Universidade Estadual de Campinas
Faculdade de Engenharia Elétrica e de Computação

Uma Linguagem para Especificação de Fluxo de Execução em Aplicações Paralelas

Autora: Cristina Enomoto

Orientador: Marco Aurélio Amaral Henriques

Dissertação de Mestrado apresentada à Faculdade de Engenharia Elétrica e de Computação da Universidade Estadual de Campinas como parte dos requisitos necessários para obtenção do título de Mestre em Engenharia Elétrica. Área de concentração: **Engenharia de Computação.**

Banca Examinadora

Prof. Dr. Célio Cardoso Guimarães .....IC/Unicamp
Prof. Dr. Eleri Cardozo .....FEEC/Unicamp
Prof. Dr. José Raimundo de Oliveira.....FEEC/Unicamp
Prof. Dr. Marco Aurélio Amaral Henriques .....FEEC/Unicamp

Campinas
Agosto, 2005

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA - BAE - UNICAMP

Enomoto, Cristina
En58L Uma linguagem para especificação de fluxo de execução em aplicações paralelas / Cristina Enomoto. --Campinas, SP: [s.n.], 2005.

Orientador: Marco Aurélio Amaral Henriques
Dissertação (Mestrado) - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Computação em grade (Sistema de computadores). 2. Programação paralela (Computação). 3. Processamento paralelo (Computadores). 4. Fluxo de trabalho. I. Henriques, Marco Aurélio Amaral. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

RMS-BAE

Titulo em Inglês: A specification language for execution flow in parallel applications
Palavras-chave em Inglês: Computational grids (Computer systems), Parallel programming, Parallel processing e Workflow
Área de concentração: Engenharia de Computação
Titulação: Mestre em Engenharia Elétrica
Banca examinadora: Célio Cardoso Guimarães, Eleri Cardozo e José Raimundo de Oliveira.
Data da defesa: 22/08/2005

Resumo

Vários sistemas de *grid* e computação distribuída existentes só permitem a execução de aplicações com um fluxo de execução de tarefas básico, no qual é feita a distribuição das tarefas executadas em paralelo e depois a coleta de seus resultados. Outros sistemas permitem definir uma relação de dependências entre as tarefas, formando um grafo direcionado acíclico. Porém, mesmo com este modelo de fluxo de execução não é possível executar vários tipos de aplicações que poderiam ser paralelizadas, como, por exemplo, algoritmos genéticos e de cálculo numérico que utilizam algum tipo de processamento iterativo. Nesta dissertação é proposta uma linguagem de especificação para fluxo de execução de aplicações paralelas que permite um controle de fluxo de tarefas mais flexível, viabilizando desvios condicionais e laços com iterações controladas. A linguagem é baseada na notação XML (eXtensible Markup Language), o que lhe confere características importantes tais como flexibilidade e simplicidade. Para avaliar estas e outras características da linguagem proposta, foi feita uma implementação sobre o sistema de processamento paralelo JoiN. Além de viabilizar a criação e execução de novas aplicações paralelas cujos fluxos de tarefas contêm laços e/ou desvios condicionais, a linguagem se mostrou simples de usar e não causou sobrecarga perceptível ao sistema paralelo.

Palavras-chave: Computação em grade, Programação paralela, Processamento paralelo, Fluxo de trabalho.

Abstract

Many distributed and parallel systems allow only a basic task flow, in which the parallel tasks are distributed and their results collected. In some systems the application execution flow gives support to a dependence relationship among tasks, represented by a directed acyclic graph. Even with this model it is not possible to execute in parallel some important applications as, for example, genetic algorithms. Therefore, there is a need for a new specification model with more sophisticated flow controls that allow some kind of iterative processing at the level of task management. The purpose of this work is to present a proposal for a specification language for parallel application execution workflow, which provides new types of control structures and allows the implementation of a broader range of applications. This language is based on XML (eXtensible Markup Language) notation, which provides characteristics like simplicity and flexibility to the proposed language. To evaluate these and other characteristics of the language, it was implemented on the JoiN parallel processing system. Besides allowing the creation and execution of new parallel applications containing task flows with loops and conditional branches, the proposed language was easy to use and did not cause any significant overhead to the parallel system.

Keywords: Computational Grids, Parallel programming, Parallel processing, Workflow

Agradecimentos

Aos meus pais e meu irmão.

Ao meu marido, Pedro, pelo apoio e compreensão.

Ao meu orientador, Prof. Marco Aurélio, pela atenção e confiança depositada.

Aos meus amigos, membros do grupo de desenvolvimento da plataforma JoiN, pelo incentivo e sugestões que contribuíram para a conclusão deste trabalho.

Glossário

BoT	Bag of Tasks
GGF	Global Grid Forum
DAG	Directed Acyclic Graph
DTD	Document Type Definition
JVM	Java Virtual Machine
LAN	Local Area Network
MPP	Massively Parallel Processing
XML	eXtensible Markup Language
VB	Visual Basic
WAN	Wide Area Network
WWW	World Wide Web

Artigos derivados deste trabalho

Cristina Enomoto e Marco Aurélio Amaral Henriques, "A Flexible Specification Model based on XML for Parallel Applications", Proceedings of 17th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), pp. 109-116, Rio de Janeiro, Outubro/2005.

Cristina Enomoto e Marco Aurélio Amaral Henriques, "Implementação de uma Linguagem de Especificação de Aplicações Paralelas baseada em XML para o Sistema JoiN", VI Workshop em Sistemas Computacionais de Alto Desempenho – WSCAD 2005, pp. 81-88, Rio de Janeiro, Outubro/2005.

Sumário

Lista de Figuras

Capítulo 1 Introdução

Os *grids* computacionais permitem a resolução de problemas complexos e simulações científicas cuja execução não seria possível com a utilização de um único processador [FOS99]. Um *grid* consiste de toda a infra-estrutura de *hardware* e *software* para prover um sistema capaz de coordenar recursos computacionais heterogêneos e distribuídos, oferecendo-lhes um acesso transparente.

Com a popularização da Internet, surgiram os primeiros projetos que procuravam explorar a capacidade computacional ociosa dos milhões de computadores conectados à rede, como GIMPS - *Great Internet Mersenne Prime Search* - e Distributed.net (decodificação de mensagens cifradas). Dentre esses projetos, um dos que obteve mais êxito foi SETI@home [AND02], que conseguiu agregar um grande número de computadores em torno da resolução de um problema comum: a análise de um gigantesco volume de dados obtidos de radiotelescópios.

A utilização de computadores domésticos para a realização de computação científica é denominada computação distribuída pública (*Public-resource computing*) e uma característica de aplicações apropriadas para este tipo de computação é que a comunicação entre tarefas paralelas é mínima ou inexistente [AND03].

Todo sistema paralelo possui algum modelo de especificação do fluxo de execução das aplicações. Várias destas especificações permitem apenas um fluxo básico, onde a execução de uma aplicação se restringe à execução de um único bloco de computação de tarefas de forma paralela, havendo:

- a transferência do código e dados de entrada de uma máquina coordenadora para as máquinas remotas;
- execução da tarefa em cada máquina remota;
- transferência de cada um dos resultados para a máquina coordenadora.

Além deste modelo básico, outros sistemas permitem a execução de vários grupos distintos de tarefas numa aplicação. Neste caso, o fluxo de execução é baseado no relacionamento de dependências entre grupos de tarefas, representado por um grafo direcionado acíclico ou DAG (*Directed Acyclic Graph*). Contudo, mesmo com este modelo de fluxo de execução, não é possível executar vários tipos de aplicações que poderiam ser paralelizadas, como, por exemplo, algoritmos genéticos ou de cálculo numérico que utilizam um processamento iterativo para a resolução de problemas.

Com isso, surge a necessidade de um modelo de especificação de aplicações paralelas para sistemas de computação pública que ofereça um controle de fluxo de execução mais

sofisticado e permita, por exemplo, iterações e desvio condicional no nível do gerenciamento de tarefas.

Alguns sistemas paralelos permitem que as tarefas se comuniquem durante sua execução em paralelo; porém este trabalho focou a especificação do fluxo de execução de aplicações em que cada tarefa é processada independentemente ou com um mínimo de comunicação. Estas tarefas são executadas de forma atômica na máquina remota sem depender de informações de outras. Enquanto o sistema paralelo provê o controle de distribuição dos dados e o recolhimento dos resultados das tarefas executadas em paralelo, o modelo de especificação de fluxo realiza o controle do fluxo de execução, utilizando os resultados das tarefas de uma etapa para determinar a próxima etapa a ser executada.

Neste trabalho são analisados diversos sistemas paralelos, principalmente em relação ao modelo de especificação do fluxo de execução das aplicações, e são levantadas algumas características e limitações dos modelos existentes. Em seguida, é proposta uma nova linguagem de especificação de aplicações paralelas para *grids* computacionais. Esta linguagem, baseada em XML [BRA05], define o fluxo de execução de tarefas em aplicações paralelas. Ela permite um controle de fluxo com as estruturas de controle básicas do modelo DAG e também estruturas mais complexas como laços de repetição (para um número fixo de iterações ou não) e a estrutura de desvio condicional (*switch/case*), que permite executar conjuntos de tarefas paralelas distintas a partir da análise dos dados de entrada. Também é apresentada a implementação desta linguagem sobre a plataforma de processamento paralelo JoiN. Apesar da linguagem ter sido implementada em uma plataforma específica, acreditamos que a maior parte das idéias propostas possam ser igualmente implementadas em outros sistemas paralelos. Algumas considerações e sugestões são apresentadas neste sentido.

1.1 Objetivos do trabalho e organização do texto

O objetivo deste trabalho é propor e implementar um novo modelo de especificação de fluxo de execução em aplicações paralelas para sistemas de computação distribuída pública ou *grids* que executam aplicações com tarefas fracamente acopladas ou independentes. Esta especificação de fluxo de execução suporta outros tipos de estruturas de controle além da estrutura de controle seqüencial básica, viabilizando a implementação de vários tipos de aplicações paralelas que não poderiam ser implementadas no modelo seqüencial com relacionamentos de dependência entre tarefas.

Esta dissertação está organizada nos seguintes capítulos:

- *Cap. 2 Conceitos básicos* - apresentação dos modelos de máquinas e programação paralela; também são mostrados os conceitos de *grids* computacionais com ênfase na computação distribuída pública e as áreas de aplicações que se beneficiam desses sistemas; são analisadas ainda as características de sistemas paralelos em relação ao controle de fluxo de execução de aplicações e às limitações dos sistemas existentes;

- *Cap. 3 Sistemas paralelos e linguagens de especificação de fluxo de execução de tarefas* - neste capítulo são analisados alguns sistemas paralelos e as características do modelo de especificação de aplicações de cada um;

- *Cap. 4 XPWSL: uma nova linguagem para especificação de fluxo de aplicações paralelas* – neste capítulo é proposto um novo modelo de controle de fluxo de execução de tarefas além de uma linguagem de especificação deste modelo. Nesta linguagem o fluxo de execução de aplicações pode ser mais flexível que em outros existentes, já que ela permite, além do fluxo de execução seqüencial, os fluxos iterativo e de desvio condicional;

- *Cap. 5 Implementação e testes da linguagem XPWSL na plataforma JoiN* - neste capítulo é mostrada como a linguagem proposta no Capítulo 4 pode ser implementada em um sistema paralelo, no caso a plataforma paralela JoiN. Também são apresentados testes que mostram a viabilidade da linguagem e o impacto no desempenho do sistema;

- *Cap. 6 Conclusões e trabalhos futuros* - neste capítulo são resumidos os principais resultados deste trabalho e são apresentadas aprimoramentos que poderiam ser feitos a partir do novo modelo de especificação para fluxo de aplicações paralelas proposto.

Capítulo 2 Conceitos básicos

Neste capítulo é apresentado um breve resumo dos modelos de máquinas e de programação paralela e a motivação da utilização de sistemas paralelos, com ênfase nos *grids* computacionais e na computação distribuída pública. Também são apresentados os modelos de especificação de aplicações paralelas e as características e limitações destes modelos.

2.1 O modelo de máquina paralela

Existem diversas classificações e modelos de arquiteturas paralelas de computadores. Uma das classificações mais conhecidas é a taxonomia de Flynn [FLY96], que se preocupa não só com a estrutura do sistema como também com os fluxos de instruções e dados. De acordo com Flynn as arquiteturas paralelas se classificam em:

Single Instruction Single Data (SISD) stream: nesta classe, um único fluxo de instruções opera sobre um único fluxo de dados, como no modelo seqüencial da máquina de Von Newmann. Arquiteturas SISD possuem uma única unidade de controle, mas podem possuir mais de uma unidade funcional;

Single Instruction Multiple Data (SIMD) stream: esta classificação corresponde a um único fluxo de instruções, operando sobre vários dados. Nesta arquitetura o programa ainda segue uma organização seqüencial, mas opera sobre múltiplos dados em paralelo, como nos computadores vetoriais e matriciais;

Multiple Instruction Single Data (MISD) stream: neste modelo existem diversas unidades de controle executando fluxos de instruções distintos, operando sobre o mesmo conjunto de dados. Este modelo é difícil de ser encontrado na prática;

Multiple Instruction Multiple Data (MIMD) stream: esta é a classe mais abrangente, na qual várias unidades de controle executam diversos fluxos de instruções. Praticamente qualquer grupo de máquinas operando em conjunto se encaixa nesse modelo, como os servidores multiprocessados, redes de estações de trabalho e as arquiteturas de processamento paralelo baseadas na Internet.

Além da taxonomia de Flynn, no livro "*Designing and Building Parallel Programs*" de Ian Foster [FOS95] são classificados os seguintes modelos de máquinas paralelas:

Multicomputadores: são formados por vários computadores de von Newmann (nós), conectados por um rede de interconexão. Cada computador executa o seu próprio programa e acessa a sua memória local. A comunicação entre nós é feita pelo envio e recebimento de mensagens pela rede. Uma característica deste modelo é que os acessos locais têm um custo menor que os acessos a memória remota, ou seja, operações de *read/write* são muito mais rápidas que operações de *send/receive*;

Computador com memória distribuída MIMD (*Multiple Instruction Multiple Data*): modelo muito semelhante ao multicomputador, no qual cada computador executa seu próprio programa com dados locais. A diferença é que no computador com memória distribuída MIMD, o custo de envio de mensagens entre dois nós pode ser dependente da localização do nó e tráfego de rede. Alguns exemplos são os servidores multiprocessados, redes de estações e as arquiteturas maciçamente paralelas;

Computador com memória compartilhada MISD (*Multiple Instruction Single Data*) ou multiprocessadores: neste modelo todos os processadores acessam uma memória comum. Programas desenvolvidos para multicomputadores podem ser executados nos multiprocessadores de forma eficiente, porque a memória compartilhada permite uma implementação mais eficiente da passagem de mensagens. O SMP (*Symmetric MultiProcessing*) é um exemplo de arquitetura na qual várias CPUs compartilham a mesma memória num mesmo gabinete. O sistema operacional SMP utiliza as CPUs como um *pool* de recursos de processamento;

Computador SIMD (*Single Instruction Multiple Data*): neste modelo todos os processadores executam o mesmo código para conjuntos diferentes de dados. Isto reduz a complexidade de *hardware* e de *software*, mas é apropriado apenas para problemas específicos, que possuem um alto grau de regularidade, como processamento de imagens e simulações numéricas. Os algoritmos de multicomputadores geralmente não podem ser executados eficientemente nesse modelo;

***Local Area Network* e *Wide Area Network*:** LANs e WANs também podem ser utilizadas como computadores paralelos, porém precisam ter um tratamento mais rigoroso quanto a confiabilidade e a segurança;

MPP (***Massively Parallel Processing***): Esta é uma arquitetura de multiprocessamento que utiliza até milhares de processadores, porém o número de CPUs nem sempre é o principal. Os sistemas MPP podem utilizar um paradigma de programação onde cada CPU possui a sua própria memória e cópia da aplicação. Cada subsistema se comunica com os outros utilizando uma rede de interconexão de alta velocidade. Para uma utilização efetiva do MPP a aplicação precisa ser quebrada em muitas partes que podem ser resolvidas simultaneamente.

2.2 Programação paralela

As arquiteturas paralelas requerem uma programação diferente dos modelos seqüenciais. A programação paralela possui uma complexidade maior que a programação seqüencial já que precisa coordenar o trabalho de vários processadores e as interações entre eles. Para isso, deve se preocupar com abstração, modularidade, concorrência, escalabilidade e localidade, envolvendo:

- decomposição do algoritmo e/ou dados em partes;
- distribuição das partes como tarefas que serão executadas por vários processadores simultaneamente;
- coordenação do trabalho e da comunicação entre os processadores;

Além disso, a programação depende também do tipo de arquitetura paralela e do tipo de comunicação dos processadores.

O processamento maciçamente paralelo pode envolver uma computação que pode ser dividida em um grande número de tarefas computacionais totalmente independentes, método conhecido como *task farming*. Outro tipo de paralelismo pode ser utilizado em situações que envolvem um grande número de dados que são divididos em nós. Nesse caso cada nó realiza o mesmo processamento porém com partes diferentes da estrutura de dados, o chamado *data parallelism*, ou paralelismo de dados. O paralelismo de dados é mais eficaz, pois o volume de comunicação é normalmente baixo em comparação com o volume de processamento realizado em cada nó.

2.3 *Grids* computacionais

O desenvolvimento de sistemas com alto poder de processamento foi motivado principalmente pela modelagem de problemas complexos e simulações científicas. No processamento paralelo, um problema é dividido em tarefas que são distribuídas entre diversas máquinas, o que permite um aumento de desempenho que não seria possível com a utilização de um único processador.

Com isso surgiu o conceito de *grid* computacional apresentado no livro "*The Grid : Blueprint for a New Computing Infrastructure*" [FOS99]. O termo *grid* surgiu a partir de uma analogia às redes de energia elétrica (*electrical power grid*). Da mesma forma que a rede elétrica conecta fontes de energia e provê a distribuição e o acesso à energia elétrica, um *grid* computacional é a infraestrutura de *hardware* e de *software* que provê acesso a serviços computacionais de grande porte de forma consistente e a um baixo custo a partir do compartilhamento e agregação de recursos distribuídos. O termo *grid* vem sendo associado com o conceito da criação de um ambiente computacional gigantesco com uma coleção distribuída de arquivos, banco de dados, computadores e outros dispositivos externos.

A grande diferença entre um *grid* e os ambientes distribuídos convencionais é que o *grid* utiliza o compartilhamento de recursos heterogêneos distribuídos geograficamente e

numa grande escala. A Tabela 1, extraída da ref. [NEM02], mostra uma comparação entre os ambientes distribuídos convencionais e os *grids*.

	Ambientes Distribuídos Convencionais	**Grids**
1	*pool* virtual de nós computacionais	*pool* virtual de recursos
2	usuário tem acesso a todos os nós do sistema	usuário tem acesso ao *pool* e não a recursos individuais
3	acesso a um nó significa acesso a todos os recursos do nó	pode-se restringir o acesso ao recurso
4	usuário tem acesso a todas as características do nó	usuário tem pouco conhecimento a respeito dos recursos
5	nós pertencem a um único domínio confiável	recursos estão distribuídos em vários domínios confiáveis
6	elementos no *pool*: de 10 a100, estático	elementos no *pool*: de 1000 a 10000, dinâmico

Tabela 1. Comparação dos ambientes distribuídos convencionais e *grids*

2.4 Classificações de *grids* e aplicações paralelas

Os *grids* podem ser classificados em cinco classes principais [FOS99], de acordo com o modelo de aplicação:

- *distributed supercomputing*: utilizado para agregar recursos computacionais em problemas que não podem ser executados em um único sistema por questões de desempenho;
- *high-throughput computing*: utilizado para executar um grande número de tarefas fracamente acopladas ou independentes, utilizando ciclos de CPU ociosos;
- *on-demand computing*: utilizado para compartilhar recursos que não estão disponíveis localmente, porém ao contrário do *distributed supercomputing*, a preocupação é a relação custo-desempenho e não apenas desempenho;
- *data-intensive computing*: utilizado para processos com comunicação e computação intensiva;
- *collaborative computing*: tem o objetivo de permitir e aprimorar as interações humanas, através do compartilhamento de acesso a dados e recursos.

As aplicações de *grid* devem ser particionadas para mapear os recursos e suas tarefas com os recursos disponíveis. De acordo com a ref. [VAR04] é proposta a seguinte taxonomia para aplicações distribuídas em um *grid*:

- tarefas independentes: é o modelo mais simples de aplicação onde as tarefas são independentes, também conhecido como *bag-of-tasks*;
- tarefas fracamente acopladas: nesse tipo de modelo existem pontos compartilhados, como se a aplicação fosse dividida em fases, permitindo alguma comunicação entre cada fase;
- tarefas fortemente acopladas: estas aplicações podem ser representadas por grafos complexos onde os dados trocados entre as tarefas são volumosos em relação a computação realizada.

Na Fig. 1, extraída do artigo [VAR04], é mostrada a representação dos grafos de aplicação para cada uma das categorias.

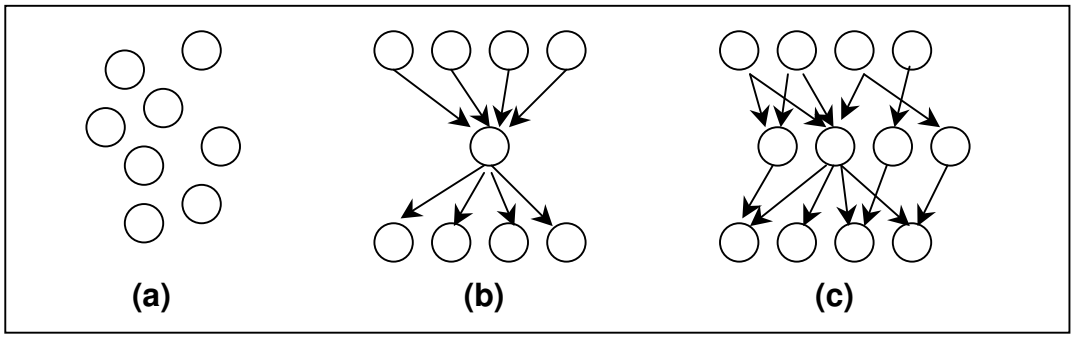

Figura 1. Classes de aplicações de *grid*: (a) tarefas independentes, (b) tarefas fracamente acopladas e (c) tarefas fortemente acopladas

2.5 Computação distribuída pública

No início dos anos 80, com o surgimento dos computadores pessoais e a interconexão dos computadores em rede, passou-se a dar uma grande importância à comunicação de dados. Os computadores de alto desempenho possuem um custo muito alto, mas com o advento e rápido avanço das tecnologias de comunicação surgiram outras alternativas ao grande "supercomputador", por meio da cooperação de um número potencialmente grande de computadores menores, que juntos possuiriam uma maior capacidade a um custo menor que os "supercomputadores" individuais [STA04].

Como já foi mencionado no Cap. 1, a popularização da Internet incentivou o surgimento de projetos que buscavam explorar ciclos ociosos de milhares de computares conectados a ela. Entre estes projetos destaca-se SETI@home [AND02], que vem de *Search for Extraterrestrial Intelligence*. Ele é um esforço científico que procura determinar se existe vida inteligente em outros planetas. Os pesquisadores do SETI utilizam vários métodos, sendo mais popular, o método Radio SETI, que procura sinais artificiais de rádio vindos de outros planetas. O SETI@home é um projeto do Radio SETI, que permite que qualquer computador com acesso a Internet participe do projeto. Para participar o usuário precisa apenas fazer o *download* da aplicação cliente, que é executada como um *screensaver* e realiza os cálculos quando ninguém está utilizando o teclado ou o *mouse*.

Os termos computação distribuída pública (*Public-resource computing*), *Global Computing*, ou ainda *Peer-to-peer computing* [AND03] denotam a utilização de computadores domésticos para a realização de computação científica pesada. Este processamento só é efetivo quando existe participação de muitos voluntários (no caso do SETI@home houve a participação de mais de quatro milhões de voluntários). Como o canal de comunicação com os participantes de um sistema de computação distribuída pública não é homogêneo nem é garantido que tenha alta velocidade, uma característica de

aplicações de computação distribuída pública é que a comunicação entre tarefas deve ser mínima ou inexistente.

Apesar da computação distribuída pública e a computação de *grid* terem o mesmo objetivo que é a melhor utilização dos recursos computacionais, existem várias diferenças entre os dois sistemas. Geralmente o *grid* envolve recursos relativamente poderosos de algumas organizações conectados em tempo integral por uma rede de alto desempenho. Já com a computação pública, geralmente os projetos são acadêmicos, com recursos limitados e, do outro lado, os participantes estão conectados à Internet e participam apenas se tiverem algum incentivo. Os projetos de computação pública não podem controlar os participantes nem evitar comportamentos maliciosos. Essas diferenças refletem em requisitos específicos para cada sistema.

2.6 Aplicações *bag-of-tasks* e os sistemas paralelos

A definição do paradigma *bag-of-tasks* [RAM05] é:

"Vários processos trabalhadores compartilham um repositório que contém tarefas independentes. Cada trabalhador remove repetidamente uma tarefa do repositório e a executa. Durante o processamento, o trabalhador pode gerar novas tarefas que são colocadas no repositório. Um processo gerenciador implementa o repositório, distribui as tarefas, coleta os resultados e detecta o término da aplicação."

A partir dessa definição pode-se dizer que aplicações *bag-of-tasks* (BoT) são aplicações paralelas cujas tarefas são independentes que não se comunicam entre si. Este tipo de computação também se aplica a sistemas do tipo *master-worker*, *public (resource) computing*, sistemas gerenciadores de *jobs* ou *batch queuing*, entre outros. Um sistema paralelo que executa uma aplicação BoT será referenciado como um sistema BoT.

As principais vantagens deste modelo são a escalabilidade e a facilidade em se obter o balanceamento de carga. Apesar da simplicidade, aplicações BoT são aplicáveis em diversos cenários, incluindo *data mining*, quebra de chaves de segurança, simulações de Monte Carlo, fractais, biologia computacional, entre outros.

Alguns sistemas paralelos utilizam aplicações onde cada tarefa realiza o mesmo tipo de computação e o sistema gerenciador se encarrega de distribuir os dados de entrada entre os diversos processos trabalhadores, havendo o paralelismo de dados como no modelo SIMD. Além dos sistemas paralelos que executam exclusivamente aplicações BoT, existem outros que também utilizam o conceito de tarefas independentes, porém permitindo um relacionamento de precedência entre grupos de tarefas. A independência das tarefas nesses sistemas evita que a comunicação se torne um gargalo durante a execução, como ocorre em aplicações fortemente acopladas.

2.7 Exemplos de aplicações

O processamento paralelo pode ser aplicado em diversas áreas, como:

- previsão meteorológica: é feita a simulação de condições futuras, baseada em condições iniciais já observadas. Outras aplicações incluem a modelagem das interações atmosféricas e dos oceanos e a criação de modelos 3-D das correntes oceânicas. Com um poder computacional suficiente, poderíamos simular a evolução de todo o planeta em um grande intervalo;
- engenharia: um grande poder computacional é necessário nas pesquisas e desenvolvimento da área de engenharia. Cada vez mais são utilizadas aproximações computacionais em vez de teoria analítica e experimentos de laboratório. Alguns exemplos são os modelos de turbulência e equações de fluido que exigem um alto poder computacional, no qual a computação de um único ponto pode envolver horas de um supercomputador;
- ciência de materiais: necessidade de simulações, que envolvem alto poder computacional, com dispositivos e materiais em 2-D e 3-D;
- física do Plasma: plasmas são gases ionizáveis em alta temperatura. Seu estudo é complexo porque os gases estão sujeitos a forças magnéticas, elétricas e também à pressão. Os plasmas possuem propriedades não lineares, e sua modelagem requer computação com alto desempenho;
- economia: modelagem e predição econômica é bem diferente das pesquisas de ciências naturais, porque o desafio é implementar teorias que sejam verificáveis na prática do relacionamento entre homens e comunidades. Modelos computacionais são utilizados para interpretar todos os dados disponíveis e também para guiar decisões econômicas e simular/testar diversas hipóteses;
- inteligência artificial: requer alto poder computacional para processar bases de conhecimento;
- cálculo numérico: vários algoritmos de cálculo numérico utilizam um processo iterativo com critérios de parada (geralmente uma determinada precisão) e, dependendo dos resultados desejados, sistemas paralelos de alto desempenho tornam-se fundamentais;
- simulações de Monte Carlo: são simulações baseadas em amostragem, geralmente utilizadas para determinar uma distribuição estatística e também pode envolver grandes volumes de cálculos;
- quebra de chaves de criptografia por meio de algoritmos sofisticados;
- algoritmos genéticos;

entre outras aplicações.

