um exemplo de paralelismo sobre execução.

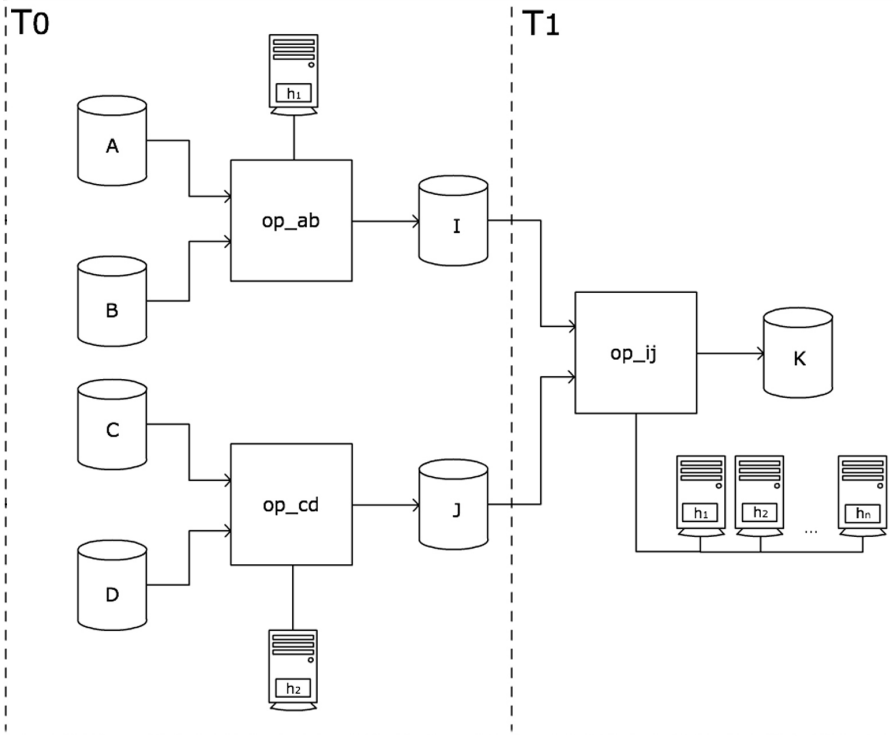

Figura 4.4: Exemplo de paralelização e dependência de dados

Nota-se dois instantes de tempo T_0 e T_1. No momento T_0 são utilizados dois nós que executam em paralelo op_{ab} e op_{cd} gerando como resultado as entradas para o fluxo op_{ij} que é executado no momento T_1. Essa execução utiliza a técnica de *paralelismo sobre execução*. Em T_0 cada nó executa sua tarefa sequencialmente, e gera um resultado. Em

T_1 utiliza-se uma coleção de n nós que executam op_{ij} de forma paralela, essa execução utiliza a técnica de *paralelização direta*.

4.7 Planejamento de Custos

Ao aplicar a técnica de planejamento para a resolução de um problema de escalonamento, busca-se resultados mais refinados utilizando-se as informações disponíveis para a geração de um plano. A implementação do planejamento de custos buscou a validação do modelo e propõe uma versão simplificada do modelo proposto nesta seção. Na implementação, verifica-se a taxa de ocupação de um recurso do sistema e aplica-se um custo pré-definido caso o recurso esteja ocupado ou parcialmente ocupado. Maiores detalhes estão descritos na seção 4.8.

Para um modelo de custos mais refinado, propõe-se uma técnica de aplicação de custos às ações que compõem um plano de execução. Uma ação, nesse contexto, é a possibilidade de escalonar uma tarefa T em um nó N utilizando para isso o operador O.

Formalmente, um custo de uma ação é uma função: $\omega(E, H, A)$, onde:

- E é um número inteiro que representa o tempo estimado em segundos. Esse tempo é fornecido pelo *usuário* no momento da descrição do fluxo de trabalho. Esta informação é coletada de forma empírica;

- H é um coeficiente gerado pela média de tempo de execução extraído do histórico de uma determinada tarefa. Como todas as execuções no sistema são monitoradas e armazenadas, é possível saber em média quanto o operador em questão demorou em execuções anteriores;

- A é a situação do ambiente no momento do escalonamento. Essencialmente A é um conjunto de duplas $A = \{\{n_1, to_1\}, \{n_2, to_2\}, \ldots, \{n_n, to_n\}\}$. Onde, n representa um nó e to representa o taxa de ocupação deste nó;

- A função $\omega(E, H, A)$ é implementada no componente *OráculoDeMetaDados*. Utiliza as informações de E, H e A para a geração do coeficiente de custo de um determinado

operador:

$$\omega = \frac{(E + H)}{2} + \alpha(A, H) \tag{4.2}$$

A função $\alpha(A, H)$ utiliza informações da tarefa, do ambiente de execução e do histórico de execuções para gerar um coeficiente. Com base no histórico de execuções H, extraí-se o custo de uma tarefa de forma semelhante a apresentada nas técnicas de *paralelização por plano* (seção 4.5.2). Se existe um conjunto de tarefas $T_1, T_2, \ldots, T_n$ no histórico de execução exatamente igual a T, assume-se que o custo de T é uma média dos custos de $T_1, T_2, \ldots, T_n$. A situação do ambiente de execução A influencia no custo de uma tarefa de acordo com a ocupação dos recursos. Por exemplo, se o sistema está sobrecarregado, o custo de uma tarefa baseada em seu histórico, pode aumentar. Se o ambiente está livre, este custo deve diminuir.

A função para geração do custo pode agregar ainda outros parâmetros, para a elaboração de um plano com um objetivo específico. Por exemplo, se fossem armazenados dados de quanto cada operador gasta em energia elétrica, esse índice poderia ser considerado na função de geração do custo. Desta forma, a política de escalonamento poderia maximizar ou minimizar o gasto de energia elétrica.

O componente *CaixaDeTradução* obtém essas informações e as utiliza para a composição dos arquivos PDDL. Cada custo é calculado pelo componente *OráculoDeMetaDados*. Assim, cada ação no domínio do problema é descrito com um custo diferente. O planejador busca pelo plano que minimize a somatória total dos custos.

Para ilustrar a potencialidade do plano de execução gerado pelo planejador, supõe-se o seguinte exemplo hipotético: seja uma tarefa T_1 composta por dois operadores O_1 e O_2. O operador O_1 executa a tarefa sequencial e leva para isso 10 segundos. O operador O_2 executa a tarefa de forma paralela em 5 nós e leva 8 segundos. A princípio o tempo de execução é menor. Entretanto se considerarmos a quantidade de nós alocados, o primeiro operador gasta usa um nó enquanto o segundo usa cinco nós. A escolha de O_1 ou O_2 depende da situação atual dos recursos, logo se existem nós livres, o melhor operador a ser escolhido é O_2, entretanto se o sistema já está com certa sobre-carga a melhor opção é

O_1. O planejador é capaz de atribuir custos a esses operadores, e em uma situação em que os recursos estão livres $\omega(O_1) < \omega(O_2)$ e quando o sistema já possui processos executando no ambiente $\omega(O_1) > \omega(O_2)$. Desta forma a busca pelo menor operador implica na geração do melhor plano.

O modelo de custos apresentadas nessa seção pode contribuir para um planejamento mais refinado, levando em consideração outras características para um escalonamento personalizado. Com a atribuição de outras funções de custos é possível, por exemplo, considerar a quantidade de energia elétrica que se gasta para processar determinada operação, possibilitando assim a geração de um plano que minimize o gasto de energia elétrica.

4.8 Método de Tradução

O método de tradução é um padrão que atua como interface entre as informações do sistema (*dados*, *operadores*, *estado atual dos recursos* e *base de metadados*) e o planejador. Essas informações são utilizadas para compor dois arquivos no formato PDDL, que servirão de entrada para o planejador. O componente responsável pelo método de tradução é a *CaixaDeTradução*.

Em nossa implementação, a estrutura básica adotada para geração dos arquivos PDDL pode ser observada nos códigos 4.2 e 4.3. Nesta versão do método de tradução, explora-se o *paralelismo direto* (seção 4.5.1) e o *paralelismo sobre execução* (seção 4.6)

Código 4.2: Estrutura básica do problema utilizado

```
(define (problem problem_matrix)
(:domain matrix)
    (:objects
        <data>  - data
        <task>  - task
        <hosts> - hosts
    )
    (:init
        (have $D)
        (input_$X $D $T)
        (output_$X $D $T)
        (availiable $H)
        (half-availiable $H)
        (= (total-cost) 0)
    )
    (:goal
        (and
            (have $D)
        )
    )
    (:metric minimize (total-cost))
)
```

O arquivo PDDL no código 4.2 representa a estrutura do problema. Importa-se as informações contidas na descrição dos fluxos de trabalho para que sejam utilizadas pelo planejador. No código, <data>, <task> e <hosts> representam coleções que indicam os dados, tarefas e os nós, que serão utilizados no escalonamento. O conjunto de dados é obtido por todos os dados de entrada e saída, que não se repetem. As tarefas são os nomes dos operadores dos fluxos de trabalhos. Ambas informações são obtidas através do arquivo *aplicação.xml*. O conjunto de nós é obtido pelo componente *OráculoDeMetaDados*.

Os elementos iniciados por $ representam variáveis. $D é um conjunto de dados. $X é um número. $T é uma tarefa. $H é um nó.

Para todos os dados de entrada, atribui-se uma proposição *(have $D)* que informa ao planejador qual dado está disponível.

As definições de *input* e *output* são semelhantes. Definem-se proposições diferentes com um indexador numérico ($X) que representa o numero de dados ($D) para uma tarefa ($T). Com as informações de entrada e saída é possível informar ao planejador a dependência entre os os operadores dos fluxos. Assim, se uma tarefa A necessita de um dado D que só é fornecido pela tarefa B, então B deve ser executado antes de A.

O componente *OráculoDeMetaDados* também retorna informações sobre a ocupação

do nó, que no modelo é definido como disponível (*availiable* $H) ou parcialmente disponível (*half-availiable* $H). Quando disponível, um nó pode trabalhar com sua capacidade total, pois está livre para executar qualquer tarefa. Um nó parceialmente disponível já está executando alguma tarefa e por isso aplica-se um custo maior para a execução dessa tarefa.

Uma variável de controle (*total-cost*) é utilizada para acumular os custos das tarefas realizadas. É possível ainda pedir ao planejador que retorne um plano que minimize esse custo, para isso a métrica (*minimize (total-cost)*). Na seção 4.7 descreve-se a possibilidade da adaptação de outras métricas para a geração de um plano de execução personalizado, nesse caso outras métricas deveriam ser incluídas e computadas.

Código 4.3: Estrutura básica do domínio utilizado

```
1  (define (domain matrix)
2  (:requirements :typing :fluents)
3      (:types data task hosts - object)
4      (:predicates
5          (have ?d - data)
6          (availiable ?h - hosts)
7          (half-availiable ?h - hosts)
8          (working-full ?h - hosts ?t - task)
9          (working-half ?h - hosts ?t - task)
10         (inputX D - data ?t - task)
11         (outputY D - data ?t - task)
12     )
13     (:functions
14         (total-cost)
15     )
16     (:action start-task-full_$X
17     :parameters (?h - hosts ?t - task $D - data)
18     :precondition (and (availiable ?h) (have $D) (input_$X $D ?t))
19     :effect (and (working-half ?h ?t) (not (availiable ?h)) (increase (total-cost) 1))
20     )
21     (:action start-task-half_$X
22     :parameters (?h - hosts ?t - task $D - data)
23     :precondition (and (half-availiable ?h) (have $D) (input_$X $D ?t))
24     :effect (and (working-full ?h ?t) (not (half-availiable ?h)) (increase (total-cost)↩
           10))
25     )
26     (:action end-task-full_$X
27     :parameters (?h - hosts ?t - task $D - data)
28     :precondition (and (working-half ?h ?t) (output_$X $D ?t))
29     :effect (and (availiable ?h) (have $D) (not (half-availiable ?h)) (not (working-↩
           half ?h ?t)))
30     )
31     (:action end-task-half_$X
32     :parameters (?h - hosts ?t - task  $D - data)
33     :precondition (and (working-full ?h ?t) (output_$X $D ?t))
34     :effect (and (half-availiable ?h) (have $D) (not (working-full ?h ?t)))
35     )
36 )
```

No domínio, código 4.3, definem-se as ações que caracterizam o início e fim de cada

tarefa. Cada uma dessa ações utiliza um indexador $X. Para as ações de início *(start-task)* $X define o número de dados de entrada para o *input*, então se $\$X = 2$ então $|D| = 2$. A mesma lógica é aplicada as ações de fim (*end-task*).

Para cada indexação de $X são definidas duas opções de início e fim de tarefa: *full* e *half*. A primeira utiliza apenas nós que estejam totalmente disponíveis para a execução de uma tarefa. Essa ação incrementa em 1 o custo total (*total-cost*). A segunda utiliza nós que já estejam trabalhando, entretanto incrementa em 10 o custo total. Os custos aplicados são fixos, entretanto aplicados automaticamente de acordo com a ocupação do nó.

Desta forma, consegue-se o paralelismo pretendido entre os nós; visto que mesmo que um nó já esteja ocupado, ainda é vantagem distribuir a nós diferentes tarefas que podem ser executadas em paralelo.

O apêndice C transcreve os arquivos de domínio e problema, respectivamente em C.1 e C.2. O plano gerado é transcrito no aprêndice C.3.

4.9 Considerações

Quanto à classificação (seção 2.1), a descrição dos fluxos de trabalho em um arquivo XML é a etapa de *composição*. O planejador atua como *orquestrador*, pois define as regras de dependências e escalonamento, e ainda atua como componente de *mapeamento*, pois dá origem ao plano de execução que será enviado ao gerenciador de execução. O componente de *execução* é definido pelas adaptações do sistema *PeerUnit*.

A implementação utiliza a linguagem Java. A motivação para o uso da linguagem é relacionada a linguagem de desenvolvimento da camada de execução (*PeerUnit* seção 4.2.1). A estrutura da codificação segue os padrões do projeto *PeerUnit*, fazendo dele uma ramificação direta, porém com um objetivo diferente, ou seja, ao invés de testar um sistema P2P, utiliza-se a mesma estrutura para a construção de um ambiente de execução inteligente. Além disso, o planejador *CRIKEY* (seção 2.5.4) também está implementado na linguagem java.

