Uma Visão Geral dos Sistemas distribuídos de Cluster e Grid e suas Ferramentas para o processamento paralelo de dados

Luiz Antônio Vivacqua Corrêa Meyer

Resumo

Aplicações convencionais e não convencionais tais como as aplicações científicas e estatísticas podem usufruir do processamento paralelo para o aumento de desempenho. Este trabalho apresenta uma visão geral das arquiteturas e paradigmas de programação paralela. Em particular, são discutidos no campo do desenvolvimento de aplicações, o uso da biblioteca MPI para a troca de mensagens e dos sistemas Globus e Condor para a gerência de tarefas no Grid. Em termos de arquitetura, este trabalho descreve os principais componentes de um *cluster* e de um Grid apresentando suas características e vantagens .

1. Introdução

Os programas científicos e estatísticos, em geral processam grandes conjuntos de dados. Portanto, um problema que pode surgir durante a execução destes programas é o tempo necessário para finalizar o seu processamento. Uma forma de melhorar o desempenho destas aplicações é utilizar o paralelismo. Uma possibilidade de explorar o paralelismo é utilizar uma máquina com processamento paralelo. Entretanto, esta solução apresenta algumas desvantagens tais como o alto custo de hardware e software. Outra possibilidade é explorar um ambiente distribuído como um grupo de PC (*cluster*). Este ambiente pode prover desempenho equivalente ao obtido com as máquinas paralelas, porém com custos substancialmente inferiores. Uma terceira alternativa é a utilização de um ambiente de Grid. Os Grids vem emergindo como plataformas para alto desempenho e para a integração de recursos em rede. Um Grid pode ser definido como um ambiente de processamento onde recursos heterogêneos e distribuídos, administrados por organizações independentes, podem ser compartilhados e agregados para formar um supercomputador virtual.

Este trabalho descreve as principais características dos ambientes de *cluster* e Grid e algumas ferramentas para utilização dos recursos nestes ambientes. Ele está organizado da seguinte forma: na segunda seção é introduzido alguns conceitos básicos referentes ao processamento de dados paralelo, como os modelos de memória e os paradigmas de programação. Na seção 3 é descrita a arquitetura de um *cluster* e a ferramenta de programação baseada na troca de mensagem. A quarta seção discute o Grid e os sistemas básicos para a sua utilização, enquanto que a quinta seção conclui este trabalho.

2. Processamento Paralelo

A computação paralela é alcançada através da divisão do problema em tarefas menores que podem ser simultaneamente processadas por múltiplos processadores. Por exemplo, a adição de dois vetores de números A e B pode ser feita por dois processadores. O primeiro, executa a soma da primeira metade de A à primeira metade de B, enquanto que o segundo, adiciona a segunda metade de A à segunda metade de B.

Duas métricas são utilizadas para avaliar a eficiência de um sistema paralelo: aceleração linear e crescimento linear. A aceleração linear não leva em consideração o tamanho do problema e sim o tamanho do sistema. Se o "hardware" for dobrado, a tarefa deverá ser executada na metade do tempo. A aceleração linear é medida pelo ganho que é definido como a razão entre o tempo de execução com *1* processador e o tempo de execução com *n* processadores.

O crescimento linear mede a habilidade de crescimento do sistema e do problema. Se o hardware for duplicado, o sistema será capaz de executar um problema duas vezes maior no mesmo espaço de tempo. A figura 1 ilustra as duas propriedades de um sistema paralelo.