Algumas destas aplicações admitem o uso de um modelo BoT, mas outras necessitam de estruturas de controle mais elaboradas para as tarefas, mesmo sendo estas relativamente independentes.

2.8 Modelos de especificação de fluxos de execução em aplicações paralelas

Existe uma grande diversidade de *grids* computacionais, e cada um possui seu próprio modelo de especificação de aplicações e principalmente de fluxo de execução de tarefas.

A especificação de uma aplicação define o seu comportamento sem definir sua implementação. No caso do fluxo de execução de uma aplicação paralela, uma especificação pode determinar o fluxo de dados entre as tarefas indicando um relacionamento de dependência entre elas.

O modelo mais simples de execução de uma aplicação paralela consiste basicamente em três passos:
- um coordenador prepara e transfere todos os dados e código executável para os trabalhadores;
- cada trabalhador realiza o processamento dos seus dados de entrada;
- após o término da tarefa, o coordenador coleta os resultados de cada trabalhador.

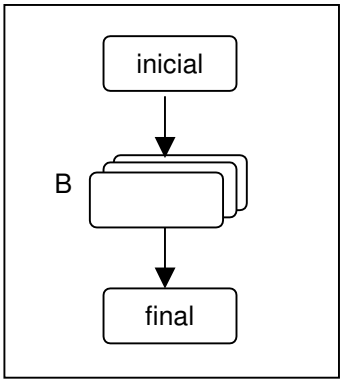

Figura 2. Exemplo do fluxo de uma aplicação paralela básica

Neste modelo o coordenador é o componente responsável pelo controle da execução das tarefas. Entende-se que um trabalhador é qualquer máquina conectada ao *grid* que esteja habilitada para a execução de tarefas. A computação útil realizada em uma tarefa pode ser denominada *workunit, job ou task.* Um conjunto de tarefas do mesmo tipo, isto é, que implementam o mesmo *workunit*, é denominado *lote de tarefas*.

Observa-se que neste exemplo o coordenador não realiza nenhuma computação útil; ele é responsável apenas pelo gerenciamento das tarefas e geralmente cada trabalhador realiza o mesmo tipo de computação porém sobre um conjunto de dados distintos (SIMD- *Single Instruction Multiple Data model*). Na Fig. 2 é ilustrado este modelo, que pode ser utilizado em vários cenários, incluindo *data mining*, simulações de Monte Carlo, biologia computacional, simulações paramétricas entre outros.

Nem todas as aplicações se encaixam neste modelo; por isto vários sistemas permitem pelo menos a execução de várias tarefas paralelas de forma ordenada. Nestes sistemas a

aplicação consiste de um conjunto de lotes de tarefas e de um relacionamento de dependência entre os lotes, formando um grafo direcionado acíclico ou DAG (*Directed Acyclic Graph*). Este tipo de aplicação pode ser representado na estrutura de árvore da Fig. 3. Neste exemplo, B_1 e B_2 são lotes de tarefas que podem ser executadas somente depois que o lote B_0 tenha completado. Da mesma forma, B_3 e B_4 dependem do término de B_2. Considerando que a execução de cada lote é iniciada somente depois que todas as tarefas do lote anterior estejam terminadas, esta transição de B_n a B_{n+1} pode se tornar um gargalo no sistema e por isso deve ser analisada com cuidado.

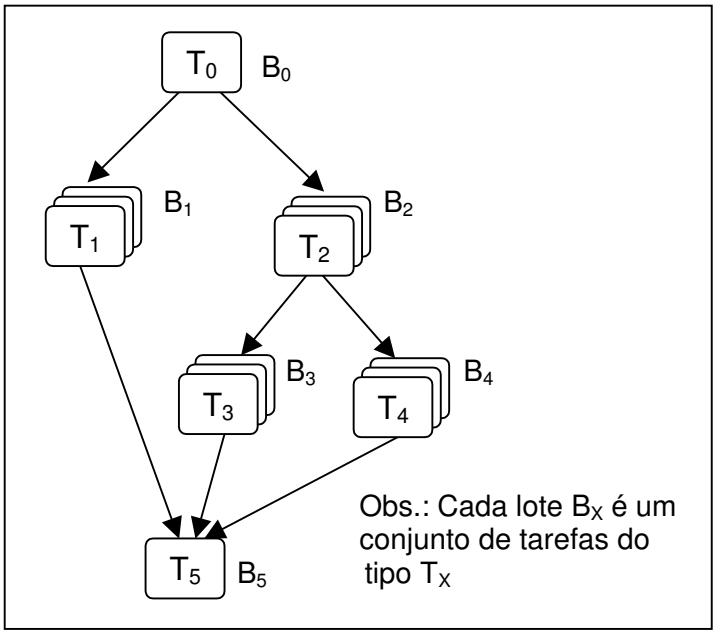

Figura 3. Exemplo de fluxo de uma aplicação paralela com dependência entre lotes de tarefas

Para executar uma aplicação paralela, são necessários pelo menos os seguintes itens:

- código executável com a computação a ser realizada em cada lote de tarefas, possuindo os parâmetros de entrada e saída bem definidos;
- a especificação da aplicação, que define o fluxo entre as entradas e saídas de cada passo.

Além do modelo DAG, existe o conceito de *master-worker* em *grids* que foi analisado em [HEY00], que consiste em duas entidades: um *master* e múltiplos *workers*. O *master* é responsável pela decomposição do problema em pequenas tarefas que são distribuídas entre processos *worker*. Ele também é responsável por coletar os resultados parciais para gerar o resultado final da computação. O *worker* executa um ciclo simples, no qual recebe a tarefa, processa e envia o resultado de volta.

Um algoritmo básico *master-worker* é exibido a seguir:

```
Initialization
Do
     For task = 1 to N
          PartialResult = Function (task) // tarefas dos workers
     end
     act on batch complete()
while (end condition not met)
```

O processo *master* resolve as N tarefas, procurando por *workers* que possam executá-las e passa a descrição da tarefa para os mesmos. Uma vez completada a tarefa, o *worker* envia o resultado para o *master*. O processo *master* realiza alguma computação com o os resultados de cada trabalhador. Esse processo pode ser repetido várias vezes até terminar a aplicação. Todo o controle do algoritmo é feito pelo *master*. Na Fig. 4 percebe-se que o controle do fluxo é feito por uma aplicação e não pelo sistema.

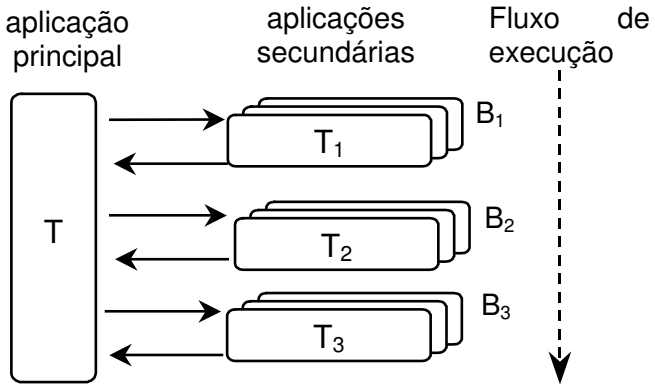

Figura 4. Exemplo de fluxo de controle master-worker

A especificação de aplicações pode ser definida de várias formas. Entretanto, em vez de se definir o controle do fluxo no próprio código da aplicação, uma forma mais flexível é a utilização de um arquivo específico, separado do código executável da aplicação. Este arquivo pode ser tão simples quanto um arquivo texto com um formato particular, ou mais complexo utilizando documentos XML com elementos específicos do sistema.

Um arquivo de especificação de aplicações paralelas deve conter pelo menos as seguintes informações:

- *workunit*: é a unidade de processamento que transforma um conjunto de dados de entrada em dados de saída. Dependendo do sistema, ele pode ser chamado de *processor*, *job* ou *task*;
- *datalink*: especifica o fluxo de dados entre os *workunits*. O *datalink* determina o relacionamento de dependência entre os lotes de tarefas determinando o fluxo de execução;
- restrições de coordenação e execução: estas restrições são utilizadas para fazer o balanceamento e selecionar os recursos mais adequados para a execução das tarefas. Os recursos podem utilizar as restrições para selecionar o tipo de aplicação que eles desejam executar como, por exemplo, definir a utilização máxima de CPU e memória, enquanto que as aplicações podem definir o tipo de sistema operacional, requisitos mínimos de memória e velocidade da CPU, por exemplo.

Apesar das restrições de execução serem importantes para que o sistema faça a alocação das tarefas (escalonamento) aos recursos disponíveis mais adequados, é possível tratar separadamente controle do fluxo de execução das tarefas e o escalonamento das mesmas.

2.9 Limitações do modelo de fluxo de execução de tarefas seqüencial

Considera-se um modelo seqüencial um grafo direcionado acíclico sem repetições de blocos, como apresentado na Seção 2.8 Encaixam-se nesse modelo diversas aplicações paralelas; porém este modelo se mostra pouco eficaz ou inadequado em aplicações que utilizam um processo iterativo para a resolução de um problema,como mostrado a seguir.

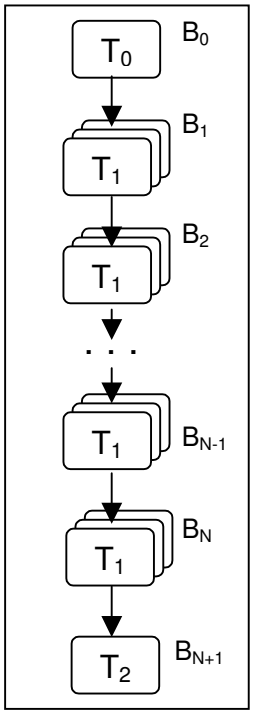

Figura 5 Exemplo de um DAG utilizado para realizar um processamento iterativo

Normalmente os algoritmos evolutivos são baseados em um processamento iterativo. Um exemplo é a ferramenta computacional chamada *Phylogenetic Tool Project* (PTP) que permite encontrar soluções quase ótimas em espaços de estados com elevado número de candidatos ao longo de um processo de busca [PRA03]. Esta ferramenta precisa administrar populações de candidatos que devem ser tratadas individualmente ao longo das gerações. Estas populações podem ser distribuídas em diferentes tarefas e processadas em paralelo, o que faz de PTP uma aplicação candidata a tirar proveito do processamento paralelo. Porém, a característica iterativa de PTP apresenta uma necessidade para a qual o modelo DAG não está preparado, isto é, o processamento paralelo deve ser repetido um número de vezes que é dependente da convergência do algoritmo para um resultado esperado. Se este número de

iterações fosse fixo, a ferramenta PTP poderia ser especificada em um DAG como uma seqüência (provavelmente longa) de lotes de tarefas, cada um dedicado a uma iteração ou geração das populações, como é ilustrado na Fig. 5. Observe que o mesmo tipo de tarefa é utilizado em cada uma das etapas: se o laço tivesse que ser repetido N vezes, seria necessário especificar N lotes para a mesma tarefa que contém a computação realizada no laço.

Porém, o número de iterações do algoritmo PTP não é fixo e depende da convergência dos resultados. Para implementar o PTP em um sistema que utiliza o modelo DAG seria necessária uma adaptação do sistema de forma que nele fosse executada apenas a parte paralela de processamento de uma população e fosse utilizado algum outro mecanismo externo para o controle das gerações e a decisão de término da aplicação. Apesar de ser possível fazer uma adaptação do sistema, esta não é uma solução ideal. O PTP é apenas um exemplo de aplicações que, apesar de possuírem características que permitam o processamento paralelo, não podem ser implementadas de forma eficiente em diversos *grids* existentes.

Com o advento das tecnologias de *grid*, cada vez mais são desenvolvidas aplicações complexas para gerenciamento de processos e execução de experimentos científicos em paralelo. Para que o sistema em *grid* suporte diversos tipos de aplicações, ele precisa prover um controle de fluxo que permita estruturas de controle mais complexas que o simples relacionamento de dependências. São necessárias estruturas iterativas e de desvio condicional, que permita a seleção das tarefas em tempo de execução.

2.10 Conclusões

Neste capítulo foram apresentados alguns conceitos básicos de programação paralela, *grids* computacionais e sistemas de computação pública distribuída.

Também foram apresentadas as características do modelo de especificação de aplicações em relação ao fluxo de execução das tarefas. O fluxo de execução utilizado por diversos *grids* consiste de apenas um fluxo básico ou de um relacionamento de dependências, o que limita os tipos de aplicação que podem ser implementados nesses sistemas.

No próximo capítulo será mostrada uma análise de diversos sistemas paralelos existentes, com o foco no fluxo de execução de cada um.

Capítulo 3 Sistemas paralelos e linguagens de especificação de fluxo de execução de tarefas

Aplicações paralelas com tarefas independentes podem ser implementadas em diversos tipos de sistemas paralelos e, mesmo considerando que cada sistema possui características próprias como *kernel* e o modelo de aplicação, é possível comparar o modo com que o paralelismo da aplicação é tratado em cada sistema e analisar as vantagens e desvantagens de cada um.

Neste capítulo são analisados os modelos de fluxo de aplicação de alguns sistemas paralelos existentes tais como: Condor, BOINC, Nimrod, OurGrid e JoiN. Cada um deles é mostrado com mais detalhes no Apêndice A.

Além disso, serão mostrados alguns trabalhos recentes de linguagens de especificação de fluxo de execução.

3.1 Sistemas paralelos

Analisou-se diversos sistemas paralelos, com o objetivo de avaliar como os sistemas existentes tratam o relacionamento entre as tarefas. O modelo de aplicação de cada sistema é mostrado a seguir.

3.1.1 Condor

Condor é um sistema tipo *high-throughput* (Seção 2.4) que realiza o gerenciamento de recursos e tarefas, utilizando uma política de escalonamento e monitoramento de recursos [THA03].

As tarefas do Condor são chamadas *jobs*. O usuário submete *jobs* para o Condor e ele decide quando e onde eles serão executados, seguindo o seguinte fluxo:

- usuário submete o *job* para um agente (escalonador);
- o *job* é armazenado em uma estrutura persistente enquanto o agente procura por recursos que possam executá-lo;
- agentes e recursos utilizam um *matchmaker* para analisar a compatibilidade entre eles;
- o agente contata o recurso para executar o *job*.

Cada um dos *jobs* e máquinas possuem atributos para qualificá-los, que são chamados *ClassAds* (*Classified Advertisements*). Estes atributos definem os requisitos e as características de *jobs* e recursos. Eles são combinados e ordenados de forma a selecionar o recurso mais apropriado para executar o *job* .

As aplicações Condor são definidas pelo *submit description file*, que é um arquivo texto com informações do *job* (código executável, arquivos de entrada e saída, parâmetros, requisitos, etc.). Este arquivo é passado para o comando *condor_submit*, que faz uma varredura e uma verificação de erros gerando um *ClassAdd* com a especificação do *job*. O *ClassAdd* e os executáveis são enviados ao *condor_schedd*, que os coloca em uma fila, aguardando o momento de execução.

```
# Exemplo de um arquivo condor_submit file
# (# indica comentário)
Universe = vanilla
Executable = /home/dummy/my_job.condor
Input  = my_job.stdin
Output  = my_job.stdout
Error  = my_job.stderr
Arguments = -arg1 -arg2
InitialDir = /home/dummy/condor/run_1
Queue
```

Também é possível descrever múltiplos *jobs* em um arquivo, neste caso ele é chamado de cluster. Cada *job* no cluster é chamado processo. No exemplo abaixo, em vez de submeter um único *job*, são enviados 600 *jobs* de uma única vez. Os arquivos de entrada e saída de cada processo estão localizados em diretórios distintos, no caso diretórios run_0 a run_599.

```
# Exemplo de um arquivo condor_submit que define um cluster
# de 600 jobs com diferentes iwd
Universe  = vanilla
Executable = my_job
Arguments = -arg1 -arg2
InitialDir = run_$(Process)
Queue 600
```

Condor também permite definir relacionamentos entre os *jobs* que são controlados pelo *DAGMan* (*Directed Acyclic Graph Manager*). O *DAGMan* é um meta-escalonador que trata as dependências *inter-jobs* não permitindo ciclos ou *loops* no grafo.

A estrutura de dados utilizada pelo DAGMan para representar as dependências é um grafo direcionado acíclico, no qual cada *job* é representado por um nó. Cada nó pode ter vários nós pais e filhos, desde que não tenha ciclos.

Um DAG é definido por um arquivo de especificação (*dag file*) que lista os nós e suas dependências. Este arquivo de entrada especifica 4 seções:

- JOB: define os programas que fazem parte do DAG, relacionando um identificador para cada arquivo *submit description file* (arquivos ".condor");
- PARENT e CHILD: definem as relações de dependência dos *jobs*;

- SCRIPT: define um processamento a ser realizado antes da submissão de um programa do DAG ao Condor (PRE), ou após a execução de um programa do DAG (POST);
- RETRY: número de re-tentativas a serem feitas no caso de um nó do DAG falhar.

A Fig. 6 ilustra o arquivo com o fluxo e o grafo associado, onde cada nó irá executar o *job* especificado pelo *Condor submit file*.

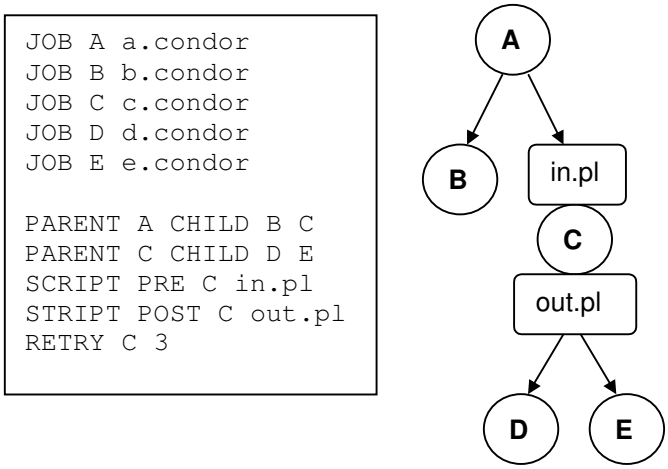

Figura 6. A linguagem do DAGMan e seu grafo direcionado acíclico

O elemento *job* associa um nome abstrato (A) ao arquivo (a.condor) que descreve um *job* do Condor. Os elementos *PARENT-CHILD* descrevem o relacionamento entre dois ou mais *jobs*. Os *jobs* que são independentes podem ser executados em qualquer ordem ou simultaneamente.

Existe um novo projeto chamado Condor-G que permite a utilização de recursos distribuídos em diversos domínios, utilizando o Globus *toolkit* [GLO05a], que é uma coleção de componentes projetados para permitir o desenvolvimento de aplicações em ambientes distribuídos. Este *toolkit* é uma implementação de uma arquitetura *bag of services* onde o usuário pode utilizar um conjunto de serviços isoladamente (*stand alone)*. Cada componente Globus provê serviços básicos como: autenticação, alocação de recursos, comunicação, detecção de falhas e acesso a dados remotos. Este serviços são utilizados para implementar serviços de alto nível, ferramentas e modelos de programação.

O Globus também provê uma linguagem de especificação RSL (*Resource Specification Language*) [GLO05b], para descrição dos *jobs* e recursos necessários para executá-los. O RSL utiliza expressões que associam pares *"atributo=valor"* que são combinadas com a utilização de conjunções. As *tags* definem: arquivo executável, lista de argumentos, ambiente, número de processos, quantidade de máquinas para rodar os processos e restrições como máximos e mínimos de memória, CPU.

Condor possui um *framework* flexível e simples com os mecanismos de gerenciamento de recursos (*ClassAds*) e o meta-escalonador DAGMan. Porém, como foi apresentado no Cap. 2, o modelo de aplicação DAG não é adequado para algoritmos paralelos que necessitem de um processamento iterativo.

3.1.2 BOINC

BOINC é um sistema de código aberto que suporta uma grande variedade de aplicações distribuídas. Para isso possui um mecanismo flexível e escalável para distribuição de dados e escalonamento de tarefas [AND05a]. Várias aplicações foram adaptadas ou desenvolvidas utilizando o BOINC, como: SETI@home, Climateprediction.net, Predictor@home, Folding@home, Climate@home, Einstein@home e vários projetos do CERN [BOI05].

BOINC foi projetado para ser utilizado em computação distribuída pública, suportando aplicações que requerem um elevado grau de computação, com um alto nível de paralelismo e sem a comunicação entre os nós.

Um projeto BOINC é um grupo de aplicações distribuídas. Os projetos são independentes, sendo que cada um possui suas próprias aplicações, banco de dados e servidores; um projeto não é afetado pelo status de outro projeto. Um projeto é identificado por um *master URL*, um documento XHMTL que, além de uma *home page* do projeto é o diretório do servidor utilizado para o escalonamento das tarefas. A maioria das informações são armazenadas em um banco de dados, incluindo: descrição das aplicações, plataforma, versões do programa, resultados, contas de usuários, *timestamps*, *workunits*, etc. O modelo de armazenamento de uma aplicação é baseado em arquivos (entrada, saída, código executável, bibliotecas) que são transferidos via HTTP entre o servidor e a máquina que irá processar a tarefa.

Uma aplicação consiste de um programa e um conjunto de *workunits* e resultados que são armazenados em tabelas do banco de dados do BOINC. O *workunit* descreve a computação a ser realizada.

BOINC também provê APIs para o desenvolvimento de aplicações compostas que consistem de um programa principal e um ou mais programas trabalhadores. O programa principal executa cada programa trabalhador em seqüência e mantém um controle de estado da execução dos trabalhadores. Mesmo este modelo de fluxo de aplicações compostas é mais limitado que o modelo DAG, já que ele permite apenas um fluxo com uma única seqüência e não um grafo de dependências como o DAG.

3.1.3 Nimrod e Nimrod/G

Nimrod é uma ferramenta para realizar simulações paramétricas em uma rede distribuída de *workstations*, sendo utilizado principalmente para simulações numéricas [ABR95]. Ele combina um sistema de gerenciamento de filas com o controle de execução de aplicações distribuídas, controlando a distribuição das tarefas para as máquinas e coletando os resultados.

A estrutura básica do sistema Nimrod é mostrada abaixo:
- o usuário especifica os parâmetros de simulação;
- Nimrod realiza uma combinação de todos os valores possíveis dos parâmetros, gera um *job* para cada conjunto e faz o gerenciamento da distribuição dos *jobs* para as máquinas. Após o término da execução dos *jobs*, ele organiza a coleta dos resultados;
- em vez de utilizar um sistema de arquivos compartilhado ele realiza cópias dos arquivos nas máquinas remotas que realizarão a operação;
- quando o programa termina, os arquivos de saída são copiados no servidor.

Um experimento, ou aplicação, do Nimrod é definido por um arquivo texto chamado *declarative plan* que contém todos os parâmetros, valores-padrão e comandos para executar a tarefa. Estas informações são utilizadas para o escalonamento e transferência de arquivos para as máquinas disponíveis. O *declarative plan* contém duas seções principais, uma seção de parâmetros e outra de tarefas. Não existe comunicação entre as tarefas e o algoritmo de escalonamento utilizado é que define como será feita a alocação dos recursos. Como as execuções das tarefas são independentes, este sistema é utilizado apenas para aplicações do tipo BoT (*Bag of Tasks*).

Nimrod separa o desenvolvimento da aplicação do gerenciamento de recursos. Para permitir a criação de aplicações paramétricas paralelas, ele provê uma linguagem simples de especificação dos parâmetros. A execução das aplicações é gerenciada pelo Nimrod, onde o usuário define os requisitos de QoS (*Quality of Service*) como alocação de recursos e prazo limite para a execução dos *jobs*.

Nimrod/G estende o modelo do Nimrod para ser utilizado em um ambiente de *grid* dinâmico e heterogêneo. Da mesma forma que o Condor, ele também utiliza o Globus *toolkit* [GLO05a] para prover o suporte ao *global grid*, onde um grande número de computadores são conectados globalmente formando um único super computador [ABR00].

As aplicações do Nimrod e Nimrod/G se encaixam no modelo básico de execução apresentado na Seção 2.8, no qual a mesma computação é realizada para diferentes combinações dos parâmetros de entrada e depois são coletados os resultados de cada tarefa paralela, não provendo, portanto, suporte para fluxos de execução mais sofisticados.

3.1.4 OurGrid

OurGrid é uma solução para executar aplicações BoT em *grids* computacionais [OUR05]. Ele forma uma comunidade *peer-to-peer* onde uma máquina disponibiliza recursos ociosos para participar da comunidade. Um usuário precisa executar seu próprio sistema MyGrid [CIR03] para participar da comunidade OurGrid. MyGrid é um *grid* que combina recursos locais, comunitários e recursos de *midlleware* como o Globus.

Os principais componentes do OurGrid são:
- GuMs – vem de *Grid Machine*; são as máquinas onde as tarefas são executadas;

- GuM Providers – são máquinas que coordenam e organizam os GuMs em um domínio determinado;
- MyGrid – é o *frontend* do usuário, utilizado para executar e monitorar *jobs*.

No OurGrid as aplicações são chamadas *jobs* e um *job* é um conjunto de tarefas independentes. O *job* é especificado por um arquivo texto chamado *job description file*. Este arquivo contém as características do *job*: que arquivos serão transferidos para o GuM, o comando ou código executável que será executado para processar a tarefa e quais arquivos de saída deve ser retornados para o servidor. Os *job description files* contêm a descrição do *job* (aplicação) e também apresentam os requisitos de execução, como no exemplo abaixo:

```
job:
    label: myjob
    requirements: (os == linux and mem >=100)
task:
    remote: mytask
```

Este arquivo informa o nome do *job* "myjob" e que a tarefa "mytask" pode ser executada apenas em máquinas que utilizam o sistema operacional Linux e possuam mais de 100 MB de memória disponível.

Assim como no Nimrod, as aplicações do OurGrid se classificam no modelo de fluxo básico de aplicação da Seção 2.8, uma vez que ele executa apenas aplicações BoT. Logo ele sofre das mesmas limitações do modelo BoT já discutidas.

3.1.5 A plataforma JoiN

JoiN é um sistema maciçamente paralelo baseado em computação distribuída pública que tira proveito das facilidades oferecidas pela linguagem Java [HEN99] [YER05]. Esse projeto teve início com as pesquisas iniciadas em 1996 com a proposta de criar um sistema maciçamente paralelo baseado na tecnologia usada pela *World Wide Web* (WWW) e as facilidades oferecidas pela linguagem Java. Desde então, o JoiN tem sido desenvolvido e aprimorado no Departamento de Engenharia de Computação e Automação Industrial da FEEC- UNICAMP.

A plataforma JoiN foi totalmente desenvolvida em Java, o que lhe garante portabilidade. Como o código é independente da plataforma de *hardware* e do sistema operacional, qualquer máquina com a JVM (*Java Virtual Machine*) instalada pode participar do sistema. Outra vantagem é possuir um ambiente de execução auto-contido e configurável. A linguagem Java tem tido aceitação em diversas áreas de aplicação, podendo ser executada desde computadores de alto desempenho, passando por PCs e chegando até mesmo a dispositivos móveis como PDAs e celulares.

O sistema é composto por máquinas heterogêneas e com baixo acoplamento que formam uma estrutura com alto poder computacional. Um de seus objetivos é simplificar o

modo com que os computadores participam do sistema e estimular a participação de voluntários.

- JoiN foi projetado visando eficiência, escalabilidade e robustez. Ele é baseado em serviços, o que oferece flexibilidade de operação e configuração. Novos serviços podem ser adicionados à plataforma com facilidade. O relacionamento entre seus quatro componentes básicos é ilustrado na Fig. 7.