O trabalho que envolve IA e fluxo de execução [20] (seção 3.6) é o principal trabalho

relacionado. Utiliza como base o projeto *Pegasus* [13] (seção 3.3) e aprimora o escalonamento utilizando técnicas de planejamento para a distribuição das tarefas no ambiente de execução. O trabalho usa um modelo colaborativo genérico que funciona com o compartilhamento de metadados pela Internet. A principal diferença entre [20] e este trabalho está no ambiente de execução. Nosso trabalho utiliza técnicas P2P que focam em um sistema de execução escalável, enquanto que [20], busca a colaboração entre ambientes de grade.

Uma limitação do modelo está relacionada ao tempo limite de execução para uma ação. Como o *PeerUnit* foi concebido inicialmente para testes que validam se um dado pode ser encontrado em um ambiente P2P, estabelece-se um tempo limite para assumir que o dado foi encontrado. Apesar da utilização de um ambiente P2P, assume-se que este ambiente deve estar controlado para a extração das metainformações sobre os recursos que ajudam na política de escalonamento. Por isso, não espera-se um tempo limite, pois há a garantia de resposta do um nó. Em nossas implementações este problema foi contornado inserindo em uma primeira execução um tempo limite infinito, e nas próximas execuções um tempo perto do esperado.

A implementação focou na construção básica para o gerenciamento dos fluxos de trabalhos científicos. Essencialmente o componente *OráculoDeMetaDados* foi implementado de maneira simplista, e não está preparado para responder a perguntas mais complexas. Nos experimentos realizados o oráculo é capaz de responder apenas uma visão global dos recursos do ambiente, classificando-os em: disponível, indisponível ou parcialmente disponível. Quanto aos tipos de paralelização, implementou-se basicamente o modelo direto, no qual o usuário define a quantidade de nós que irão trabalhar em um operador paralelo, e o paralelismo sobre execução, que permite a execução paralela de múltiplos operadores.

Na seção de planejamento de custos (seção 4.7) expõe-se a potencialidade do modelo, com a discussão sobre as opções de escalonamento levando em consideração a situação do ambiente distribuído.

CAPÍTULO 5

EXPERIMENTOS

Neste capítulo apresentam-se os experimentos realizados no sistema desenvolvido. A seção 5.1 mostra um experimento que demonstra a técnica de paralelização direta. Na seção 5.2 apresenta-se um experimento para avaliação do comportamento utilizando vários encadeamentos no fluxo de trabalho. Na seção 5.3 apresenta-se uma abordagem que demonstra a implementação de um operador que trabalha com múltiplas entradas e saídas. Na seção 5.4 faz-se uma analise dos resultados obtidos.

O principal objetivo na análise dos resultados está na corretude dos experimentos bem como a qualidade dos planos gerados. Para os experimentos 5.1 e 5.3 utilizou-se um ambiente de execução local no qual simulou-se um ambiente distribuído. Desta forma, possibilitou-se a reprodução dos experimentos com nós emulados na mesma máquina. O experimento 5.2 foi um teste para a validação do modelo, executado em um ambiente distribuído com 4 nós.

A máquina utilizada para execução local possui um processador 2.66 GHz Intel Core 2 Duo com 4GB de memória. As máquinas do ambiente distribuído possuem um AMD Opteron com 2.4 GHz e 2GB de memória.

Os testes utilizaram como base os dados extraídos de matrizes. A motivação para a utilização de matrizes é a necessidade de cálculos simples sobre um grande volume de dados científicos. Para a validação do experimento foram utilizadas matrizes quadradas preenchidas por números aleatórios. Dois operadores foram implementados: soma e multiplicação de matrizes.

Nas figuras dispostas nesse capítulo, cada quadrado representa um operador. Cada operador possui uma identificação dos dados que são trabalhados. Por exemplo, op_a_b significa que o operador está trabalhando com os dados A e B. Dentro do quadrado,

descreve-se ainda uma informação que mostra qual a operação efetuada com os dados em questão. Por exemplo, *SUM* para soma ou *MULT* para multiplicação. Os dados são representados por cilindros, cada conjunto de dados representa uma matriz e sua identificação se dá por letras em maiúsculo. As setas indicam o fluxo dos dados.

Nas tabelas estão apresentados os resumos de execução dos fluxos de trabalho, bem como o plano gerado pelo planejador. A coluna ordem, define a sequencia de execução de uma ação no plano de execução. Se uma mesma ordem é repetida em mais de um linha, significa que as ações ocorrerem em paralelo. A coluna nó, indica qual o nó selecionado para a execução da ação. Se a célula apresenta o caractere $*$, significa que a ação foi executada por todos os nós.

Ainda é possível que um subconjunto específico de nós execute uma ação, nesse caso suas identificações estão separadas por vírgula. A coluna operador indica qual o operador utilizado. Se o operador é identificado como *difundir* então há uma barreira para a publicação das informações, geradas até o momento, para todos os outros nós. A coluna dados, indica os dados utilizados para a aplicação do operador. Quando um dado está na mesma linha que possui um operador difundir, esse dado é o que será publicado para todo o conjunto de nós. A coluna tempo indica o tempo de execução de cada uma das ações em Milisegundos (ms) . As últimas linhas mostram um resumo do tempo de execução de cada etapa: planejamento, tradução, difusão e processamento. A última linha mostra o tempo total de execução.

5.1 Experimento 1 - Paralelização Direta

A representação do esquema de execução dos fluxos de trabalho estão dispostos na figura 5.1.

No exemplo são definidas três operadores: $op_{\{ma,mb\}}$, $op_{\{mc,md\}}$ e $op_{\{mi,mj\}}$. O operador $op_{\{ma,mb\}}$ espera como entrada os dados MA e MB e gera como saída MI. Já $op_{\{mc,md\}}$ espera como entrada MC e MD e gera como saída MJ. O operador $op_{\{mi,mj\}}$ espera como entrada MI e MJ e gera como saída MK, que em nosso exemplo é o objetivo. A definição do arquivo que contém a descrição dos fluxos de trabalho está transcrita no

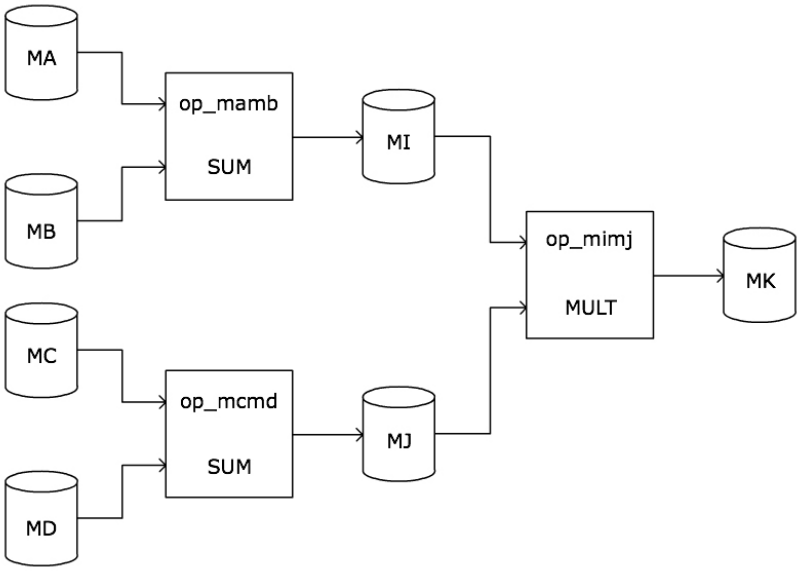

Figura 5.1: Experimento com operações entre matrizes

apêndice B.1.

Foram utilizados dois nós para a execução do experimento: h_0 e h_1. Neste experimento, analisa-se os modelos paralelos abordados no capítulo 4. A *paralelização direta* (seção 4.5.1) e o *paralelismo sobre execução* (seção 4.6). A figura 5.2 mostra a distribuição das tarefas e os respectivos nós utilizados para execução.

A paralelização direta ocorre na execução distribuída do operador $op_{\{mi,mj\}}$. O paralelismo sobre execução está exemplificado na execução simultânea de $op_{\{ma,mb\}}$ e $op_{\{mc,md\}}$.

A tabela 5.1 detalha as informações extraídas do experimento.

Tabela 5.1: Resultados do experimento com paralelização direta

Ordem	Host	Operador	Dados	Tempo (ms)
1	h_0	SUM	MA, MB	4779
1	h_1	SUM	MC, MD	3935
2	*	DIFUNDIR	MI	5617
2	*	DIFUNDIR	MJ	4203
3	h_0, h_1	MULT	MI, MJ	3170
4	*	DIFUNDIR	MK	2880
5	*	coletor de informação	metadados	31
tempo de planejamento				1014
tempo de tradução				523
tempo de difusão				12700
tempo processamento				11935
tempo total				26172

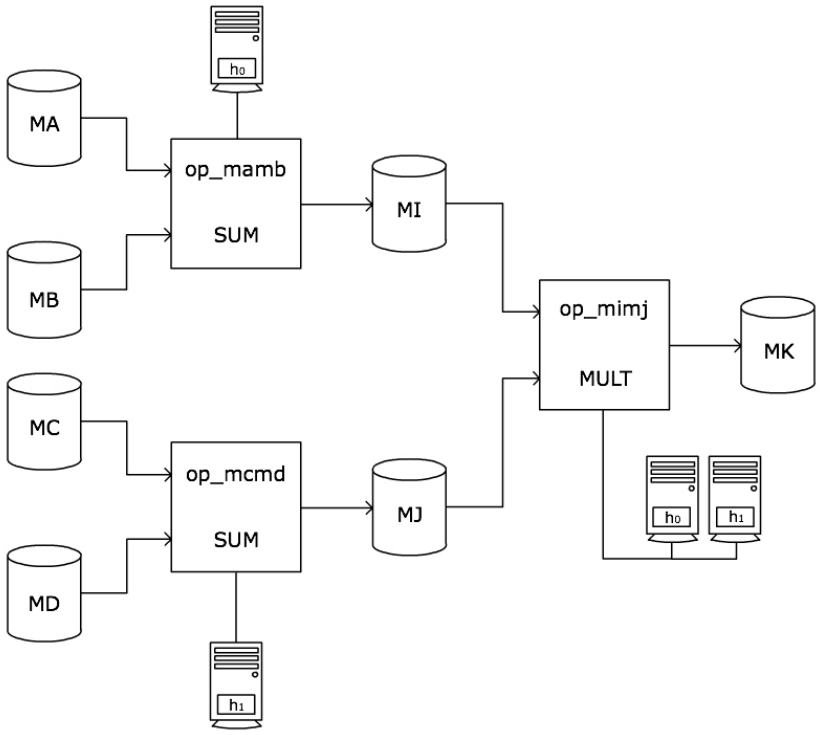

Figura 5.2: Diagrama do experimento 1

Nota-se que o planejador considerou a execução simultânea dos operadores $op_{\{ma,mb\}}$ e $op_{\{mc,md\}}$ na etapa 1, pois os operadores não possuem qualquer dependência de dados. O tempo para execução da operação de soma foi semelhante, para o operador $op_{\{ma,mb\}}$ gastou-se 4779ms, enquanto que o operador $op_{\{mc,md\}}$ terminou a execução em 3935ms.

Nas etapas 2 e 4 nota-se a operação de difusão. Essa operação distribui os resultados obtidos pelos operadores para todos os outros nós do ambiente, visto que algum operador seguinte pode utilizar os dados gerados na operação em questão.

A etapa 3 utiliza dois nós para a execução da operação de multiplicação. O tipo de paralelização adotada é direta, por isso, o desenvolvedor informa os parâmetros para a execução paralela dos dados. Ambos os nós disponíveis foram escolhidos para a divisão dos dados, e a operação completa levou 3170ms.

5.2 Experimento 2 - Encadeamento de Fluxos de Trabalho

A representação do esquema de execução do fluxo de trabalho está disposta na figura 5.3.

