4
Arquitetura

O processamento iterativo de dados se faz presente em muitas aplicações científicas nas quais os dados variam em função de uma dada dimensão como, por exemplo, o tempo. Nestas aplicações, um conjunto de operações repetitivas leva os dados de um estado inicial para um estado final. Exemplos destas aplicações incluem previsão de tempo e simulação da trajetória de partículas. Analisado sobre a visão de processamento de consulta, este é um modelo de execução interessante no qual tuplas são avaliadas repetidamente por um fragmento do plano de execução.

A execução de consultas no modelo iterativo pode requerer minutos ou até mesmo horas dependendo do número de tuplas a serem processadas, do número de iterações a serem realizadas e a complexidade do cálculo de cada iteração. Desta forma, uma escolha natural para diminuir o tempo de avaliação das consultas neste modelo é utilizar os recursos disponíveis em uma grade computacional para paralelizar o calculo das iterações.

Neste capítulo apresenta-se a arquitetura de execução de consulta desenvolvida para realizar o processamento de consultas no modelo iterativo com suporte a paralelismo em um ambiente nada compartilhado[1]. Esta solução foi implementada no contexto do projeto CODIMS-G cujo objetivo é auxiliar os usuários na execução de aplicações distribuídas em uma grade computacional.

Inicialmente apresenta-se uma descrição do subsistema de processamento de consulta do CoDIMS-G. Em seguida apresentam-se detalhes sobre a extensão realizada para adicionar suporte ao processamento de consultas no modelo Orbit em um ambiente de grade. Ao final do capítulo é realizada uma breve comparação entre as funcionalidades aqui implementadas com alguns trabalhos relacionados.

[1] O ambiente pode ser modelado como híbrido uma vez que os dados são mantidos em um nó e distribuídos durante o processamento a nós com disco, CPU e memória.

4.1. QEEF-G

O *Query Engine Execution Framework* (QEEF) (FAUSTO V.M.AYRES, FABIO PORTO et al., 2003) é um componente de *software* que visa facilitar o desenvolvimento de máquinas de execução de consultas (MEC) que suportem diferentes características de execução como, por exemplo, tipo de sincronismo (síncrono/assíncrono), distribuição (local/remoto), paralelismo(inter-operador/intra-operador), modelo de tratamento do fluxo de dados (sob demanda ou produção) e controle e estratégia de produção de resultado (resultados parciais ou completos). A abordagem utilizada consiste na utilização de composição de operadores pré-definidos denominada módulo, responsável por implementar uma determinada característica de execução. Módulos podem ser combinados entre si para formar um plano de execução.

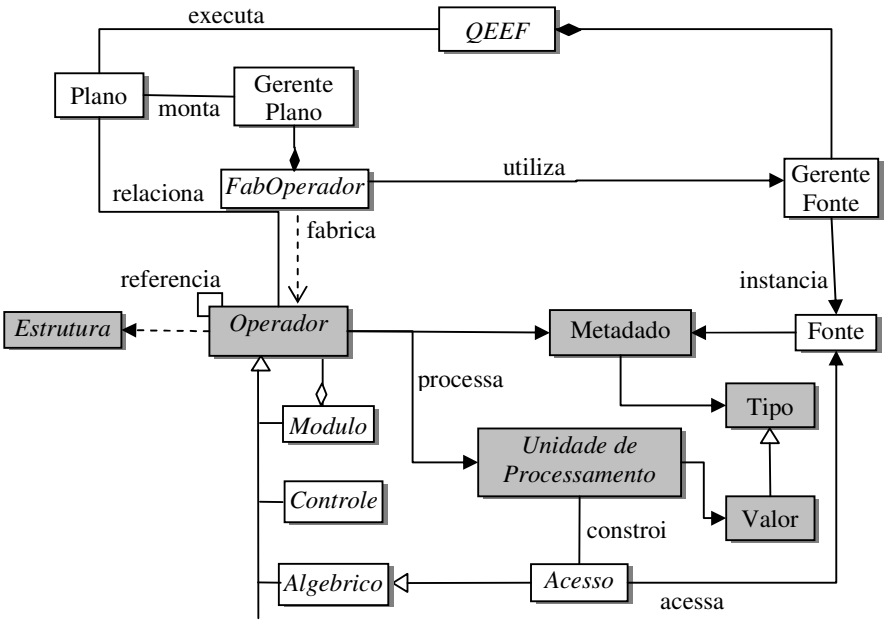

Figura 5. Framework QEEF

A figura 5 apresenta a arquitetura central do projeto QEEF-G uma instância do *framework* desenvolvida no contexto do projeto CoDIMS-G. A classe QEEF implementa o padrão de projeto *facade* (GRADY BOOCH, IVAR JACOBSON et al., 1999) e é responsável pela interface da máquina de execução com a aplicação usuária. Esta classe possui métodos para inicialização, submissão de um plano, obtenção dos resultados e finalização.