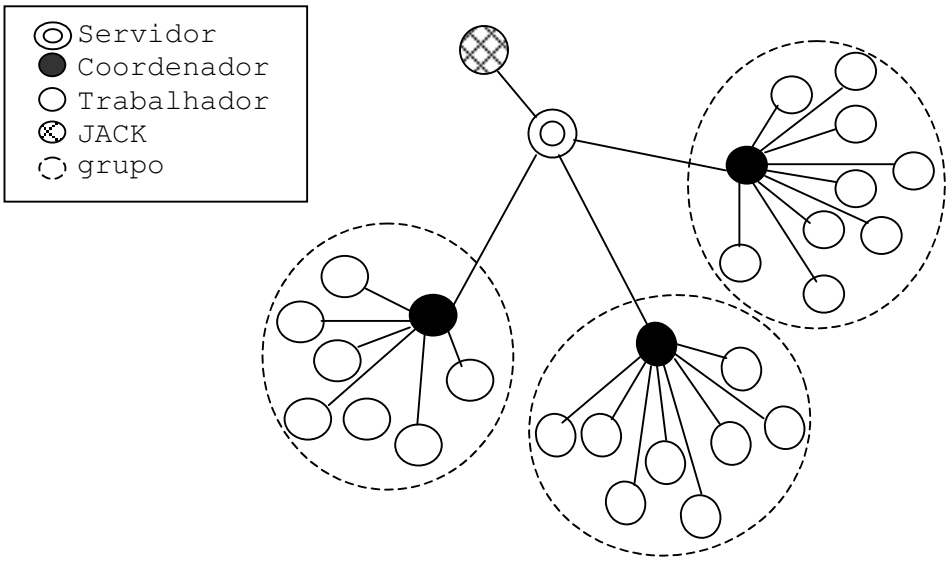

Figura 7. A plataforma JoiN

Os componentes do sistema JoiN são:

- **Servidor ou Mediador** - responsável pelo gerenciamento da plataforma, possuindo as seguintes funções:
  - receber a conexão de outros componentes: coordenador, trabalhador e jack;
  - iniciar a execução de uma aplicação;
  - associar novos trabalhadores a um coordenador;
  - atender às requisições administrativas do componente Jack;
- **Coordenador** - responsável por gerenciar a execução das aplicações em um grupo de trabalhadores, tendo como funções:
  - receber do servidor a aplicação que será executada e distribuir as tarefas entre os trabalhadores sob sua coordenação de acordo com o desempenho dos mesmos;
  - escalonamento e a execução das tarefas;
  - coletar os resultados de cada tarefa;
- **Trabalhador** - responsável pela realização do trabalho computacional útil, ou seja, pela execução das tarefas de aplicações. Um conjunto de trabalhadores forma, com um coordenador, um grupo da plataforma;
- **Jack** (*JoiN Administration and Configuration Kit*) - é o módulo de configuração do sistema. Ele conecta-se diretamente com o servidor para obter informações do sistema e é responsável por requisitar ações ao servidor, como:
  - adicionar e remover componentes de uma aplicação;

- instalação e desinstalação de aplicação;
- sincronização de versões de aplicações;
- obter a lista completa das aplicações instaladas no servidor;
- iniciar/interromper a execução de uma aplicação;

definir o número de trabalhadores que serão utilizados na execução da aplicação.

Como os componentes têm papéis distintos, os serviços JoiN também precisam ter tratamentos adequados para cada componente. Por isso, cada módulo de serviço possui quatro especialidades, isto é, possui um sub-módulo tipo servidor, um sub-módulo tipo coordenador, um sub-módulo tipo Jack e um sub-módulo tipo trabalhador.

Para garantir a escalabilidade e a modularidade, a comunicação entre componentes é restrita: trabalhadores se comunicam apenas com o coordenador, coordenadores podem se comunicar com outros coordenadores, trabalhadores e servidor e o servidor se comunica com coordenadores e jack. Além disso, a comunicação entre serviços também é restrita: serviços diferentes só podem se comunicar dentro do mesmo componente.

Uma aplicação do JoiN é formada por pelo menos três lotes de tarefas: (1) um lote inicial com uma tarefa, responsável por organizar os dados que servirão de entrada para o próximo lote, (2) um ou mais lotes de tarefas com várias tarefas paralelas e; (3) um lote final com uma tarefa responsável pela consolidação dos resultados. Para definir o fluxo de dados entre os lotes de tarefas utiliza-se o relacionamento de precedência dos lotes.

A plataforma JoiN possui uma linguagem de especificação de aplicações paralelas chamada PASL (*Parallel Application Specification Language*); que na prática é um arquivo texto formado por três seções distintas: *header*, *assignment* e *data link*.

Um exemplo de uma especificação completa e o respectivo diagrama de fluxo com a nomenclatura utilizada no PASL são mostrados na Fig. 8. Neste exemplo é apresentada uma aplicação formada por três lotes de tarefas: $\mathbf{B_1}$, $\mathbf{B_2}$ e $\mathbf{B_3}$, onde o fluxo de execução é $\mathbf{B_1 \rightarrow B_2 \rightarrow B_3}$, sendo que o os lotes $\mathbf{B_1}$ e $\mathbf{B_3}$ possuem cardinalidade unitária e o lote $\mathbf{B_2}$ possui 500 instâncias de tarefas tipo $\mathbf{T_2}$ (cardinalidade 500). Os dados a serem processados são obtidos do arquivo de entrada *infile* e o resultado é armazenado no arquivo de saída *outfile.*

Quando uma aplicação é executada, ocorrem as seguintes ações:

- o coordenador recebe do servidor a aplicação e distribui as tarefas e os dados entre os trabalhadores sob sua coordenação;
- após a distribuição de todas as tarefas de um lote e à medida que surgem trabalhadores ociosos, o coordenador inicia uma nova distribuição gradual das tarefas que ainda não retornaram seus resultados (replicação das tarefas) para estes trabalhadores, o que torna a aplicação tolerante a falhas nos trabalhadores e melhora o equilíbrio de carga;
- quando uma das réplicas de uma tarefa retorna seu resultado, as demais réplicas são finalizadas ou têm os seus resultados desprezados;
- devido ao relacionamento de precedência entre os lotes, antes de iniciar a execução das tarefas do próximo lote é necessário aguardar até que a última tarefa do lote corrente tenha terminado (apesar de ser um ponto de gargalo, a replicação das tarefas faz com que este problema seja minimizado).

```
// Seção Header
name = "Aplicação Exemplo";
description = "Exemplo de uma especificação
PASL";
%% // separador de seção

// Seção Assignment
path = "/app/test1"; //diretório com o
           //código da aplicação
T1 = "distribute.class"; //arquivo com a
       //computação realizada na
       //tarefa T1
T2 = "process.class";
T3 = "gather.class";

%% // separador de seção

// Seção Data Link
// Relacionamento entre os lotes de tarefas

B1 = T1(1) << infile //lote B1 possui 1
 // instância e recebe dados do arquivo
 // infile

B2 = T2(500) << B1; //lote B2 possui 500
  //instâncias da tarefa T2 e recebe os
  //dados da saída de B1

B3 = T3(1) << B2 >> ''outfile'';
 //lote B3 possui 1 instância da tarefa T3
 //recebe os dados de entrada da saída de
 // B2 e envia o resultado para "outfile"
```

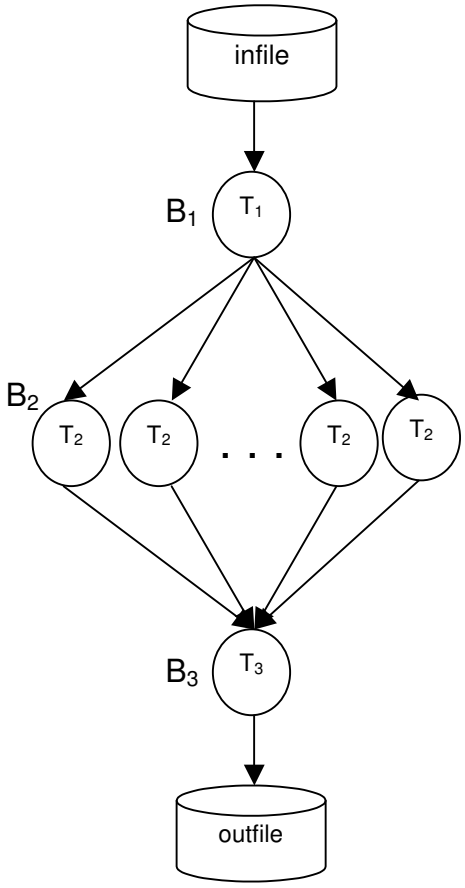

Figura 8. Exemplo de uma especificação de aplicação em JoiN

Apesar do PASL oferecer um mecanismo para a especificação de aplicações, definindo a cardinalidade dos lotes e as relações de precedência entre esses lotes, ele restringe a gama de aplicações que podem ser utilizadas no JoiN, uma vez que esse modelo também representa o modelo DAG e sofre suas limitações.

Uma comparação entre os sistemas analisados é apresentada na tabela 2.

3.1.6 Outros sistemas

Existem vários outros sistemas conhecidos, porém com características semelhantes aos apresentados neste capítulo, como os gerenciadores de filas LoadLeveler [IBM05] que é baseado no Condor e o OpenPBS/PBSPro [OPE05] que pode ser utilizado para executar aplicações BoT.

Outro exemplo é o Chimera VDS (*Virtual Data System*) que utiliza uma linguagem própria, o VDL (*Virtual Data Language*), para descrever dados e transformações representados em um grafo de dependências (DAG) [FOS02].

Sistema	Aplicação	Modelo de aplicação	Sistema Operacional
Condor	programas de computação intensiva	relacionamento de dependências DAG	Linux, Microsoft Windows, OS X, AIX
BOINC	aplicações para computação distribuída pública	básico e composto (*master-worker*)	aplicação cliente disponível para Microsoft Windows, OS X, Linux/x86, Solaris/SPARC (também é possível compilar a partir do código fonte)
Nimrod	simulações paramétricas	básico	Linux/x86, Solaris/sparc e OS X
OurGrid	aplicações paralelas do tipo *bag-of-tasks*	básico	*home machine/peer* - Linux, *grid machine* - Linux ou Windows
JoiN	aplicações para computação distribuída pública	relacionamento de dependências DAG	multiplataforma (Java)

Tabela 2 – Comparação entre os sistemas analisados

3.2 Linguagens de especificação de fluxo

Os sistemas analisados permitem a execução de aplicações BoT, ou no máximo aplicações cujo fluxo pode ser representado pelo modelo DAG. Estes sistemas não são adequados para aplicações que utilizam algoritmos iterativos ou um processamento condicional, como é o caso de diversos algoritmos genéticos, sendo necessário um controle de fluxo de execução mais sofisticado.

Recentemente, outras linguagens de controle de fluxo de execução para aplicações de *grid* foram propostas, como Condor DAGMan e PASL apresentadas neste capítulo.

Uma aplicação de controle de fluxo para *grid* pode ser vista como uma coleção de atividades (tarefas computacionais) que são processadas em uma ordem bem definida de forma a atingir um objetivo que é a solução do problema computacional.

No artigo "*A Taxonomy of Workflow Management Systems for Grid Computing*" é mostrada um análise de diversos modelos de controle de fluxo e proposta uma taxonomia

de controle de fluxo de execução em *grids* [YU05]. Em relação à estrutura, o fluxo de execução pode ser representado em duas classes: DAG e non-DAG.

No controle de fluxo baseado em DAG, a estrutura pode ser classificada como: seqüencial, paralela e controle de escolha:
- seqüencial: definida como uma série ordenada de tarefas, onde uma tarefa começa após a anterior ter terminado;
- paralela: representa as tarefas que são executadas concorrentemente;
- controle de escolha: a seleção da tarefa é feita em tempo de execução de acordo com alguma condição.

O modelo non-DAG possui as mesmas categorias que o DAG e além disso inclui a estrutura iterativa, para que sejam definidos blocos de tarefas onde é permitida a repetição da execução. A estrutura iterativa é freqüente em aplicações científicas onde uma ou mais tarefas precisam ser executadas mais de uma vez.

A seguir são apresentados alguns exemplos de especificação de fluxo para aplicações de *grid*.

3.2.1 GRID-ADL (*Grid Application Description Language*)

GRID-ADL é uma linguagem DAG que combina as características das linguagens VDL do Chimera e o Condor DAGMan, nesta linguagem o usuário especifica um arquivo descrevendo suas tarefas, arquivos de entrada e saída e também o tipo da aplicação. A partir disto o sistema infere o grafo de execução da aplicação[VAR04].

O arquivo com a descrição do fluxo de execução define: o tipo do grafo (tarefas independentes, fracamente ou fortemente acopladas) e as entradas e saídas de cada tarefa. A partir do relacionamento destes dados, é derivado o grafo de dependências para execução das tarefas (DAG).

Por ser uma linguagem DAG, ela não permite a definição de estruturas repetitivas nem a estrutura de controle de escolha.

3.2.2 AGWL (*Abstract Grid Workflow Language*)

AGWL é uma linguagem non-DAG baseada em XML que foi projetada de forma a cobrir as construções de fluxos de execução de tarefas essenciais, permitindo que o usuário descreva as aplicações científicas de uma forma intuitiva [FAH05]. Para isto ela descreve o fluxo de execução de aplicações com um alto nível de abstração, que não considera a implementação das atividades e nem como os dados são transferidos. Como esta é uma linguagem abstrata, para a execução da aplicação no *grid,* ela deve ser convertida para o CGWL (*Concrete Grid Workflow Language*). CGWL combina as informações da AGWL e da infra-estrutura em uma linguagem suportada pelo *grid*.

Ela permite controles de fluxo básicos como o fluxo seqüencial, *exclusive-choice (swich-case, if-then-else)* e laços seqüenciais (as tarefas no laço são executadas em

seqüência); também suporta estruturas complexas como execução paralela de atividades e laços paralelos (pelo menos duas tarefas no laço são executadas em paralelo)

Uma tarefa é representada pelo elemento *activity,* que possui propriedades e restrições. As propriedades descrevem atributos que podem ser úteis para o escalonamento ou predição de desempenho. As restrições são utilizadas para determinar os tipos de máquinas nas quais as tarefas podem ser executadas, usando especificações de memória, sistema operacional, etc.

AGWL é uma linguagem que procura cobrir a maioria das estruturas de controle das linguagens de programação convencionais, permitindo um controle de fluxo complexo em uma aplicação de *grid*. Como ela possui um alto nível de abstração, é necessária a conversão para outra linguagem baseada em XML que seja entendida pelo *grid*.

3.2.3 XRSL (*Extended Resource Specification Language*)

XRSL é uma linguagem baseada em XML que foi projetada para descrever *jobs* nos projeto PROGRESS [KOS04]. Apesar do nome sugerir, o XRSL não é uma versão XML do Globus RSL. Esta linguagem foi desenvolvida para descrever *jobs* num formato utilizado pelo ambiente de *grid* PROGRESS que utiliza um portal *Web* para acesso ao grid. Um fluxo de execução deste sistema pode ser representado por um DAG, onde um *job* deve definir pelo menos um dos seguintes tipos de tarefa:

- *single*: define uma única tarefa a ser executada no *grid*;
- *sequential*: descreve uma seqüencia de tarefas; precisa conter pelo menos uma tarefa ou uma estrutura paralela;
- *parallel*: descreve um conjunto de tarefas que serão executadas em paralelo no *grid*; deve conter pelo menos uma tarefa ou uma estrutura seqüencial.

Esta linguagem possui as mesmas limitações de outras linguagens DAG.

3.2.4 BPEL4WS (*Business Process Execution Language for Web Services*)

Apesar de ser recente em *grid*, o controle de fluxo de execução de aplicações tem sido estudado há bastante tempo em diversas outras áreas, tais como modelagem de negócios e de serviços *Web*. Um bom exemplo é *Business Process Execution Language for Web Services* (BPEL4WS) [AND05b].

BPEL4WS é uma linguagem para serviços *Web* que tem um modelo de integração fracamente acoplado para permitir uma integração de sistemas heterogêneos em diferentes domínios. O BPEL4WS define um modelo e a gramática para descrever o comportamento de processos de negócios baseado em interações entre processos. As interações são feitas por interfaces de serviços *Web* e o BPEL4WS define como é feita a coordenação dessas interações.

Apesar do BPELS4WS ser uma linguagem para serviços *Web*, algumas estruturas de controle de fluxo também poderiam ser estendidas para o modelo de fluxo de aplicações paralelas, como as estruturas de controle:

- *sequence*: atividades que são executadas em seqüência;
- *switch*: seleciona uma única atividade dentre várias opções;
- *while*: repete uma atividade até satisfazer um critério;
- *throw*: sinaliza uma falha no processo, permitindo que a aplicação seja cancelada, ou outra tomada de decisão;
- *compensate*: permite que seja executada uma atividade compensatória no caso de uma atividade falhar.

Acreditamos não ser possível utilizar esta linguagem diretamente em um sistema paralelo existente, já que BPEL4WS foi projetado para serviços *Web* e um sistema paralelo possui a1gumas características distintas, como a definição de um lote de tarefas que são executadas em paralelo e a utilização de restrições de execução para aplicações e recursos do *grid*. Entretanto, é possível que os recursos oferecidos por esta linguagem sejam aproveitáveis em um novo tipo de sistema de processamento paralelo projetado e construído com base nos conceitos de serviços *Web*.

O desenvolvimento de linguagens controle de fluxo de execução de tarefas para *grids* é um trabalho recente, que foi motivado pelas limitações dos modelos utilizados nos sistemas existentes. Como era de se esperar, a maioria das linguagens de controle de fluxo de execução de tarefas aplicáveis a *grids* se baseia nas estruturas de controle das linguagens de programação tradicionais.

3.3 Conclusões

Neste capítulo foram apresentados alguns sistemas paralelos com enfoque em seus fluxos de execução de aplicações.

Também foram apresentadas algumas linguagens para especificação e controle do fluxo de execução de tarefas em *grids* e suas limitações

No próximo capítulo é apresentado um novo modelo de especificação de fluxo para aplicações paralelas, que é flexível o suficiente para permitir especificações mais complexas que os dos sistemas paralelos existentes.

Capítulo 4 XPWSL: uma nova linguagem para especificação de fluxo de aplicações paralelas

Após a análise das restrições dos modelos de especificação de fluxo dos sistemas paralelos existentes feita no Cap. 3, é proposto um novo modelo de especificação de fluxo de execução de tarefas em aplicações paralelas mais flexível e sofisticado que o modelo DAG utilizado em diversos sistemas, permitindo fluxos de tarefas com laços de repetição e desvios condicionais. Também é proposta uma linguagem de especificação baseada neste modelo.

4.1 Modelo de referência para o sistema paralelo

Antes de detalhar a proposta de especificação de fluxo é preciso descrever o modelo de sistema paralelo adotado como referência. Para que esta proposta possa ter um espectro de aplicação o mais amplo possível, o modelo paralelo de referência é simples e pode ser encontrado em vários dos sistemas paralelos existentes, como aqueles apresentados no capítulo anterior.

O modelo de referência possui um módulo coordenador, responsável pela distribuição das tarefas e coleta dos resultados. Do ponto de vista do coordenador uma aplicação paralela é executada em etapas e cada etapa corresponde ao processamento de um determinado lote de tarefas, que vai desde a distribuição inicial dos dados até o retorno do resultado pela última tarefa do lote. Antes de disparar a execução do próximo lote, o coordenador aguarda os resultados de todas as tarefas do lote anterior, reordena os dados e os distribui para o próximo lote de tarefas. Esta mudança entre lotes é denominada ponto de transição, o qual pode depender de mais de um lote de tarefas de acordo com o grafo de dependências da aplicação. A Fig. 9 mostra as tarefas e o controle realizado pelo coordenador: cada caixa representa uma tarefa de um lote; caixas múltiplas indicam que o lote possui cardinalidade maior que 1; setas indicam não só o fluxo de dados, mas também o controle que é realizado pelo coordenador.

Para que o coordenador consiga distribuir os dados entre as tarefas de um lote, é necessário que estes dados estejam em um formato que permita a distribuição para as N tarefas do próximo lote. Além disso, deve haver um relacionamento entre as cardinalidades dos lotes, de forma que a cardinalidade de um lote seja um múltiplo ou sub-múltiplo da cardinalidade dos lotes de que ele depende diretamente.

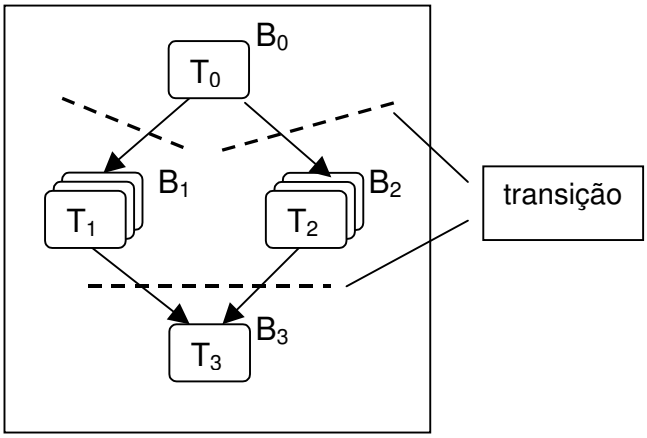

Figura 9. Etapas e pontos de transição de uma aplicação

4.2 Novo modelo de especificação de aplicações

O modelo de especificação de aplicações define e restringe os tipos de aplicações que podem ser executados em uma plataforma. Como o modelo DAG mostrou-se limitado, foram levantadas quais estruturas de controle de fluxo deveriam ser acrescentadas a este modelo, de forma que fosse coberta a maior parte dos tipos de aplicações.

Além de uma estrutura de repetição básica com um número fixo de iterações, outra estrutura de repetição essencial é aquela que permita repetições condicionais onde o número de iterações dependa de um critério de saída especial ou de um valor limite de iterações (para evitar que a execução fique em um laço infinito). Para isso deve ser utilizado um controle de decisão que verifica o resultado de cada iteração para decidir se o processamento no laço deve terminar ou não.

Uma outra estrutura desejável e necessária em alguns tipos de aplicações paralelas é aquela que permita um desvio condicional no fluxo, ou seja, que permita a escolha em tempo de execução de determinados lotes de tarefas com base em algum critério específico.

Sendo assim, o novo modelo de especificação de fluxo proposto provê suporte às seguintes estruturas de controle:

- DAG ou seqüencial: estrutura baseada nas relações de dependências entre lotes de tarefas;
- blocos de repetição (*loop*) fixo ou iterativo: esta estrutura deve ser utilizada quando o número de repetições do laço é conhecido a priori e todas as iterações devem ser realizadas;
- blocos de repetição condicional: utilizado nos processamentos iterativos em que a decisão de saída não é apenas a quantidade de repetições, mas também a verificação se um determinado critério foi satisfeito ou não. É ainda necessário o estabelecimento de um valor limite para o número de iterações para evitar que o programa fique em um laço infinito;

- desvio condicional *(switch/case)*: permite a escolha de um processamento específico de acordo com algum critério baseado nos dados de entrada. Esta estrutura é utilizada quando são necessários processamentos distintos dependendo do valor dos parâmetros.

4.3 O conceito de Bloco de Execução

Para permitir a execução de mais de uma estrutura de controle na mesma aplicação, foi introduzido o conceito de Bloco de Execução.

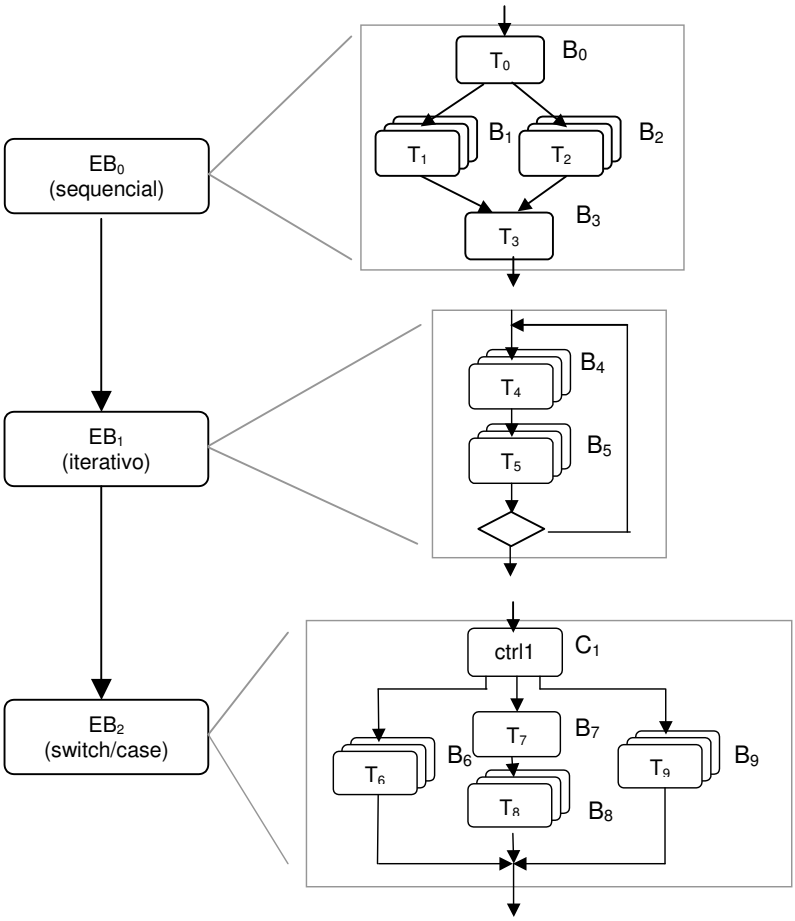

Figura 10. Exemplo de uma aplicação: seqüência de Blocos de Execução

O Bloco de Execução (EB) é um conjunto lotes de tarefas que são executadas com base em uma estrutura de controle (seqüencial, iterativa, repetição condicional ou *switch/case*). Uma aplicação paralela passa a ser vista como um conjunto de Blocos de Execução que são executados seqüencialmente. O resultado do último lote de tarefas de um bloco é passado como dado de entrada para o primeiro lote do próximo Bloco de Execução. Por exemplo, se existirem três Blocos de Execução definidos na seguinte ordem: EB_0, EB_1 e EB_2 o fluxo de execução dos Blocos de Execução será:

$$\mathbf{EB_0 \rightarrow EB_1 \rightarrow EB_2}$$

A computação dentro de uma estrutura de controle, como ilustrado na Fig. 10, representa um DAG formado por:
- um conjunto de lotes de tarefas (nós);
- um relacionamento de dependência entre os lotes (*links*).

Observe que no caso do bloco com laço de repetição, esta afirmação considera apenas a computação realizada em cada iteração isoladamente.

4.3.1 Processamento seqüencial

Este é o Bloco de Execução mais simples; ele é composto por um grupo de lotes de tarefas com um relacionamento de dependência seqüencial entre eles. A computação realizada neste bloco é representada pelo modelo de fluxo DAG.