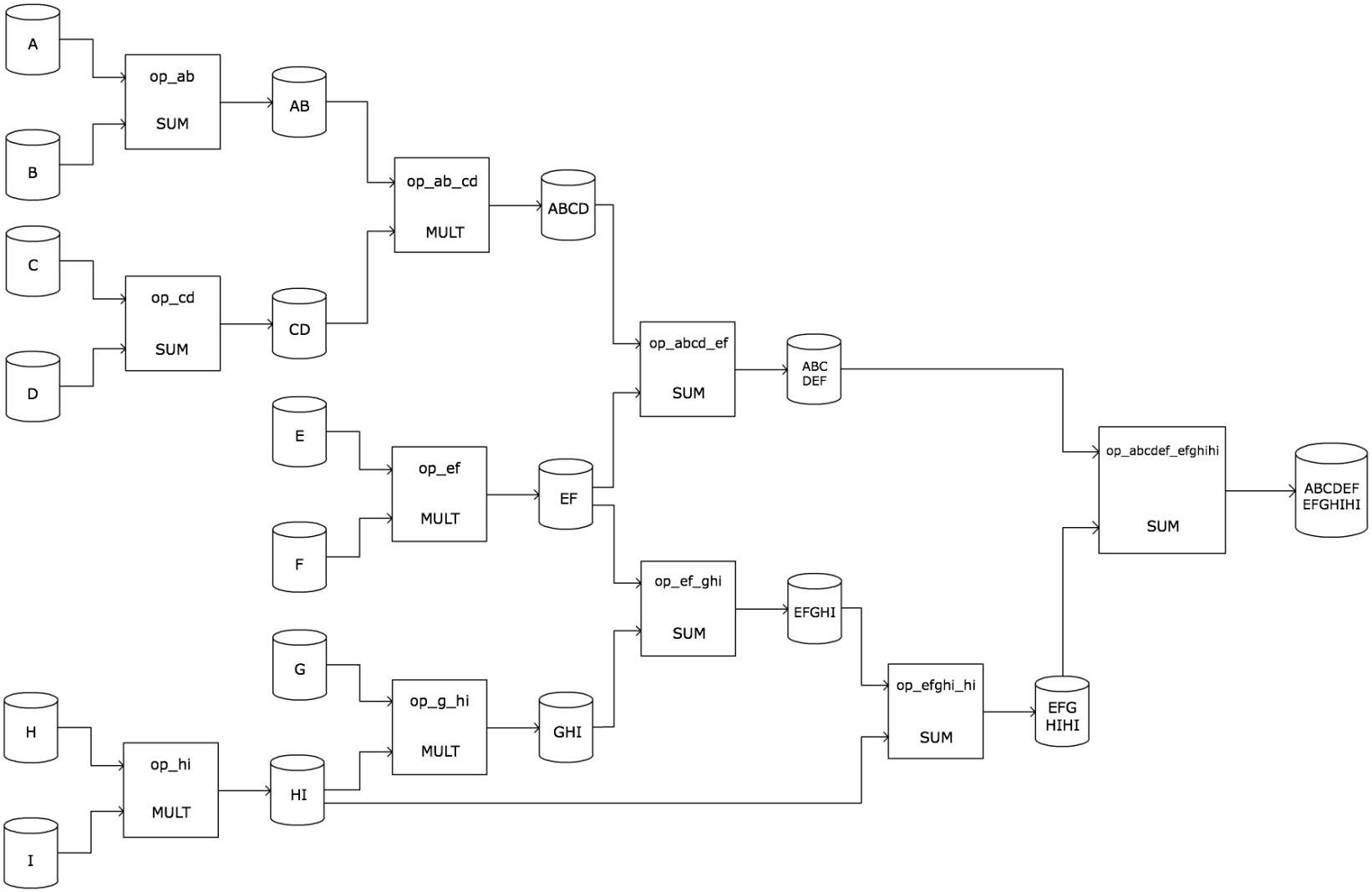

Figura 5.3: Diagrama do experimento 2

No exemplo são definidos dez operadores: $op_{\{a,b\}}$, $op_{\{c,d\}}$, $op_{\{h,i\}}$, $op_{\{e,f\}}$, $op_{\{ab,cd\}}$, $op_{\{g,hi\}}$, $op_{\{abcd,ef\}}$, $op_{\{ef,ghi\}}$, $op_{\{efghi_hi\}}$ e $op_{\{abcdef,efghihi\}}$. A indexação de cada operador define sua entrada de dados. O objetivo do experimento é a obtenção do dado final $ABCDEFEFGHIHI$.

Para o experimento foram utilizados 4 nós: h_0, h_1, h_2 e h_3. Testou-se a validade do componente *OráculoDeMetaDados* que retorna a taxa de ocupação dos nós disponíveis, em três situações diferentes:

1. quatro nós disponíveis;

2. dois nós disponíveis (h_0 e h_1) e dois nós parcialmente disponíveis (h_2 e h_3);

3. um nó disponível (h_0) e três parcialmente disponíveis (h_1, h_2 e h_3).

As tabelas 5.2, 5.3, 5.4 demonstram os resultados para as três situações, respectivamente.

Tabela 5.2: Encadeamento de fluxos - 4 nós disponíveis

Ordem	Host	Operador	Dados	Tempo (ms)
1	h_1	MULT	H, I	17400
1	h_3	MULT	E, F	17160
1	h_0	SUM	C, D	16699
1	h_2	SUM	A, B	17738
2	*	DIFUNDIR	EF	16616
2	*	DIFUNDIR	HI	15818
2	*	DIFUNDIR	AB	14650
2	*	DIFUNDIR	CD	14472
3	h_2	MULT	G, HI	13168
3	h_0	MULT	AB, CD	14475
4	*	DIFUNDIR	GHI	14908
4	*	DIFUNDIR	$ABCD$	14637
5	h_1	SUM	EF, GHI	14189
5	h_3	SUM	$ABCD, EF$	13622
6	*	DIFUNDIR	$EFGHI$	14661
6	*	DIFUNDIR	$ABCDEF$	14881
7	h_3	SUM	$EFGHI, HI$	13584
8	*	DIFUNDIR	$EFGHIHI$	15042
9	h_0	SUM	$ABCDEF, EFGHIHI$	14900
10	*	DIFUNDIR	$ABCDEFEFGHIHI$	15145
11	*	coletor de informação	metadados	39
tempo de planejamento				3523
tempo de tradução				1301
tempo de difusão				150830
tempo processamento				153016
tempo total				308670

Nota-se que com 4 nós disponíveis o planejador elabora o plano ótimo, distribuindo o máximo de operadores possível na etapa 1. A difusão dos resultados ocorre nas etapas 2, 4, 6, 8 e 10. As etapas ímpares são responsáveis pelos cálculos dos operadores enquanto que as etapas pares são responsáveis pela difusão dos resultados obtidos.

Tabela 5.3: Encadeamento de fluxos - 2 nós disponíveis e 2 parcialmente disponíveis

Ordem	Host	Operador	Dados	Tempo (ms)
1	h_1	MULT	H, I	18999
1	h_0	SUM	C, D	17012
1	h_3	SUM	A, B	17541
2	*	DIFUNDIR	HI	17243
2	*	DIFUNDIR	CD	16543
2	*	DIFUNDIR	AB	15426
3	h_1	MULT	E, F	15188
3	h_0	MULT	G, HI	13453
4	*	DIFUNDIR	EF	15377
4	*	DIFUNDIR	GHI	15170
5	h_1	SUM	EF, GHI	15286
5	h_0	MULT	AB, CD	14212
6	*	DIFUNDIR	$EFGHI$	15334
6	*	DIFUNDIR	$ABCD$	15415
7	h_2	SUM	$EFGHI, HI$	13484
7	h_3	SUM	$ABCD, EF$	13389
8	*	DIFUNDIR	$EFGHIHI$	15433
8	*	DIFUNDIR	$ABCDEF$	15423
9	h_3	SUM	$ABCDEF, EFGHIHI$	13634
10	*	DIFUNDIR	$ABCDEFEFGHIHI$	15518
11	*	coletor de informação	metadados	10
tempo de planejamento				2285
tempo de tradução				1043
tempo de difusão				156882
tempo processamento				152279
tempo total				312489

Ao trabalhar com dois nós parcialmente disponíveis, o planejador optou por um plano no qual em um primeiro momento apenas 3 nós são utilizados. Entretanto gerou-se ainda assim um plano com 10 etapas. A grande diferença é que a etapa 7 executa paralelamente o operador $op_{\{efghi_hi\}}$ e $op_{\{abcd,ef\}}$. No plano anterior, com todos os nós disponíveis, o operador $op_{\{abcd,ef\}}$ é executado na etapa 5, pois neste momento, todas as dependências já haviam sido calculadas.

Tabela 5.4: Encadeamento de fluxos - 1 nó disponível e 3 parcialmente disponíveis

Ordem	Host	Operador	Dados	Tempo (ms)
1	h_0	MULT	E, F	18146
1	h_2	MULT	H, I	17923
1	h_1	SUM	C, D	17075
1	h_3	SUM	A, B	17972
2	*	DIFUNDIR	EF	16672
2	*	DIFUNDIR	HI	15897
2	*	DIFUNDIR	AB	14519
2	*	DIFUNDIR	CD	14676
3	h_3	MULT	G, HI	13658
3	h_0	MULT	AB, CD	14333
4	*	DIFUNDIR	GHI	15110
4	*	DIFUNDIR	$ABCD$	14909
5	h_3	SUM	EF, GHI	14770
5	h_1	SUM	$ABCD, EF$	14974
6	*	DIFUNDIR	$EFGHI$	14817
6	*	DIFUNDIR	$ABCDEF$	15352
7	h_2	SUM	$EFGHI, HI$	14004
8	*	DIFUNDIR	$EFGHIHI$	15235
9	h_2	SUM	$ABCDEF, EFGHIHI$	14046
10	*	DIFUNDIR	$ABCDEFEFGHIHI$	14987
11	*	coletor de informação	metadados	32
tempo de planejamento				2256
tempo de tradução				1115
tempo de difusão				152174
tempo processamento				156973
tempo total				312518

Mesmo com apenas um nó totalmente disponível, o planejador ainda aproveita do poder de paralelização, e gera um plano bastante semelhante ao primeiro experimento.

5.3 Experimento 3 - Múltiplas Entrada e Saídas de Dados

No terceiro experimento utilizou-se um único operador para demonstrar a capacidade de múltiplas entradas e saídas de dados. Nem sempre um operador utiliza um número fixo de entradas ou saídas. Nesse experimento demonstra-se a capacidade do modelo em executar operadores mais complexos, que utilizem vários dados de entrada e gerem várias saídas.

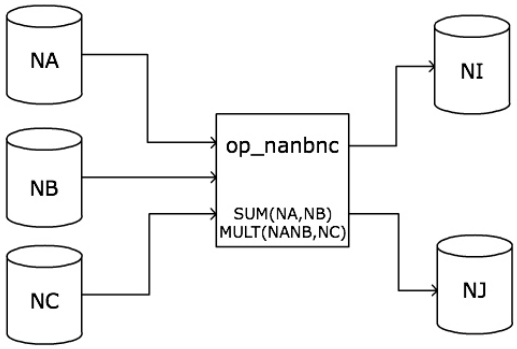

Figura 5.4: Diagrama do experimento 3

Na figura 5.4 ilustra-se um fluxo de trabalho que utiliza como entrada de dados três matrizes: NA, NB e NC. O operador aplicado, $op_{\{na,nb,nc\}}$ soma as matrizes NA e NB e a matriz resultante é multiplicada com NC.

Apenas um nó (h_1) foi utilizado para a execução deste experimento. Os detalhes da execução estão descritos na tabela 5.5.

Tabela 5.5: Resultados do experimento com encadeamento de fluxos

Ordem	Host	Operador	Dados	Tempo (ms)
1	h_1	SUM,MULT	NA, NB, NC	5809
2	*	DIFUNDIR	NI, NJ	7258
3	*	coletor de informação	metadados	91
tempo de planejamento				1030
tempo de tradução				580
tempo de difusão				7258
tempo processamento				5809
tempo total				14790

5.4 Considerações

O principal experimento 5.2 demonstra a capacidade do planejador adaptar o escalonamento de acordo com a situação dos recursos. Na primeira execução, quando os quatro nós estavam disponíveis, o melhor plano foi traçado, executando quatro operadores em paralelo $op_{\{h,i\}}$, $op_{\{e,f\}}$, $op_{\{c,d\}}$ e $op_{\{a,b\}}$, gerando o plano de execução com dez passos. Ao trabalhar com um ambiente com um nó parcialmente ocupado, a primeira parte da execução foi escalonada com três operadores. Entretanto, outras execuções paralelas foram alocadas durante o plano deixando-o também com dez passos.

A utilização da paralelização direta mostra que o ambiente está preparado para a adequação da paralelização por plano, no qual o próprio planejador escolhe qual operador e quantos nós serão utilizados para a execução de uma determinada tarefa.

No experimento que utiliza múltiplas entradas de dados, demonstra-se que sistema é capaz de agregar operadores mais elaborados, que utilizem várias operações e conjuntos de dados variados.

Os experimentos realizados neste trabalho foram direcionados para a validação do modelo (seção 4). Para isto implementou-se e testou-se as principais funcionalidades para um ambiente de execução distribuída com a geração de planos de execução por um planejador.

CAPÍTULO 6

CONCLUSÃO

Este trabalho apresenta um modelo para a construção de um sistema para execução de fluxos de trabalho científico baseado em técnicas de planejamento. Um fluxo de trabalho é formado por tarefas, que são enviadas para um ambiente de execução distribuído, visando minimizar o tempo de execução.

Para o escalonamento de tarefas entre os nós do ambiente distribuído, o modelo proposto utiliza técnicas de planejamento que leva em consideração três principais características: um tempo de execução estimado fornecido pelo usuário, um coeficiente de tempo extraído de execuções anteriores e a situação atual do ambiente distribuído. Essas informações são obtidas por um oráculo que consulta: as definições dos fluxos de trabalho, o estado atual do sistema e um banco de metadados. Com essas informações, o oráculo gera um coeficiente de custo para cada operador no espaço de estados do plano. O planejador procura por um plano que minimize o custo total das operações, resultando em um plano de execução que mapeia as tarefas para seus respectivos nós.

A implementação do sistema, utilizou o *CRIKEY* como planejador [21, 11] (seção 2.5.4) e o *PeerUnit* [2] (seção 4.2.1) como gerenciador de execução. Visto que a implementação utilizou a técnica P2P, o sistema torna-se independente das características do ambiente distribuído, pois cada *peer* tem a função de trabalhar de forma autônoma.

A fim de coletar os metadados, cada nó possui um sistema coletor de informações, que armazena em um banco de metadados quais foram as tarefas executadas, o tempo de execução e informações sobre o nó no final da execução, tais como: taxa de ocupação do processador, quantidade de memória utilizada, entre outros.

No modelo, foram explorados dois tipos de paralelismo: *paralelismo sobre os dados* e *paralelismo sobre a execução*. O paralelismo sobre os dados é dividido em dois submodelos: *direto* e *paralelo por plano*. O modelo *direto* escalona o processamento dos dados a partir

de parâmetros inseridos na descrição do fluxo de trabalho. O modelo *paralelo por plano* utiliza o planejador para dividir os dados entre os nós disponíveis. O *paralelismo sobre a execução* é a execução paralela dos fluxos de trabalho. Seu escalonamento é feito pelo planejador, que a partir das informações provenientes dos recursos disponíveis, pondera quais tarefas podem ser executadas em paralelo e quais os melhores nós para tal execução.

A implementação do modelo focou na construção de um ambiente executável, que coordena desde o mapeamento dos experimentos para os fluxos de trabalho, até a exportação dos resultados obtidos. Dentre as técnicas de paralelização, implementou-se o modelo *direto*, e o *paralelismo sobre execução*. Além disso, foi implementada uma versão mais simples do oráculo, que é capaz de responder apenas a taxa de ocupação dos nós em três estados: disponível, parcialmente disponível ou indisponível.

Com os resultados obtidos, nota-se que o sistema gera planos de qualidade e que podem ser aprimorados com o refinamento do oráculo, bem como a implementação completa do planejamento de custos abordado na seção 4.7, e a adição de novas métricas para explorar planos de execução personalizados.