A classe GerentePlano (padrão de projeto *singleton*) interpreta um PEC-XML e gerencia a montagem do plano utilizando as fábricas de operadores (padrão de projeto *factory*) disponíveis.

A classe Operador define uma interface comum a todos os operadores, e permite a organização na forma produtor/consumidor dos operadores em plano de execução. Esta interface é baseada no modelo de iterador (G.GRAEFE, 1990) que possui os métodos apresentados na tabela 3. Os métodos *get* e *put* são usados de acordo com o fluxo de controle utilizado (*Demand-driven* ou *Data-driven*). O fim de processamento é representado por uma unidade de processamento especial, que é enviada pelo método get/put.

Open	Inicia as estruturas utilizadas pelo operador e define o formato dos dados produzidos, representado pela classe Metadado.
Get	Recebe uma unidade de dado sob demanda do produtor.
Put	Produz uma unidade de dado e envia para o consumidor.
Close	Finaliza o operador.

Tabela 3. Interface Operador

De acordo com a funcionalidade desenvolvida, os operadores podem ser classificados em algébricos e de controle. Os operadores de controle realizam a comunicação entre os operadores algébricos e são responsáveis por gerir o fluxo de dados entre os operadores.

Já os operadores algébricos são responsáveis por implementar a semântica de uma aplicação através dos operadores de uma álgebra específica. Um caso especial deste tipo de operador é o Acesso, que representa uma folha na árvore de execução. Este operador é responsável por ler os dados de uma fonte de dados e instanciar as unidades de processamento. Nesta implementação utilizamos o modelo relacional e, portanto, uma unidade de processamento é representada por uma tupla composta de dados de tipos não complexos.

A classe *Módulo* generaliza os módulos de execução. Sua implementação requer a utilização de operadores de controle a serem inseridos no plano segundo a característica de execução desejada.

O Gerente de Fonte é responsável por manter informação sobre as fontes de dados disponíveis, seu formato e localização. Nesta implementação, estas informações são armazenadas no catálogo do CoDIMS-G.

Por fim, a classe Estrutura representa as estruturas de dados que podem ser utilizadas pelos operadores, tais como listas (Buffer), filas (Queue) e arquivos (File), para o armazenamento temporário das unidades de processamento.

Em extensão às funcionalidades originalmente oferecidas pelo *framework*, a instância CODIMS-G desenvolvida neste trabalho possui suporte a vários tipos de dados e oferece um avaliador de predicado que pode ser utilizado pelos operadores. O suporte a tipos de dados foi incorporado pela utilização das classes Tipo e Valor. A classe Tipo é utilizada na definição dos metadados e um valor representa um dado na unidade de processamento que até então só aceitava *strings*. Já o avaliador de predicado se baseia no conceito de árvore de avaliação e cada predicado pode ser definido em função dos operadores suportados por cada tipo.

4.2. Arquitetura de Execução

O processamento de dados no modelo de órbita é comum a várias aplicações nas quais seus dados variam em função do tempo. Do ponto de vista de processamento de consulta este modelo pode ser implementado a partir da constante avaliação de uma tupla por um mesmo fragmento do plano de execução. Onde a cada execução de uma órbita os dados são atualizados e as tuplas são levadas ao próximo estado.

Nesta seção apresentamos uma arquitetura de execução de consulta paralela, implementada a partir do QEEF-G, para a avaliação de consultas no modelo em Órbita. O objetivo é utilizar os recursos computacionais disponíveis na grade para minimizar o tempo de execução dessas consultas. Para isso, consideramos uma arquitetura de *hardware* do tipo nada compartilhado, na qual cada nó da grade executa o fragmento da consulta a ser processado de maneira cíclica. Nesta arquitetura de execução, um operador denominado Orbit é responsável por controlar as iterações de uma tupla e realizar a distribuição dos dados entre os nós de forma a minimizar o tempo de execução.