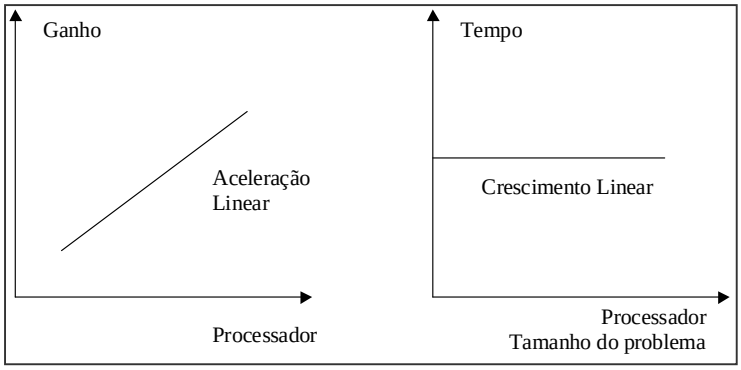


Figura 1 - Métricas de um sistema paralelo

O processamento paralelo consiste na utilização de múltiplos processadores para realizar uma determinada computação. O modo como os processadores e os dispositivos de memória comunicam entre si, definem a arquitetura da máquina paralela. Os principais modelos de memória são (BUYYA, 1999): memória compartilhada ("shared memory") e memória distribuída ("shared nothing").

2.1 Memória compartilhada

Neste tipo de arquitetura (figura 2), todos os processadores tem acesso a memória principal e as unidades de disco através de uma rede interconexão.

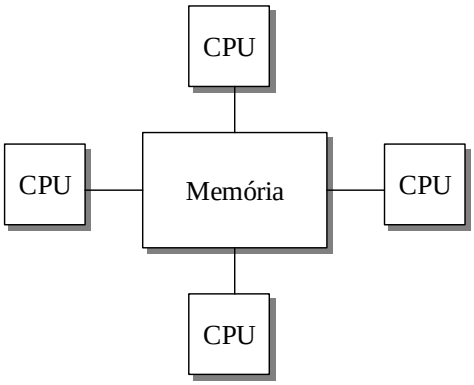


Figura 2 - Modelo de memória compartilhada

A principal vantagem da memória compartilhada é o compartilhamento dos dados entre as tarefas de forma rápida. Como desvantagens podem ser citados a expansibilidade limitada e a disponibilidade. A adição de mais processadores aumenta o trafego entre a memória e os nós além de ser limitada a um número pequeno (ordem de dezenas) de processadores.

2.2 Memória distribuída

Nesta arquitetura (Figura 3), cada nó tem exclusivo acesso a sua memória. Desta forma cada processador opera independentemente não existindo endereçamento global do espaço de memória. Quando um nó necessita de dados residentes em outro nó, é responsabilidade do programador explicitamente definir como obte-lo.

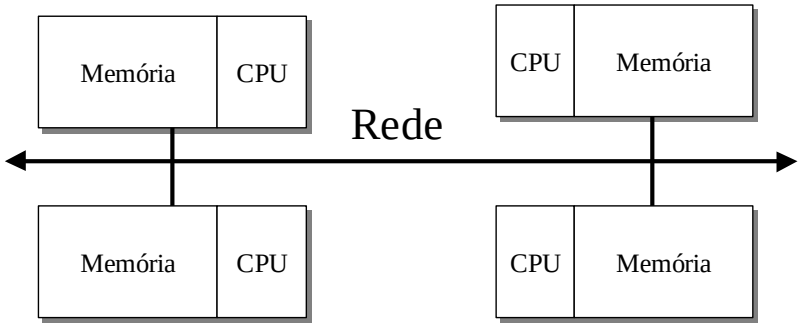


Figura 3 - Modelo de memória distribuída

A principal vantagem da memória distribuída são o baixo custo aliado a uma alta expansibilidade e disponibilidade. Permitindo um crescimento incremental, consegue suportar um número enorme de processadores minimizando interferências através da redução de

recursos compartilhados. Como desvantagens podem ser citadas a responsabilidade legada ao programador da aplicação, que necessita programar a troca de mensagens associadas a comunicação entre os nós, além de poder ser difícil mapear estruturas de dados existentes baseadas em uma memória global para este tipo de organização.

2.3 Paradigmas de programação

É amplamente reconhecido que as aplicações paralelas podem ser classificadas em um conjunto bem definido de paradigmas de programação (BUYYA, 1999). Os mais populares são: *Mestre/Escravo, Um Programa Múltiplos Dados, Fluxo de Dados, Dividir e Conquistar*.