6.1 Trabalhos Futuros

A implementação utilizada neste trabalho focou na construção de uma estrutura capaz de executar as funcionalidades básicas do modelo descrito. Por isso uma série de extensões podem ser exploradas como trabalhos futuros. O principal componente a ser trabalho é o *OráculoDeMetaDados*. Com consultas mais detalhadas aos metadados, e a situação atual dos recursos é possível obter planos ainda melhores. Como sugestão de implementação, apresenta-se o modelo descrito na seção de custos 4.7.

Ainda é possível obter detalhes dos nós que possam ajudar em um planejamento baseado em outras métricas que não se resumem a minimizar o tempo de execução. Ao extrair informações específicas, como por exemplo, qual o custo em energia elétrica para processar determinado operador, pode-se ajustar o coeficiente de custos para a obtenção de um plano de execução que resolva um determinado fluxo de trabalho com o menor gasto de energia.

Os operadores (seção 4.3.3) suportam diferentes implementações, específicas para serem executadas de forma serial ou para uma quantidade n de nós. Desta forma propõe-se que o planejador seja capaz de escolher (em conjunto com o *OráculoDeMetaDados*) qual a melhor implementação a ser utilizada. Para que isso seja feito, é necessário que se insira informações específicas sobre as tarefas nos arquivos PDDL que serão interpretados pelo planejador.

Nos testes efetuados, nota-se que o processo de difusão dos resultados consome um tempo considerável. Para amenizar este problema, sugere-se que o método de difusão seja capaz de analisar o plano de execução e propague os dados de saída apenas para os nós que irão utilizar os dados em processamentos futuros. Para isso, é necessário que os operadores de difusão sejam incluídos no planejamento, assim é possível uma propagação exata dos dados para os nós que irão utilizar os dados futuramente.

A *paralelização por plano* (seção 4.5.2) pode ser implementada, deixando por conta do planejador a distribuição de dados paralela. Após a análise dos dados oferecidos pelo oráculo, o planejador poderá escolher quantos nós serão destinados para a execução de uma tarefa. Para isso é necessário que o sistema de tradução seja alterado, e os operadores recebam como parâmetro o número de nós que serão utilizados naquela operação.

Como detalhes de implementação, sugere-se o refinamento do tratamento de tolerância a falhas, o detalhamento do sistema de *logs* e a interação com o sistema de *logs* do *PeerUnit*.

Um fluxo de trabalho científico pode conter ciclos em sua estrutura. Os ciclos aumentam a complexidade em sistemas que incluam o tratamento dessa classe de problemas [10]. Uma limitação deste modelo é a impossibilidade de tratar fluxos de trabalho que necessitem de uma especificação cíclica. Entretanto, uma das funcionalidades desenvolvidas, permite ao usuário especificar múltiplas entrada e saídas de dados (seção 5.3). Isso permite um encapsulamento de operadores que necessitem de ciclos, unindo-os em um operador maior que executa mais de uma operação.

APÊNDICE A

MODELO INICIAL ELABORADO EM JSHOP

Neste apêndice apresenta-se a primeira modelagem do problema utilizando a ferramenta JSHOP [23].

Os códigos A.1 e A.2 apresentam os arquivos de problema e domínio modelados.

Código A.1: Problema elaborado em JSHOP

```
(defproblem problem jshop-example (
    (data d1)
    (data d21)
    (data d22)
    (data d3)
    (have d1)
    (task a)
    (task b)
    (task c)
    (input d1 a)
    (input d1 b)
    (input d21 c)
    (input d22 c)
    (output d21 a)
    (output d22 b)
    (output d3  c)
    (host h1)
    (host h2)
    (host h3)
    (avaliable h1)
    (avaliable h2)
    (avaliable h3)
)
(:unordered
    (doplan c)
    (doplan b)
    (doplan a)
    (finish-task a)
    (finish-task b)
    (finish-task c)
))
```

Nota-se que a elaboração do arquivo de problema é muito semelhante a um problema elaborado na linguagem PDDL. O problema em questão utiliza três tarefas. A tarefa A e B são dependentes do dado D_1, e geram como saída D_{21} e D_{22} respectivamente. a tarefa C utiliza como entrada D_{21} e D_{22} e gera D_3 que também é o objetivo. Estão disponíveis três nós ($H1, H2$ e $H3$).

Como o JSHOP trabalha com um sistema hierárquico de decomposição de tarefas,

solicita-se que cada resolução *doplan* seja executada separadamente. Ainda foi utilizado um método para finalização de cada uma das tarefas (caso ainda não tenha sido finalizada pela estrutura recursiva).

Código A.2: Domínio elaborado em JSHOP

```
(defdomain jshop-example (
    (:operator (!start-task ?host ?task ?input)
        ((host ?host) (avaliable ?host))
        ((avaliable ?host))
        ((working ?host ?task))
    )
    (:operator (!end-task ?host ?task ?output)
        ((host ?host) (not (avaliable ?host)) (working ?host ?task))
        ((working ?host ?task))
        ((avaliable ?host) (done ?task) (have ?output))
    )
    (:method (finish-task ?task)
        ((working ?h ?task) (output ?o ?task))
        ((!end-task ?h ?task ?o))
        ()
        ()
    )
    (:method (resolve-all ?input ?task)
        ( (input ?i ?task) (not(working ?h ?task)))
        ((!start-task ?h ?task ?i) (resolve-all ?x ?task))
        ()
        ()
    )
    (:method (doplan ?task)
        ((input ?x ?task) (output ?x ?t) (not(working ?h ?t)) (not(done ?t)))
        ((doplan ?t) (doplan ?task))
        ((input ?x ?task) (output ?x ?t) (working ?h ?t))
        ((!end-task ?h ?t ?x) (doplan ?task))
        ((not(done ?task)) (input ?i ?task))
        ((resolve-all ?i ?task))
        ()
        ()
    )
))
```

Como domínio do problema descreve-se ações de inicialização e finalização de tarefa. O método *doplan* é o método principal. Ele invoca os demais métodos ou ele próprio recursivamente caso seja necessário. O método *resolve-all* é utilizado para resolver todas as dependências de uma tarefa que necessita dos dados para sua execução. O método *finish-task* é invocado e termina uma tarefa com o operador *end-task* caso ela ainda esteja em execução.

A figura A.1 mostra a decomposição utilizada para a geração do plano.

O código A.3 mostra o resultado obtido.

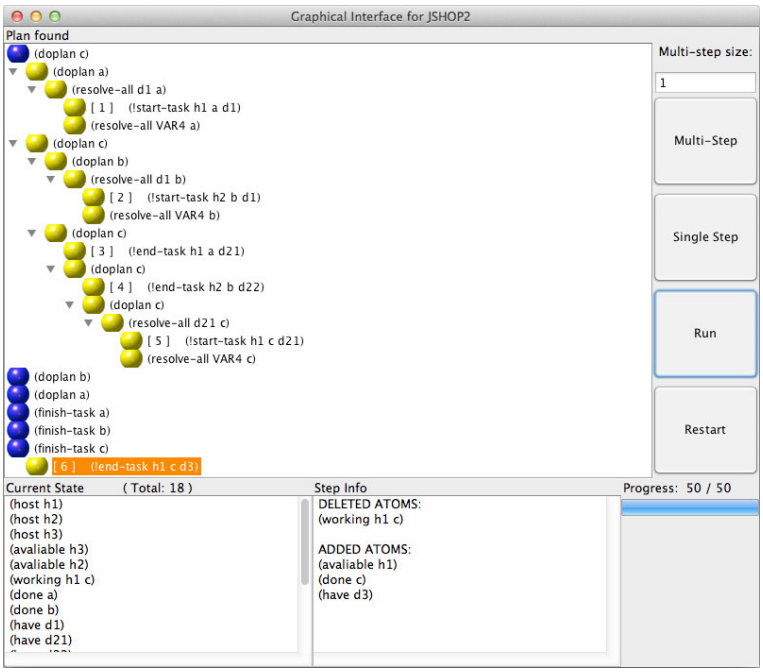

Figura A.1: Decomposição de ações no JSHOP

Código A.3: Plano de ações retornado pelo JSHOP

```
1 [ 1 ]    (!start-task h1 a d1)
2 [ 2 ]    (!start-task h2 b d1)
3 [ 3 ]    (!end-task h1 a d21)
4 [ 4 ]    (!end-task h2 b d22)
5 [ 5 ]    (!start-task h1 c d21)
6 [ 6 ]    (!end-task h1 c d3)
```

Com a decomposição das tarefas conseguiu-se inicialmente um plano de execução aceitável entretanto é fácil observar que as ações 1 e 2 poderiam ser executadas de forma paralela, visto que cada uma delas utiliza um nó diferente. Esta foi a principal motivação para a utilização de um planejador capaz de gerar planos paralelos.

APÊNDICE B

RECURSOS DO SISTEMA

Neste apêndice apresenta-se exemplos de códigos que representam as configurações utilizadas para o gerenciamento do sistema.

B.1 Aplicação

O código B.1 mostra o arquivo XML que representa uma aplicação no sistema.

Código B.1: Arquivo XML que representa uma aplicação

```
<?xml version="1.0" encoding="UTF-8"?>
<application numWorkflows="3" numHosts="3">
  <hosts>
    <host>h0</host>
    <host>h1</host>
    <host>h2</host>
  </hosts>
  <workflow>
    <id>0</id>
    <name>sum_ab</name>
    <description>This workflow sum matrix a with matrix b.</description>
    <timeout>1000</timeout>
    <operator>JoinWorkFlows.sumVet_sequential(globals, id, 0, leechLogger);</operator>
    <broadcast>JoinWorkFlows.broadcastResults_sumVet(globals, id, 0, leechLogger);</broadcast>
    <data_input>
      <data>matrix_a.xml</data>
      <data>matrix_b.xml</data>
    </data_input>
    <data_output>
      <data>matrix_i.xml</data>
    </data_output>
    <parallelism type="sequential" />
  </workflow>
  <workflow>
    <id>1</id>
    <name>sum_cd</name>
    <description>This workflow sum matrix c with matrix d.</description>
    <timeout>1000</timeout>
    <operator>JoinWorkFlows.sumVet_sequential(globals, id, 1, leechLogger);</operator>
    <broadcast>JoinWorkFlows.broadcastResults_sumVet(globals, id, 1, leechLogger);</broadcast>
    <data_input>
      <data>matrix_c.xml</data>
      <data>matrix_d.xml</data>
    </data_input>
    <data_output>
      <data>matrix_j.xml</data>
    </data_output>
    <parallelism type="sequential" />
  </workflow>
  <workflow>
    <id>2</id>
    <name>mult_ij</name>
    <description>This workflow mult matrix i with matrix j.</description>
    <timeout>1000</timeout>
    <operator>JoinWorkFlows.multVet_parallel(globals, id, 2, leechLogger);</operator>
    <broadcast>JoinWorkFlows.broadcastResults_multVet_parallel(globals, id, 2, leechLogger);</broadcast>
    <data_input>
      <data>matrix_i.xml</data>
      <data>matrix_j.xml</data>
    </data_input>
    <data_output>
      <data objective="true">matrix_out.xml</data>
    </data_output>
    <parallelism type="parallel">
      <hosts>2</hosts>
      <chunk_size>2</chunk_size>
    </parallelism>
  </workflow>
</application>
```

No código é possível observar as informações sobre os nós disponíveis, bem como os

blocos que definem os fluxos de trabalho a serem executados.

B.2 Ações

O arquivo de ações contém informações para geração de um plano de execução. Um exemplo desse arquivo pode ser observado no código B.2.

Código B.2: Arquivo XML que representa um conjunto de ações

```
<?xml version="1.0" encoding="UTF-8"?>
<execution_plan>
  <action>
    <name>sum_cd</name>
    <range>0</range>
    <order>1</order>
    <timeout>1000</timeout>
    <operator>JoinWorkFlows.sumVet_sequential(globals, id, 1, leechLogger);</operator>
    <workflow>1</workflow>
  </action>
  <action>
    <name>sum_ab</name>
    <range>1</range>
    <order>1</order>
    <timeout>1000</timeout>
    <operator>JoinWorkFlows.sumVet_sequential(globals, id, 0, leechLogger);</operator>
    <workflow>0</workflow>
  </action>
  <action>
    <name>sum_ab</name>
    <range>*</range>
    <order>2</order>
    <timeout>1000</timeout>
    <operator>JoinWorkFlows.broadcastResults_sumVet(globals, id, 0, leechLogger);</↩
        operator>
    <workflow>0</workflow>
  </action>
  <action>
    <name>sum_cd</name>
    <range>*</range>
    <order>2</order>
    <timeout>1000</timeout>
    <operator>JoinWorkFlows.broadcastResults_sumVet(globals, id, 1, leechLogger);</↩
        operator>
    <workflow>1</workflow>
  </action>
  <action>
    <name>mult_ij</name>
    <range>0-1</range>
    <order>3</order>
    <timeout>1000</timeout>
    <operator>JoinWorkFlows.multVet_parallel(globals, id, 2, leechLogger);</operator>
    <workflow>2</workflow>
  </action>
  <action>
    <name>mult_ij</name>
    <range>0-1</range>
    <order>4</order>
    <timeout>1000</timeout>
    <operator>JoinWorkFlows.broadcastResults_multVet_parallel(globals, id, 2, ↩
        leechLogger);</operator>
    <workflow>2</workflow>
  </action>
</execution_plan>
```

Esse arquivo é o intermediário entre um plano gerado pelo *planejador* e um plano de execução que é interpretado pelo *gerenciador de execução*. Ao utilizar esse arquivo intermediário garante-se a modularidade, podendo assim, alterar a máquina que escalona as tarefas, desde que essa utilize o mesmo sistema de exportação de ações.

B.3 Dados

A estrutura utilizada pode ser observada no código de exemplo B.3. A estrutura utilizada para o processamento e validação é baseada no cálculo de vetores de matrizes. Essas estruturas são mapeadas e traduzidas para uma estrutura de linhas e colunas.