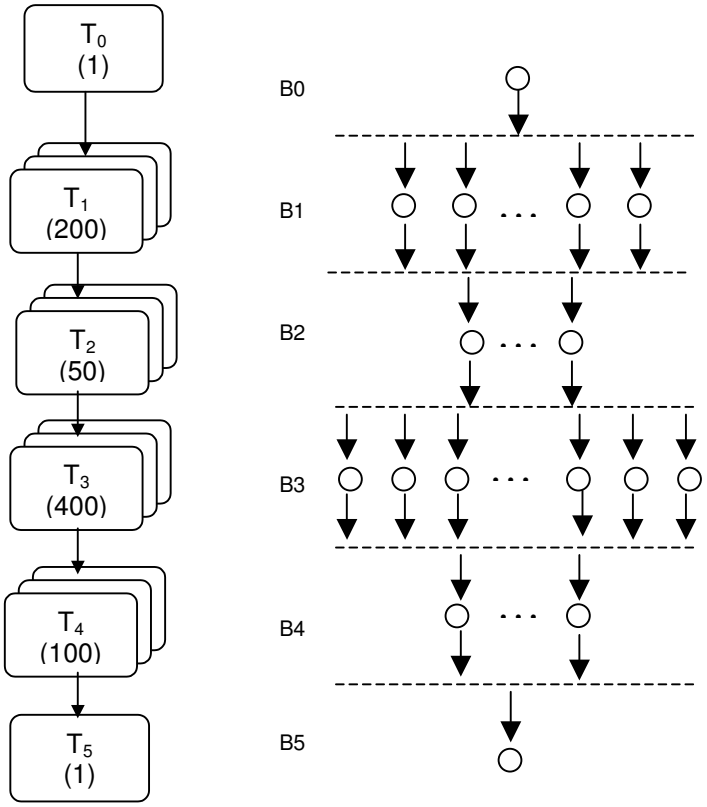

Figura 11. Diagramas em bloco e expandido do modelo EB seqüencial

Na Fig. 11 são mostrados o diagrama em blocos e o modelo expandido com os pontos de transição dos lotes de tarefas. Observa-se que o resultado de um lote é passado como entrada para o próximo lote via coordenador.

Neste modelo as relações de dependência são simples e a entrada de um lote de tarefas é o resultado de um ou mais lotes anteriores. Na Fig. 11 foi ilustrado um fluxo linear apenas

para simplificar a visualização do modelo expandido de fluxo das tarefas paralelas, porém nesta estrutura é possível representar qualquer relacionamento DAG.

4.3.2 Processamento iterativo

Este Bloco de Execução permite que um grupo de lotes de tarefas seja executado em um laço, como é ilustrado na Fig. 12. O resultado do último lote neste bloco é passado como dado de entrada para o primeiro lote se o número fixo de iterações do laço não foi ultrapassado. Caso contrário, o resultado é passado para o próximo Bloco de Execução. O controle do número de iterações é feito no coordenador. Deve ser utilizado quando o número de repetições do laço é conhecido a priori.

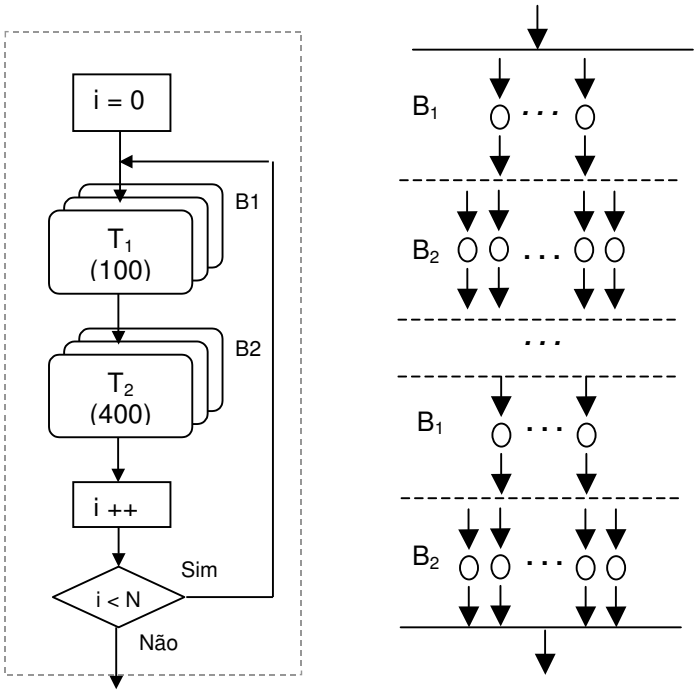

Figura 12. Diagramas em blocos e expandido do EB iterativo

Algoritmo utilizado no coordenador para o controle das iterações:

- início do bloco de repetição – inicia informações de controle
  - i = 0
  - Max_value = Num_iter
- após completar o último lote do bloco de repetição (no caso B2), incrementa o contador i
- testa número da iteração:
  - se i < Max_value
    - próximo lote: primeiro lote do Bloco de Execução (B1)
  - senão
    - próximo lote: primeiro lote do próximo Bloco de Execução

A Fig. 12 mostra um exemplo de um bloco de repetição com dois lotes de tarefas (B_1 e B_2), o controle do laço é feito com a utilização de um contador no Bloco de Execução.

Observa-se que o primeiro lote deste bloco recebe tanto dados do Bloco de Execução anterior quanto do último lote de tarefas do próprio bloco; desta forma, o resultado de B_2 deve ser do mesmo tipo que a entrada de B_1. Os lotes de tarefas em um bloco de repetição possuem as mesmas características daqueles do bloco seqüencial e, como B_1 depende de dois lotes distintos, sua cardinalidade deve ser um múltiplo ou sub-múltiplo das cardinalidades do último lote do Bloco de Execução anterior e do último lote do bloco de repetição onde se encontra B_1 .

4.3.3 Processamento de repetição condicional

Esta estrutura é utilizada nos processamentos iterativos em que a decisão de saída não é a quantidade máxima de repetições, mas sim um outro critério, que pode ser a verificação da precisão de um resultado, por exemplo. É necessário ainda o estabelecimento de um valor limite para o número de iterações para evitar que o programa fique em um laço infinito caso o critério de controle nunca seja satisfeito.

O Bloco de Execução de repetição condicional é semelhante ao processamento iterativo, tendo como única diferença a condição de saída do laço. Ele possui dois critérios de saída: (1) o limite de iterações foi atingido ou (2) uma condição definida foi satisfeita. Para verificar a condição de saída, o bloco de repetição condicional necessita de um lote extra, chamado lote de verificação ou controle. Este lote verifica os resultados do último lote executado e retorna um resultado do tipo *boolean*, indicando se são necessárias mais iterações ou não.

Conforme foi discutido na seção 4.1, deve haver um relacionamento entre o resultado de um lote e a entrada do próximo lote. O Bloco de Execução com repetição condicional permite duas condições de saída do laço, que é a satisfação de um critério de saída ou o número máximo de iterações. Para garantir que este bloco tenha o mesmo tipo/formato do resultado, independente do critério utilizado para saída do laço, o lote de controle precisa ser o último lote no Bloco de Execução, de forma que o resultado do Bloco de execução será sempre o resultado do último lote de tarefas. Portanto, não é possível usa saída do laço em outros pontos, senão o último lote do bloco de repetição.

Um exemplo deste tipo de Bloco de Execução é mostrado na Fig. 13, observa-se que devido ao lote de controle são necessárias duas verificações de saída: resultado do lote de controle ou quando atingir o número máximo de iterações. Neste exemplo, o resultado do bloco de repetição condicional é sempre o resultado do lote L2.

O lote de controle pode ser considerado um lote virtual. Ele apenas recebe o resultado da etapa anterior e verifica se o critério de saída é satisfeito ou não. Os dados recebidos como entrada são passados sem nenhuma modificação para a próxima etapa, que pode ser o primeiro lote do mesmo Bloco de Execução ou o primeiro lote do próximo Bloco de Execução.

Como o lote de controle tem cardinalidade sempre unitária e como os dados resultantes dos lotes são sempre enviados ao coordenador, é conveniente que Bc seja sempre executado pelo coordenador.

O bloco de repetição condicional é mais geral que o bloco com número fixo de iterações, já que este último pode ser representado no bloco condicional como o caso em que o lote de controle sempre retorna falso.

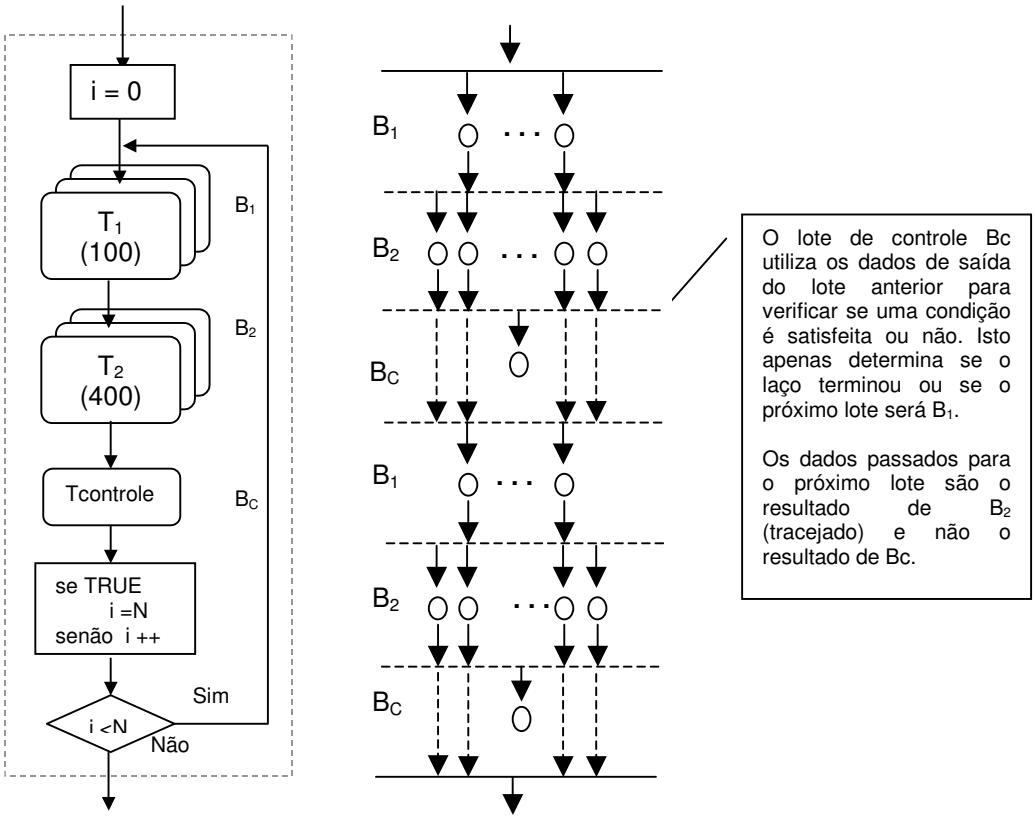

Figura 13. Diagrama em blocos e expandido do modelo de repetição condicional

Algoritmo utilizado no coordenador para o controle do bloco de repetição condicional:

- início do bloco de repetição – inicia informações de controle:
  - i = 0
  - Max_value = Num_iter
- termina a execução dos lotes de tarefas dentro do bloco de repetição, no caso B_1 e B_2;
- carrega a tarefa do lote de controle (B_C) e executa $T_{controle}$ com o resultado de B_2;
- se o resultado de $T_{controle}$ for:
  - TRUE, atualiza contador i= Max_value
  - FALSE, incrementa contador i
- testa número de iterações:
  - se i < Max_value

- próximo lote: primeiro lote do bloco de repetição (B_1);

o senão

- próximo lote: primeiro lote do próximo Bloco de Execução.

O lote de controle é carregado e liberado após cada execução do último lote de tarefas do bloco de repetição. Não é necessário ter este lote em memória durante a execução do bloco de repetição condicional porque ele apenas analisa o conjunto de dados final e indica se um critério foi satisfeito ou não. O coordenador precisa guardar apenas o limite de iterações e a iteração atual. Caso não haja restrições de memória, Bc poderá permanecer carregado enquanto acontece mais de uma iteração do laço. Entretanto, deve ser garantido que cada execução deste lote seja feita com a versão mais recente dos dados finais disponibilizados pelo último lote do laço.

4.3.4 Desvio condicional – *switch/case*

O Bloco de Execução de desvio condicional (*switch/case*) permite que um ou outro conjunto de lotes de tarefas seja executado dependendo dos dados de entrada. Neste bloco podem ser definidos diversos sub-blocos, porém apenas um deles será executado; cada sub-bloco possui um identificador e um conjunto de lotes de tarefas com um relacionamento de dependência como no Bloco de Execução seqüencial. Na Fig. 14 é apresentado um exemplo com 3 sub-blocos (a), (b) e (c). Da mesma forma que o bloco de repetição condicional, este também precisa ter um lote específico de controle, porém com uma finalidade distinta.

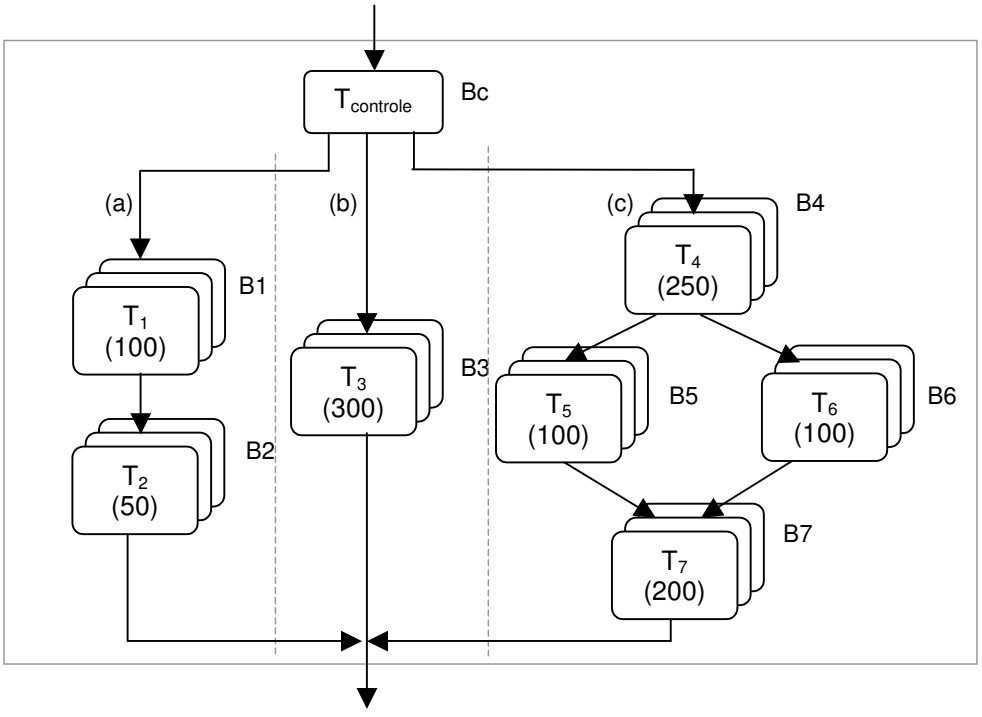

Figura 14. Diagrama do Bloco de Execução de desvio condicional (*switch/case*)

O lote de controle do *switch_case* possui as mesmas características do lote de controle do bloco de repetição condicional, só que em vez de retornar um resultado tipo *boolean*, ele retorna um identificador que informa qual dos conjuntos de lotes de tarefa será executado neste Bloco de Execução.

Algoritmo utilizado no coordenador:

- início do bloco switch/case
  - recebe os resultados do Bloco de Execução anterior;
- carrega a tarefa do lote de controle (B_C) e executa $T_{controle}$ com os resultados do lote anterior;
- resultado de $T_{controle}$ é o identificador de um sub-bloco do bloco de switch/case;
- este valor é comparado com o identificador de cada um dos sub-blocos;
- executa-se o primeiro sub-bloco cujo identificador coincidir com o resultado de $T_{controle}$.

Esta estrutura de controle é utilizada quando são necessários processamentos específicos dependendo do valor de determinados parâmetros.

4.4 Especificação em XML

Para implementar o modelo de aplicação com as novas estruturas de controle, foi desenvolvida uma nova linguagem de especificação de aplicações paralelas chamada "*XML based Parallel Workflow Specification Language*" (XPWSL).

A nova linguagem foi especificada com a notação XML (*eXtensible Markup Language*) por ser uma notação na qual é possível definir novos rótulos. Isto torna XPWSL mais flexível, pois permite que novos atributos e elementos sejam facilmente acrescentados. Outra vantagem de XML é que praticamente todas as linguagens de programação modernas já oferecem algum suporte para tratá-la, o que pode vir a simplificar futuramente a interpretação e a geração automática de especificações XPWSL com a utilização de uma interface gráfica para definição do fluxo de execução de tarefas.

Para a validação e a verificação do formato de XPWSL foi utilizado o modelo XML *Schema* [THO05]. Decidiu-se pela utilização de XML Schema em lugar de DTD (*Document Type Definition*) [DTD05], porque XML Schema oferece maior flexibilidade para definição do formato do documento. Além disso, já existem diversos pacotes Java que realizam a validação de documentos utilizando Schema, como o Xerces e o Crimson que já fazem parte do JDK 1.5. Na implementação da linguagem XPWSL a validação foi feita com a utilização do Xerces.

A nova especificação de aplicações possui três seções: *header, assignment* e *datalink*, que serão descritas a seguir. No Apêndice B é apresentada a especificação completa do XPWSL com a utilização de XML Schema.

4.4.1 Seção *Header*

Esta seção contém informações que permitem identificar uma aplicação. O sistema não precisa utilizar esta informação para controlar a execução da aplicação; porém ela permite que o usuário identifique facilmente uma aplicação entre as demais aplicações instaladas no sistema. Inicialmente foram definidos dois elementos: a identificação e a descrição da aplicação, como é apresentado na Fig. 15.

```
<header>
        <name>Nome da Aplicação</name>
        <description>Descrição da Aplicação</description>
</header>
```

Figura 15. Seção *header* de XPWSL

4.4.2 Seção *Assignment*

Esta seção define os arquivos com os códigos que serão executados em cada lote de tarefas. Cada tipo de tarefa é definido pelo elemento ***task*** e possui um identificador determinado pelo atributo ***id***. Este identificador deve ser único para cada tipo de tarefa e está associado ao arquivo com a implementação da tarefa. Uma tarefa de controle é definida pelo elemento ***control*** e possui os mesmos atributos de qualquer tarefa. A diferença entre as tarefas de controle e as tarefas ordinárias, é a interface que cada uma deve implementar (ver Cap. 5).

```
<assignment>
        <task id="T0" delay="250">
                <path>/app/test</path>
                <code>mytask.class</code>
        </task>
        ...
        <control id="C0" delay="5">
                <path>/app/test</path>
                <code>loop_control.class</code>
        </control>
</assignment>
```

Figura 16. Seção *assignment* de XPWSL

Esta seção é exemplificada na Fig. 16, onde:

- o elemento *task* define cada uma das tarefas da aplicação e seu identificador é definido no atributo *id.* O *id* não possui um padrão rígido para sua formação, podendo ser uma combinação qualquer de letras e números. Nos exemplos utilizados neste documento foram utilizados índices numéricos para identificar cada tipo de lote e tarefa apenas para facilitar a visualização do fluxo, isto não é uma imposição da linguagem XPWSL.
- o atributo *delay*, é utilizado para associar uma estimativa do tempo de execução a cada tarefa; esta informação seria utilizada por ferramentas de predição de desempenho, como aquela apresentada na ref. [HER04]. Este atributo define apenas um valor e não a unidade; fica a cargo da ferramenta de predição especificar qual unidade deve ser

utilizada em cada uma das estimativas. Este atributo é um exemplo de como é possível adicionar atributos a cada elemento de XPWSL de acordo com as características e funcionalidades de cada sistema paralelo; da mesma forma, também é possível acrescentar outros atributos às tarefas dependendo das necessidades do sistema;
- a localização e o nome do arquivo executável da tarefa são definidos pelos elementos *path* e *code*, respectivamente.
- a computação realizada pela tarefa de controle é definida no elemento *control,* que possui os mesmos elementos e atributos que as demais tarefas.

4.4.3 Seção *Datalink*

Esta seção define os Blocos de Execução de uma aplicação e o relacionamento entre eles, como mostrado na Fig. 17. Neste exemplo o diagrama da aplicação é semelhante ao diagrama apresentado na Fig. 10.

Cada Bloco de Execução é definido pelo elemento *block* e o tipo do bloco é identificado pelo seu atributo *type*. Se o *Bloco de Execução* possuir uma tarefa de controle, o identificador dessa tarefa deve ser especificado no atributo *control*, como pode ser observado para o caso do *desvio condicional* e *repetição condicional*. A ordem de execução dos blocos segue a ordem com que foram descritos na XPWSL.

Cada Bloco de Execução (*block*) possui um conjunto de lotes que são definidos pelo elemento *batch,* que tem como atributo o identificador *id* do lote. O elemento *batch* define o tipo de tarefa que será executada (*task*), a multiplicidade das tarefas do lote (*multi*), de onde são obtidos os dados de entrada (*input*) e opcionalmente, um arquivo de saída (*output*) para armazenar os resultados do lote.

O relacionamento de precedência entre os lotes de um Bloco de Execução é determinado pelo atributo *input*, ele é utilizado para indicar a origem dos dados de entrada, que pode ser:
- vazio – se o elemento *input* estiver vazio, o lote não precisa de dados de entrada, ou seja, não possui nenhuma dependência para sua execução;
- arquivo – nome de um arquivo de onde os dados de entrada são lidos;
- outros lotes – identificadores de lotes dos quais serão obtidos os dados de entrada.

O primeiro lote de um Bloco de Execução não define nenhum valor para o atributo *input* porque os dados de entrada são obtidos do Bloco de Execução anterior. Apenas no caso do primeiro Bloco de Execução é necessário definir a fonte dos dados de entrada.

No caso do *desvio condicional*, o Bloco de Execução possui cláusulas para determinar que conjunto de lotes será executado. Estas cláusulas são definidas pelo elemento *case*. Será executada a cláusula que tiver o mesmo valor (atributo *value*) que o resultado da tarefa de controle *C1*.

Em qualquer uma das estruturas de controle pode ser realizado o processamento de um relacionamento de dependências entre lotes (DAG). Porém, na implementação da

linguagem XPWSL, não é permitido que as estruturas de controles sejam aninhadas. Nessa implementação, uma aplicação pode fazer uma combinação em seqüência das estruturas de controle, mas a combinação de estruturas não é permitida. Por exemplo, não é possível utilizar uma estrutura de repetição que utiliza durante o processamento de cada iteração uma estrutura de desvio condicional.

```
<datalink>
        <block type="sequential">
                <batch id="B0">
                        <task_id>T0</task_id>
                        <multi>250</multi>
                        <input></input>
                </batch>
                <batch id= "B1">
                        …
                </batch>
        </block>
        <block type="loop" control="C0" max_iter="9999">
                <batch id="B2">
                        <task_id>T2</task_id>
                        <multi>250</multi>
                        <input></input>
                </batch>
                <batch id= "B3">
                        …
                </batch>
        </block>
        <block type="switch" control="C1">
                <case value="0">
                        <batch id="B4">
                                <task_id>T4</task_id>
                                <multi> 500 </multi>
                                <input></input>
                        </batch>
                        <batch id= "B5">
                                …
                        </batch>
                </case>
                <case value="1">
                        …
                </case>
        </block>
        <block type="sequential">
                <batch id="B6">
                        <task_id>T6</task_id>
                        <multi>1</multi>
                        <output>resultados.data</output>
                </batch>
        </block>
</datalink>
```

Figura 17. Seção *datalink* de XPWSL

Por restrições de tempo e pelo fato de ser possível mapear grande parte das aplicações paralelas em uma seqüência de blocos de execução, a implementação atual não permite a combinação de estruturas do mesmo tipo ou de tipos diferentes. Entretanto, acreditamos que adicionar suporte a estruturas combinadas é possível e é simples, afetando apenas a especificação em XML Schema da linguagem XPWSL e as classes de controle do *workflow*.

A seção *datalink* descreve apenas o fluxo de execução das tarefas. Para isso utiliza apenas os identificadores das tarefas, o que permite que a XPWSL seja combinado com outras linguagens como, por exemplo, JSDL (*Job Submission Description Language*). Esta é uma linguagem padronizada proposta pelo GGF (*Global Grid Forum*) para definição de *jobs* e não de fluxos de tarefas [ANJ05]. JSDL é utilizada para descrever os requisitos de submissão de *jobs* computacionais, tais como quantidade mínima de memória necessária, por exemplo. Da mesma forma que a XPWSL, esta linguagem é baseada em XML com a utilização de XML Schema.

A complexidade de uma linguagem de especificação está diretamente relacionada com a complexidade de implementação do suporte a esta linguagem no sistema paralelo. A linguagem de especificação apresentada neste trabalho não possui todas as estruturas de controle de fluxo como o AGWL; porém nossa proposta é definir uma linguagem flexível que permita a especificação de grande parte das aplicações paralelas que podem ser executadas em um *grid.* Além disso, a linguagem deve ser de implementação simples e viável nos *grids* existentes que satisfaçam o modelo de sistema paralelo tomado como referência.

4.5 Comparação com outras linguagens de especificação de fluxo

Na seção 3.2 foram apresentadas algumas linguagens de especificação de fluxo existentes, cada uma com suas características.

As linguagens GRID-ADL e XRSL são variações do modelo DAG sendo portanto mais limitadas que a XPWSL ou qualquer outra linguagem que suporte o modelo non-DAG (por exemplo: AGWL e BPEL4WS).

BPEL4WS é uma linguagem para serviços *Web.* Apesar de ser padronizada e possuir diversas ferramentas e implementações, não é possível utilizar esta linguagem diretamente, sem adaptações, em um sistema de computação distribuída pública ou um sistema paralelo em que os lotes possuem um número elevado de tarefas sendo executadas em paralelo.

AGWL é uma linguagem para *grids* que descreve o fluxo de execução com um alto nível de abstração, sem considerar detalhes de implementação nem controle do *grid*. Esta linguagem cobre a maioria das estruturas de controle utilizadas nas linguagens de programação tradicionais; portanto, permite um controle de fluxo complexo em uma aplicação de *grid*. O custo da flexibilidade é um aumento na complexidade: para executar

uma aplicação AGWL em um *grid* é necessário converter esta linguagem abstrata em uma outra que é compreendida pelo *grid* (CGWL). Uma outra característica é que, por ser de alto nível, a linguagem AGWL é independente de plataforma e toda a parte específica para um sistema paralelo é feita pela CGWL.

XPWSL foi desenvolvida com o propósito de ser uma linguagem simples. Ela possui um conjunto básico de estruturas de controle para representar um modelo non-DAG. Além de permitir a execução de grande parte das aplicações paralelas, é possível adicionar o suporte à XPWSL em um sistema paralelo existente, desde que este possua um módulo coordenador responsável pela distribuição das tarefas e coleta dos resultados. Da mesma forma que as linguagens BPEL4WS e AGWL, ela é baseada em XML o que lhe confere flexibilidade.

O objetivo de XPWSL não é ser a linguagem de especificação de fluxo mais completa. Seu objetivo é ser uma linguagem simples, mas que permita a execução de ampla gama de aplicações que poderiam se beneficiar do processamento paralelo, mas que não podem ser executadas em um modelo DAG.

4.6 Representação gráfica do XPWSL e o diagrama de atividades UML

Por ser uma especificação de controle de fluxo de execução, a XPWSL poderia se beneficiar do XML e utilizar diagramas de atividades UML para visualização e definição de uma aplicação. A grande vantagem da utilização dos diagramas de atividades é a facilidade para a conversão/geração da especificação em XML a partir de uma representação gráfica.