2.3.1 Mestre/Escravo

Este paradigma consiste de duas entidades: mestre e escravos. O mestre é o responsável pela decomposição do problema em tarefas menores, pela distribuição destas tarefas entre os processos escravos e pela coleta dos resultados intermediários para poder produzir o resultado final da computação. Os escravos por sua vez, recebem uma mensagem com a tarefa, processam a tarefa e retornam o resultado para o mestre. Normalmente, a comunicação entre os processos é feita somente entre o mestre e os escravos.

O balanceamento de carga entre os nós pode ser feito de forma estática ou dinâmica. No primeiro caso, a distribuição das tarefas aos escravos é feita no início da computação. No balanceamento dinâmico, a distribuição é feita durante a execução da aplicação. Ela é mais apropriada quando o número de tarefas excede o número de processadores, quando o número de tarefas é desconhecido no início da computação ou ainda, quando o problema é desbalanceado. Este paradigma pode alcançar elevadas acelerações lineares e graus de expansibilidade. Entretanto, se o número de processadores for muito grande, o controle centralizado exercido pelo mestre pode constituir um gargalo no sistema.

2.3.2 Um Programa Múltiplos Dados (SPMD)

Este paradigma, também chamado de paralelismo de dados, é o mais popular e utilizado em diversas áreas da ciência da computação. Cada nó basicamente executa o mesmo pedaço de código porém em uma parte diferente dos dados. Isto envolve na divisão dos dados da aplicação entre os processadores disponíveis. SPMD pode ser bem eficiente caso o dado esteja bem distribuído entre os processadores e o sistema for homogêneo. Entretanto, ele é sensível a perda de algum processador, visto que em geral, os processos comunicam-se entre si. Neste caso, uma única perda é suficiente para causar a impossibilidade de concluir a computação.

2.3.3 Fluxo de dados

Este paradigma é baseado na decomposição funcional das tarefas que compõe a aplicação em porções que podem ser executadas concorrentemente. Cada nó representa um estágio no fluxo e é responsável pela sua tarefa. A comunicação entre os processos pode ser assíncrona.

2.3.4 Dividir e conquistar

Neste caso, um problema é dividido em subproblemas. Cada subproblema é resolvido independentemente e os seus resultados são combinados para fornecer o resultado final. É uma modificação do paradigma mestre-escravo. Ao invés de haver um processo mestre responsável pela divisão da carga de trabalho, dois ou mais nós realizam a distribuição das tarefas em uma hierarquia.

3. Computação em Cluster

Um *cluster* pode ser definido como um tipo de sistema de processamento paralelo ou distribuído, que consiste de uma coleção de computadores interconectados que trabalham em conjunto como se fossem um único recurso computacional (BUYYA, 1999). A figura 4 ilustra uma arquitetura típica de um *cluster*.

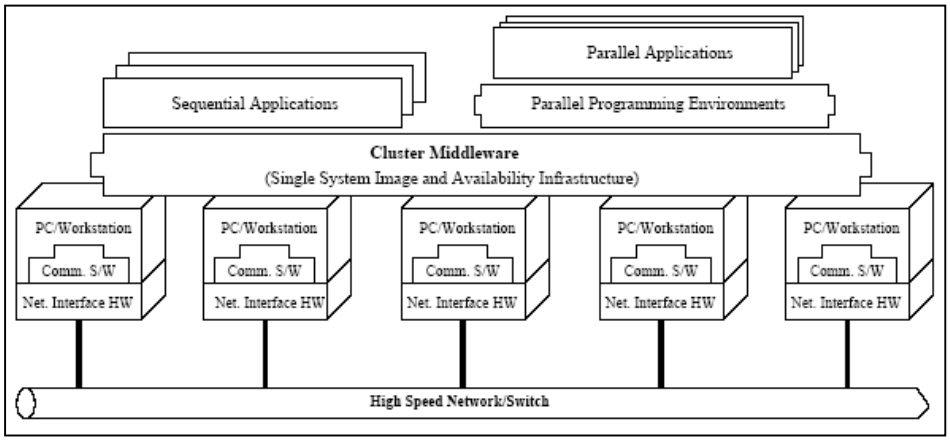


Figura 4 – Arquitetura de um cluster (BUYYA, 1999)