Código B.3: Arquivo XML que representa um conjunto de dados

```
<?xml version="1.0" encoding="UTF-8"?>
<data type="int" lines="5" columns="5">
  <line>
    <column>2358</column>
    <column>2813</column>
    <column>2739</column>
    <column>3311</column>
    <column>2553</column>
  </line>
  <line>
    <column>2237</column>
    <column>3062</column>
    <column>2366</column>
    <column>2970</column>
    <column>2226</column>
  </line>
  <line>
    <column>2441</column>
    <column>3588</column>
    <column>2856</column>
    <column>3692</column>
    <column>2693</column>
  </line>
  <line>
    <column>4158</column>
    <column>5452</column>
    <column>4320</column>
    <column>5462</column>
    <column>4112</column>
  </line>
  <line>
    <column>3029</column>
    <column>4182</column>
    <column>3102</column>
    <column>4005</column>
    <column>2966</column>
  </line>
</data>
```

B.4 Logs e Proveniência

No código de exemplo B.5 estão dispostas as informações sobre os nós. Estes são dados sobre o ocupação do nó assim que a execução é finalizada. No código de exemplo B.6 são armazenadas quais as tarefas foram executadas por um nó, além do tempo de execução de cada tarefa. Os arquivos de *log* de tarefas executadas é alimentado assim que uma

tarefa é iniciada ou finalizada, dessa forma é possível informar ao *OracleMetaData* qual o estado atual dos nós.

No código B.4 pode-se observar a saída do sistema de *logs* aplicado ao modelo.

Código B.4: Arquivo de log de execução do sistema

```
[INIT_SWEP_LOGGER_03-01-2012-13:59:20]
(03-01-2012 13:59:20)[INFO] Exporting Application.
(03-01-2012 13:59:20)[INFO] Executing Translation Box.
(03-01-2012 13:59:20)[INFO] Calling the Planner.
(03-01-2012 13:59:21)[INFO] Exporting Actions.
(03-01-2012 13:59:21)[INFO] Generating Plan.
(03-01-2012 13:59:21)[INFO] Making JoinWorkFlows.
(03-01-2012 13:59:21)[INFO] Making Bash File.
(03-01-2012 13:59:21)[INFO] Executing Application
(03-01-2012 13:59:21)[INFO] Executing mvn install.
(03-01-2012 13:59:33)[INFO] Executing PeerUnit.
[END_SWEP_LOGGER]
```

Código B.5: Arquivo que representa a informação coletada de um nó

```
<?xml version="1.0" encoding="UTF-8"?>
<leech_information>
    <host_information>
        <host_id>0</host_id>
        <date>03-01-2012</date>
        <time>13:43:34</time>
        <available_processors>2</available_processors>
        <free_memory>73744352</free_memory>
        <max_memory>129957888</max_memory>
        <roots_number>1</roots_number>
            <roots>
            <total_space>319213174784</total_space>
            <free_space>164763983872</free_space>
            <usable_space>164501839872</usable_space>
            </roots>
        <cpu_average>23,63%</cpu_average>
        <tasks>
            <task>
                <name>sumVet_sequential</name>
                <runtime>30.0</runtime>
            </task>
            <task>
                <name>broadcastData_./src/main/resources/data/matrix_i.xml;</name>
                <runtime>135.0</runtime>
            </task>
            <task>
                <name>multVet_parallel</name>
                <runtime>14.0</runtime>
            </task>
            <task>
                <name>broadcastData_./src/main/resources/data/matrix_out.xml;</name>
                <runtime>67.0</runtime>
            </task>
        </tasks>
    </host_information>
</leech_information>
```

Código B.6: Arquivo que representa a execução de tarefas de um nó

```
<?xml version="1.0" encoding="UTF-8"?>
<log_tasks>
  <start-task>sumVet_sequential03-01-2012-13:43:32</start-task>
  <end-task>sumVet_sequential03-01-2012-13:43:32</end-task>
  <start-task>broadcastData03-01-2012-13:43:33</start-task>
  <start-task>broadcastData03-01-2012-13:43:33</start-task>
  <end-task>broadcastData_./src/main/resources/data/matrix_i.xml;03-01-2012-13:43:33</end-task>
  <start-task>broadcastData03-01-2012-13:43:33</start-task>
  <end-task>broadcastData_./src/main/resources/data/matrix_i.xml;03-01-2012-13:43:33</end-task>
  <start-task>multVet_parallel03-01-2012-13:43:34</start-task>
  <end-task>multVet_parallel03-01-2012-13:43:34</end-task>
  <start-task>broadcastData03-01-2012-13:43:34</start-task>
  <start-task>broadcastData03-01-2012-13:43:34</start-task>
  <end-task>broadcastData_./src/main/resources/data/matrix_out.xml;03-01-2012-13:43:34</end-task>
  <end-task>broadcastData_./src/main/resources/data/matrix_out.xml;03-01-2012-13:43:34</end-task>
  <start-task>sumVet_sequential03-01-2012-13:59:36</start-task>
  <end-task>sumVet_sequential03-01-2012-13:59:36</end-task>
  <start-task>broadcastData03-01-2012-13:59:38</start-task>
  <end-task>broadcastData_./src/main/resources/data/matrix_j.xml;03-01-2012-13:59:38</end-task>
  <start-task>broadcastData03-01-2012-13:59:38</start-task>
  <end-task>broadcastData_./src/main/resources/data/matrix_j.xml;03-01-2012-13:59:38</end-task>
  <start-task>broadcastData03-01-2012-13:59:38</start-task>
  <end-task>broadcastData_./src/main/resources/data/matrix_j.xml;03-01-2012-13:59:38</end-task>
  <start-task>multVet_parallel03-01-2012-13:59:38</start-task>
  <end-task>multVet_parallel03-01-2012-13:59:38</end-task>
  <start-task>broadcastData03-01-2012-13:59:38</start-task>
  <end-task>broadcastData_./src/main/resources/data/matrix_out.xml;03-01-2012-13:59:38</end-task>
  <start-task>broadcastData03-01-2012-13:59:38</start-task>
  <end-task>broadcastData_./src/main/resources/data/matrix_out.xml;03-01-2012-13:59:38</end-task>
</log_tasks>
```

APÊNDICE C

ARQUIVOS DE PLANEJAMENTO

Neste apêndice apresenta-se exemplos de códigos que representam os arquivos de domínio (seção C.1), problema (seção C.2) e o plano gerado (seção C.3).

Os arquivos apresentados foram gerados automaticamente pelo tradutor que é apresentado em F.4. O método de tradução é detalhado em 4.8.

C.1 Domínio

Código C.1: Código representa o domínio de um problema em PDDL

```
(define (domain matrix)
(:requirements :typing :fluents)
    (:types data task hosts - object)
    (:predicates
        (have ?d - data)
        (availiable ?h - hosts)
        (half-availiable ?h - hosts)
        (working-full ?h - hosts ?t - task)
        (working-half ?h - hosts ?t - task)
        (input2 ?d1 - data ?d2 - data ?t - task)
        (output1 ?d1 - data ?t - task)
    )
    (:functions
        (total-cost)
    )
    (:action start-task-full2
    :parameters (?h - hosts ?t - task ?i1 - data ?i2 - data)
    :precondition (and (availiable ?h) (have ?i1) (have ?i2) (input2 ?i1 ?i2 ?t))
    :effect (and (working-half ?h ?t) (not (availiable ?h)) (increase (total-cost) 1))
    )
    (:action start-task-half2
    :parameters (?h - hosts ?t - task ?i1 - data ?i2 - data)
    :precondition (and (half-availiable ?h) (have ?i1) (have ?i2) (input2 ?i1 ?i2 ?t))
    :effect (and (working-full ?h ?t) (not (half-availiable ?h)) (increase (total-cost)↩
        10))
    )
    (:action end-task-full1
    :parameters (?h - hosts ?t - task ?i1 - data)
    :precondition (and (working-half ?h ?t) (output1 ?i1 ?t))
    :effect (and (availiable ?h) (have ?i1) (not (half-availiable ?h)) (not (working-↩
        half ?h ?t)))
    )
    (:action end-task-half1
    :parameters (?h - hosts ?t - task ?i1 - data)
    :precondition (and (working-full ?h ?t) (output1 ?i1 ?t))
    :effect (and (half-availiable ?h) (have ?i1) (not (working-full ?h ?t)))
    )
)
```

C.2 Problema

Código C.2: Código representa um problema em PDDL

```
(define (problem problem_matrix)
(:domain matrix)
    (:objects
        matrix_a matrix_b matrix_c matrix_d matrix_i matrix_j matrix_out - data
        sum_ab sum_cd mult_ij - task
        h0 h1 - hosts
    )
    (:init
        (have matrix_a)
        (have matrix_b)
        (have matrix_c)
        (have matrix_d)
        (input2 matrix_a matrix_b sum_ab)
        (input2 matrix_c matrix_d sum_cd)
        (input2 matrix_i matrix_j mult_ij)
        (output1 matrix_i sum_ab)
        (output1 matrix_j sum_cd)
        (output1 matrix_out mult_ij)
        (availiable h0)
        (availiable h1)
        (= (total-cost) 0)
    )
    (:goal
        (have matrix_out)
    )
    (:metric minimize (total-cost))
)
```

C.3 Plano Gerado

Nota-se que o plano gerado pela execução do plaejador *CRIKEY* (seção 2.5.4) apresenta ações paralelas. A ação 0.01 é formada por duas instruções, bem como a ação 0.02.

Código C.3: Saída que representa um plano gerado pelo *CRIKEY*

```
0.01: (start-task-full2 h0 sum_cd matrix_c matrix_d)
0.01: (start-task-full2 h1 sum_ab matrix_a matrix_b)
0.02: (end-task-full1 h1 sum_ab matrix_i)
0.02: (end-task-full1 h0 sum_cd matrix_j)
0.03: (start-task-full2 h0 mult_ij matrix_i matrix_j)
0.04: (end-task-full1 h0 mult_ij matrix_out)
```

APÊNDICE D

DESCRIÇÃO DOS COMPONENTES

Neste apêndice apresenta-se a descrição dos componentes apresentados Visão Detalhada do Sistema (seção 4.3), bem como a descrição dos componentes auxiliares utilizados para a implementação do sistema. Cada componente representa uma classe Java.

A organização dos componentes pode ser observada na figura D.1.

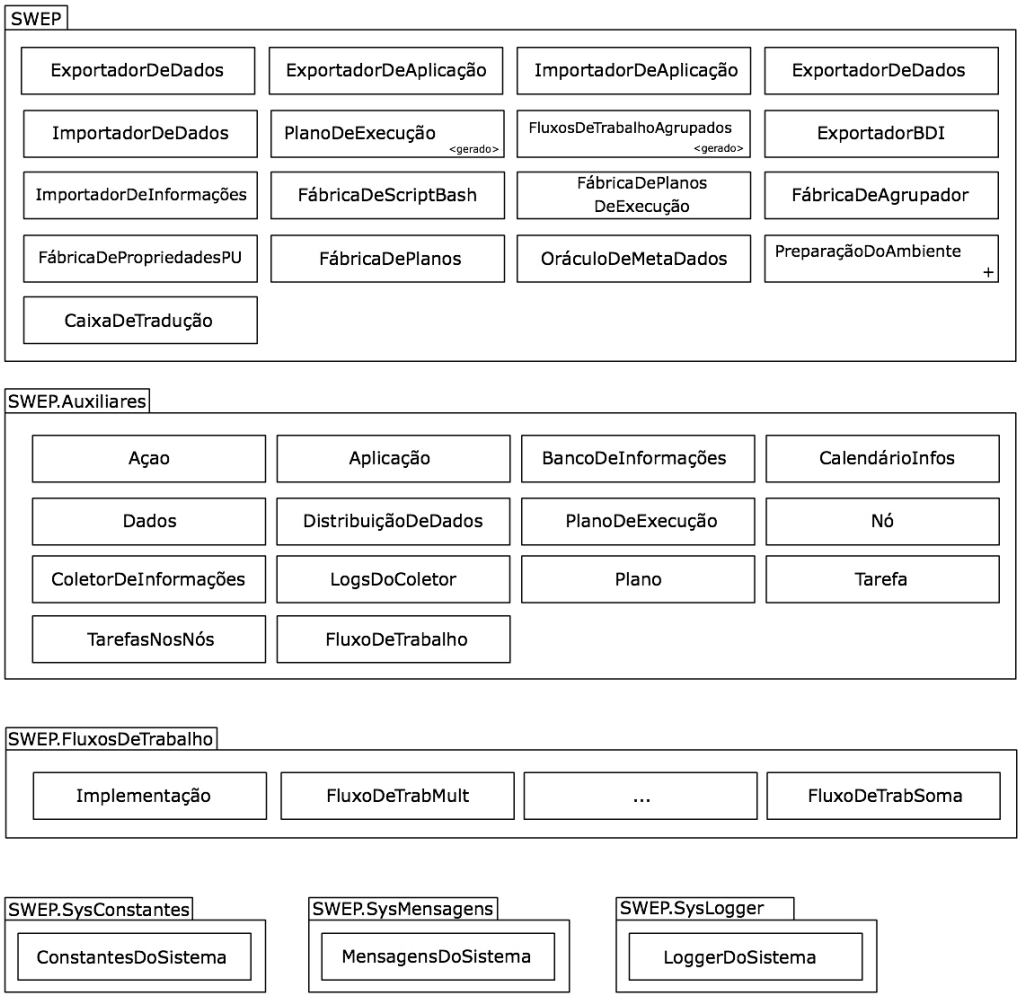

Figura D.1: Componentes do sistema

A tabela D.1 mostra a descrição de cada um dos componentes.

Tabela D.1: Descrição dos componentes.