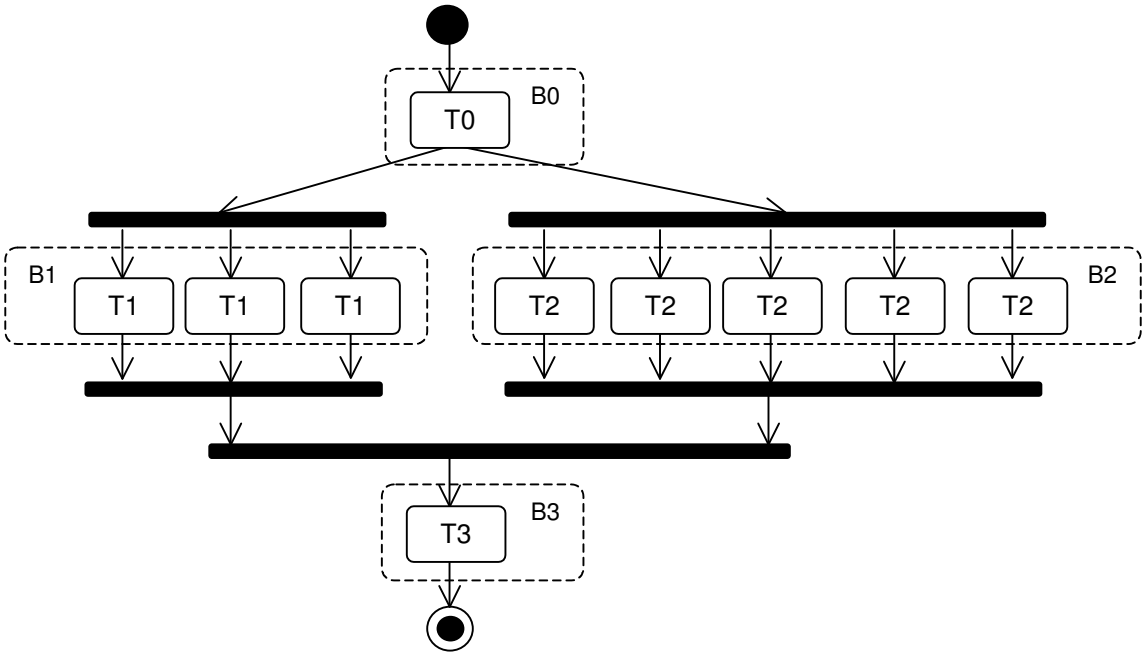

Figura 18. Representação da XPWSL como um diagrama de atividades

A dificuldade em representar a XPWSL com esse tipo de diagrama é a característica dos lotes de tarefas. Por ser voltado para o processamento paralelo, uma das características da XPWSL é a definição da cardinalidade dos lotes de tarefas, em que um elevado número de instâncias da mesma tarefa é executadao de forma concorrente.

As figuras 18 e 19 mostram como a XPWSL poderia ser mapeado em diagramas de atividades. A Fig. 18 é a representação do exemplo da Fig. 9 e a Fig. 19 representa o fluxo de um bloco de execução de repetição condicional como o da Fig. 13.

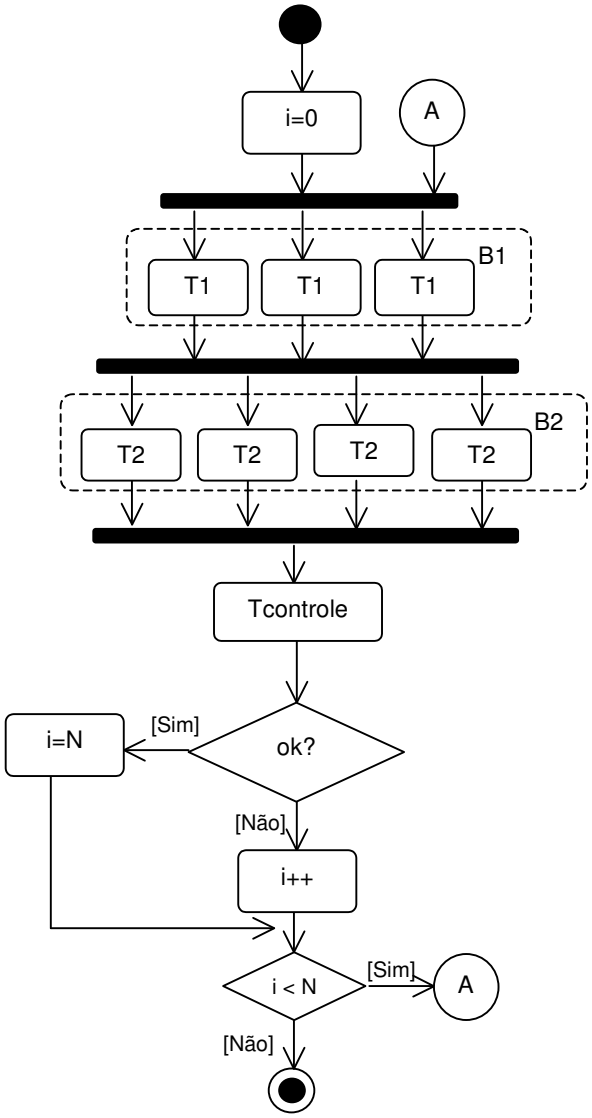

Figura 19. Representação do bloco de repetição condicional

Observa-se que neste tipo de diagrama é necessário definir cada uma das instâncias de tarefas explicitamente. Apesar de ser possível representar a XPWSL por um diagrama de atividades, isso se torna inviável em uma aplicação paralela real em que cada lote possui um número elevado de tarefas. Portanto, deve ser buscada uma forma alternativa para representação gráfica da XPWSL.

4.7 Conclusões

Neste capítulo foi apresentada a proposta de um novo modelo de fluxo de execução de tarefas em aplicações paralelas e uma linguagem de especificação para este modelo.

A linguagem XPWSL define além das estruturas de controle DAG, novas estruturas de controle como: repetição com um número fixo de iterações, repetição condicional e também o desvio condicional que permite a seleção de lotes de tarefas em tempo de execução.

Recentemente diversas linguagens têm sido desenvolvidas para especificar o fluxo de execução de aplicações de *grids*. Algumas dessas linguagens apenas refinam o modelo DAG, enquanto outras, como o AGWL, utilizam um alto nível de abstração e permitem praticamente todas as estruturas de controle utilizadas nas linguagens de programação convencionais, mas a um custo elevado.

A proposta da linguagem XPWSL não é cobrir todas as estruturas de controle possíveis, mas definir um conjunto de estruturas que possibilitem a implementação de grande parte das aplicações que poderiam se beneficiar da computação paralela em *grids*. Além de ser uma linguagem flexível, a implementação do suporte à esta linguagem em um sistema paralelo é simples, como será apresentado no próximo capítulo, já que é possível tirar proveito do suporte à notação XML provida pelas linguagens de programação modernas.

Capítulo 5 Implementação e testes da linguagem XPWSL na plataforma JoiN

Neste capítulo será mostrado como a linguagem de especificação proposta no capítulo anterior pode ser implementada em um sistema paralelo. Neste caso a implementação foi feita na plataforma JoiN, mas acreditamos que poderia ter sido feita em outro sistema desde que este possua um módulo coordenador/gerenciador de execução das tarefas, como descrito no modelo de sistema paralelo (Seção 4.1).

Inicialmente será apresentado o modelo de aplicações de JoiN e em seguida apresentada a implementação da linguagem XPWSL nesta plataforma.

5.1 O modelo de aplicações de JoiN

Em JoiN [LUC02] é definido um modelo de aplicação baseado em blocos de dados, tarefas, lotes de tarefas e a relação de dependência de dados. Basicamente, uma aplicação paralela é formada por:

- um conjunto de lotes de tarefas, em que um lote é composto por várias cópias de uma mesma tarefa (dentro de um lote são executadas as mesmas operações em conjuntos distintos de dados);
- um conjunto de dependências de dados entre lotes de tarefas, com restrições para que não haja dependências cíclicas.

As dependências entre lotes de tarefa são definidas por meio de uma linguagem de especificação de aplicações paralelas, que foi descrita superficialmente na Seção 3.1.5 e que agora será abordada com mais detalhes.

Qualquer aplicação em JoiN precisa ter, além do código a executar, uma especificação da estrutura da aplicação mostrando como é o relacionamento de precedência entre os lotes e a multiplicidade das tarefas, atendendo os seguintes os requisitos:

- definir tarefas da aplicação atribuindo um identificador lógico a um bloco lógico que possua interfaces de entrada e saída bem definidas;
- definir lotes de tarefas da aplicação;
- definir as relações de precedência entre os lotes de tarefas da aplicação descrevendo os fluxos de dados entre eles;
- descrever o tipo de computação realizado pelas tarefas.

Essa especificação é definida por meio de uma linguagem de especificação de aplicações paralelas (PASL – *Parallel Application Specification Language*). Na prática, a especificação PASL é um arquivo texto formado por três seções distintas:

- *header:* contém o nome e a descrição da aplicação;
- *assignment*: definição dos caminhos (*paths*) até os códigos e identificação dos tipos de tarefas; os identificadores de tarefas são formados pela justaposição da letra T com um índice numérico;
- *datalink*, definição dos lotes de tarefas que formam a aplicação e as relações de precedência entre lotes. Este relacionamento possui a seguinte sintaxe:

```
B[bid] = T[tid](m)<< [B[bid][,B[bid]]*| arquivo | NULL] [>> arquivo]*;
```

  onde:
  - B identifica um lote de tarefas;
  - T identifica um tipo de tarefa;
  - *bid* representa um identificador de lote e *tid* um identificador de tarefa, ambos formados por números inteiros;
  - o termo *m* indica a multiciplidade das tarefas;
  - o operador << é utilizado para indicar a origem dos dados entregues ao lote; estes dados podem vir de um ou mais lotes ou de um repositório de dados, tal como um arquivo local ou remoto;
  - a utilização da constante NULL indica que o lote de tarefas não necessita de dados de entrada para ser executado;
  - o operador >> é opcional e deve ser utilizado quando os dados de saída de um lote tiverem que ser armazenados em um repositório de dados. Apenas um arquivo de saída pode ser especificado.

Uma característica da linguagem PASL é que as operações de leitura e escrita em um repositório de dados só podem ser realizadas por lotes com cardinalidade unitária. Isto é necessário já que não é possível prever como os dados do repositório seriam distribuídos entre as *m* tarefas do lote ou como estas *m* tarefas escreveriam concorrentemente em um repositório.

Para que a ligação de dados entre lotes de cardinalidades distintas seja possível, supõe-se que as suas respectivas tarefas possuirão interfaces de entrada e saída de dados que suportarão essa ligação.

JoiN foi totalmente desenvolvido na linguagem Java para facilitar sua portabilidade em uma ampla gama de arquiteturas de computadores e de sistemas operacionais. Além disso é necessário que suas aplicações paralelas também sejam desenvolvidas nesta linguagem, de forma que possa haver uma maior integração entre a aplicação e os serviços oferecidos pela plataforma paralela, resultando em uma maior eficiência na execução.

Para que uma aplicação seja executada na plataforma Join é necessário que:

- cada bloco de código implemente a classe *AppCode* e o método *taskrun;*
- a multiplicidade e a ordem de execução dos blocos de código estejam definidos pela linguagem de especificação PASL.

No JoiN cada tarefa é uma classe Java que implementa a interface AppCode:

```
public interface AppCode {
 public Serializable taskrun(Serializable par);
}
```

Esta interface define o método `taskrun`, que implementa a computação realizada em um trabalhador. No Apêndice C é mostrado mais detalhadamente como uma tarefa JoiN é implementada. Basicamente, quando o trabalhador recebe uma tarefa do coordenador, ele apenas executa o método `taskrun` com os parâmetros recebidos e em seguida retorna o resultado.

A execução de uma aplicação na plataforma JoiN é controlada pelo módulo coordenador, que é responsável pelo controle da execução das tarefas da aplicação utilizando os componentes `TaskScheduler` e `ExecutionEngine`. O `TaskScheduler` utiliza uma avaliação de desempenho (*benchmark*) de cada trabalhador para definir o escalonamento das tarefas aos mesmos. O `ExecutionEngine` realiza a determinação das tarefas para os trabalhadores, sendo responsável pelo controle das aplicações. Ele mantém uma lista com as tarefas a serem executadas para cada aplicação e, à medida que as tarefas são concluídas e as respostas devolvidas, o `ExecutionEngine` ordena estas respostas e monta/particiona os dados que serão enviados aos trabalhadores que executarão as tarefas dos próximos lotes que dependam desses dados.

Na Seção 4.3.1, foi observado que, na estrutura do laço de repetição, o primeiro lote do laço pode receber os dados de entrada do lote anterior ou do último lote do laço. Mas como não existe a persistência de informações de estado no JoiN, esta duplicidade nas relações de precedência em um bloco de repetição inviabiliza a implementação de um laço neste sistema.

5.2 Implementação do novo modelo de aplicações na plataforma JoiN

O controle da execução de uma aplicação é feito pelo módulo coordenador e, para isso, ele utiliza o serviço `ApplicationManager`. O `ApplicationManager` possui três componentes principais: `TaskScheduler`, `ExecutionEngine` e o `JAS`. O relacionamento entre as principais classes do `ApplicationManager` é ilustrado na Fig. 20.

O `JAS` (*JoiN Application Specification*) é responsável pela interpretação do arquivo PASL da aplicação.

O `TaskScheduler` realiza o escalonamento das tarefas nos trabalhadores.

O `ExecutionEngine` controla o fluxo da aplicação utilizando uma lista com todas as tarefas da aplicação e organizando os dados de entrada e os resultados dos trabalhadores. O `ExecutionEngine` não realiza um controle de estado da aplicação; ele agrupa as tarefas de acordo com o relacionamento de precedência em “prontas” e “não prontas” para

execução. São consideradas "prontas" as tarefas que não dependam de nenhuma entrada de dados ou que os dados de que elas dependam já estejam disponíveis. A execução dos lotes de tarefas é feita a partir da lista de tarefas "prontas" e, à medida que elas vão sendo executadas, as outras tarefas que delas dependam e que ainda eram consideradas "não prontas" são removidas da lista de "não prontas" e passadas para a lista de "prontas". Este processo é repetido até que não existam mais tarefas na lista de "não prontas". No PASL não é permitido nenhum ciclo no grafo de dependências, garantindo que o fluxo nunca entra em um laço infinito.

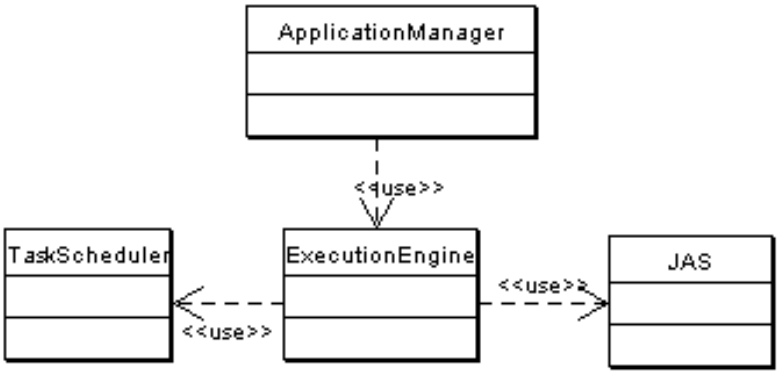

Figura 20. Relacionamento das principais classes do ApplicationManager de JoiN

Para suportar as novas estruturas de controle foi acrescentada uma nova definição em JoiN: o Bloco de Execução. Como foi discutido no Cap. 4, no novo modelo uma aplicação consiste de um conjunto de Blocos de Execução; cada bloco contém um objeto `ExecutionEngine` com as tarefas que serão processadas no mesmo; ele também pode possuir um método de controle que vai depender de seu tipo.

Foi criada também uma nova classe, o `AppEngine`, para fazer o controle do fluxo de execução entre os Blocos de Execução. A função do `AppEngine` é ser apenas um envoltório (*wrapper*). O escalonamento dos lotes de tarefas no `ApplicationManager` continua sendo baseado no `ExecutionEngine` com a única diferença de que, em lugar de processar uma aplicação, apenas um Bloco de Execução será processado; isso é transparente para o `ApplicationManager`. O `AppEngine` apenas controla a ordem de execução dos Blocos de Execução. O relacionamento entre as classes do novo modelo de especificação e o `ApplicationManager` é ilustrado na Fig. 21.

O `JAS_Manager` substitui o `JAS`, fazendo a interpretação do novo formato em XPWSL e realizando o controle dos Blocos de Execução.

Todos os tipos de Bloco de Execução devem derivar da interface básica `ExecutionBlock`:

```
public interface ExecutionBlock {
 public void initialize(Object[] _result);
 public ExecutionEngine getExecutionEngine ( JAS_Manager jasMan,
                                                    long appNumber);
 public boolean moreIterations(Object[] _result);
 public void finalize();
}
```

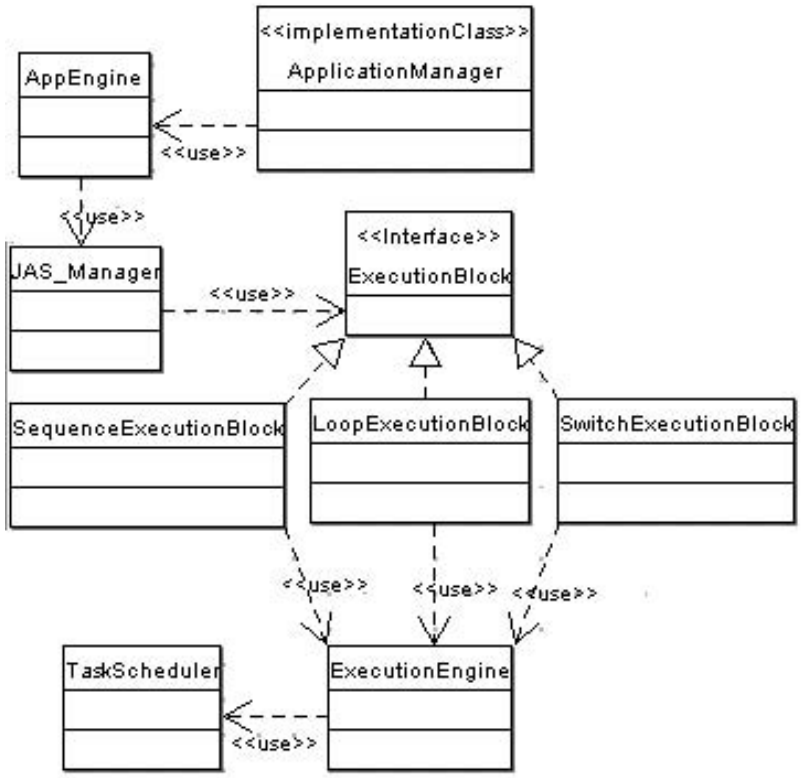

Figura 21. Relacionamento de classes no novo modelo para especificação de fluxo de aplicações

Esta interface comum especifica os seguintes métodos:

- **`initialize`***:* é o primeiro método chamado quando o controle passa para o Bloco de Execução. Ele recebe como parâmetro o resultado do último `ExecutionBlock` executado;
- **`getExecutionEngine`***:* retorna o `ExecutionEngine` apropriado para o Bloco de Execução ativo, contendo os lotes de tarefas a serem processados e as relações de dependência. Este método é chamado no momento em que o `ExecutionEngine` for executado;
- **`moreIterations`***:* é um método *boolean* utilizado pela tarefa de controle do Bloco de Execução iterativo para determinar se o controle deve ser passado para o próximo Bloco de Execução ou o mesmo bloco deve ser re-executado. Como todos os tipos de blocos devem implementar esta interface, nos Blocos de Execução que não são iterativos esse método retorna sempre falso. O controle de execução é independente do tipo do bloco, sendo feito apenas com a utilização dos métodos definidos nesta interface, por isso, apesar deste método ser utilizado para o controle de iterações ele

precisa fazer parte da interface do ExecutionBlock. Este método é chamado após a execução do último lote de tarefas do Bloco de Execução;
- **`finalize`***:* método utilizado para o processamento de finalização do bloco. Ele é chamado imediatamente antes do controle ser passado para o próximo Bloco de Execução.

Cada uma das estruturas de controle deve implementar a interface `ExecutionBlock` e as classes são detalhadas a seguir.

`public class SequenceExecutionBlock`

Esta classe representa a estrutura básica onde não é aplicado nenhum método de controle nos dados de entrada. O método `getExecutionEngine` retorna um `ExecutionEngine` com os lotes de tarefas a serem executados.

`public class SwitchExecutionBlock`

Nesta classe, o método `initialize` recebe os dados de entrada para este Bloco de Execução. É necessário definir na XPWSL qual a classe condicional que está associada a esse Bloco de Execução. Esta classe de controle, que implementa a interface `AppSwitchControl`, é carregada durante a execução e o método `getConditional` retorna uma string que é comparada com os valores das cláusulas definidas no *switch* para determinar quais lotes serão executados. O método `getExecutionEngine` retorna um `ExecutionEngine` com os lotes selecionados.

`public class LoopExecutionBlock`

Este Bloco de Execução pode ser executado com ou sem a definição de uma classe de controle. A classe de controle do laço deve implementar a interface `AppLoopControl`. Se for especificada uma classe de controle do laço na XPWSL, o resultado do último lote de tarefas é verificado utilizando o método `moreIterations`. Este método carrega a classe de controle e passa o resultado do lote de tarefas como parâmetro. O controle de execução passará para o próximo Bloco de Execução se o número máximo de iterações for atingido ou se o `moreIterations` retornar o *boolean* falso. Se a classe de controle não for especificada, apenas o número máximo de iterações é verificado.

No JoiN, os lotes com cardinalidade unitária, isto é, que são executados por uma única tarefa, geralmente são utilizados para realizar a distribuição de dados (preparar um vetor com os dados a serem enviados para cada trabalhador) ou para coletar os resultados dos trabalhadores. Por esse motivo, são executados no próprio coordenador.

As classes de controle (`AppSwitchControl` e `AppLoopControl`) também possuem cardinalidade unitária. Como estas classes devem implementar uma verificação de um conjunto de dados que as tarefas de um lote retornaram ao coordenador e como esta verificação é feita por meio de um lote de cardinalidade unitária, o novo modelo com os Blocos de Execução também se beneficia do tratamento de lotes unitários no próprio coordenador da plataforma JoiN.

Para um desenvolvedor de aplicações paralelas em JoiN não há grandes diferenças na preparação dos programas ao adotar a nova linguagem de especificação de fluxo XPWSL. As diferenças principais do novo modelo em relação ao sistema original baseado em PASL está na linguagem de especificação de aplicações e nas classes de controle para suportar as novas estruturas de controle. A implementação das classes com o código das tarefas continua sendo feita da mesma forma que no modelo PASL (Apêndice C).

Para utilizar a estrutura condicional e a de repetição condicional, é necessário implementar o tratamento para realizar o controle das mesmas, visto que cada uma possui uma interface específica.

O bloco de repetição condicional utiliza uma classe de controle *boolean* que recebe o resultado do último lote de tarefas do bloco de execução e retorna um *boolean* indicando se o critério de saída do laço foi satisfeito ou não. A classe de controle deve implementar a interface *AppLoopControl*:

```
public interface AppLoopControl {
     public boolean exitLoop(Serializable par);
}
```

O bloco *switch/case* utiliza uma classe de controle que retorna uma *string*. Esta classe é carregada assim que o controle passa para esse bloco de execução; os dados de entrada do bloco de execução são passados para o método *getConditional* e este retorna uma *string* que será comparada com os valores de cada cláusula *case*. A utilização da classe de controle para definição da escolha da cláusula permite que o desenvolvedor defina qualquer critério e um número indeterminado de tratamentos diferenciados. Abaixo é mostrada a interface `AppSwitchControl` utilizada pelo controle `switch/case`.

```
public interface AppSwitchControl {
     public String getConditional(Serializable par);
}
```

5.3 O processo de execução de uma aplicação

O processo de execução de uma aplicação no JoiN não foi alterado com a implementação da linguagem XPWSL. Do ponto de vista do usuário, a única diferença é o formato do arquivo de especificação da aplicação, que antes era em PASL.

O primeiro passo é instalar a aplicação no sistema JoiN. A instalação é feita com o módulo Jack.

Após a seleção do arquivo com a especificação da aplicação, o Jack utiliza a classe Xml2JasManager para fazer a interpretação do arquivo de especificação. Esta classe realiza a validação do XML (utilizando o Xerces) e retorna um objeto JAS_Manager com todas as informações da aplicação. Após a validação, os arquivos com os códigos das tarefas e classes auxiliares são transferidos para o sistema. Uma vez que a instalação tenha sido feita

com sucesso, a aplicação é disponibilizada na lista de aplicações que podem ser executadas no JoiN.

Quando o usuário solicita a execução de uma aplicação, o Jack carrega um objeto JAS_Manager com as informações da aplicação e o envia ao módulo coordenador.

Finalmente, o coordenador adiciona a aplicação à lista de aplicações que estão sendo executadas e realiza o controle de execução e distribuição das tarefas aos trabalhadores conectados.

5.4 Testes e resultados

Cada uma das novas estruturas de controle foi implementada, testada e validada separadamente. Em seguida, algumas aplicações paralelas que requerem tais estruturas foram implementadas e testadas com a XPWSL. Vários algoritmos, tais como algoritmos genéticos, por exemplo, que não poderiam ser implementadas na plataforma JoiN sem as novas estruturas de XPWSL, se tornaram viáveis nesta plataforma, abrindo o leque de aplicações que podem tirar proveito da mesma. A seguir são apresentados dois destes algoritmos.

5.4.1 Implementação de algoritmo com número de iterações fixo

Entre as aplicações implementadas e testadas, encontra-se uma que utiliza um pacote em Java para algoritmos genéticos [GAJ05]. Esta aplicação considera uma população de cromossomos e um conjunto de operadores genéticos que define a probabilidade de cada um ser escolhido entre os demais. A partir dessa população é iniciado um processo iterativo de atribuição de um grau de *fitness* e a geração de uma nova população. Este programa exercitou com sucesso as estruturas de repetição com um número fixo de iterações e a estrutura seqüencial. Sua especificação em JoiN usando PASL torna-se inviável à medida que aumenta o número de iterações desejadas, pois passa a exigir um número muito elevado de lotes de tarefas em seqüência. Usando XPWSL, a especificação da aplicação é praticamente a mesma para qualquer número desejado de iterações. A seguir são apresentados os detalhes da implementação desta aplicação, bem como toda a sua especificação de fluxo usando XPWSL.

```
<?xml version="1.0"?>
<pas xmlns="http://www.dca.fee.unicamp.br/projects/join"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://www.dca.fee.unicamp.br/projects/join
PAS.xsd">

<header>
     <name>GAGS new</name>
     <description>Genetic Algorithm program</description>
</header>

<assignment>
```

```
<task id="init" delay="100">
     <path>br.unicamp.fee.dca.join.applications.Gags</path>
     <code>InitPopul.class</code>
 </task>
<task id="eval" delay="800">
     <path>br.unicamp.fee.dca.join.applications.Gags</path>
     <code>Eval.class</code>
 </task>
<task id="result" delay="100">
     <path>br.unicamp.fee.dca.join.applications.Gags</path>
     <code>Results.class</code>
 </task>
<task id="aux1" delay="0">
     <path>br.unicamp.fee.dca.join.applications.Gags</path>
     <code>Gags_constants.class</code>
 </task>
<task id="aux2" delay="0">
     <path>br.unicamp.fee.dca.join.applications.Gags.gajit</path>
     <code>Chrom.class</code>
 </task>
<task id="aux3" delay="0">
     <path>br.unicamp.fee.dca.join.applications.Gags.gajit</path>
     <code>ChromItem.class</code>
 </task>
<task id="aux4" delay="0">
     <path>br.unicamp.fee.dca.join.applications.Gags.gajit</path>
     <code>FixView.class</code>
 </task>
<task id="aux5" delay="0">
     <path>br.unicamp.fee.dca.join.applications.Gags.gajit</path>
     <code>View.class</code>
 </task>
<task id="aux6" delay="0">
     <path>br.unicamp.fee.dca.join.applications.Gags.gajit</path>
     <code>DupOp.class</code>
 </task>
<task id="aux7" delay="0">
     <path>br.unicamp.fee.dca.join.applications.Gags.gajit</path>
     <code>Population.class</code>
 </task>
<task id="aux8" delay="0">
<path>br.unicamp.fee.dca.join.applications.Gags.util</path>
     <code>ListIterator.class</code>
 </task>
<task id="aux9" delay="0">
     <path>br.unicamp.fee.dca.join.applications.Gags.util</path>
     <code>ExtendedBitSet.class</code>
 </task>
<task id="aux10" delay="0">
     <path>br.unicamp.fee.dca.join.applications.Gags.util</path>
     <code>RandomInt.class</code>
 </task>
<task id="aux11" delay="0">
     <path>br.unicamp.fee.dca.join.applications.Gags.util</path>
     <code>SimpleList.class</code>
 </task>
<task id="aux12" delay="0">
```

```
          <path>br.unicamp.fee.dca.join.applications.Gags.util</path>
          <code>SortedList.class</code>
      </task>
     <task id="aux13" delay="0">
          <path>br.unicamp.fee.dca.join.applications.Gags.util</path>
          <code>Comparable.class</code>
      </task>
</assignment>

<datalink>
  <block type="loop" max_iter="500"  >
          <batch id="B0">
               <task_id>init</task_id>
               <multi> 1 </multi>
               <input></input>
          </batch>
           <batch id= "B1">
               <task_id>eval</task_id>
               <multi>1</multi>
               <input>B0</input>
          </batch>
  </block>
  <block type="sequence">
      <batch id="B2">
            <task_id>result</task_id>
          <multi> 1 </multi>
          <input></input>
      </batch>
  </block>
</datalink>
</pas>
```

Apesar da seção *datalink* especificar apenas o relacionamento entre os lotes de tarefa B1, B2 e B3, todas as classes auxiliares utilizadas por suas tarefas devem ser especificadas na seção *assignment*.