Cada nó de um *cluster* pode ser um único computador ou um computador com múltiplos processadores simétricos (Memória Compartilhada). Cada nó tem a sua própria memória, dispositivos de entrada/saída e sistema operacional. Em termos físicos, um cluster pode estar montado em um único gabinete, ou seus nós podem estar fisicamente separados e conectados via uma rede local (LAN). Com relação ao seu tamanho, existem hoje em dia *clusters* com um pequeno número de nós até aqueles encontrados nos grandes centros de

pesquisa do mundo, que são constituídos por milhares de processadores. Tipicamente, um *cluster* irá aparecer como um sistema único para seus usuários e administradores.

Os *Clusters* vem sendo utilizados para resolver problemas de larga escala em diversas áreas científicas e comerciais. Este tipo de plataforma é construída utilizando-se componentes comuns ("hardware de prateleira") e "software" livre ou largamente utilizado, como por exemplo os sistemas Linux e Windows. O seu surgimento remonta ao início da década de 90, motivado pelos baixos custos dos seus componentes - processador, redes de inter-conexão e memória, assim como do surgimento de ferramentas de alto desempenho para computação distribuída e do aumento da necessidade de poder de computação das aplicações científicas e comerciais.

A construção de um *cluster* é tida como extremamente fácil. Entretanto, fazer com que computadores individuais operem como um sistema integrado com o propósito de resolver um dado problema é mais difícil. Os maiores desafios residem no desenvolvimento de aplicações que tirem proveito da infra-estrutura do *cluster*. As aplicações necessitam ser "paralelizadas", ou seja, precisam estar aptas a tirar proveito do processamento paralelo e serem desenvolvidas com as ferramentas adequadas ao ambiente.

Várias alternativas para a paralelização de código tem sido desenvolvidas. Entretanto, estratégias baseadas na troca de mensagens explícitas tem tido mais aceitação e sucesso em uma grande variedade de aplicações e plataformas. Os mais populares ambientes e interfaces de programação paralela são o PVM ("Parallel Virtual Machine") (GEIST et al., 1994) e o MPI ("Message Passing Interface") (SNIR et al. 1996). As aplicações paralelas codificadas com estas duas biblioteca são altamente portáveis podendo serem executadas em ambientes que variam desde computadores portáteis até supercomputadores. Ambas bibliotecas são gratuitas e estão disponíveis para uma grande gama de plataformas computacionais e permitem a confecção de programas paralelos utilizando as linguagens de programação C, C++ e Fortran.

3.1 - MPI

O MPI fornece uma biblioteca para troca de mensagens portável, eficiente e flexível. O objetivo desta subseção é fornecer uma visão geral, através de um exemplo, de como é construído uma aplicação paralela com esta ferramenta. O paradigma a ser explorado é o mestre/escravo, visto ser o mais utilizado com o MPI. A estrutura geral de um programa MPI é mostrado na figura 5. Basicamente, um programa em MPI utiliza um conjunto de seis rotinas básicas: MPI_Init, MPI_Comm_size, MPI_Comm_rank, MPI_Send, MPI_Recv e MPI_Finalize.

- MPI_Init – inicializa o ambiente de execução. Esta rotina deve ser chamada em todo programa um única vez.
- MPI_Comm_size – informa o número de processos a serem criados.
- MPI_Comm_rank – informa a identificação (sequencial numérica) de cada processo.
- MPI_Send – operação para o envio de mensagem para outro processo.
- MPI_Recv – operação para o recebimento de uma mensagem.
- MPI_Finalize – termina o ambiente de execução. Assim como a rotina *init*, deve ser chamada uma única vez.