SWEP	
ExportadorDeAções	Exporta as ações para geração de um plano de execução.
ExportadorDeAplicação	Exporta as informações da aplicação.
ImportadorDeAplicação	Importa as informações da aplicação.
ExportadorDeDados	Exporta os dados para um formato aceitável pelo sistema.
ImportadorDeDados	Importa os dados.
PlanoDeExecução	É um plano de execução (Gerado automaticamente).
FluxosDeTrabalhoAgrupados	Junção de todos os operadores diponíveis (Gerado automaticamente).
ExportadorBDI	Exporta uma base de dados do coletor de informações.
ImportadorDeInformações	Importa uma base de dados do coletor de informações.
FábricaDeScriptBash	Cria um arquivo bash para a execução do ambiente.
FábricaDePlanosDeExecução	Cria um plano de execução com base na ações exportadas.
FábricaDeAgrupador	Cria o arquivo JoinWorkFlow que reune todos os operadores disponíveis.
FábricaDePropriedadePU	Cria o arquivo de propriedades do gerenciador de execução com as informações do sistema.
FábricaDePlanos	Invoca o planejador e gera um plano.
OráculoDeMetaDados	Retorna informações referente aos nós.
PreparaçãoDoAmbiente	Arquivo executável que coordena a execução do ambiente.
CaixaDeTradução	É responsável pela tradução das informações do sistema nos arquivos PDDL.
Auxiliares	
Ação	Representa uma ação.
Aplicação	Representa a aplicação.
BancoDeInformações	Representa uma base de coletores de informação.
CalendárioInfos	Retorna informações de data e hora.
Dados	Representa uma coleção de dados.
DistribuiçãoDeDados	Representa um sistema de paralelização de dados simplista.
PlanoDeExecução	Representa um plano de execução.
Nó	Representa um nó.
ColetorDeInformações	Representa um coletor de informação em um nó.
LogsDoColetor	Representa um coletor de log de tarefas executadas.
Plano	Representa um plano.
Tarefa	Representa uma tarefa.
FluxoDeTrabalho	Representa quais tarefas foram executadas por cada nó.
WorkFlow	Representa um fluxo de trabalho.
Fluxos de Trabalho	
Implementação	É a base para implementação dos demais operadores.
FLuxoDeTrabMult	Representa um operador que executa a multiplicação de matrizes.
FluxoDeTrabSoma	Representa um operador que executa a soma de matrizes.
SysConstantes	
ConstantesDoSistema	Armazena as constantes do sistema.
SysMensagens	
MensagensDoSistema	Armazena as mensagens do sistema.
SysLogger	
LoggerDoSistema	Contém esquema de geração de logs do sistema.

APÊNDICE E

ARQUIVO DE CONFIGURAÇÃO DO PEERUNIT

Neste apêndice apresenta-se o arquivo de configuração do ambiente de execução *PeerUnit* descrito na seção 4.2.1.

Código E.1: Arquivo de configuração do PeerUnit

```
1  # Number of peers that will be in the simulation
2  tester.hostfile=hosts.txt
3  tester.peers=4
4  tester.server=127.0.0.1
5  tester.port=8181
6  # Log level order SEVERE,WARNING,INFO,CONFIG,FINE,FINER,FINEST
7  tester.log.level=FINEST
8  tester.logfolder=./logs
9  # Coordination Type
10 # (0) centralized, (1) btree
11 test.coordination=2
12 test.treeOrder=5
13 test.jar=${basedir}/target/Benchmark-1.0.jar
14 testcase.class=test.MainTest
15 # Test strategy - fr.inria.peerunit.coordinator.<SequencialStrategy|DependencyStrategy|↩
       HierarchicalStrategy|GlobalStrategy>
16 fr.inria.peerunit.coordinator.strategy=fr.inria.peerunit.coordinator.GlobalStrategy
```

APÊNDICE F

CÓDIGOS FONTE

Neste apêndice apresenta-se exemplos de códigos fonte utilizados para na implementação do sistema.

F.1 Chamada do Planejador

O código F.1 mostra o arquivo que corresponde ao componente *FábricaDePlanos*. Nesse componente o método *doMakePlan()* é que invoca o planejador *Crikey*. O sistema aguarda até que o arquivo de saída seja gravado, e em seguida o utiliza para a geração do arquivo contendo as ações.

Código F.1: Exemplo que invoca o planejador

```
1  package swep;
2  import java.io.File;
3  import java.util.LinkedList;
4  import swep.Beans.Action;
5  import swep.Beans.Plan;
6  import swep.SysConstants.SystemConstants;
7  import swep.SysLogger.SystemLogger;
8  import swep.SysMessages.SystemMessages;
9  /**
10  * Class responsible to manage the generation and exportion in actions of plan.
11  * @author Diogo Cezar Teixeira Batista
12  * @version 0.0.2
13  * @since aug-2011
14  */
15 public class MakePlan {
16     /**
17      * Path of problem file.
18      */
19     private String problemFileOut;
20     /**
21      * Path of domain file.
22      */
23     private String domainFileOut;
24     /**
25      * The plan.
26      */
27     private Plan plan;
28     /**
29      * Default constructor.
30      */
31     public MakePlan(){
32     }
33     /**
```

```
34      * Alternative constructor.
35      * @param problemFileOut File of problem.
36      * @param domainFileOut File of domain.
37      * @param uniqueKey Unique key for each execution.
38      */
39     public MakePlan(String problemFileOut, String domainFileOut, String uniqueKey){
40         this.setProblemFileOut(problemFileOut);
41         this.setDomainFileOut (domainFileOut);
42         this.setPlan(new Plan(SystemConstants.PLAN + uniqueKey + ".txt", uniqueKey));
43         this.doMakePlan();
44         this.getPlan().fillPlan();
45         LinkedList<Action> actions = this.getPlan().getActions();
46         SystemLogger.addInfo("Exporting Actions.");
47         // calling export actions
48         this.getPlan().exportActions(actions, uniqueKey);
49     }
50     private void doMakePlan(){
51         try {
52             if(this.checkFile(this.getDomainFileOut()) && this.checkFile(this.↩
                   getProblemFileOut())){
53                 String command = "java -jar ./lib/CRIKEY.jar " + this.getDomainFileOut↩
                       () + " " + this.getProblemFileOut() + " " + this.getPlan().↩
                       getFileName();
54                 Runtime rt = Runtime.getRuntime();
55                 Process pr = rt.exec(command);
56                 pr.waitFor();
57             }
58         } catch (Exception ex) {
59             System.out.println(SystemMessages.EXMSG_MAKING_PLAN + ex.getMessage());
60         }
61     }
62     /**
63      * Method that checks if a file exists
64      * @param fileName Path of file.
65      * @return If file exists.
66      */
67     private boolean checkFile(String fileName){
68         boolean exists = (new File(fileName)).exists();
69         if (exists){
70             return true;
71         }
72         else{
73             return false;
74         }
75     }
76     /**
77      * @return the problemFileOut
78      */
79     public String getProblemFileOut() {
80         return problemFileOut;
81     }
82     /**
83      * @param problemFileOut the problemFileOut to set
84      */
85     private void setProblemFileOut(String problemFileOut) {
86         this.problemFileOut = problemFileOut;
87     }
88     /**
89      * @return the domainFileOut
90      */
91     public String getDomainFileOut() {
92         return domainFileOut;
93     }
94     /**
95      * @param domainFileOut the domainFileOut to set
96      */
97     private void setDomainFileOut(String domainFileOut) {
98         this.domainFileOut = domainFileOut;
99     }
100    /**
101     * @return the plan
102     */
103    private Plan getPlan() {
```

```
104         return plan;
105     }
106     /**
107      * @param plan the plan to set
108      */
109     private void setPlan(Plan plan) {
110         this.plan = plan;
111     }
112 }
```

F.2 Plano de Execução

O código F.2 exemplifica um plano de execução gerado automaticamente a partir do arquivo de ações gerado pelo planejador.

Código F.2: Exemplo de um plano de execução

```
1  package swep;
2  import fr.inria.peerunit.remote.GlobalVariables;
3  import fr.inria.peerunit.parser.SetGlobals;
4  import fr.inria.peerunit.parser.TestStep;
5  import fr.inria.peerunit.parser.AfterClass;
6  import fr.inria.peerunit.parser.BeforeClass;
7  import fr.inria.peerunit.parser.SetId;
8  import java.rmi.RemoteException;
9  import swep.SysConstants.SystemConstants;
10 import swep.Beans.LeechInformation;
11 import swep.Beans.LeechLogger;
12 /**
13  * Execution Plan
14  * This file were generated automatically.
15  */
16 public class ExecutionPlanGenerated {
17     /**
18      * globals is accessible on all peers
19      */
20     private GlobalVariables globals;
21     /**
22      * id is a variable to get a peer identification
23      */
24     private int id;
25     /**
26      * leech
27      */
28     private LeechLogger leechLogger;
29     /**
30      * Unique Key Execution
31      */
32     public static String uniqueKey = "03-01-2012-13:59:20";
33     @SetId
34     public void setId(int i) {
35         id = i;
36     }
37     @BeforeClass(range="*")
38     public void begin(){
39         /**
40          * Starting LeechLogger
41          */
42          leechLogger = new LeechLogger();
43     }
44     @TestStep(range = "0", order = 1, timeout = 1000)
```

```
45      public void host_0_order_1_workflow_1() throws RemoteException {
46          JoinWorkFlows.sumVet_sequential(globals, id, 1, leechLogger);
47      }
48      @TestStep(range = "1", order = 1, timeout = 1000)
49      public void host_1_order_1_workflow_0() throws RemoteException {
50          JoinWorkFlows.sumVet_sequential(globals, id, 0, leechLogger);
51      }
52      @TestStep(range = "*", order = 2, timeout = 1000)
53      public void host_all_order_2_workflow_0() throws RemoteException {
54          JoinWorkFlows.broadcastResults_sumVet(globals, 1, 0, leechLogger);
55      }
56      @TestStep(range = "*", order = 2, timeout = 1000)
57      public void host_all_order_2_workflow_1() throws RemoteException {
58          JoinWorkFlows.broadcastResults_sumVet(globals, 0, 1, leechLogger);
59      }
60      @TestStep(range = "0-1", order = 3, timeout = 1000)
61      public void host_0_1_order_3_workflow_2() throws RemoteException {
62          JoinWorkFlows.multVet_parallel(globals, id, 2, leechLogger);
63      }
64      @TestStep(range = "0-1", order = 4, timeout = 1000)
65      public void host_0_1_order_4_workflow_2() throws RemoteException {
66          JoinWorkFlows.broadcastResults_multVet_parallel(globals, id, 2, leechLogger);
67      }
68      @TestStep(range = "*", order = 5, timeout = 1000)
69      public void leechInformation() throws RemoteException {
70          LeechInformation li = new LeechInformation(id, leechLogger);
71          String fileName = SystemConstants.METADATA + "information_h" + id + ".xml";
72          li.appendToXML(fileName);
73      }
74      @AfterClass(range="*", timeout = 1500)
75      public void end() {
76      }
77      @SetGlobals
78      public void setGlobals(GlobalVariables gv) {
79          globals = gv;
80      }
81  }
```

F.3 Fluxo de Trabalho de Exemplo

Apresenta-se um exemplo da implementação de um operador. O código F.3 apresenta o componente que é herdado pelos operadores. O código F.4 mostra a implementação de um operador para soma de matrizes.

F.3.1 Implementação Base

Código F.3: Implementação básica para fluxos de trabalho

```
1  package swep.WorkFlows;
2  import fr.inria.peerunit.remote.GlobalVariables;
3  import java.rmi.RemoteException;
4  import java.util.LinkedList;
5  import swep.ApplicationImporter;
6  import swep.Beans.Data;
7  import swep.Beans.DataDistribution;
8  import swep.Beans.LeechLogger;
```

```
9  import swep.DataExporter;
10 import swep.SysConstants.SystemConstants;
11 import swep.SysMessages.SystemMessages;
12 import swep.Beans.WorkFlow;
13 import swep.ExecutionPlanGenerated;
14 /**
15  * This class is inherited by WorkFLows
16  * @see WorkflowMultVet
17  * @see WorkflowSumVet
18  * @author Diogo Cezar Teixeira Batista
19  * @version 0.0.2
20  * @since aug-2011
21  */
22 public abstract class Implementation {
23     /**
24      * Object that will distribut data in a basic model
25      */
26     private DataDistribution dataDistribution;
27     /**
28      * Object that represents a workflow that will be executed.
29      */
30     private WorkFlow workFlow;
31     /**
32      * Default constructor.
33      * @param idWorkFlow Id of workflow
34      */
35     public Implementation(int idWorkFlow){
36         this.setWorkFlow(new WorkFlow());
37         this.populate(idWorkFlow);
38     }
39     /**
40      * Method that popule workflow with definitions in XML application file.
41      * @param idWorkflow
42      */
43     private void populate(int idWorkflow){
44         ApplicationImporter ai = new ApplicationImporter(SystemConstants.APPLICATION + ↩
               ExecutionPlanGenerated.uniqueKey + ".xml");
45         this.setWorkFlow(ai.getWorkflowById(idWorkflow));
46     }
47     /**
48      * Method used to debug system. Prints a matrix.
49      * @param matrix Matrix to print.
50      */
51     public static void print(int[][] matrix){
52         int limit = matrix.length;
53         for(int i=0; i<limit; i++){
54             for(int j=0; j<limit; j++){
55                 System.out.print(matrix[i][j] + " ");
56             }
57             System.out.println();
58         }
59         System.out.println();
60         System.out.println();
61     }
62     /**
63      * Abstract method that will be implemented on heirs.
64      */
65     abstract void populateData();
66     /**
67      * Method that broadcast result of a workflow execution.
68      * @param globals Global variable of PeerUnit.
69      * @param hostId Id of host that is executing broadcast.
70      * @param listFileName List of files that will be saved informations.
71      * @param leechLogger Object to control log.
72      * @param IdWorkFlow The id of workflow that will be executed.
73      */
74     public static void broadcastData(GlobalVariables globals, int hostId, LinkedList<↩
           Data> listFileName, LeechLogger leechLogger, int IdWorkFlow){
75         try{
76             leechLogger.startTask(hostId, "broadcastData");
77             Object obj = Implementation.getValueGlobal(globals, hostId);
78             String logFiles = "";
79             if(obj instanceof LinkedList){
```

```
80                  LinkedList<Object> listObjects = (LinkedList<Object>) obj;
81                  for(int i=0; i<listObjects.size(); i++){
82                      if(listObjects.get(i) instanceof int[][]){
83                              int[][] matrix = new int[5][5];
84                              matrix = (int[][]) listObjects.get(i);
85                              DataExporter de = new DataExporter();
86                              Data data       = new Data(matrix, listFileName.get(i).↩
                                    getFileName());
87                              de.exportData(data);
88                              logFiles += listFileName.get(i).getFileName() +  ";";
89                      }
90                  }
91                  leechLogger.endTask(hostId, "broadcastData_" + logFiles);
92              }
93              else{
94                  leechLogger.endTask(hostId, SystemMessages.EMSG_EXPORTING_DATA);
95              }
96          }
97          catch(Exception ex){
98          }
99      }
100     /**
101      * Method that broadcast result of a workflow execution in a simple parallel model.
102      * @param globals Global variable of PeerUnit.
103      * @param leechLogger Object to control log.
104      * @param listFileName List of files that will be saved informations.
105      */
106     public static void broadcastDataParallel(GlobalVariables globals, LeechLogger ↩
            leechLogger, LinkedList<Data> listFileName){
107         try{
108             leechLogger.startTask(0, "broadcastData");
109             Object obj = Implementation.getValueGlobal(globals, 0);
110             String logFiles = "";
111             if(obj instanceof LinkedList){
112                 LinkedList<Object> listObjects = (LinkedList<Object>) obj;
113                 for(int i=0; i<listObjects.size(); i++){
114                     if(listObjects.get(i) instanceof int[][]){
115                         int[][] matrix = new int[5][5];
116                         matrix = (int[][]) listObjects.get(i);
117                         DataExporter de = new DataExporter();
118                         Data data       = new Data(matrix, listFileName.get(i).↩
                                getFileName());
119                         de.exportData(data);
120                         logFiles += listFileName.get(i).getFileName() +  ";";
121                     }
122                 }
123                 leechLogger.endTask(0, "broadcastData_" + logFiles);
124             }
125             else{
126                 leechLogger.endTask(0, SystemMessages.EMSG_EXPORTING_DATA);
127             }
128         }
129         catch(Exception ex){
130         }
131     }
132     /**
133      * Method that insert a variable in globals.
134      * @param globals Global variable of PeerUnit.
135      * @param hostId Id of host.
136      * @param objectPut Object to save.
137      */
138     public static void insertValueGlobal (GlobalVariables globals, int hostId, Object ↩
            objectPut){
139         try {
140             globals.put(hostId, objectPut);
141         } catch (RemoteException ex) {
142             System.out.println(SystemMessages.EMSG_PUT_GLOBAL + ex.getMessage());
143         }
144     }
145     /**
146      * Method that return a variable in globals.
147      * @param globals Global variable of PeerUnit.
148      * @param hostId Id of host.
```

```
149      * @return Object to return.
150      */
151     public static Object getValueGlobal (GlobalVariables globals, int hostId){
152         try{
153             return globals.get(hostId);
154         } catch (RemoteException ex) {
155             System.out.println(SystemMessages.EMSG_GET_GLOBAL + ex.getMessage());
156             return null;
157         }
158     }
159     /**
160      * Method that distribute automatically the data in simple parallel model.
161      * @param hostId
162      */
163     protected void distribute(int hostId){
164         if(this.getWorkFlow().getChunk() == -1){
165             this.distributeByHosts(hostId);
166         }
167         else{
168             this.distributeByChunk(hostId);
169         }
170     }
171     /**
172      * Method that distribut data by chunk size in basic parallel model.
173      * @param hostId Id of host.
174      */
175     protected void distributeByChunk(int hostId){
176         this.getDataDistribution().distributeByChunk(hostId, this.getWorkFlow());
177     }
178     /**
179      * Method that distribut data by number of hosts in basic parallel model.
180      * @param hostId
181      */
182     protected void distributeByHosts(int hostId){
183         this.getDataDistribution().distributeByHosts(hostId, this.getWorkFlow());
184     }
185     /**
186      * @return the dataDistribution
187      */
188     public DataDistribution getDataDistribution() {
189         return dataDistribution;
190     }
191     /**
192      * @param dataDistribution the dataDistribution to set
193      */
194     protected void setDataDistribution(DataDistribution dataDistribution) {
195         this.dataDistribution = dataDistribution;
196     }
197     /**
198      * @return the workFlow
199      */
200     public WorkFlow getWorkFlow() {
201         return workFlow;
202     }
203     /**
204      * @param workFlow the workFlow to set
205      */
206     private void setWorkFlow(WorkFlow workFlow) {
207         this.workFlow = workFlow;
208     }
209 }
```