Nosso modelo de execução é bastante semelhante ao modelo de consulta do Eddy. Neste trabalho, um operador de controle n-ário é responsável por ler tuplas de uma fonte de dados e distribuí-las para os demais operadores do plano, que as processarão e as devolverão para serem mapeados ao próximo operador. Uma política de roteamento define para quais operadores uma tupla pode ser encaminhada e uma política de controle de fluxo determina para qual destes operadores a tupla deve seguir. No contexto deste trabalho, o operador Orbit possui as mesmas funcionalidades. A política de roteamento define se uma tupla deve ser processada novamente e o controle de fluxo é responsável por definir para qual nó uma tupla deve ser processada.

A implementação desta arquitetura é baseada no QEEF-G e segue o princípio da organização em módulos de execução, a seguir são apresentados os módulos desenvolvidos e detalhes da implementação de cada um.

4.2.1. Módulo Iterativo

O módulo iterativo tem como objetivo a execução cíclica de um fragmento do plano de execução. Neste módulo uma unidade de dados é avaliada continuamente até que atinja certo número de iterações, ou seja, eliminada por algum operador do FEC (Fragmento de Execução Cíclica). Sua implementação é baseada na adição do operador de controle *Orbit*, que é ilustrado pela figura 6. As setas representam o fluxo de dados.

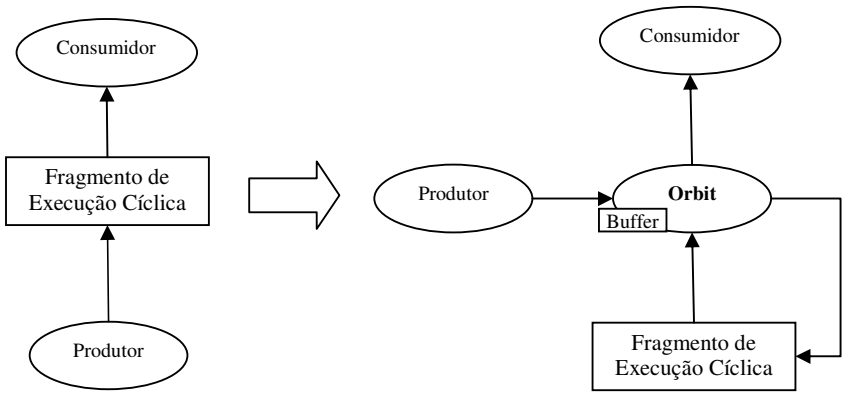

Figura 6. Módulo de Execução Iterativo

Inicialmente, de acordo com o volume de memória disponível para o operador *Orbit*, um conjunto de dados é consumido e disponibilizado para processamento em um buffer do operador. Para identificar o número de iterações realizadas por cada instância uma propriedade denominada *iteração* é adicionada a cada uma antes de ser processada pelo FEC.

A produção de dados pelo FEC acontece segundo a demanda do consumidor. Cada instância processada é enviada ao operador *Orbit* que incrementa o número de iterações realizadas pela mesma e determina se deve ser novamente avaliada. De acordo com a necessidade da aplicação, os resultados intermediários podem ou não ser enviados ao consumidor. À medida que as instâncias carregadas inicialmente atingem o número máximo de iterações desejadas ou são eliminadas pelo FEC, novas tuplas do produtor são consumidas e disponibilizadas para processamento. A fim de evitar que o processamento seja interrompido para que seja realizada a carga de novas instâncias, este processo acontece por uma linha de execução independente que é controlada por um semáforo, sempre que uma instância deve ser carregada, uma permissão é liberada.

A detecção de final de processamento é um pouco mais difícil de ser percebida no processamento iterativo, uma vez que o operador *Orbit* deve ser notificado sempre que uma instância de dados é eliminada por algum operador do FEC, que pode inclusive estar sendo executado em outra máquina. Para solucionar este problema foi criada uma especialização na hierarquia de classes de unidade de dados para representar instâncias com informações de controle. Sempre que algum operador do FEC eliminar uma instância de dado ele deve criar uma instância de controle para que o operador *Orbit* seja notificado. Este mecanismo possui a vantagem de permitir que dados e informações de controle utilizem o mesmo canal de comunicação, e pode ser estendido para auxiliar na resolução de outros problemas.

O modelo iterativo é ortogonal ao modelo de comunicação adotado. Assim, podem-se ter órbitas em um ambiente mono processável, bem como numa arquitetura paralela híbrida, como a focada neste trabalho. As seções 4.2.2 e 4.2.3 discutem a implementação do modelo orbit para a arquitetura paralela.