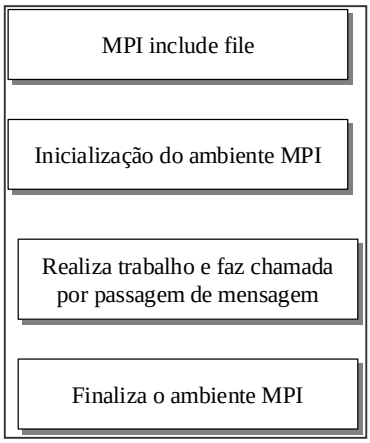


Figura 5 - Estrutura de um programa MPI

```
#include <stdio.h>
#include "mpi.h"
main(int argc, char **argv)
{
        int rank, size, tag, rc, i;
        MPI_Status status; char message[20];
        rc = MPI_Init(&argc, &argv);
        rc = MPI_Comm_size(MPI_COMM_WORLD, &size);
        rc = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        tag = 100;
        if(rank == 0)
        {
                strcpy(message, "Hello, world");
                for (i=1; i<size; i++)
                    rc = MPI_Send(message, 13, MPI_CHAR, i, tag,
                    MPI_COMM_WORLD);
        }
        else rc = MPI_Recv(message, 13, MPI_CHAR, 0, tag,
        MPI_COMM_WORLD, &status);
        printf( "node %d : %.13s\n", rank,message);
        rc = MPI_Finalize();
}
```

Figura 6 – Programa C/MPI

O exemplo na figura 6 ilustra um programa C/MPI. Neste exemplo, cópias do mesmo programa executam em diferentes nós. Cada processo faz a sua inicialização (MPI_INIT), determina o número de processos (MPI_COMM_SIZE) e obtém a sua identificação (MPI_COMM_RANK). Um dos processos (RANK=0) envia mensagens (MPI_SEND). Os demais processos recebem a mensagem enviada e todos os processos imprimem a sua identificação e saem do MPI (MPI_FINALIZE).

4. GRID

O modelo de computação em Grid tem sido descrito fazendo-se uma analogia as redes de transmissão de eletricidade. Quando ligamos algum aparelho elétrico na tomada, nós não fazemos idéia de onde a eletricidade é gerada. A companhia elétrica responsável pelo fornecimento deste serviço fornece uma interface para um sistema complexo de geração e transmissão. A visão de Grid é similar (ou almeja ser), ou seja, diversos recursos computacionais geograficamente distribuídos podem ser agregados para formar um supercomputador virtual. Porém, estes recursos não necessitam ser visíveis ao usuário, da mesma forma que um consumidor não tem visibilidade de onde a eletricidade disponível em sua residência é gerada. A infra-estrutura e os recursos tecnológicos que formam um Grid devem suportar o compartilhamento e o uso coordenado destes recursos. A figure 7 ilustra o ambiente de Grid do ponto de vista do usuário.

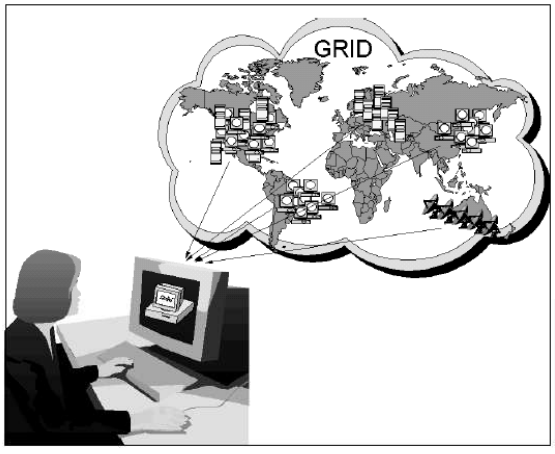


Figura 7 - O Grid do ponto de vista do usuário

A Internet fornece uma infra estrutura de rede que conecta milhares de computadores espalhados pelo mundo. A WEB, é um serviço de compartilhamento de informações construído sobre a Internet. A Internet também é usada para o correio eletrônico e para a troca de arquivos. De forma análoga, o Grid também é um serviço construído sobre a Internet.

Porém o Grid vai um passo adiante pois através de sua tecnologia, além de informação, processadores, unidades de disco, instrumentos, programas, bancos de dados, etc., também podem ser compartilhados (FOSTER, KESSELMAN, 2002).