5.4.2 Implementação de algoritmo com número de iterações variável

A ferramenta PTP apresentada brevemente na Seção 2.9 é um outro exemplo de aplicação que poderia tirar proveito de uma plataforma paralela. *Phylogenetic Tool Project* (PTP) permite encontrar soluções quase ótimas em espaços de estados com elevado número de candidatos ao longo de um processo de busca [PRA03]. Porém, a característica iterativa de PTP apresenta uma necessidade para a qual a maioria dos sistemas paralelos não está preparada, isto é, o processamento paralelo deve ser repetido um número de vezes que é dependente da convergência do algoritmo para um resultado esperado.

A plataforma JoiN também não estava preparada para este tipo de aplicação e, para permitir a implementação de PTP, seriam necessárias várias adaptações de forma que a aplicação pudesse determinar quando as repetições do processamento paralelo deveriam terminar [PRA03]. Em vez de PTP ser uma aplicação totalmente contida em JoiN, seria necessário que parte dela fosse executada fora do sistema. A plataforma JoiN teria que ser alterada para este caso específico e ficar responsável pelo processamento paralelo de uma

população, enquanto que o controle das gerações e a decisão de término da aplicação seriam feitos por um módulo externo.

A plataforma Join foi desenvolvida para ser auto-contida e eficiente. Por isso uma aplicação JoiN não pode ser composta de uma parte executando dentro e outra fora da plataforma. As alterações que precisariam ser feitas no componente Jack de JoiN para permitir que uma aplicação fosse iniciada por código externo gerariam uma vulnerabilidade grave na plataforma.

Apesar destas modificações terem sido testadas na prática e demonstrado como o PTP se beneficiava do paralelismo oferecido pela plataforma JoiN, ficou claro que esta era uma solução específica para aquele problema e de difícil aplicação em outras situações. A Fig. 22 exemplifica a arquitetura do PTP com o JoiN modificado, mostrando que um módulo externo (originalmente escrito em Visual Basic) era responsável por gerar um arquivo em XML contendo os dados a serem processados em paralelo (candidatos). Além de criar estes dados, o módulo externo precisava se comunicar por meio de uma interface especial com a plataforma para poder controlar o início de cada nova iteração do processamento.

A utilização da linguagem XPWSL tornou possível a implementação integral desse tipo de aplicação na plataforma JoiN, já que XPWSL oferece todo o suporte necessário para o controle, em tempo de execução, do número de iterações de acordo com os resultados obtidos nos cálculos.

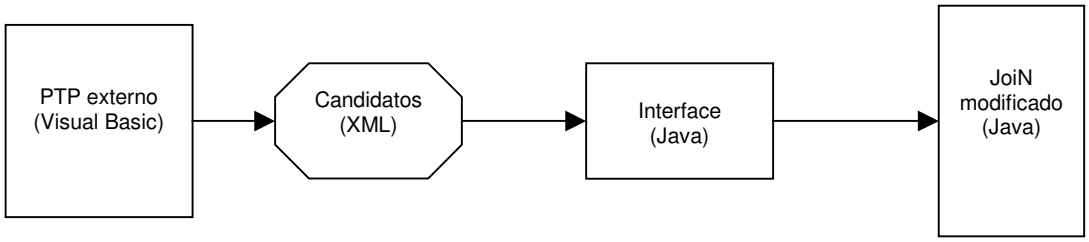

Figura 22. Integração JoiN - PTP

A especificação completa da aplicação PTP em XPWSL é mostrada a seguir.

```
<?xml version="1.0"?>
<pas xmlns="http://www.dca.fee.unicamp.br/projects/join"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://www.dca.fee.unicamp.br/projects/join
PAS.xsd">

<header>
     <name>PTP</name>
     <description>Aplicacao PTP - LOOP</description>
</header>

<assignment>
     <task id="dispatcher" delay="10">
          <path>br.unicamp.fee.dca.join.applications.PTP</path>
```

```
     <code>PTP_Dispatcher.class</code>
</task>
<task id="compute" delay="100">
     <path>br.unicamp.fee.dca.join.applications.PTP</path>
     <code>PTP_DataCompute.class</code>
</task>
<task id="gather" delay="10">
     <path>br.unicamp.fee.dca.join.applications.PTP</path>
     <code>PTP_Gather.class</code>
</task>
<task id="aux1" delay="0">
     <path>br.unicamp.fee.dca.join.applications.PTP</path>
     <code>Population.class</code>
</task>
<task id="aux2" delay="0">
     <path>br.unicamp.fee.dca.join.applications.PTP</path>
     <code>Candidate.class</code>
</task>
<task id="aux3" delay="0">
     <path>br.unicamp.fee.dca.join.applications.PTP</path>
     <code>AdvMatrix.class</code>
 </task>
<task id="aux4" delay="0">
     <path>br.unicamp.fee.dca.join.applications.PTP</path>
     <code>BasicParser.class</code>
</task>
<task id="aux5" delay="0">
     <path>br.unicamp.fee.dca.join.applications.PTP</path>
     <code>CandidateDB.class</code>
</task>
<task id="aux6" delay="0">
     <path>br.unicamp.fee.dca.join.applications.PTP</path>
     <code>CandidateDistMatrix.class</code>
</task>
<task id="aux7" delay="0">
     <path>br.unicamp.fee.dca.join.applications.PTP</path>
     <code>CandidateFactory.class</code>
</task>
<task id="aux8" delay="0">
     <path>br.unicamp.fee.dca.join.applications.PTP</path>
     <code>CandidateMLK1.class</code>
</task>
<task id="aux9" delay="0">
     <path>br.unicamp.fee.dca.join.applications.PTP</path>
     <code>CandidateMLK2.class</code>
</task>
<task id="aux10" delay="0">
     <path>br.unicamp.fee.dca.join.applications.PTP</path>
     <code>Log.class</code>
</task>
<task id="aux11" delay="0">
     <path>br.unicamp.fee.dca.join.applications.PTP</path>
     <code>PopulationDB.class</code>
</task>
<task id="aux12" delay="0">
     <path>br.unicamp.fee.dca.join.applications.PTP</path>
     <code>PopulHandler.class</code>
```

```
    </task>
    <task id="aux13" delay="0">
         <path>br.unicamp.fee.dca.join.applications.PTP</path>
         <code>PTP_Constants.class</code>
    </task>
    <task id="aux14" delay="0">
         <path>br.unicamp.fee.dca.join.applications.PTP</path>
         <code>WorkControl.class</code>
    </task>
    <task id="aux15" delay="0">
         <path>br.unicamp.fee.dca.join.applications.PTP</path>
         <code>WorkerLH.class</code>
    </task>
    <task id="aux16" delay="0">
         <path>br.unicamp.fee.dca.join.applications.PTP</path>
         <code>MailBox.class</code>
    </task>
    <control id="ctrl" delay="1">
         <path>br.unicamp.fee.dca.join.applications.PTP</path>
              <code>Ptpcontrol.class</code>
    </control>
  </assignment>

  <datalink>
  <block type="loop" control="ctrl" max_iter="10">
      <batch id="init">
           <task_id>dispatcher</task_id>
           <multi> 1 </multi>
           <input></input>
      </batch>
       <batch id="eval">
           <task_id>compute</task_id>
           <multi>6</multi>
           <input>init</input>
      </batch>
      <batch id="analysis">
          <task_id>gather</task_id>
          <multi>1</multi>
          <input>eval</input>
      </batch>
  </block>

 </datalink>

</pas>
```

5.4.3 Impacto de XPWSL no desempenho de JoiN

Para analisar o desempenho do novo modelo de especificação em relação ao modelo original do JoiN, foi executada uma mesma aplicação nas duas versões do sistema usando especificação em PASL e em XPWSL.

Como o suporte às novas estruturas foi todo implementado no módulo coordenador e como o objetivo dos testes é analisar o impacto das novas estruturas de controle, foram utilizados apenas lotes com cardinalidade unitária de forma que todo o processamento fosse

feito no coordenador. Tanto na linguagem PASL quanto na XPWSL os lotes unitários foram executados no coordenador e, como os tipos de tarefas realizadas são exatamente os mesmos, a diferença dos tempos de execução representa o impacto da XPWSL no desempenho.

Foi utilizado um único tipo de tarefa na qual há um *sleep* de 10 segundos. Para cada um dos testes a aplicação foi executada 5 vezes seguidas. Em seguida o sistema foi reiniciado e os testes repetidos, sendo totalizadas 10 amostras para cada modelo. Procurou-se utilizar o mesmo ambiente em relação à carga da máquina para cada cenário e para isso, a execução de cada série foi alternada entre os cenários.

A implementação e os arquivos de especificação de cada um dos testes realizados encontra-se no Apêndice D.

Os testes foram executados com a seguinte configuração:
- computador: Pentium III com 800 MHz e 256MB de RAM;
- sistema operacional: Windows 98.
- versão Java: 1.4.1_02

Bloco seqüencial

Para este teste utilizou-se uma aplicação seqüencial com 3 lotes de tarefas, já que uma aplicação básica possui:
- preparação dos dados
- processamento
- análise dos resultados

A especificação da seção *datalink* em PASL e XPWSL é mostrada a seguir.

PASL

```
//Datalink section
B0 = T0(1) << NULL;
B1 = T0(1) << B0;
B2 = T0(1) << B1;
```

XPWSL

```
<datalink>
  <block type="sequence" >
       <batch id="lote1">
            <task_id>tid</task_id>
             <multi>1</multi>
             <input></input>
       </batch>
       <batch id="lote2">
            <task_id>tid</task_id>
            <multi>1</multi>
            <input>lote1</input>
       </batch>
       <batch id="lote3">
            <task_id>tid</task_id>
```

```
            <multi>1</multi>
            <input>lote2</input>
        </batch>
    </block>
</datalink>
```

Conforme foi explicado anteriormente cada aplicação foi executada 10 vezes e obteve-se um valor médio:

Aplicação de teste em PASL: seq3pasl.jas

```
INFO: It took 30150 ms for this application to execute.
INFO: It took 30150 ms for this application to execute.
INFO: It took 30160 ms for this application to execute.
INFO: It took 30150 ms for this application to execute.
INFO: It took 30150 ms for this application to execute.
INFO: It took 30100 ms for this application to execute.
INFO: It took 30160 ms for this application to execute.
INFO: It took 30160 ms for this application to execute.
INFO: It took 30150 ms for this application to execute.
INFO: It took 30090 ms for this application to execute.
```

min.: 30090 ms max.: 30160 ms média: 30142 ms

Aplicação de teste em XPWSL: seq3.XML

```
INFO: It took 30160 ms for this application to execute.
INFO: It took 30150 ms for this application to execute.
INFO: It took 30150 ms for this application to execute.
INFO: It took 30150 ms for this application to execute.
INFO: It took 30100 ms for this application to execute.
INFO: It took 30150 ms for this application to execute.
INFO: It took 30160 ms for this application to execute.
INFO: It took 30150 ms for this application to execute.
INFO: It took 30160 ms for this application to execute.
INFO: It took 30160 ms for this application to execute.
```

min.: 30100 ms max.: 30160 ms média: 30149 ms

Considerando que em cada tarefa foi colocado apenas um *sleep* de 10 segundos; então, teoricamente o tempo total gasto com a execução das tarefas seria de 30 segundos. A diferença entre o tempo obtido e o teórico pode ser considerada como o tempo gasto com o controle feito por JoiN.

Observa-se que a diferença entre o tempo médio é mínima (7ms). Além disso existe uma grande variação entre as medições não sendo possível dizer com certeza que o tempo gasto com a XPWSL é menor que o PASL. De qualquer forma, é possível afirmar que não houve degradação significativa de desempenho com a implementação da XPWSL.

Bloco de repetição com um número fixo de iterações

A vantagem da utilização do bloco de repetição fica mais clara com o exemplo do arquivo de especificação XPWSL.

PASL

```
//Datalink section
B0 = T0(1) << NULL;
B1 = T0(1) << B0;
```

```
B2=  T0(1) << B1;
B3 = T0(1) << B2;
B4=  T0(1) << B3;
B5 = T0(1) << B4;
B6=  T0(1) << B5;
B7 = T0(1) << B6;
B8=  T0(1) << B7;
B9 = T0(1) << B8;
```

XPWSL

```
<datalink>
  <block type="loop" max_iter="10"  >
       <batch id="lote">
            <task_id>teste</task_id>
            <multi> 1 </multi>
            <input></input>
       </batch>
  </block>
</datalink>
```

Observa-se que ao contrário do PASL, onde o mesmo lote de tarefas é repetido 10 vezes, com esta estrutura o lote é definido apenas uma vez. Esta vantagem se mostra de maneira ainda mais acentuada quando se trata de laços com um número elevado de iterações.

Resultados:

Aplicação de teste em PASL: seq10pasl.jas

```
INFO: It took 100630 ms for this application to execute.
INFO: It took 100510 ms for this application to execute.
INFO: It took 100570 ms for this application to execute.
INFO: It took 100730 ms for this application to execute.
INFO: It took 100570 ms for this application to execute.
INFO: It took 100730 ms for this application to execute.
INFO: It took 100620 ms for this application to execute.
INFO: It took 100510 ms for this application to execute.
INFO: It took 100510 ms for this application to execute.
INFO: It took 100510 ms for this application to execute.
```

min.: 100510 ms max.: 100730 ms média: 100589 ms

Aplicação de teste em XPWSL: Loop.XML

```
INFO: It took 100630 ms for this application to execute.
INFO: It took 100540 ms for this application to execute.
INFO: It took 100790 ms for this application to execute.
INFO: It took 100680 ms for this application to execute.
INFO: It took 100570 ms for this application to execute.
INFO: It took 100460 ms for this application to execute.
INFO: It took 100680 ms for this application to execute.
INFO: It took 100790 ms for this application to execute.
INFO: It took 100570 ms for this application to execute.
INFO: It took 100460 ms for this application to execute.
```

min.: 100460 ms max: 100790 ms média: 100617 ms.

Era de se esperar que o tempo em PASL fosse menor que o em XPWSL. Porém observa-se que o aumento em XPWSL é mínimo, principalmente se for comparado ao tempo total de processamento da aplicação. Nesse exemplo cada uma das tarefas gastou apenas 10 segundos; porém na prática, o tempo de computação de uma tarefa costuma ser bem maior que este para que o tempo de comunicação seja compensado e para que valha a pena executar a aplicação no sistema paralelo.

O aumento de 28ms no tempo médio em XPWSL parece estar mais relacionado com a carga de processamento da máquina no momento de execução dos testes do que um custo real, já que houve uma grande variação com os valores medidos. Da mesma forma que na estrutura seqüencial pode-se dizer que esta estrutura é viável e não degrada o sistema de forma significativa.

Bloco de repetição condicional e o switch/case

A mesma aplicação utilizada na análise anterior foi modificada para executar um *loop* condicional. Nesta análise de custo considerou-se o pior caso onde a condição de saída foi o número máximo de iterações de forma a facilitar a comparação com o laço com um número fixo de iterações.

```
<datalink>
     <block type="loop" control="controle" max_iter="10"  >
          <batch id="lote">
               <task_id>teste</task_id>
               <multi>1</multi>
               <input></input>
          </batch>
     </block>
</datalink>
```

Resultados:

```
Aplicação de teste (laço fixo) em XPWSL
   min.: 100460 ms    max: 100790 ms    média: 100617 ms.

Aplicação de teste (laço condicional) em XPWSL: LoopCond.xml
   INFO: It took 100620 ms for this application to execute.
   INFO: It took 100570 ms for this application to execute.
   INFO: It took 100620 ms for this application to execute.
   INFO: It took 100790 ms for this application to execute.
   INFO: It took 100620 ms for this application to execute.
   INFO: It took 100680 ms for this application to execute.
   INFO: It took 100620 ms for this application to execute.
   INFO: It took 100680 ms for this application to execute.
   INFO: It took 100730 ms for this application to execute.
   INFO: It took 100630 ms for this application to execute.
   min.: 100570 ms    max: 100790 ms    média: 100656 ms
```

A diferença entre o bloco de repetição com um número fixo de iterações e o condicional é que este último realiza o carregamento dinâmico e a execução da classe de controle para verificar se a condição de saída do laço foi satisfeita ou não. A necessidade de

carregar a classe de controle a cada iteração depende da política de *cache* utilizada, normalmente a classe é carregada na primeira execução e nas iterações seguintes ela já esta carregada na *cache*. A diferença entre o tempo médio de execução foi de 39 ms, o que daria aproximadamente um custo por iteração de 3,9ms para carregar e executar a classe AppLoopControl. Neste exemplo, o carregamento da classe de controle afetou muito pouco o desempenho já que esta classe não realiza praticamente nenhum processamento. Mas vale lembrar que o custo total na prática depende da complexidade do critério de parada implementado nessa classe. O programador da aplicação deve estar ciente de que não é recomendável fazer um processamento muito complexo nessa classe de controle. O número de iterações nesta aplicação foi de apenas 10; porém em aplicações reais o número de iterações provavelmente será bem maior (da ordem de centenas) e a probabilidade de se sair do laço a partir da satisfação do critério deve ser muito maior que a saída pelo limite de iterações.

No caso do desvio condicional, o custo é o mesmo que o custo de um único passo do *loop* condicional, porque a classe de controle é carregada uma única vez no momento de decisão. Como foi observado no teste anterior, o impacto dessa classe no desempenho vai depender da implementação do conteúdo da classe.

Com estes testes foi comprovado que o impacto do novo modelo de aplicação no desempenho da plataforma JoiN é mínimo, uma vez que as novas estruturas de controle de fluxo praticamente não realizam computação. A operação de maior custo computacional presente no novo modelo é o carregamento dinâmico das classes de controle utilizadas nas estruturas de repetição e de desvio condicional . Mas mesmo nestes casos o impacto no desempenho final é desprezível, já que a maior parte do tempo é gasto na execução das tarefas paralelas.

5.5 Conclusões

Neste capítulo foi apresentada a implementação da linguagem XPWSL na plataforma paralela JoiN. Para suportar a nova linguagem foram necessárias modificações na plataforma e a maior mudança foi a adição de novas classes para o suporte ao Bloco de Execução e estruturas de controle. Além disso foi possível reutilizar praticamente todo o código que JoiN já utilizava para o controle do DAG e escalonamento de tarefas.

Para validar a linguagem foram realizados diversos testes isolados e também aplicações completas, como no caso das aplicações PTP e outra com a utilização do pacote GAJIT. No caso do PTP foi feita uma comparação com a implementação realizada com a linguagem PASL, em que foi necessária uma customização do sistema JoiN para permitir um controle de iterações externo. Com XPWSL foi possível implementar a aplicação completa na plataforma JoiN sem a necessidade de adaptações ou de interações com outros sistemas para o controle da aplicação. Além de permitir a implementação de aplicações que não eram viáveis no sistema JoiN usando PASL, a utilização da linguagem XPWSL não causou nenhuma degradação significativa no desempenho do sistema.

A implementação do novo modelo de especificação de aplicações paralelas em JoiN pode ser considerada bem sucedida já que passou a possibilitar a execução de uma gama maior de aplicações sem acarretar em um custo de desempenho perceptível. Além disso, o uso do XML na implementação oferece um novo grau de liberdade e flexibilidade não existentes no modelo anterior e que poderá permitir a adição de novas funcionalidades no futuro sem maiores esforços.

Apesar da implementação ter sido feita na plataforma JoiN, entendemos que a linguagem XPWSL pode ser implementado em outros sistemas paralelos, desde que contemplem o modelo de sistema paralelo adotado como referência neste trabalho.

Capítulo 6 Conclusões e trabalhos futuros

Neste trabalho foi proposto um novo modelo de especificação de fluxo de execução de tarefas em sistemas paralelos e também uma linguagem para este modelo.

Foram analisados diversos sistemas paralelos existentes com enfoque no fluxo de execução das aplicações de cada um. A maioria dos sistemas permite apenas a execução de aplicações *bag of tasks* ou então baseadas no modelo de dependências DAG. Estes sistemas não são adequados para aplicações que utilizam algoritmos iterativos. Devido às limitações dos modelos de especificação de fluxo dos sistemas existentes, recentemente, diversos trabalhos têm sido realizados com o objetivo de criar novas linguagens de especificação de fluxo para aplicações em *grid*, mas muitas dessas linguagens não suportam iterações ou são complexas.

O foco desse trabalho foi o desenvolvimento de um novo modelo de especificação de fluxo de tarefas paralelas baseado em Blocos de Execução, que permite especificar a maioria das aplicações paralelas usando apenas 4 estruturas de controle. Também foi proposta uma nova linguagem de especificação para este modelo que é: flexível (admite a combinação de estruturas de controle) e extensível (a criação de novos atributos é simples). A linguagem proposta, XPWSL, contempla, além das tradicionais estruturas de controle DAG, novas estruturas de controle como repetição com um número fixo ou não de iterações e o desvio condicional, sendo possível a combinação, em seqüência, de diversas estruturas de controle na mesma aplicação.

A proposta de XPWSL não é cobrir todas as estruturas de controle possíveis, mas definir um conjunto de estruturas que possibilitem a implementação de grande parte das aplicações que poderiam se beneficiar com a computação paralela em *grids*. Além de ser uma linguagem flexível, a implementação do suporte à esta linguagem em um sistema paralelo é simples, como foi mostrado com a implementação na plataforma JoiN. Apesar da implementação ter sido feita nesta plataforma específica, acreditamos que ela também poderia ser implementada em outros sistemas que sejam compatíveis com o modelo de sistema paralelo adotado como referência neste trabalho.

Também foi comprovado que o controle adicionado pelas novas estruturas de controle tem um impacto mínimo no desempenho do sistema JoiN. O tempo gasto com o controle do fluxo de tarefas acaba sendo desprezível, já que em um sistema paralelo é esperado que o tempo de processamento gasto com as tarefas seja grande o suficiente a ponto de justificar o tempo gasto com a distribuição e coleta de dados.

Ainda existem vários pontos que poderiam ser melhorados em XPWSL:

- refinar o modelo XPWSL para permitir a definição de novos atributos como as restrições de execução, que poderiam ser requisitos mínimos de quantidade de memória e velocidade de CPU para cada tarefa. Com a utilização de restrições de execução será possível realizar um escalonamento mais eficiente das tarefas para os processadores, reduzindo o tempo de execução das aplicações;
- desenvolver um componente gráfico para definição de aplicações em XPWSL, permitindo a visualização gráfica das estruturas de controle e relacionamento das tarefas, já que uma das vantagens de se ter escolhido o XML para a definição da linguagem XPWSL é facilitar a geração automática do arquivo de especificação;
- permitir combinações mais complexas de estruturas de controle; apesar de qualquer uma das estruturas de controle poder possuir um conjunto de lotes com um relacionamento de dependências (DAG) e uma estrutura seqüencial poder conter qualquer uma das outras estruturas suportadas, uma estrutura de controle iterativa não pode conter dentro do laço uma estrutura de desvio condicional e vice-versa. A linguagem poderia ser expandida para permitir que o processamento realizado em uma estrutura de controle possa conter qualquer uma das outras estruturas de controle;
- analisar novas estruturas de controle para tratamentos de erro e outros eventos.

Referências bibliográficas

[ABR00] D. Abramson, J. Giddy, L. Kotler, "High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid?", Proceedings of the 14th International Symposium on Parallel and Distributed Processing (IPDPS), 2000

[ABR95] D. Abramson, R. Sosic, J. Giddy, B. Hall, "Nimrod: A Tool for Performing Parametised Simulations using Distributed Workstations", Proceedings of the 4th IEEE Symposium on High Performance Distributed Computing, pp. 112-121, Virginia, Agosto 1995.

[AND02] D. Anderson, J. Cobb, E. Korpela, M. Lebofsky and D. Werthimer, "SETI@home: An experiment in public resource computing", Communications of the ACM, Vol.45 No.11, pp 56-61, novembro/2002 (disponível em SETI@Home Project home page: http://www.seti.org, último acesso em Junho/2005)

[AND03] D. Anderson, "Public Computing: Reconnecting People to Science", apresentado na conferência "Shared Knowledge and the Web", Residencia de Estudiantes, Madrid, Spain, Novembro/2003

[AND05a] D. Anderson, "BOINC: A System for Public-Resource Computing and Storage", disponível em BOINC Home Page - http://boinc.berkeley.edu (último accesso em Junho/2005)

[AND05b] T. Andrews, et. Al., "Business Process Execution Language for Web Services Specification, version 1.1. IBM, Microsoft, BEA, SAP and Siebel Systems, Maio/2003 (disponível em http://www-128.ibm.com/developerworks/library/ws-bpel/ , último acesso em Julho/2005)

[ANJ05] A. Anjomshoaa, F. Brisard, M. Drescher, D. Fellows, A. Ly, S. McGough, D. Pulsipher, A. Savva, "Job Submission Description Language (JSDL) Specification version 1.0", Junho/2005 (disponível em http://forge.gridforum.org/projects/jsdl-wg último acesso em Julho/2005)

[BOI05] BOINC Synergy projects - http://www.boincsynergy.com (último accesso em Junho/2005)

[BRA05] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau, "Extensible Markup Language (XML) 1.0 (Third Edition)", W3C Recommendation, disponível em http://www.w3.org/TR/REC-xml/ (último acesso em Agosto/2005)

[DTD05] DTD Tutorial, diponível em http://www.w3schools.com/dtd/default.asp (último acesso em Agosto/2005)

[FOS02] I. Foster, “Chimera: A virtual data system for representing, querying and automating data derivation”, Proceedings of the 14th Conference on Scientific and Statistical Data Based Management (SSDBM '02), pp. 37-46, Edinburgh, Scotland, 2002

[CIR03] W. Cirne, D. Paranhos, L. Costa, E. Santos-Neto, F. Brasileiro, J. Sauvé, F. A. B. da Silva, C. O. Barros, C. Silveira, “Running Bag-of-Tasks Applications on Computational Grids: The MyGrid Approach”, The 2003 International Conference on Parallel Processing (ICPP-03), Kaohsiung, Taiwan, ROC, 2003

[FAH05] T. Fahringer, J. Qin, S. Hainzer, “Specification of Grid Workflow Applications with AGWL: An Abstract Workflow Language”, IEEE International Symposium on Cluster Computing and the Grid 2005 (CCGrid 2005), Cardiff, UK, Maio/2005.

[FER05] Condor User Tutorial, Fermilab, Janeiro/2005, disponível em http://www.cs.wisc.edu/condor (último acesso em Julho/2005)

[FLY96] M. J. Flynn, “Very High-Speed Computing Systems”, Proceeding of the IEEE, 54(12), pp. 1901 – 1909, Dezembro/1996

[FOS95] I. Foster, “Designing and Building Parallel Programs”, ISBN: 0201575949 , 1995, (também disponivel em www-unix.mcs.anl.gov/dbpp, último acesso em Julho/2005)

[FOS99] I. Foster and C. Kesselman, "Computational Grids" do livro "The Grid : Blueprint for a New Computing Infrastructure", Morgan Kaufmann, 1999. ISBN 1-55860-475-8

[GAJ05] "GAJIT - A Simple Java Genetic Algorithms Package" home page http://www.micropraxis.com/gajit/ (último acesso em Julho/2005)

[GLO05a] The Globus Alliance, http://www.globus.org/ (último acesso em Julho/2005)

[GLO05b] The Grid Resource Allocation and Management, http://www-unix.globus.org/toolkit/docs/3.2/gram/ws/ (último acesso em Julho/2005)

[HEN99] M. A. A. Henriques, "A proposal for java based massively parallel processing on the web", Proceedings of the FirstAnnual Workshop on Java for High-Performance Computing, ACM International Conference on Supercomputing, pp. 59–66, Rhodes, Grécia, 1999

[HER04] R. Herai and M. A. A. Henriques, "Ferramentas de Modelagem para a Predição de Performance Analítica em uma Plataforma de Processamento Paralelo", V Workshop em Sistemas Computacionais de Alto Desempenho - WSCAD 2004, Vol. 1, pp. 3-10, Foz do Iguaçu, Brasil, Outubro/2004.