4.2.2. Módulo de Comunicação

O modelo de comunicação implementado se baseia na utilização de blocos de comunicação de tamanhos variáveis, oferecendo flexibilidade na distribuição de dados entre instâncias que participam da execução paralela de um fragmento do plano de execução. O objetivo é favorecer a adaptatividade durante a execução, de tal forma que o tamanho dos blocos de comunicação possa ser definido em tempo de execução segundo o desempenho de cada nó.

A implementação deste modelo se baseia na utilização de quatro operadores de controle: *Sender/Receiver (DONALD KOSSMAN, 2000) e Instance2Block/Block2Instance*. Os operadores *Sender/Receiver* são responsáveis por deixar transparentes aspectos como tecnologia de comunicação utilizada e serialização/deserialização dos dados dos demais operadores do fragmento do plano de execução. Já os operadores *Instance2Block/Block2Instance* têm a função de converter a unidade de processamento de um fragmento para bloco/instância respectivamente. A figura 7 ilustra como estes operadores podem ser dispostos de forma a deixar transparente para os demais operadores do plano o processo de comunicação.

Para oferecer uma implementação flexível do módulo de comunicação, o tamanho do bloco utilizado pode ser definido de forma estática ou a cada chamada. De forma a minimizar o overhead de comunicação, os blocos de dados são enviados como anexos na mensagem SOAP, evitando que cada unidade de dados tenha que ser codificada em XML.

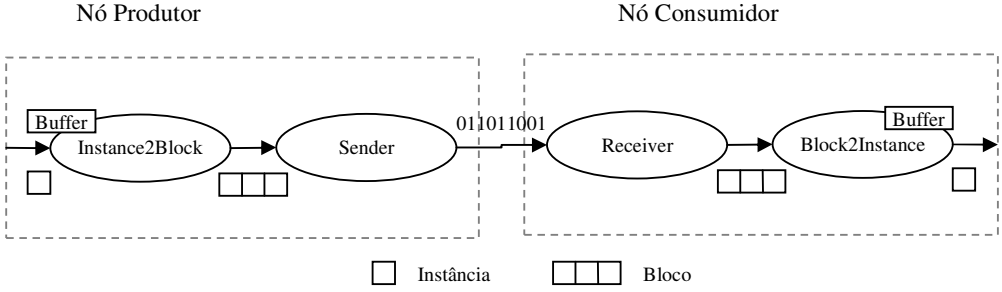

Figura 7. Operadores de Comunicação

A solicitação por blocos ao operador *Instance2Block* é realizada através da emissão de *tickets* que são registrados no quadro de comunicação em uma lista de

pedidos do operador. Cada *ticket* é identificado pelo *id* do operador consumidor e especifica o tamanho e o tempo máximo que se pode gastar para montar o bloco. Este tempo funciona como um mecanismo de *timeout* para garantir que mesmo sob uma eventual diminuição do fluxo de dados no sistema os blocos sejam criados, não atrasando o processamento do subconjunto de instâncias disponíveis. Uma vez que o ticket foi registrado, o consumidor deve realizar uma requisição por dados ao *Instance2Block* que pegará o pedido do consumidor correspondente e começara a consumir as instâncias necessárias para montar o bloco. O operador *Instance2Block* funciona de maneira assíncrona em relação ao seu consumidor e, desta forma, as instâncias consumidas são armazenadas provisoriamente em um *buffer* até serem encaminhadas para algum bloco. Para proporcionar algumas otimizações e evitar perda de tempo desnecessária durante a montagem dos blocos, os *tickets* podem ser atualizados mesmo depois de emitidos, por exemplo, reduzindo o seu tamanho ou o tempo máximo de espera.

Já o funcionamento do operador *Block2Instance* é relativamente simples, inicialmente um bloco de dados é consumido de seu produtor e as instâncias do bloco são enviadas a um buffer. Sempre que existe alguma requisição por dados uma instância é retirada do buffer e é enviada, se o buffer estiver vazio um novo bloco de dados é consumido.

4.2.3. Módulo Paralelo Distribuído

O módulo de paralelismo é responsável por introduzir suporte ao paralelismo intra-operador em uma arquitetura nada compartilhado. Sua função é dividir o fluxo de execução, controlar a distribuição de dados entre as instâncias e unir novamente o fluxo de execução, deixando transparente para os demais operadores do plano a presença do paralelismo.