Para que este ambiente possa funcionar, é necessário estabelecer padrões. O "*Global Grid forum*" é uma entidade que reúne desenvolvedores e usuários do Grid e tem por meta "promover e dar suporte ao desenvolvimento, instalação e implementação de tecnologia e aplicações de Grid através da criação de especificações técnicas e da documentação das melhores práticas". A principal especificação do GGF é a arquitetura de serviços denominada "*Open Grid Services Architecture*" (OGSA). Na OGSA, os recursos computacionais, de armazenamento, redes, bancos de dados são tratados como serviços. Um serviço de Grid ("*Grid Service*") é um serviço Web com um conjunto definido de interfaces. Um serviço Web por sua vez pode ser definido como um sítio na Internet que é acessado por programas ao invés de pessoas. Nas próximas subseções, são descritos as principais características dos sistemas de infra estrutura básica para desenvolvimento e utilização de aplicações em Grid.

4.1 Globus Toolkit

Diversos sistemas de infra estrutura básica para Grid vem sendo desenvolvidos. No entanto, o "Globus Toolkit" vem sendo adotado como o padrão de fato. O "Globus Toolkit (GT4)" (FOSTER, 2005) é um sistema aberto, desenvolvido em conjunto por diversas instituições de pesquisa, universidades e colaboradores individuais, e provê um conjunto de recursos para a construção de sistemas computacionais em Grid.

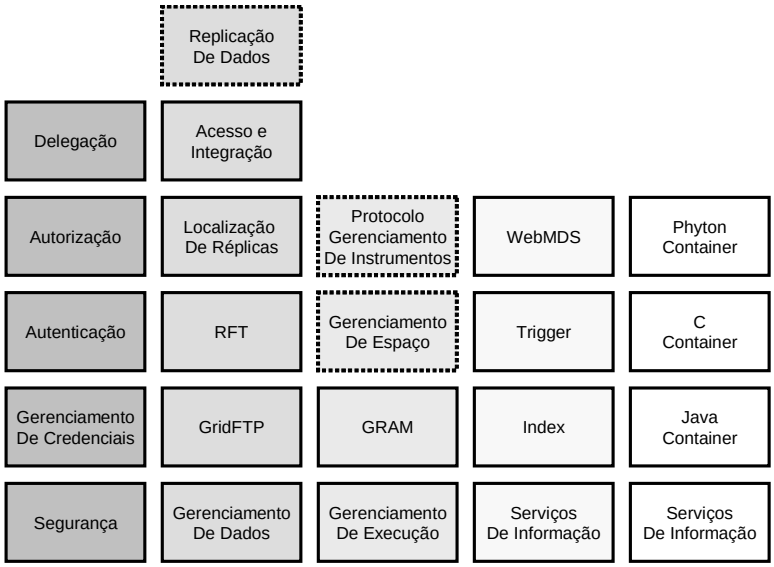


Figura 8 - Componentes principais do GT4 (adaptado de FOSTER, 2005)

A figura 3 ilustra os principais componentes do GT4. Um conjunto de serviços de infra-estrutura implementa interfaces para segurança de acesso, gerenciamento de dados, gerenciamento e execução de programas, monitoramento e descobrimento de recursos, além de ferramentas para instalação e execução de serviços Web desenvolvidos nas linguagens Java, C e Phyton. As caixas com bordas pontilhadas indicam serviços disponibilizados no *toolkit*, mas ainda em fase de testes.

4.1.1 Segurança

Os componentes de segurança do GT4 implementam protocolos para proteção de mensagens, autenticação de usuários, e delegação. Por *default*, todo usuário e recurso computacional possui uma credencial de chave pública. Cada entidade pode validar a credencial de outra entidade, usar estas credenciais para estabelecer um canal seguro de comunicação e agir em nome de um usuário remoto por um determinado período de tempo.