[HEY00] E. Heymann, M. A. Senar, E. Lugue, M.Livny, "Adaptive Scheduling for Master-Worker Applications on the Computational Grid", Proceedings of the First IEEE/ACM International Workshop on Grid Computing, pp. 214-227, 2000

[IBM05] IBM LoadLeveler for AIX 5L: "Using and Administering", http://publib.boulder.ibm.com/doc_link/en_US/a_doc_lib/sp34/LoadL/am2ugmst.html, (último acesso em Junho/2005)

[KOS04] M. Kosiedowski, K. Kurowski, C. Mazurek, J. Nabrzyski, J. Pukacki, "Work-flow applications in PROGRESS and GridLab environments", Global Grid Forum 2004 Special Issue of Concurrency and Computation:Practice and Experience, 2004

[LUC02] F. de O.Lucchese, "Um Mecanismo para Distribuição de Carga em Ambientes Virtuais de Computação Maciçamente Paralela", dissertação de mestrado, Faculdade de Engenharia Elétrica e de Computação, Unicamp, 2002

[NEM02] Z. Nemeth and V. Sunderam, "A comparison of conventional distributed computing environments and computational grids", Proceedings of the International Conference on Computational Science (ICCS, 2002), pp. 729-738, Amsterdam, Netherlands, Abril/2002

[OPE05] OpenBPS home page, http://www.openpbs.org, último acesso em Junho/2005

[OUR05] OurGrid 3.0 User Manual, http://www.ourgrid.org/, último acesso em Julho/2005

[PRA03] O. Prado, F. J. Von Zuben, M. A. A. Henriques. "PTP and JoiN as Software Packages fo Phylogenetic Inference", 1st International Conference on Bioinformatics and Computational Biology, Ribeirão Preto, SP, Brasil, 2003.

[RAM05] R. Ramanujam, "A foundation for parallel programming", disponível em www.imsc.res.in/~parapp/jam.pdf (último acesso em Julho/2005)

[STA04] M. Stanton, "Computação Distribuída Pública", artigo publicado no jornal "O Estado de São Paulo" em 28/nov/2004

[THA03] D. Thain, T. Tannenbaum, M. Livny, "Condor and the Grid" do livro "Grid Computing – Making the Global Infrastructure a Reality" ISBN: 0-470-85319-0, 2003

[THO05] H. S. Thompson, D. Beech, M.Maloney, N. Mendelsohn, "XML Schema Part 1:

Structures Second Edition", W3C Recommendation, disponível em http://www.w3.org/TR/xmlschema-1/ (último acesso em Julho/2005)

[VAR04] P. K. Vargas, I. C. Dutra, C. F. R. Geyer, " Application partitioning and hierarchical management in grid environments", em Proceedings of the 1st international doctoral symposium on Middleware, pp 314-318, Toronto, Canadá, Outubro/2004

[YER05] E. J. H. Yero, F. O. Lucchese, F. S. Sambatti and M. A. A. Henriques., "Join: The Implementation of a Java-based Massively Parallel Grid", Future Generation Computing Systems, Elsevier, Vol. 21, pp. 791-810, 2005

[YU05] J. Yu and R. Buyya, "A Taxonomy of Workflow Management Systems for Grid Computing", Technical Report, GRIDS-TR-2005-1, Grid Computing and Distributed Systems Laboratory, University of Melbourne, Australia, Março, 2005.

Apêndice A Descrição detalhada dos sistemas paralelos analisados

A.1 O projeto Condor

O projeto Condor surgiu a partir de um trabalho de doutorado em processamento cooperativo em 1983 na Universidade de Wisconsin e, ao contrário dos sistemas centralizados dominantes na época, permitia que cada participante contribuísse com apenas o que desejasse.

Ao longo dos anos, Condor foi atingindo maturidade e flexibilidade, sendo adaptado aos novos ambientes de programação, como: PVM, MPI e Java.

Atualmente, Condor também engloba outras atividades e projetos como:
- aproveitamento do poder dos recursos dedicados e oportunos (Condor);
- serviços de gerenciamento de *jobs* para aplicações *grid* (Condor-G, DaPSched);
- serviços de gerenciamento de recursos *grid* (Condor, Glide-In, NeST);
- busca de recursos, monitoramento e gerenciamento (ClassAds, Hawkeye);
- ambientes de resolução de problemas (MW, DAGMan);
- tecnologia distribuída de I/O (Bypass, PFS, Kangaroo, NeST);
- gerenciamento de fluxo de *jobs* (DAGMan, Condor, Hawk);
- *packaging* e Integração(NMI, VDT).

Devido à importância do Condor em *grids* computacionais o livro: "*Grid Computing"* dedica um capítulo exclusivo para o sistema [THA03]. As informações do sistema e exemplos utilizados neste trabalho foram obtidos deste livro e do tutorial do Condor [FER05] além de artigos relacionados.

A.1.1 Condor

O Condor é um sistema que provê mecanismos de gerenciamento de *jobs* e recursos, política de escalonamento e monitoramento de recursos e prioridades. O usuário submete seus *jobs* ao Condor, e ele escolhe "quando" e "onde" o *job* será executado.

Além das funcionalidades de um sistema *batch queuing*, o Condor inclui outros mecanismos importantes como:
- classAds: um *framework* flexível para associar as requisições de recursos feitas pelos *jobs* com os recursos disponíveis;

- *job checkpoint* e migração: mecanismo de tolerância a falhas que permite gravar *checkpoints* e continuar a execução a partir deles. Também é possível a migração do *job* para outra máquina e continuar a execução a partir do *checkpoint*;
- chamadas de sistemas remotas: durante a execução de *jobs* em máquinas remotas, o Condor permite preservar o ambiente local utilizando chamadas remotas com o redirecionamento de todas as operações de entrada/saída para a máquina que fez a submissão do *job*.

O *kernel* do Condor, exemplificado na Fig. 23, consiste em:

- usuário submete um *job* para um agente do Condor;
- os *jobs* são armazenados em uma estrutura persistente enquanto o agente procura recursos para executá-los;
- agentes e recursos utilizam um *matchmaker* que analisa a compatibilidade entre eles;
- o agente contata o recurso e verifica se ele ainda é compatível com o *job* antes de iniciar a execução;
- para executar o *job*, os dois lados iniciam um novo processo: no agente o *shadow* é responsável por prover todas as informações para executar o *job* enquanto que no recurso, o *sandbox* é responsável por criar um ambiente seguro para a execução do *job*.

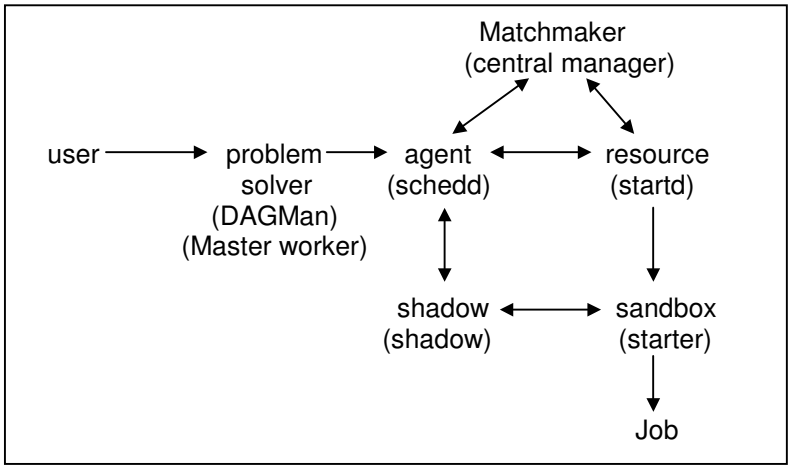

Figura 23. *Kernel* do Condor, com os principais processos e o relacionamento entre eles

A.1.1.1 ClassAds

Cada um dos *jobs* e máquinas conectados ao Condor possui atributos para qualificá-los, que são chamados *ClassAds* (*Classified Advertisements*). Estes atributos definem os requisitos e as características de *jobs* e recursos. No escalonador, eles são combinados e ordenados de forma a selecionar o recurso mais apropriado para executar o *job* .

Um ClassAd é um conjunto de atributos na forma:

```
attribute_name = attribute_value
```

Como ele é *schema-free*, os participantes podem definir atributos que ainda não existem e as expressões podem ter três resultados: *true, false* ou *undefined*.

Dois ClassAds "casam" se a expressão de requisitos de ambos são TRUE quando avaliados no contexto do outro. A seguir é mostrado um exemplo de um ClassAdd de *job* e de recurso extraído de [THA03].

```
Job ClassAd
[
  MyType = "Job"
  TargetType = "Machine"
  Requirements =
    ((other.Arch == "INTEL") &&
    (other.OpSys == "WINDOWS") &&
    (other.Disk > my.DiskUsage))
  Rank = (Memory *5000)
  Cmd = "/home/dummy/bin/exe"
  Department = "Comp"
  Owner = "dummy"
  DiskUsage = 6000
]

Machine ClassAd
[
  MyType = "Machine"
  TargetType = "Job"
  Machine= "machine.domain"
  Requirements =
    (LoadAvg <=0.1) &&
    (KeyboardIdle > (15*60))
  Rank = other.Department == self.Department
  Arch = "INTEL"
  OpSys = "WINDOWS"
  DiskUsage = 100000
]
```

Neste exemplo, os requisitos do *job* são que a máquina tem que ser Windows e precisa ter no mínimo 6K de espaço em disco livre. Das máquinas que satisfazem os requisitos, será selecionada aquela que tiver mais memória. Por outro lado, a máquina deseja apenas *jobs* que tenham baixa carga e que sejam executados apenas quando o teclado estiver ocioso por mais de 15 minutos. O *rank* define que o recurso deseja rodar apenas *jobs* do seu departamento.

A.1.1.2 Aplicações Condor

As aplicações Condor são definidas pelo *submit description file*, que é um arquivo texto com informações do *job.* Este arquivo é passado para o comando *condor_submit*, que faz uma varredura e uma verificação de erros gerando um *ClassAdd* com a especificação do *job*. *ClassAdd* e os executáveis são enviados ao *condor_schedd*, que os coloca em uma fila, aguardando o momento de execução.

O *submit description file* contém: informações do *job,* executável, universo, entrada, saída e arquivos de erro, argumentos da linha de comando; variáveis de ambiente; e os requisitos/preferências. Um exemplo desse arquivo é mostrado a seguir.

```
# Simple condor_submit input file
# (Lines beginning with # are comments)
# NOTE: the words on the left side are not
#       case sensitive, but filenames are!
Universe   = vanilla
Executable = my_job
Queue
```

O arquivo *submit file* é passado para o comando *condor_submit*, que faz a validação e verificação de erros, gerando um arquivo *classAdd* com a especificação do *job*. O arquivo de *classAdd* e os executáveis são enviados ao *condor_schedd*, que os coloca em uma fila, aguardando o momento de execução:

```
% condor_submit my_job.submit-file
Submitting job(s).
1 job(s) submitted to cluster 1.
% condor_q
-- Submitter: perdita.cs.wisc.edu : <128.105.165.34:1027> :
 ID   OWNER      SUBMITTED   RUN_TIME ST PRI SIZE CMD
 1.0  frieda      6/16 06:52  0+00:00:00 I 0  0.0 my_job
1 jobs; 1 idle, 0 running, 0 held
```

Pode-se descrever mais de um *job* de uma vez, cada um com informações distintas. Quando um *submit file* descreve mais de um *job* ele é chamado *cluster*. Cada *job* em um cluster é chamado *process* ou simplesmente *proc*.

Condor ainda possui um mecanismo chamado DAGMan (*Directed Acyclic Graph Manager*) que permite definir a dependências entre os *jobs* para que o Condor faça o gerenciamento automático do fluxo.

DAGMan utiliza a estrutura DAG para representar as dependências entre os *jobs*, onde cada *job* é um nó do grafo e os arcos direcionados definem a dependência entre os nós, sendo possível ter qualquer número de nós pais ou filhos, contanto que não haja *loops*. Para definir os nós e dependências utiliza-se um arquivo chamado *dag file*, onde cada nó é especificado em um Condor *submit file*.

Para executar um DAG, utiliza-se o comando *condor_submit_dag* com o arquivo *dag file*:

```
% condor_submit_dag diamond.dag
```

O arquivo diamond.dag possui um formato do tipo:

```
# diamond.dag
Job A a.sub
Job B b.sub
Job C c.sub
Job D d.sub
Parent A Child B C
Parent B C Child D
```

DagMan funciona como um meta-escalonador, gerenciando a submissão dos *jobs* para o Condor *queue* baseando-se nas dependências do DAG.

A.1.2 Condor-G

Desenvolveu-se um outro projeto chamado Condor-G que permite a utilização de recursos distribuídos em diversos domínios, para isso o Condor utiliza o Globus *toolkit* [GLO05a]. Globus define um *toolkit* com serviços de baixo-nível para segurança, comunicação, localização e alocação de recursos, gerenciamento de processos e acesso a dados remotos. Estes serviços são utilizados para implementar serviços de alto-nível como ferramentas e modelos de programação.

Condor-G é uma combinação das tecnologias dos projetos Condor e Globus. Do Globus é aproveitada a utilização de protocolos para comunicação segura inter-domínios e acesso padronizado para vários sistemas remotos de *batch*. Do Condor vem o conceito de submissão e alocação de *jobs*, recuperação de erros e criação de um ambiente amigável de execução.

A Fig. 24 mostra o Condor em um *grid Middleware* com a utilização do Globus.

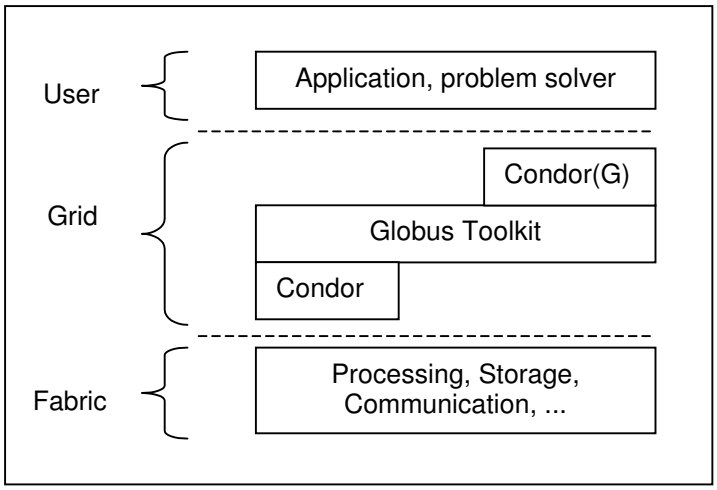

Figura 24. As tecnologias Condor em um *Grid Middleware*

Condor é um projeto que atingiu uma maturidade e flexibilidade que permitem a adaptação com outros sistemas, sem sacrificar os princípios do projeto original. ClassAds é uma ferramenta flexível que permite a definição de características para fazer o casamento dos *jobs* e recursos. Ele também permite que recursos definam restrições para a sua utilização.

A.2 BOINC

BOINC (*Berkeley Open Infrastructure for Network Computing*) é um sistema que facilita a criação e a operação de projetos com a utilização de computação pública distribuída. Ele é um projeto de código-aberto que suporta diversos tipos de aplicações,

com um mecanismo flexível e escalável de distribuição de dados e escalonamento de tarefas.

As características principais do BOINC são:
- *framework* flexível para aplicações: aplicações escritas em outras linguagens (C, C++, Fortran) podem ser executadas no BOINC com poucas ou nenhuma modificação. As aplicações também podem utilizar vários executáveis coordenados por um script;
- segurança: oferece proteção contra vários tipos de ataques utilizando assinatura digital baseada em chave pública;
- tolerância a falhas e múltiplos servidores: cada projeto pode ter seus próprios escalonadores e servidores de dados;
- ferramentas de monitoramento: utiliza um sistema baseado em *Web* para exibir dados estatísticos (carga de CPU, tráfego de rede, dimensão das tabelas), o que simplifica o diagnóstico de problemas;
- código fonte disponível: o BOINC é distribuído sob o GNU *Public License*, mas as aplicações não precisam ter código-aberto;
- suporte para um grande volume de dados: a distribuição e a coleta de dados pode ser distribuída entre vários servidores. Os usuários podem especificar limites de utilização de disco e banda de rede.

Vários projetos foram adaptados ou criados com a utilização do BOINC, como:
- SETI@home: executa o processamento de sinais digitais de dados do radio-telescópio do observatório Arecibo. A versão BOINC do SETI@home, utiliza discos do cliente para armazenar os dados, eliminando a necessidade de um arquivo central;
- Predictor@home: estuda o comportamento da proteína utilizando um programa FORTRAN para mecânica e dinâmica macromolecular;
- Folding@home: estudo do *folding, misfolding, aggregation* e outras enfermidades da proteína, como Alzheimer's, *Mad Cow* (BSE), CJD, ALS, Huntington's e Parkinson's;
- Climateprediction.net: procura quantificar e reduzir as incertezas da predição climática a longo prazo baseado em simulações. Para isso, utiliza um grande número de simulações com variações de cenários e parâmetros do modelo de predição;
- Climate@home: semelhante ao Climateprediction, mas utiliza o modelo do CCSM (*Community Climate System Model*) do NCAR;
- Eistein@home: procura detectar certos tipos de ondas gravitacionais, os *spinning neutron stars*, que podem ser detectadas apenas com a utilização de técnicas de filtragem muito seletivas que requerem alto poder computacional.

Além desses, existem diversos projetos realizados com parceria da Intel, CERN (*European Organization for Nuclear Research*), etc.

A.2.1 Implementação

Um projeto BOINC é um grupo de aplicações de uma única organização. Ele é identificado por uma *master* URL, um documento XHTML, que além de *home page* serve como diretório dos servidores de escalonamento. Os participantes se registram nos projetos que podem ter uma ou mais aplicações.

BOINC utiliza um banco de dados relacional para armazenar todas as informações dos projetos como: descrições das aplicações, plataformas, versões, *workunits*, resultados, contas, times, entre outros. As funções do servidor são executadas por serviços *Web* e processos *daemon*, como os servidores de escalonamento, servidores de dados e interfaces *Web* para monitoramento e submissão de aplicações.

Os participantes precisam fazer o *download* da aplicação cliente para poder fazer parte do projeto. A aplicação pode ser um *screensaver*, um serviço Windows ou ainda um executável UNIX.

O modelo de armazenamento do BOINC é baseado em arquivos: entradas e saídas de aplicações, bibliotecas e executáveis de aplicações. A transferência de arquivos é feita através de HTTP, onde os arquivos são descritos por elementos XML. Os arquivos são relacionados com *workunits*, resultados e aplicações. Estas associações também são representadas em XML.

O projeto é compilado de acordo com a plataforma que será utilizada, por isso o Database precisa manter um conjunto de plataformas. Cada programa é associado às plataformas que são suportadas.

Os componentes de um projeto são exibidos na Fig. 25.

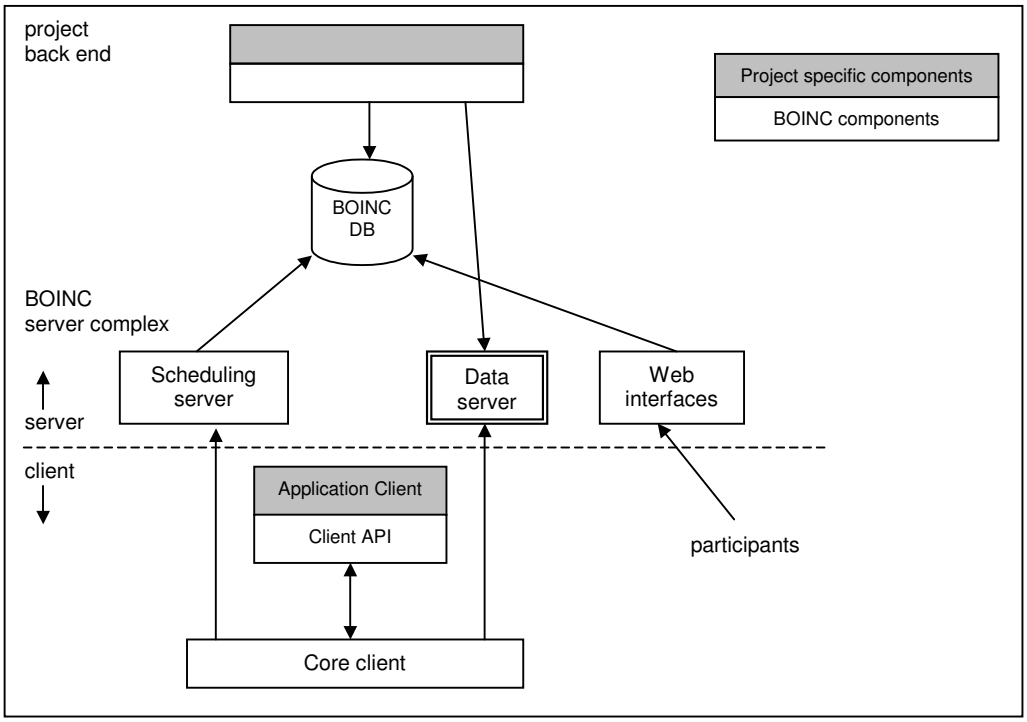

Figura 25. Componentes de um projeto do BOINC

A.1.1.3 Aplicações

Uma aplicação consiste de um programa (que pode ter diferentes versões de acordo com a plataforma) e um conjunto de *workunits* e *results*, que são mantidos em uma tabela do banco de dados do BOINC.

Uma versão de aplicação pode ter vários arquivos como: *scripts* para controle, arquivos para pré e pós-processamento e um programa principal.

O trabalho é organizado na seguinte hierarquia:

$$file \rightarrow workunit \rightarrow result$$

Um *workunit* descreve a computação que será processada e é uma entrada na tabela *workunit*. Ele possui vários atributos como: nome, aplicação, arquivos de entrada, prioridade, estimativas de recursos e atributos de redundância e escalonamento.

O *result* descreve uma instância de computação, que pode ser: não iniciado, em progresso ou completo. A tabela *result* possui diversos atributos como: identificação do *workunit*, arquivos de saída, estado de saída, tempo de CPU utilizado, máquina onde foi executado, entre outros.

A.2.2 Aplicações compostas

Uma aplicação composta consiste de um programa principal e um ou mais programas trabalhadores. O programa principal executa os programas trabalhadores em seqüência e mantém um arquivo de estado com os trabalhadores que já terminaram. O programa principal associa a cada trabalhador um *subrange* da fração realizada de 0 a 1. Por exemplo, se existem 2 trabalhadores com mesmo *runtime*, o primeiro teria um range de 0..0.5 e o segundo de 0.5 a 1.

Um exemplo da lógica do programa principal é mostrado a seguir:

```
BOINC_OPTIONS options;

options.main_program = true;
...
boinc_init_options(options)
read main state file
for each remaining worker program:
  aid.fraction_done_start = x
  aid.fraction_done_end = y
  boinc_write_init_data_file()
  run the app
     wait for the app to finish
          poll
  write main state file
     if last app:
          break
  boinc_parse_init_data_file()   // reads CPU time from app_init.xml file
boinc_finish()
```

onde x e y são os limites apropriados do range da fração realizada.

A lógica do programa trabalhador seria:

```
options.main_program = false;
...
boinc_init_options(options);
...
do work, calling boinc_fraction_done() with values from 0 to 1,
and boinc_time_to_checkpoint(), occasionally
...
boinc_finish(); // this writes final CPU time to app_init.xml file
```

A importância do BOINC é ser uma plataforma de desenvolvimento de aplicações para computação pública distribuída. Antes dele, os projetos de computação pública distribuída eram dedicados para um único propósito, como o SETI@HOME ou o GIMPS. Esta plataforma provê todo o suporte para desenvolvimento, execução e monitoramentos das aplicações.

A.3 Nimrod

Nimrod é uma ferramenta para realizar simulações parametrizáveis em redes distribuídas de *workstations*, sendo muito utilizada para simulação numérica. Nimrod realiza o controle da distribuição dos *jobs* para as máquinas remotas e coleta os resultados.

Para descrever um experimento no Nimrod o usuário define um arquivo *declarative plan* que descreve os parâmetros, seus valores padrão e os comandos necessários para realizar o trabalho. O sistema utiliza essas informações para transportar os arquivos necessários e submeter o trabalho para uma máquina disponível.

O *declarative plan* é composto de 2 seções principais: a seção de parâmetros e a seção de tarefas. No exemplo abaixo, o experimento consiste na variação do parâmetro *thickness* e cada execução recebe um *seed* diferente. O Nimrod gera um *job* para cada combinação única dos valores dos parâmetros, utilizando o produto cruzado de todos os valores, no exemplo abaixo seriam gerados 400 *jobs*:

```
Parameter issed integer range from 100 to 4000 step 100
Parameter thick label "BUC thickness"
Float range from 1.1 to 2.0 step 0.1
parameter jseed integer compute thick*1000;

task nodestart
     Copy ccal.$OS node: ./ccal
     Copy dummy node:.
     Copy ccal.data node:.
     Copy skel.inp node:.
endtask

task main
     Node: substitute skel.inp ccal.inp
     Node: execute ./ccal
```

```
    Copy node:ccal.op ccalout.$jobname
endtask
```

Neste exemplo existem duas tarefas – *main* e *nodestart*. A tarefa *nodestart* é executada uma única vez e faz a transferência dos arquivos para as máquinas remotas. A *main* é executada para cada conjunto de parâmetros. Ela executa uma simulação chamada *ccal*, em um nó e depois copia o arquivo de saída da simulação.

Da mesma forma que no Condor, foi desenvolvida uma versão do Nimrod chamada Nimrod/G que estende o sistema para a utilização em um *grid* computacional dinâmico e heterogêneo, utilizando para isso o Globus *toolkit.*

Nimrod não permite a comunicação entre as tarefas durante a execução das mesmas. Cada uma das tarefas realiza a mesma computação com um conjunto distinto de dados que é obtido através da combinação de parâmetros de entrada. Ele é um sistema para aplicações *bag of tasks*.

A.4 OurGrid

OurGrid é uma solução para a execução de aplicações do tipo BoT em *grids* computacionais [OUR05]. Ele forma uma comunidade *peer-to-peer*, onde cada *site* disponibiliza recursos que não estão sendo utilizados para participar do OurGrid. O protocolo do sistema garante que os *sites* que disponibilizam mais recursos, tenham uma prioridade maior (direito a mais recursos) quando forem executar suas aplicações.

O usuário tem acesso a comunidade OurGrid executando o MyGrid [CIR03]. O MyGrid combina os recursos locais, os recursos de outros *sites* da mesma comunidade e recursos que o usuário tem acesso (acesso remoto e transferência de arquivos, ou outro *middleware* como o Globus). Todos os recursos que compõem o MyGrid são utilizados para a execução de aplicações.

O MyGrid faz o escalonamento das tarefas e a atribuição das tarefas às máquinas, procurando melhorar o desempenho da aplicação e tratando a heterogeneidade dos recursos.

A.4.1 Arquitetura básica

Os principais componentes do OurGrid são:

- GuMs ou *grid machine*: são as máquinas onde as tarefas são executadas. O OurGrid suporta 3 tipos de gums:
    - MyGrid User Agent*:* permite que a máquina *home* acesse as outras *GuMs;*
    - Script based access*:* permite o uso de scripts *shell* para transferência de arquivos e execução remota. A vantagem dos scripts é que não requerem que as GuMs possuam o software do MyGrid, a desvantagem é que é bem menos eficiente que os "MyGrid User Agents";
    - Globus access: uma alternativa para utilizar os serviços providos pelo Globus *toolkit*.