O funcionamento do módulo de paralelo distribuído é baseado na utilização dos operadores de controle *Split/Merge (DONALD KOSSMAN, 2000)* e os operadores do módulo de distribuição que foram apresentados anteriormente. A figura 8 ilustra o posicionamento destes operadores no plano de execução. Nesta arquitetura um par de operadores *Sender/Receiver* é utilizado para enviar os dados ao fragmento onde serão processados e um outro par de operadores é utilizado para devolvê-los. Os operadores *Instance2Block/Block2Instance* são utilizados

sempre que a conversão da unidade de dados for necessária, observe que no nó de distribuição apenas uma instância destes operadores é necessária.

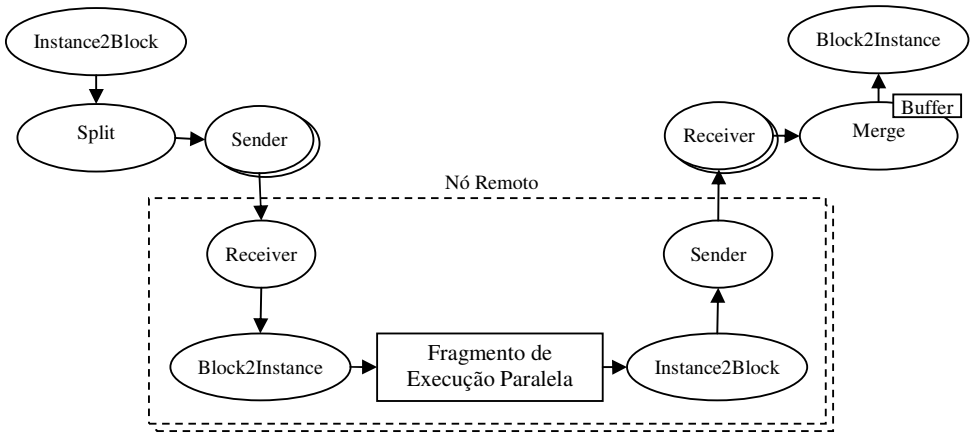

Figura 8. Módulo de Execução Paralelo Distribuído

O operador *Split* tem a função de dividir o fluxo de execução e controlar a distribuição dos dados entre os nós remotos. Seu posicionamento no PEC é determinado de forma a permitir o cálculo do tamanho de blocos a serem utilizados. Ao receber a requisição de dados do fragmento remoto (seu respectivo *Sender*) o tamanho do bloco de comunicação é calculado e o ticket emitido pelo *Sender* é atualizado. A política de distribuição dos dados e tamanho de blocos é apresentada em detalhes no capítulo 5.

O operador Merge tem a função de coletar os dados produzidos pelas instâncias paralelas e disponibilizá-las para processamento. Todos os blocos de dados recebidos são armazenados em um buffer de onde podem ser consumidos quando necessário. No funcionamento sob demanda (*get*) uma linha de execução é aberta para cada nó remoto, de forma a não deixar que os dados produzidos estourem os *buffers* dos fragmentos remotos e impeçam que a produção continue.

4.3. Combinando os Módulos de Execução

Os módulos de execução apresentados nas seções anteriores podem ser compostos de acordo com a figura 9 para formar um plano de execução para as consultas no modelo *Orbit*. Nesta figura as setas indicam o fluxo dos dados. A comunicação entre os operadores se dá segundo o modelo de *get*, no qual o consumidor aciona o produtor sempre que necessita de dados.