4.1.2 Gerenciamento de Dados

Os componentes para o gerenciamento de dados do GT4 são: *GridFTP, RFT, RLS, OGSA-DAI* e *DRS*. O *GridFTP* implementa um serviço de alto desempenho para transferência de arquivos, enquanto que o serviço *RFT* (*Reliable File Transfer*) gerencia múltiplas transferências de arquivos que utilizem o *GridFTP*. O registro e a localização de réplicas são implementados pelo serviço *RLS* (*Replica Location Service*). O serviço *OGSA-DAI* (*Globus Data Access and Integration*) possibilita o acesso e a integração de dados residentes em banco de dados relacionais ou em formato XML. O *DRS* (*Data Replication Service*) combina os recursos do *GridFTP* e do *RLS* para fornecer um serviço para replicação de dados.

4.1.3 Gerenciamento de execução

O *GRAM* (*Globus Resource and Allocation Manager*) é o componente básico do GT4 para submeter, monitorar e controlar tarefas em computadores remotos e possui interface para vários escalonadores como Condor e PBS. O gerenciamento de espaço é responsabilidade do componente *WMS* (*Workspace Management Service*) enquanto que o *GTPC* (*Grid TeleControl Protocol*) é um serviço voltado para a gerência instrumentos e tem sido usado em microscópios e equipamentos para detecção de terremotos.

4.1.4 Serviços de informação

Os serviços de informação permitem que o usuário obtenha uma descrição dos recursos do Grid. O GT4 fornece dois serviços agregadores que coletam dados de fontes de informação: *Index Service* e *Trigger Service*. O primeiro serviço prove informações sobre o estado de outros serviços no Grid, enquanto que o segundo pode ser usado para construir notificações por email para avisar administradores de sistemas em caso de falhas. O serviço *WebMDS* pode usado para criar visões especializadas dos dados coletados pelo *Index Service*.

4.1.5 Construção de serviços

O GT4 inclui software para possibilitar o desenvolvimento de componentes que implementem interfaces para serviços Web. O GT4 disponibiliza "containers" para a instalação e execução de serviços Web escritos em Java, C ou Phyton.

4.2 Condor/DagMan

Um segundo sistema muito utilizado em ambientes distribuídos é o Condor (THAIN, D., TANNENBAUM, T., LIVNY, M., 2005). Condor é um sistema em lote especializado no gerenciamento de tarefas que executem em estações de trabalho que se comuniquem através de uma rede. Os usuários submetem suas tarefas para o Condor, que as coloca em uma fila, executa-as e informa o resultado ao usuário quando as tarefas terminam. Condor descobre as máquinas para executar os programas, mas não faz o escalonamento das tarefas baseado em dependências.

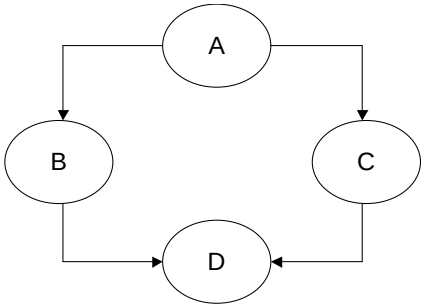
Condor-G (FREY et al., 2001) é uma extensão do Condor, e permite que as tarefas sejam descritas no arquivo de submissão do Condor. Porém, quando submetidas para execução, o Condor utiliza a interface Globus GRAM para submeter à tarefa para execução em um recurso remoto.

DagMan (*Directed Acyclic Graph Manager*), (CONDOR TEAM, 2003), é um meta-escalonador para tarefas Condor. DagMan submete tarefas para o Condor na ordem representada pelo grafo e processa os resultados. O grafo de tarefas é definido em um arquivo de entrada que especifica a lista dos programas no grafo, pré e pós-processamento para cada programa, a descrição das dependências entre os programas e o número de re-execuções para o caso de falhas. DagMan é um meta-escalonador para tarefas Condor. DagMan submete tarefas para o Condor na ordem representada pelo grafo de tarefas e processa os resultados. O grafo que representa o workflow é definido em um arquivo e especifica a lista das tarefas, a

localização das rotinas para pré e pós- processamento de cada tarefa, caso existam, a descrição das dependências e o número de re-execuções para o caso de falhas.