- GuMProviders: GuMs que pertencem ao mesmo domínio são organizadas e coordenadas pelo GuMProvider, que é um serviço que disponibiliza e controla as GuMs utilizadas para executar as tarefas.
- MyGrid: é o *frontend*, ele provê todo o suporte para descrever, executar e monitorar *jobs* nas GuMs. Durante a execução de um *job* o MyGrid utiliza o GuMProvider para obter as GuMs que executarão as tarefas. O myGrid escalona as tarefas que são executadas nas GuMs. A máquina que roda o MyGrid é chamada *home machine* e as demais máquinas que participam do OurGrid são chamadas *grid machines*.

O relacionamento entre os componentes é mostrado na Fig. 26.

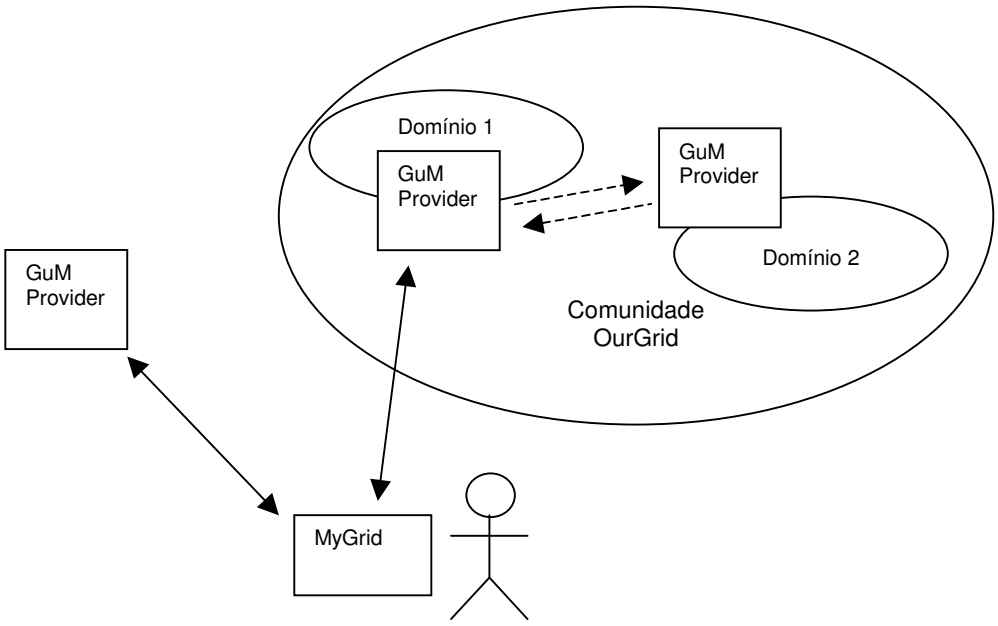

Figura 26. MyGrid e OurGrid

A.4.2 Especificação das aplicações paralelas

No MyGrid as aplicações são chamadas *jobs* e como as aplicações são do tipo *bag-of-tasks*, um *job* é um conjunto de tarefas que executam de forma independente.

Uma tarefa possui 3 fases, que são executada seqüencialmente:
- initial
- remote
- final

As fases *initial* e *final* são executadas na *home machine*, enquanto a *remote* é executada nas máquinas do *grid*. Para executar *jobs* no MyGrid é necessário criar um *job description file*.

O *job description file* é um arquivo texto que contém a descrição de um único *job*, como no exemplo a seguir:

```
job:
     label: myjob2
task:
     init:     put input input
               store mytask mytask
     remote:   mytask < input > output
     final:    get output output
```

A cláusula de *task* contém todas as fases de uma tarefa:
- *initial*: define que arquivos serão transferidos para a GuM antes de executar a tarefa;
- remote: define o comando que será executado pela GuM para processar a tarefa;
- final: define que arquivos serão retornados pela GuM após a execução da tarefa.

A aplicação do exemplo contém apenas uma tarefa que inicia com a transferência do arquivo input e *mytask* para a máquina remota. A tarefa então executa o *mytask* na GuM, recebendo o input como entrada padrão e produzindo o output como saída padrão. Quando a execução do *mytask* termina, a fase final da tarefa é executada, e o output é enviado ao *home machine*.

É possível acrescentar condições para determinar que arquivos serão transferidos, como por exemplo, checar o sistema operacional e enviar o arquivo executável apropriado dependendo do sistema.

As operações do MyGrid também podem ser executadas condicionalmente, utilizando uma estrutura do tipo *if-then-else*:

```
if condition then
     command 1
     command 2
     …
     command n
[else
     elsecommand 1
     elsecommand 2
     …
     elsecomand n ]
endif
```

onde:
condition: expressão *boolean*; pode conter variáveis de ambiente ou atributos do *job*;
command: comandos executados quando a condição é verdadeira. Deve-se colocar apenas um comando MyGrid por linha, pode ser *get*, *put* ou *store*;
elsecommand: comandos que são executados se a condição for falsa.

Uma vez que foi definido o arquivo de descrição do *job*, ele deve ser enviado para o MyGrid para que ele seja executado usando o comando "*addjob*" e se o arquivo estiver sintaticamente correto ele será escalonado para a execução no ambiente MyGrid.

O OurGrid é uma solução para rodar aplicações *bag of tasks* em *grids* computacionais, formando uma comunidade *peer-to-peer* onde os *sites* doam recursos ociosos para utilização do OurGrid. A sua proposta é ser um sistema simples, específico para aplicações *bag of tasks*, mas completo de forma que o usuário se preocupe apenas com a sua aplicação e não com detalhes do *grid*.

Apêndice B Especificação da linguagem XPWSL utilizando XML Schema

Optou-se pela utilização do XML Schema para especificar a linguagem XPWSL porque o Schema oferece as seguintes vantagens em relação ao DTD:
- possui vários tipos de dados primitivos;
- permite a definição de novos tipos de dados;
- permite a definição de chaves (elementos/conteúdos únicos dentro do XML);
- define relacionamentos entre os elementos.

B.1 Definição do XML Schema da aplicação

O trecho a seguir define a estrutura básica do modelo de aplicação, com as seções que devem estar presentes na definição de uma aplicação XPWSL:
- *header* – informações da aplicação;
- *assignment* – localização e definição das classes;
- *datalink* – dinâmica da aplicação, definição do fluxo de execução.

```
<element name="pas">
  <complexType>
   <sequence>
     <element name="header" type="r:headerType" maxOccurs="1"/>
     <element name="assignment" type="r:assignmentType"
                                                    maxOccurs="1" />
     <element name="datalink" type="r:datalinkType"  maxOccurs="1" />
   </sequence>
  </complexType>
</element>
```

B.2 Especificação das seções

A seção de header deve aparecer uma única vez no XML e contém apenas dois elementos do tipo string com informações da aplicação:

```
<complexType name="headerType">
  <sequence>
   <element name="name" type="string"/>
   <element name="description" type="string"/>
  </sequence>
</complexType>
```

A seção de assignment define a localização e o nome de cada uma das classes utilizadas no programa, podendo ser classes de tarefas, controle ou auxiliares. Cada arquivo é associado a um identificador único:

```
<complexType name="assignmentType">
    <sequence>
      <element name="task" minOccurs="1" maxOccurs="unbounded">
         <complexType>
              <all>
                   <element name="path" type="string" />
                   <element name="code" type="string" />
              </all>
        <attributeGroup ref="r:taskAttributes" />
         </complexType>
       </element>

       <element name="control" minOccurs="0" maxOccurs="unbounded">
         <complexType>
              <all>
                   <element name="path" type="string" />
                   <element name="code" type="string" />
              </all>
         <attributeGroup ref="r:controlAttributes" />
         </complexType>
       </element>
    </sequence>
</complexType>
```

A seção de *datalink* é mais complexa, porque existem 3 tipos de tarefas que podem ser executadas; não podemos ter uma restrição quanto à ordem e à quantidade de cada tipo de tarefa nesta seção.

Foi utilizado um elemento extra para selecionar um dos tipos de tarefa, em cada passo. Este elemento não tem restrição quanto à quantidade de vezes que aparece dentro da seção *datalink*.

```
<complexType name="datalinkType">
    <sequence>
       <element name="block" type="r:allBlocksType" minOccurs="1"
maxOccurs="unbounded"/>
   </sequence>
</complexType>

<complexType name="allBlocksType">
  <sequence>
     <choice>
      <element name="batch" type="r:batchType" minOccurs="1"
maxOccurs="unbounded">
      </element>
       <element name="case" type = "r:caseType" minOccurs="1"
maxOccurs="unbounded">
      </element>
   </choice>
```

```
  </sequence>
  <attributeGroup ref="r:blockAttributes" />
</complexType>
```

Com essa definição um *block* é considerado um conjunto de elementos *batch* ou *case*. Os Blocos de Execução seqüencial e de repetição possuem a mesma estrutura, sendo a única diferença os atributos do *block*. Apenas o *switch/case* tem um tratamento diferenciado, já que ele não possui um único conjunto de lotes de tarefas e sim várias alternativas. Por isso há a necessidade de se definir o *batch* e o *case*.

A estrutura de um lote de tarefas:

```
<complexType name="batchType">
  <sequence>
   <element name="task_id" type="r:taskIDType" />
   <element name="multi" type="positiveInteger" />
   <element name="input" type="r:batchInputType" />
   <element name="output" type="r:fileNameType" minOccurs="0"
maxOccurs="1" />
  </sequence>
  <attributeGroup ref="r:batchAttributes" />
</complexType>
```

No *batchType* são definidos: o tipo de tarefa, o número de instâncias e os e relacionamentos do lote, observa-se que o *output* é um elemento opcional (`minOccurs="0"`). .

Para a definição da nomenclatura, são utilizadas expressões regulares (*patterns)*, como pode ser observado a seguir:

Definição do arquivo com o Bloco de Execução:

```
<simpleType name="fileNameType">
  <restriction base="string">
   <pattern value="([a-zA-Z0-9]*\\)*[a-zA-Z0-9]+\.?[a-zA-Z]*" />
  </restriction>
</simpleType>
```

Identificadores de lotes, tarefas e delay:

```
<simpleType name="batchIDType">
    <restriction base="string">
         <pattern value="[a-zA-Z0-9]+" />
    </restriction>
</simpleType>

<simpleType name="controlIDType">
    <restriction base="string">
         <pattern value="[a-zA-Z0-9]+" />
    </restriction>
</simpleType>
```

```
<simpleType name="taskIDType">
    <restriction base="string">
         <pattern value="[a-zA-Z0-9]+" />
    </restriction>
</simpleType>


<simpleType name="delayTimeType">
    <restriction base="string">
         <pattern value="[0-9]+\.?[0-9]*" />
    </restriction>
</simpleType>

<simpleType name="batchInputType">
    <restriction base="string">
         <pattern value="([a-zA-Z0-9]*\\)*[a-zA-Z0-9]+\.?[a-zA-Z]*|[ ]*"
/>
    </restriction>
</simpleType>
```

B.3 Especificação completa do XPWSL

A seguir á apresentado o documento completo da especificação do XPWSL com o Schema.

```
<?xml version="1.0"?>

<schema
     targetNamespace="http://www.dca.fee.unicamp.br/projects/join"
     xmlns="http://www.w3.org/2001/XMLSchema"
     xmlns:r="http://www.dca.fee.unicamp.br/projects/join"
     elementFormDefault="qualified">

<element name="pas">
  <complexType>
   <sequence>
     <element name="header" type="r:headerType" maxOccurs="1"/>
     <element name="assignment" type="r:assignmentType" maxOccurs="1" />
     <element name="datalink" type="r:datalinkType" maxOccurs="1">
     </element>
   </sequence>
  </complexType>

  <key name="batchIdKey">
     <selector xpath="r:assignment/r:task"/>
     <field xpath="@id"/>
  </key>

  <key name="controlIdKey">
     <selector xpath="r:assignment/r:control"/>
     <field xpath="@id"/>
  </key>

  <key name="taskIdKey">
     <selector xpath="r:datalink/r:task"/>
```

```
    <field xpath="@id"/>
  </key>
</element>

<complexType name="headerType">
  <sequence>
   <element name="name" type="string"/>
   <element name="description" type="string"/>
  </sequence>
</complexType>

<complexType name="assignmentType">
  <sequence>
   <element name="task" minOccurs="1" maxOccurs="unbounded">
     <complexType>
      <all>
        <element name="path" type="string" />
        <element name="code" type="string" />
      </all>
      <attributeGroup ref="r:taskAttributes" />
     </complexType>
   </element>

   <element name="control" minOccurs="0" maxOccurs="unbounded">
     <complexType>
      <all>
        <element name="path" type="string" />
        <element name="code" type="string" />
      </all>
      <attributeGroup ref="r:controlAttributes" />
     </complexType>
   </element>
  </sequence>
</complexType>

<complexType name="datalinkType">
  <sequence>
   <element name="block" type="r:allBlocksType" minOccurs="1"
maxOccurs="unbounded"/>
  </sequence>
</complexType>

<complexType name="allBlocksType">
  <sequence>
   <choice>
     <element name="batch" type="r:batchType" minOccurs="1"
maxOccurs="unbounded">
     </element>
     <element name="case" type = "r:caseType" minOccurs="1"
maxOccurs="unbounded">
     </element>
   </choice>
  </sequence>
  <attributeGroup ref="r:blockAttributes" />
</complexType>

<complexType name="batchType">
```

```
  <sequence>
   <element name="task_id" type="r:taskIDType" />
   <element name="multi" type="positiveInteger" />
   <element name="input" type="r:batchInputType" />
   <element name="output" type="r:fileNameType" minOccurs="0"
maxOccurs="1" />
  </sequence>
  <attributeGroup ref="r:batchAttributes" />
</complexType>

<complexType name="caseType">
  <sequence>
   <element name="batch" type = "r:batchType" minOccurs="1"
maxOccurs="unbounded">
   </element>
  </sequence>
  <attributeGroup ref="r:caseAttributes" />
</complexType>

<simpleType name="fileNameType">
  <restriction base="string">
   <pattern value="([a-zA-Z0-9]*\\)*[a-zA-Z0-9]+\.?[a-zA-Z]*" />
  </restriction>
</simpleType>

<simpleType name="batchIDType">
  <restriction base="string">
   <pattern value="[a-zA-Z0-9]+" />
  </restriction>
</simpleType>

<simpleType name="controlIDType">
  <restriction base="string">
   <pattern value="[a-zA-Z0-9]+" />
  </restriction>
</simpleType>

<simpleType name="taskIDType">
  <restriction base="string">
   <pattern value="[a-zA-Z0-9]+" />
  </restriction>
</simpleType>

<simpleType name="delayTimeType">
  <restriction base="string">
   <pattern value="[0-9]+\.?[0-9]*" />
  </restriction>
</simpleType>

<simpleType name="batchInputType">
  <restriction base="string">
   <pattern value="([a-zA-Z0-9]*\\)*[a-zA-Z0-9]+\.?[a-zA-Z]*|[ ]*" />
  </restriction>
</simpleType>

<attributeGroup name="taskAttributes">
  <attribute name="id" type="r:taskIDType" use="required">
```

```
   <annotation>
     <documentation>
      Task identificator
     </documentation>
   </annotation>
  </attribute>

  <attribute name="delay" type="r:delayTimeType" >
   <annotation>
     <documentation>
      Datatype representing the delay related to the
      task execution time.
      This delay may be used by a performance prediction tool.
     </documentation>
   </annotation>
  </attribute>
</attributeGroup>

<attributeGroup name="controlAttributes">
  <attribute name="id" type="r:controlIDType" use="required">
   <annotation>
     <documentation>
      task control identificator
     </documentation>
   </annotation>
  </attribute>

  <attribute name="delay" type="r:delayTimeType" >
   <annotation>
     <documentation>
      Datatype representing the delay related to the
      batch execution time.
      This delay may be used by a performance prediction tool.
     </documentation>
   </annotation>
  </attribute>
</attributeGroup>


<attributeGroup name="batchAttributes">
  <attribute name="id" type="r:batchIDType" use="required">
   <annotation>
     <documentation>
      Batch identificator
     </documentation>
   </annotation>
  </attribute>
</attributeGroup>


<attributeGroup name="caseAttributes">
  <attribute name="value" type="string" use="required">
   <annotation>
     <documentation>
      Case indentificator (switch/case Execution Block
     </documentation>
   </annotation>
```

```
  </attribute>
</attributeGroup>

<attributeGroup name="blockAttributes">
  <attribute name="type" type="string" use="required">
  </attribute>
  <attribute name="control" type="r:controlIDType" >
  </attribute>
  <attribute name="max_iter" type="positiveInteger" >
  </attribute>

</attributeGroup>

</schema>
```

Apêndice C Implementação de uma tarefa na plataforma JoiN usando PASL

O modelo de aplicação do JoiN [LUC02] consiste em relações de dependência entre lotes de tarefas. Para entender este modelo é necessário definir antes os seguintes conceitos:

- *bloco de dados*: estruturas de dados serializáveis;
- *tarefa*: seqüência ordenada de operações aritméticas, matemáticas, estruturas de controle, atribuição/entrada e saída de dados que implementam algum tipo de processamento determinístico e finito sobre um bloco de dados de entrada a fim de produzir um bloco de dados de saída.;
- *lote de tarefas*: aplicação de uma determinada tarefa sobre blocos de entrada;
- *relação de dependência*: relação entre lotes de tarefas, onde os blocos de saída de um (ou vários) lote(s) são utilizados como dados para os blocos de entrada de outro (ou vários) lote(s).

As dependências entre os lotes de tarefas podem ser definidas por uma linguagem de especificação de aplicações paralelas denominada PASL. Esta linguagem foi descrita na seção 3.1.5 e nesta seção será mostrado como é implementado um tipo de tarefa.

Cada tarefa de uma aplicação do JoiN deve implementar a interface AppCode (join/src/app/AppCode.java):

```
public interface AppCode {
  public Serializable taskrun(Serializable par);
}
```

A implementação dessa interface fica a critério do programador. O importante é que a computação a ser executada na tarefa deve ser feita dentro do método *taskrun*. Os dados de entrada da tarefa são passados através do argumento *par*, e os dados de saída devem ser formatados e retornados com um tipo *Serializable.*

Como os relacionamentos entre os lotes de tarefas se baseiam em juntar os resultados de um lote de tarefas no coordenador e distribuí-los para o lote seguinte, cabe ao programador da aplicação determinar a estrutura em que os dados são passados entre os lotes, utilizando como base um vetor com elementos *Object*.

Atualmente, cada tarefa pode ser considerada como uma unidade atômica, onde os dados de entrada são manipulados localmente (não é permitida a comunicação entre os trabalhadores) e o resultado é retornado para ser passado para o próximo lote.

A plataforma é responsável por juntar o resultado de cada uma das tarefas de um lote e distribuir entre as tarefas do próximo lote. Para isso ele monta um vetor com os resultados e

realiza o particionamento do vetor de acordo com a cardinalidade do próximo lote de tarefas.

Este relacionamento entre a cardinalidade dos lotes é explicado utilizando a seguinte aplicação como exemplo.

```
// Header Section
name = "numberSieve1";
description = "An application that searches through intervals looking for
prime numbers";

%%   // This is a separator
// Assigment section
path = "numberSieve";
T0 =  "DistributeData.class";
T1 =  "NumberSieve.class";
T2 =  "GatherResults.class";

%%   // This is a separator
// Data link section
B0 = T0(1)  << NULL;
B1 = T1(50) << B0;
B2 = T2(1)  << B1;

%%   // This is a separator
// Model section
T0=delay(t1);
T1=delay(t2);
T2=delay(t3);
```

- A tarefa "DistributeData.class" é executada em uma única tarefa, não precisa de dados de entrada e retorna como dados de saída um vetor com X elementos, onde X é um múltiplo de 50, visto que estes dados serão distribuídos para 50 tarefas. Se este lote de tarefas fornecesse entrada para outros lotes, o número de elementos do vetor de saída deveria ser um múltiplo comum das cardinalidades de todos os lotes que dependem destes dados.
- JoiN recebe o resultado e particiona o vetor no número de instâncias necessárias para o próximo bloco e inicia a execução das instâncias da tarefa “NumberSieve.class”.
- JoiN aguarda todas as 50 tarefas de B1 terminarem a execução das tarefas, junta todos os 50 vetores em um único e envia para execução em "GatherResults.class".
- Após receber o resultado desta última tarefa, a aplicação é concluída.

Apêndice D Aplicações utilizadas nos testes comparativos do PASL e XPWSL

Neste Apêndice são apresentados os arquivos de especificação e as tarefas das aplicações utilizadas nos testes das seção 5.3

D.1 Especificação em PASL

Seq3PASL.jas

```
// Header Section
name =          "seq3 pasl";
description =   "3 lotes";

%%      // This is a separator
// Assigment section
path =  "br.unicamp.fee.dca.join.applications.vazio";
T0 =    "EvalPopul.class";

%%      // This is a separator
// Data link section
B0 = T0(1) << NULL;
B1 = T0(1) << B0;
B2=  T0(1) << B1;

%%      // This is a separator
// Model section
T0=delay(t1);
```

Seq10PASL.jas

```
// Header Section
name =          "seq10 pasl";
description =   "10 lotes";

%%      // This is a separator
// Assigment section
path =  "br.unicamp.fee.dca.join.applications.vazio";
T0 =    "EvalPopul.class";

%%      // This is a separator
// Data link section
B0 = T0(1) << NULL;
B1 = T0(1) << B0;
B2=  T0(1) << B1;
B3 = T0(1) << B2;
B4=  T0(1) << B3;
B5 = T0(1) << B4;
```

```
B6=  T0(1) << B5;
B7 = T0(1) << B6;
B8=  T0(1) << B7;
B9 = T0(1) << B8;

%%      // This is a separator
// Model section
T0=delay(t1);
```

EvalPopul.class

```
package br.unicamp.fee.dca.join.applications.vazio;

import java.io.Serializable;
import br.unicamp.fee.dca.join.framework.app.AppCode;

public class EvalPopul implements AppCode {
     public Serializable taskrun(Serializable par) {
          Serializable result = null;
          if (par == null)
          {
               Object [][] part = new Object[1][1];
               part[0][0]=new Integer(1);

               result= (Serializable)part;

          } else
          {
               result=par;
          }
          try {
               Thread.sleep(10000);
          } catch (InterruptedException ie) {
          }
          return result;
     }
}
```

D.2 Especificação em XPWSL

Especificação das aplicações utilizadas para o teste de cada estrutura de controle.

Seq3.xml

```
<?xml version="1.0"?>
<pas xmlns="http://www.dca.fee.unicamp.br/projects/join"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://www.dca.fee.unicamp.br/projects/join
PAS.xsd">

<header>
     <name>Sequencial</name>
     <description>sequencial básico com 3 lotes (10
seg)=30seg</description>
</header>
```

```
<assignment>
     <task id="tid" delay="10">
          <path>br.unicamp.fee.dca.join.applications.vazio</path>
          <code>teste.class</code>
     </task>
</assignment>

<datalink>
     <block type="sequence" >
          <batch id="lote1">
               <task_id>tid</task_id>
                <multi>1</multi>
                <input></input>
          </batch>
          <batch id="lote2">
               <task_id>tid</task_id>
               <multi>1</multi>
               <input>lote1</input>
          </batch>
          <batch id="lote3">
               <task_id>tid</task_id>
               <multi>1</multi>
               <input>lote2</input>
          </batch>
     </block>
</datalink>
</pas>
```

Seq10.xml

```
<?xml version="1.0"?>
<pas xmlns="http://www.dca.fee.unicamp.br/projects/join"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://www.dca.fee.unicamp.br/projects/join
PAS.xsd">

<header>
     <name>Sequencial</name>
     <description>sequencial com 10 lotes (10 seg)=100seg</description>
</header>

<assignment>
     <task id="tid" delay="10">
          <path>br.unicamp.fee.dca.join.applications.vazio</path>
          <code>teste.class</code>
     </task>
</assignment>

<datalink>
     <block type="sequence" >
          <batch id="lote1">
               <task_id>tid</task_id>
                <multi>1</multi>
                <input></input>
          </batch>
```

```
            <batch id= "lote2">
                 <task_id>tid</task_id>
                 <multi>1</multi>
                 <input>lote1</input>
            </batch>
            <batch id= "lote3">
                 <task_id>tid</task_id>
                 <multi>1</multi>
                 <input>lote2</input>
            </batch>
            <batch id= "lote4">
                 <task_id>tid</task_id>
                 <multi>1</multi>
                 <input>lote3</input>
            </batch>
            <batch id= "lote5">
                 <task_id>tid</task_id>
                 <multi>1</multi>
                 <input>lote4</input>
            </batch>
            <batch id= "lote6">
                 <task_id>tid</task_id>
                 <multi>1</multi>
                 <input>lote5</input>
            </batch>
            <batch id= "lote7">
                 <task_id>tid</task_id>
                 <multi>1</multi>
                 <input>lote6</input>
            </batch>
            <batch id= "lote8">
                 <task_id>tid</task_id>
                 <multi>1</multi>
                 <input>lote7</input>
            </batch>
            <batch id= "lote9">
                 <task_id>tid</task_id>
                 <multi>1</multi>
                 <input>lote8</input>
            </batch>
            <batch id= "lote10">
                 <task_id>tid</task_id>
                 <multi>1</multi>
                 <input>lote9</input>
            </batch>
       </block>
</datalink>
</pas>
```

Loop.xml

```
<?xml version= "1.0"?>
<pas xmlns= "http://www.dca.fee.unicamp.br/projects/join"
 xmlns:xsi= "http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation= "http://www.dca.fee.unicamp.br/projects/join
PAS.xsd">
```

```
<header>
     <name>Loop</name>
     <description>10 iteracoes (10 cada) =100seg</description>
</header>

<assignment>
     <task id= "teste" delay= "10">
          <path>br.unicamp.fee.dca.join.applications.vazio</path>
          <code>teste.class</code>
     </task>
</assignment>

<datalink>
     <block type= "loop" max_iter= "10"  >
          <batch id= "lote">
               <task_id>teste</task_id>
               <multi> 1 </multi>
               <input></input>
          </batch>
     </block>
</datalink>

</pas>
```

LoopCond.xml

```
<?xml version= "1.0"?>
<pas xmlns= "http://www.dca.fee.unicamp.br/projects/join"
     xmlns:xsi= "http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation= "http://www.dca.fee.unicamp.br/projects/join
PAS.xsd">

<header>
     <name>Loop condicional</name>
     <description>10 iteracoes (10 cada)=100seg</description>
</header>

<assignment>
     <task id= "teste" delay= "10">
          <path>br.unicamp.fee.dca.join.applications.vazio</path>
          <code>teste.class</code>
     </task>
     <control id= "controle" delay= "1">
          <path>br.unicamp.fee.dca.join.applications.vazio</path>
          <code>LoopControl.class</code>
     </control>
</assignment>

<datalink>
     <block type= "loop" control= "controle" max_iter= "10"  >
          <batch id= "lote">
               <task_id>teste</task_id>
               <multi>1</multi>
               <input></input>
          </batch>
```

```
     </block>
</datalink>

</pas>
```

EvalLoop.class

```
package br.unicamp.fee.dca.join.applications.vazio;

import java.io.Serializable;
import br.unicamp.fee.dca.join.framework.app.AppCode;

public class EvalLoop implements AppCode {

     public Serializable taskrun(Serializable par) {
          Serializable result = null;
          if (par == null)
          {
               Object [][] part = new Object[1][1];
               part[0][0]=new Integer(1);

               result= (Serializable)part;
          }
          else
          {
               result=par;
          }
          try {
               Thread.sleep(10000);
          } catch (InterruptedException ie) {
          }

          return result;
     }
}
```

LoopControl.class

```
package br.unicamp.fee.dca.join.applications.vazio;

import java.io.Serializable;
import br.unicamp.fee.dca.join.framework.app.AppLoopControl;

public class LoopControl implements AppLoopControl {
    public boolean exitLoop(Serializable par) {
        return false;
    }
}
```