F.3.2 Fluxo de Trabalho para Soma de Matrizes

Código F.4: Fluxo de trabalho para soma de matrizes

```
1  package swep.WorkFlows;
2  import fr.inria.peerunit.remote.GlobalVariables;
3  import java.util.LinkedList;
4  import swep.Beans.Data;
5  import swep.Beans.LeechLogger;
6  import swep.DataImporter;
7  import swep.SysMessages.SystemMessages;
8  /**
9   * This class implements an operator that sum vectors.
10  * @see Implementation
11  * @see WorkflowMultVet
12  * @author Diogo Cezar Teixeira Batista
13  * @version 0.0.2
14  * @since aug-2011
15  */
16 public class WorkflowSumVet extends Implementation {
17     /**
18      * Source matrix a.
19      */
20     private int[][] matrix_a;
21     /**
22      * Source matrix b.
23      */
24     private int[][] matrix_b;
25     /**
26      * Result matrix c.
27      */
28     private int[][] matrix_c;
29     /**
30      * List of results that will be saved.
31      */
32     private LinkedList<Object> listObjects;
33     /**
34      * Default constructor.
35      * @param idWorkFlow Id of workflow.
36      */
37     public WorkflowSumVet(int idWorkFlow){
38         super(idWorkFlow);
39         int size = 5;
40         this.setMatrix_a(new int[size][size]);
41         this.setMatrix_b(new int[size][size]);
42         this.setMatrix_c(new int[size][size]);
43         this.setListObjects(new LinkedList<Object>());
44         this.populateData();
45     }
46     /**
47      * Method that reset the matrix of results.
48      */
49     private void resetMatrix(){
50         int limit = getMatrix_c().length;
51         for(int i=0; i<limit; i++){
52             for(int j=0; j<limit; j++){
53                 getMatrix_c()[i][j] = 0;
54             }
55         }
56     }
57     /**
58      * Method used to debug system. Prints a matrix.
59      * @param matrix Matrix to print.
60      */
61     private void printMatrix(int[][] matrix){
62         int limit = matrix.length;
63         for(int i=0; i<limit; i++){
64             for(int j=0; j<limit; j++){
65                 System.out.print(matrix[i][j] + " ");
66             }
67             System.out.println();
68         }
69         System.out.println();
70         System.out.println();
71     }
```

```
72      /**
73       * Method that saves a result in a list of results.
74       * @param globals Global variable of PeerUnit.
75       * @param hostId Id of host will save information.
76       */
77      private void save(GlobalVariables globals, int hostId){
78          Implementation.insertValueGlobal(globals, hostId, getListObjects());
79      }
80      /**
81       * Method that sum a sequential matrix.
82       * @param globals Global variable of PeerUnit.
83       * @param hostId Id of host will process matrix.
84       * @param leechLogger Leech logger to record this action.
85       */
86      public void sumVet_sequential(GlobalVariables globals, int hostId, LeechLogger ↩
            leechLogger){
87          try{
88              leechLogger.startTask(hostId, "sumVet_sequential");
89              int limit = getMatrix_a().length;
90              this.resetMatrix();
91              for(int i=0; i<limit; i++){
92                  for(int j=0; j<limit; j++){
93                      getMatrix_c()[i][j] = getMatrix_a()[i][j]+getMatrix_b()[i][j];
94                  }
95              }
96
97              this.getListObjects().add(getMatrix_c());
98              this.save(globals, hostId);
99              leechLogger.endTask(hostId, "sumVet_sequential");
100         }
101         catch(Exception ex){
102             System.out.println(SystemMessages.EMSG_EXECUTE_TASK + ex.getMessage());
103         }
104     }
105     /**
106      * Method that broadcast the result for all other hosts.
107      * @param globals Global variable of PeerUnit.
108      * @param hostId Id of host will broadcast information.
109      * @param leechLogger Leech logger to record this action.
110      * @param idWorkFlow  Id of workflow that will be executed.
111      */
112     public void broadcastData(GlobalVariables globals, int hostId, LeechLogger ↩
            leechLogger, int idWorkFlow){
113         Implementation.broadcastData(globals, hostId, this.getWorkFlow().↩
                getListDataOutput(), leechLogger, idWorkFlow);
114     }
115     /**
116      * Method that broadcast parallel the result for all other hosts.
117      * @param globals Global variable of PeerUnit.
118      * @param leechLogger Leech logger to record this action.
119      */
120     public void broadcastDataParallel(GlobalVariables globals, LeechLogger leechLogger)↩
            {
121         Implementation.broadcastDataParallel(globals, leechLogger, this.getWorkFlow().↩
                getListDataOutput());
122     }
123     /**
124      * Method that execute a simple parallel model.
125      * @param globals Global variable of PeerUnit.
126      * @param hostId Id of host will process matrix.
127      * @param leechLogger Leech logger to record this action.
128      */
129     public void sumVet_parallel(GlobalVariables globals, int hostId, LeechLogger ↩
            leechLogger){
130         leechLogger.startTask(hostId, "sumVet_parallel");
131         super.distribute(hostId);
132
133         int limit = getMatrix_a().length;
134         int start_data = this.getDataDistribution().getStartData();
135         int end_data   = this.getDataDistribution().getEndData();
136         this.resetMatrix();
137         for(int i=start_data; i<end_data; i++){
138             for(int j=0; j<limit; j++){
```

```
139                 getMatrix_c()[i][j] = getMatrix_a()[i][j]+getMatrix_b()[i][j];
140             }
141         }
142         this.getListObjects().add(getMatrix_c());
143         this.save(globals, hostId);
144         leechLogger.endTask(hostId, "sumVet_parallel");
145     }
146     /**
147      * Method that join results of a simple parallel model.
148      * @param globals Global variable of PeerUnit.
149      */
150     public void joinResult(GlobalVariables globals){
151         int numHosts = this.getWorkFlow().getHosts();
152         this.resetMatrix();
153         for(int i=0; i<numHosts; i++){
154             Object obj = Implementation.getValueGlobal(globals, i);
155             if(obj instanceof LinkedList){
156                 LinkedList<Object> listObjectsByHost = (LinkedList<Object>) obj;
157                 for(int j=0; j<listObjectsByHost.size(); j++){
158                     if(listObjectsByHost.get(j) instanceof int[][]){
159                         int[][] matrix = new int[5][5];
160                         matrix = (int[][]) listObjectsByHost.get(j);
161                         for(int k=0; k<matrix.length; k++){
162                             for(int l=0; l<matrix.length; l++){
163                                 if(matrix[k][l] != 0){
164                                     getMatrix_c()[k][l] = matrix[k][l];
165                                 }
166                             }
167                         }
168                     }
169                 }
170             }
171         }
172         this.getListObjects().add(getMatrix_c());
173         this.save(globals, 0);
174     }
175     /**
176      * Method that populate sources matrix with content in xml files.
177      */
178     final void populateData() {
179         String input_matrix_a = this.getWorkFlow().getListDataInput().get(0).↩
              getFileName();
180         String input_matrix_b = this.getWorkFlow().getListDataInput().get(1).↩
              getFileName();
181         Data data_matrix_a = new Data(this.getMatrix_a(), input_matrix_a);
182         Data data_matrix_b = new Data(this.getMatrix_b(), input_matrix_b);
183         DataImporter di_matrix_a = new DataImporter();
184         DataImporter di_matrix_b = new DataImporter();
185         di_matrix_a.importData(data_matrix_a);
186         di_matrix_b.importData(data_matrix_b);
187         this.setMatrix_a(di_matrix_a.getData().getDataInt());
188         this.setMatrix_b(di_matrix_b.getData().getDataInt());
189     }
190     /**
191      * @return the matrix_a
192      */
193     public int[][] getMatrix_a() {
194         return matrix_a;
195     }
196     /**
197      * @param matrix_a the matrix_a to set
198      */
199     private void setMatrix_a(int[][] matrix_a) {
200         this.matrix_a = matrix_a;
201     }
202     /**
203      * @return the matrix_b
204      */
205     public int[][] getMatrix_b() {
206         return matrix_b;
207     }
208     /**
209      * @param matrix_b the matrix_b to set
```

```
210         */
211        private void setMatrix_b(int[][] matrix_b) {
212            this.matrix_b = matrix_b;
213        }
214        /**
215         * @return the matrix_c
216         */
217        public int[][] getMatrix_c() {
218            return matrix_c;
219        }
220        /**
221         * @param matrix_c the matrix_c to set
222         */
223        private void setMatrix_c(int[][] matrix_c) {
224            this.matrix_c = matrix_c;
225        }
226        /**
227         * @return the listObjects
228         */
229        public LinkedList<Object> getListObjects() {
230            return listObjects;
231        }
232        /**
233         * @param listObjects the listObjects to set
234         */
235        private void setListObjects(LinkedList<Object> listObjects) {
236            this.listObjects = listObjects;
237        }
238    }
```

F.4 Tradutor

O código F.5 demonstra todo o modelo detalhado na seção 4.8.