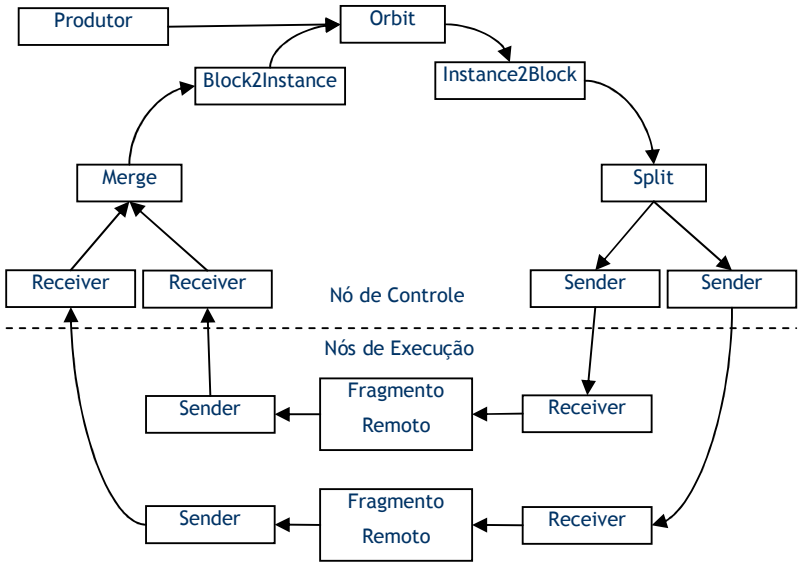

Figura 9. Arquitetura de Execução

O operador de controle *Orbit* consome tuplas de seus produtores. Pode-se verificar que, neste modelo, o operador apresenta dois produtores. Aquele identificado como *Produtor* na Figura 9 corresponde ao fragmento do plano que produz dados ao *Orbit*. Já o operador *Block2Instance* realimenta o *Orbit* com tuplas processadas pelo nó remoto. *Orbit* obtém dados de seu produtor apenas quando não há suficientemente dados realizando órbitas. Nestas situações *Orbit* tenta obter dados de seu *Produtor*. A existência do produtor, externo ao ciclo, determina o fim do laço que, de outra forma, seria produzido pelo modelo *Orbit*. O caso em que não há dados suficientes para alimentar os nós remotos é tratado na seção 5.3.2 como um processo de adaptatividade.

Do ponto de vista arquitetural, a Figura 9 distingue dois papéis entre os operadores em uma *órbita*, o nó dito controle e os nós de execução. O primeiro tem como peça central o operador *Orbit* e os componentes do módulo paralelo-distribuído e comunicação. O segundo, corresponde a implementação paralela de um fragmento do PEC. Os nós de execução implementam os componentes do módulo de comunicação que complementam seus pares no nó de controle.

No nó de controle, especificamente no operador *Merge*, uma thread é aberta para cada nó remoto. Estas consumirão dados dos fragmentos até que recebam um sinal de fim de execução. Esta mensagem indica que este nó não participará mais da execução ou que a execução chegou ao fim. Uma vez que o

fluxo de dados entre os operadores ocorre sobre demanda, o operador *Merge* pode ser visto como coordenador da execução.

No nó remoto o funcionamento acontece de maneira assíncrona, enquanto o operador *Instance2Block* do fragmento possuir espaço em buffer as tuplas serão processadas. Sempre que houver requisições pendentes do operador *Merge* um bloco de tuplas será montado e enviado ao nó central. Ainda no fragmento remoto, o operador *Block2Instance* realiza o armazenamento temporário de todas as tuplas a serem processadas, sempre que não houver tuplas disponíveis requisições por dados serão realizadas ao nó de controle.

Ao receber as requisições dos fragmentos, o operador *Split* definirá um tamanho de bloco adequado àquele fragmento – segundo o modelo de custo apresentado no capítulo 5 – e solicitará a sua montagem ao operado *Instance2Block*, que por sua vez consumirá as tuplas do operador *Orbit*. O processamento fica neste ciclo até que não existam mais tuplas a serem processadas.

4.4. Síntese do Capítulo

Neste capítulo apresentou-se a arquitetura de execução desenvolvida para suportar o processamento de consulta no modelo em órbita. A implementação realizada foi desenvolvida a partir do QEEF-G e utiliza o conceito de módulos de execução. Em especial foram desenvolvidos os módulos de: comunicação, paralelismo e processamento em órbita. O módulo Orbit é bastante semelhante ao modelo de execução proposto em Eddies, e difere-se pela política de roteamento, o controle de fluxo e mecanismo de detecção de final de processamento, para a qual foi desenvolvido um mecanismo especial de comunicação que permite que mensagens de dados e controle sejam trocadas da mesma.

Uma das dificuldades encontradas no desenvolvimento deste trabalho foi a falta de uma linguagem para definição dos planos de execução que suportasse o conceito de módulo. Esta ausência dificulta a criação dos planos de execução uma vez que faz com que o usuário ou o otimizador tenham que conhecer o funcionamento de cada módulo, seus operadores de controle e como estes devem se relacionar com os demais operadores do plano.

No próximo capítulo apresentarmos uma estratégia de adaptação que permite ao sistema reagir as constantes variações de desempenho dos nós utilizados e redefinir o nível de paralelismo utilizado de acordo com o volume de dados disponível.