Código F.5: Código que faz a tradução do sistema para os arquivos PDDL

```
1   package swep.WorkFlows;
2   import fr.inria.peerunit.remote.GlobalVariables;
3   import java.util.LinkedList;
4   import swep.Beans.Data;
5   import swep.Beans.LeechLogger;
6   import swep.DataImporter;
7   import swep.SysMessages.SystemMessages;
8   /**
9    * This class implements an operator that sum vectors.
10   * @see Implementation
11   * @see WorkflowMultVet
12   * @author Diogo Cezar Teixeira Batista
13   * @version 0.0.2
14   * @since aug-2011
15   */
16  public class WorkflowSumVet extends Implementation {
17      /**
18       * Source matrix a.
19       */
20      private int[][] matrix_a;
21      /**
22       * Source matrix b.
23       */
24      private int[][] matrix_b;
25      /**
26       * Result matrix c.
```

```
27          */
28         private int[][] matrix_c;
29         /**
30          * List of results that will be saved.
31          */
32         private LinkedList<Object> listObjects;
33         /**
34          * Default constructor.
35          * @param idWorkFlow Id of workflow.
36          */
37         public WorkflowSumVet(int idWorkFlow){
38             super(idWorkFlow);
39             int size = 5;
40             this.setMatrix_a(new int[size][size]);
41             this.setMatrix_b(new int[size][size]);
42             this.setMatrix_c(new int[size][size]);
43             this.setListObjects(new LinkedList<Object>());
44             this.populateData();
45         }
46         /**
47          * Method that reset the matrix of results.
48          */
49         private void resetMatrix(){
50             int limit = getMatrix_c().length;
51             for(int i=0; i<limit; i++){
52                 for(int j=0; j<limit; j++){
53                     getMatrix_c()[i][j] = 0;
54                 }
55             }
56         }
57         /**
58          * Method used to debug system. Prints a matrix.
59          * @param matrix Matrix to print.
60          */
61         private void printMatrix(int[][] matrix){
62             int limit = matrix.length;
63             for(int i=0; i<limit; i++){
64                 for(int j=0; j<limit; j++){
65                     System.out.print(matrix[i][j] + " ");
66                 }
67                 System.out.println();
68             }
69             System.out.println();
70             System.out.println();
71         }
72         /**
73          * Method that saves a result in a list of results.
74          * @param globals Global variable of PeerUnit.
75          * @param hostId Id of host will save information.
76          */
77         private void save(GlobalVariables globals, int hostId){
78             Implementation.insertValueGlobal(globals, hostId, getListObjects());
79         }
80         /**
81          * Method that sum a sequential matrix.
82          * @param globals Global variable of PeerUnit.
83          * @param hostId Id of host will process matrix.
84          * @param leechLogger Leech logger to record this action.
85          */
86         public void sumVet_sequential(GlobalVariables globals, int hostId, LeechLogger ↩
                leechLogger){
87             try{
88                 leechLogger.startTask(hostId, "sumVet_sequential");
89                 int limit = getMatrix_a().length;
90                 this.resetMatrix();
91                 for(int i=0; i<limit; i++){
92                     for(int j=0; j<limit; j++){
93                         getMatrix_c()[i][j] = getMatrix_a()[i][j]+getMatrix_b()[i][j];
94                     }
95                 }
96 
97                 this.getListObjects().add(getMatrix_c());
98                 this.save(globals, hostId);
```

```
99              leechLogger.endTask(hostId, "sumVet_sequential");
100         }
101         catch(Exception ex){
102             System.out.println(SystemMessages.EMSG_EXECUTE_TASK + ex.getMessage());
103         }
104     }
105     /**
106      * Method that broadcast the result for all other hosts.
107      * @param globals Global variable of PeerUnit.
108      * @param hostId Id of host will broadcast information.
109      * @param leechLogger Leech logger to record this action.
110      * @param idWorkFlow  Id of workflow that will be executed.
111      */
112     public void broadcastData(GlobalVariables globals, int hostId, LeechLogger ↩
            leechLogger, int idWorkFlow){
113         Implementation.broadcastData(globals, hostId, this.getWorkFlow().↩
                getListDataOutput(), leechLogger, idWorkFlow);
114     }
115     /**
116      * Method that broadcast parallel the result for all other hosts.
117      * @param globals Global variable of PeerUnit.
118      * @param leechLogger Leech logger to record this action.
119      */
120     public void broadcastDataParallel(GlobalVariables globals, LeechLogger leechLogger)↩
            {
121         Implementation.broadcastDataParallel(globals, leechLogger, this.getWorkFlow().↩
                getListDataOutput());
122     }
123     /**
124      * Method that execute a simple parallel model.
125      * @param globals Global variable of PeerUnit.
126      * @param hostId Id of host will process matrix.
127      * @param leechLogger Leech logger to record this action.
128      */
129     public void sumVet_parallel(GlobalVariables globals, int hostId, LeechLogger ↩
            leechLogger){
130         leechLogger.startTask(hostId, "sumVet_parallel");
131         super.distribute(hostId);
132 
133         int limit = getMatrix_a().length;
134         int start_data = this.getDataDistribution().getStartData();
135         int end_data   = this.getDataDistribution().getEndData();
136         this.resetMatrix();
137         for(int i=start_data; i<end_data; i++){
138             for(int j=0; j<limit; j++){
139                 getMatrix_c()[i][j] = getMatrix_a()[i][j]+getMatrix_b()[i][j];
140             }
141         }
142         this.getListObjects().add(getMatrix_c());
143         this.save(globals, hostId);
144         leechLogger.endTask(hostId, "sumVet_parallel");
145     }
146     /**
147      * Method that join results of a simple parallel model.
148      * @param globals Global variable of PeerUnit.
149      */
150     public void joinResult(GlobalVariables globals){
151         int numHosts = this.getWorkFlow().getHosts();
152         this.resetMatrix();
153         for(int i=0; i<numHosts; i++){
154             Object obj = Implementation.getValueGlobal(globals, i);
155             if(obj instanceof LinkedList){
156                 LinkedList<Object> listObjectsByHost = (LinkedList<Object>) obj;
157                 for(int j=0; j<listObjectsByHost.size(); j++){
158                     if(listObjectsByHost.get(j) instanceof int[][]){
159                         int[][] matrix = new int[5][5];
160                         matrix = (int[][]) listObjectsByHost.get(j);
161                         for(int k=0; k<matrix.length; k++){
162                             for(int l=0; l<matrix.length; l++){
163                                 if(matrix[k][l] != 0){
164                                     getMatrix_c()[k][l] = matrix[k][l];
165                                 }
166                             }
```

```
167                         }
168                     }
169                 }
170             }
171         }
172         this.getListObjects().add(getMatrix_c());
173         this.save(globals, 0);
174     }
175     /**
176      * Method that populate sources matrix with content in xml files.
177      */
178     final void populateData() {
179         String input_matrix_a = this.getWorkFlow().getListDataInput().get(0).↩
                getFileName();
180         String input_matrix_b = this.getWorkFlow().getListDataInput().get(1).↩
                getFileName();
181         Data data_matrix_a = new Data(this.getMatrix_a(), input_matrix_a);
182         Data data_matrix_b = new Data(this.getMatrix_b(), input_matrix_b);
183         DataImporter di_matrix_a = new DataImporter();
184         DataImporter di_matrix_b = new DataImporter();
185         di_matrix_a.importData(data_matrix_a);
186         di_matrix_b.importData(data_matrix_b);
187         this.setMatrix_a(di_matrix_a.getData().getDataInt());
188         this.setMatrix_b(di_matrix_b.getData().getDataInt());
189     }
190     /**
191      * @return the matrix_a
192      */
193     public int[][] getMatrix_a() {
194         return matrix_a;
195     }
196     /**
197      * @param matrix_a the matrix_a to set
198      */
199     private void setMatrix_a(int[][] matrix_a) {
200         this.matrix_a = matrix_a;
201     }
202     /**
203      * @return the matrix_b
204      */
205     public int[][] getMatrix_b() {
206         return matrix_b;
207     }
208     /**
209      * @param matrix_b the matrix_b to set
210      */
211     private void setMatrix_b(int[][] matrix_b) {
212         this.matrix_b = matrix_b;
213     }
214     /**
215      * @return the matrix_c
216      */
217     public int[][] getMatrix_c() {
218         return matrix_c;
219     }
220     /**
221      * @param matrix_c the matrix_c to set
222      */
223     private void setMatrix_c(int[][] matrix_c) {
224         this.matrix_c = matrix_c;
225     }
226     /**
227      * @return the listObjects
228      */
229     public LinkedList<Object> getListObjects() {
230         return listObjects;
231     }
232     /**
233      * @param listObjects the listObjects to set
234      */
235     private void setListObjects(LinkedList<Object> listObjects) {
236         this.listObjects = listObjects;
237     }
```

```
238 }
```

BIBLIOGRAFIA

[1] A. Akram, D. Meredith, e R. Allan. Evaluation of BPEL to Scientific Workflows. *Proceedings of 6th IEEE International Symposium on Cluster Computing and the Grid*, páginas 269–274. IEEE Computer Society, 2006.

[2] Eduardo Cunha De Almeida, Gerson Sunyé, Yves Le Traon, e Patrick Valduriez. A Framework for Testing Peer-to-Peer Systems. *2008 19th International Symposium on Software Reliability Engineering (ISSRE)*, páginas 167–176, novembro de 2008.

[3] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, e S. Mock. Kepler: an Extensible System for Design and Execution of Scientific Workflows. *Proceedings of the 16th International Conference on Scientific and Statistical Database Management*, páginas 423–424. IEEE Computer Society, 2004.

[4] Stephanos Androutsellis-Theotokis e Diomidis Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Comput. Surv.*, 36(4):335–371, dezembro de 2004.

[5] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, e M. Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. Relatório técnico, EECS Department, University of California, Berkeley, 2009.

[6] K. Belhajjame, K. Wolstencroft, O. Corcho, T. Oinn, F. Tanoh, A. William, e C. Goble. Metadata Management in the Taverna Workflow System. *Journal of 8th IEEE International Symposium on Cluster Computing and the Grid*, 0:651–656, maio de 2008.

[7] Avrim L. Blum e Merrick L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1):1636–1642, 1995.

[8] Blai Bonet e Hector Geffner. Planning as heuristic search. *Artificial Intelligence*, 129:5–33, 2001.

[9] S.P. Callahan, Juliana Freire, Emanuele Santos, C.E. Scheidegger, C.T. Silva, e H.T. Vo. VisTrails: visualization meets data management. *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, páginas 745–747. ACM, 2006.

[10] Yongsun Choi. A two phase verification algorithm for cyclic workflow graphs. Jian Chen, editor, *ICEB*, páginas 137–143. Academic Publishers/World Publishing Corporation, 2004.

[11] Andrew Coles, Maria Fox, Keith Halsey, Derek Long, e Amanda Smith. Managing concurrency in temporal planning using planner-scheduler interaction. *Artif. Intell.*, 173(1):1–44, 2009.

[12] Daniel de Oliveira, Eduardo S. Ogasawara, Fernanda Araujo Baião, e Marta Mattoso. Scicumulus: A lightweight cloud middleware to explore many task computing paradigm in scientific workflows. *IEEE CLOUD*, páginas 378–385, 2010.

[13] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M. H. Su, K. Vahi, e M. Livny. *Pegasus: Mapping Scientific Workflows onto the Grid*, volume 0086044, páginas 131–140. Springer Berlin / Heidelberg, 2004.

[14] Ewa Deelman, Dennis Gannon, Matthew Shields, e Ian Taylor. Workflows and e-science: An overview of workflow system features and capabilities. *Future Generation Computer Systems*, 25(5):528 – 540, 2009.

[15] T. Dörnemann, E. Juhnke, T. Noll, D. Seiler, e B. Freisleben. Data Flow Driven Scheduling of BPEL Workflows Using Cloud Resources. *Proceedings of 3rd International Conference on Cloud Computing*, páginas 196–203. IEEE Computer Society, julho de 2010.

[16] Johan Eker, Jorn Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Sonia Sachs, e Yuhong Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, January de 2003.

[17] Richard E. Fikes e Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4):189–208, janeiro de 1971.

[18] Ian Foster, J. Vockler, M. Wilde, e Y. Zhao. Chimera: A virtual data system for representing, querying, and automating data derivation. *Scientific and Statistical Database Management, 2002. Proceedings. 14th International Conference on*, páginas 37–46. IEEE, 2002.

[19] Maria Fox e Derek Long. pddl2 . 1 : An Extension to pddl for Expressing Temporal Planning Domains. *Information Sciences*, 20:61–124, 2003.

[20] Yolanda Gil, Ewa Deelman, Jim Blythe, Carl Kesselman, e H. Tangmunarunkit. Artificial Intelligence and Grids : Workflow Planning and Beyond. *Intelligent Systems, IEEE*, 19(1):26–33, 2004.

[21] Keith Halsey. The workings of CRIKEY - a temporal metric planner. *Proc. Int. Conf. on Automated Planning and Scheduling (ICAPS-2004) International Planning Competition*, páginas 35–37, 2004.

[22] Jorg Hoffmann. Extending ff to numerical state variables. *Proceedings of the 15Th European Conference on Artificial Intelligence*, páginas 571–575. Wiley, 2002.

[23] Okhtay Ilghami. Documentation for JSHOP2. *Relatório técnico, Department of Computer Science, University of Maryland*, 51, 2006.

[24] Henry Kautz. Planning as satisfiability. *Proceedings of the European Conference on*, (August 1992):1–12, 1992.

[25] Henry A. Kautz, Bart Selman, e Jörg Hoffmann. SatPlan: Planning as satisfiability. *Abstracts of the 5th International Planning Competition*, 2006.

[26] Kevin Lee, Norman W. Paton, Rizos Sakellariou, Ewa Deelman, Alvaro A. A. Fernandes, e Gaurang Mehta. Adaptive Workflow Processing and Execution in Pegasus. *Proceedings of the 2008 The 3rd International Conference on Grid and Pervasive Computing - Workshops*, páginas 99–106, Washington, DC, USA, 2008. IEEE Computer Society.

[27] Simon Miles, Paul Groth, Miguel Branco, e Luc Moreau. The requirements of recording and using provenance in e-science experiments. Relatório técnico, Journal of Grid Computing, 2005.

[28] Simon Miles, Sylvia C. Wong, Weijian Fang, Paul Groth, Klaus-Peter Zauner, e Luc Moreau. Provenance-based validation of e-science experiments. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5(1):28–38, março de 2007.

[29] D. Nau, Y. Cao, A. Lotem, e H. Muñoz Avila. SHOP: Simple hierarchical ordered planner. *Proceedings of the 16th international joint conference on Artificial intelligence-Volume 2*, páginas 968–973. Morgan Kaufmann Publishers Inc., 1999.

[30] Dana Nau, T.C. Au, O. Ilghami, U. Kuter, J.W. Murdock, Dan Wu, e F. Yaman. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research*, 20(1):379–404, 2003.

[31] Raksha Sharma, Vishnu Kant Soni, Manoj Kumar Mishra, e Prachet Bhuyan. A Survey of Job Scheduling and Resource Management in Grid Computing. *Engineering and Technology*, páginas 461–466, 2010.

[32] Yogesh L. Simmhan, Beth Plale, e Dennis Gannon. A survey of data provenance in e-science. *ACM SIGMOD Record*, 34(3):31, setembro de 2005.

[33] B. Sotomayor, R. S. Montero, I. M. Llorente, e I. Foster. Virtual Infrastructure Management in Private and Hybrid Clouds. *Journal of IEEE Internet Computing*, 13(5):14–22, setembro de 2009.

[34] D.S. Weld. Recent advances in AI planning. *AI magazine*, 20(2):93, 1999.

DIOGO CEZAR TEIXEIRA BATISTA

UM MODELO DE EXECUÇÃO DE FLUXOS DE TRABALHO CIENTÍFICO UTILIZANDO TÉCNICAS DE PLANEJAMENTO AUTOMÁTICO

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Fabiano Silva

CURITIBA

2012