A figura 9 ilustra um arquivo de descrição com o respectivo DAG e um arquivo de submissão da tarefa do Condor. Cada nó do DAG representa uma tarefa que é descrita no arquivo Condor para submissão. Este arquivo define os locais onde se encontra o executável além dos arquivos de saída, erros e log. DagMan também permite que o número de rotinas de pré e pós-processamento bem como o número de tarefas executando concorrentemente seja definido por meio de parâmetros de execução. Uma vez submetido, o workflow pode ser monitorado através de comandos que listam o estado das tarefas.

```
# Filename: diamond.dag
#
Job A A.condor
Job B B.condor
Job C C.condor
Job D D.condor
#
Script PRE A top_pre.csh
Script PRE B mid_pre.perl $JOB
Script POST B mid_post.perl $JOB
$RETURN
Script PRE C mid_pre.perl $JOB
Script POST C mid_post.perl $JOB
$RETURN
Script PRE D bot_pre.csh
#
PARENT A CHILD B C
```




```
# Filename:diamond_job.condor
executable = /path/diamond.exe
output = diamond.out.$(cluster)
error = diamond.err.$(cluster)
log = diamond_condor.log
universe = vanilla
```

Figura 9 – Arquivos de descrição do DAG e submissão de tarefa (CONDOR TEAM, 2003)

5. Conclusão

As características de processamento dos dados encontradas nas aplicações científicas e comerciais atuais demandam cada vez mais recursos computacionais para serem atendidas em tempo hábil. Os ambientes para o processamento distribuído e paralelo constituem uma nova realidade e vem sendo utilizados para prover um melhor desempenho e para compartilhar os diversos recursos. Este trabalho apresentou uma visão geral destes ambientes e algumas de suas ferramentas básicas de uso.

O baixo custo e o crescimento incremental fazem com que os "*clusters*" de computadores sejam uma alternativa viável para aumentar a capacidade de processamento. Entretanto, de

nada adianta dispor de um "*hardware*" paralelo se o "*software*" não tirar proveito desta arquitetura. Se por um lado diversos fabricantes como a Oracle e a IBM já possuem versões de seus sistemas explorando estas arquiteturas, os programadores de aplicação em geral, desconhecem como trabalhar neste tipo de ambiente.

Um Grid por sua vez, pode ser visto como um "*cluster*" de "*clusters*". Apesar dos esforços da comunidade científica e acadêmica, a sua utilização e administração não é trivial e ainda requer uma mão de obra altamente especializada (MEYER et al., 2006). No entanto, na medida em que as barreiras para utilização desta tecnologia vierem a ser superadas, a construção destes ambientes irá extrapolar os ambientes de computação científica atuais e se tornarão uma realidade no processamento e acesso aos dados dentro e entre as corporações.

Referêcias Bibliográficas

BUYYA, R., 1999, "Parallel Programming Models and Paradigms" In: "*High Performance Cluster Computing*", v.1, pp 4-26, Prentice Hall

CONDOR TEAM, 2003 http://www.cs.wisc.edu/condor/manual/v6.4/ref.html

FOSTER, I., 2005, "Globus Toolkit Version 4: Software for Services Oriented Systems", In: *International Conference on Network and parallel Computing, Lecture Notes in Computer Science*, v. 3779, pp. 2-13.

FOSTER,I., KESSELMAN, C., 1999, "Computational Grids", In: Foster, Kesselman, *The Grid: Blueprint for a New Computing Infrastructure*, first edition, chapter 2, Morgan-Kaufman.

FREY, J., TANNENBAUM, T., FOSTER, I., LIVNY, M., TUECKE, S., 2001, "Condor-G: A Computation Management Agent for Multi-Institutional Grids", In: *Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing (HPDC10)*, pp. 237-246, San Francisco, California.

GEIST, A., et al., 1994, "*PVM:Parallel Virtual Machine*", MIT Press.

MEYER, L., MATTOSO, M., FOSTER, I., et al., 2006b, "An Opportunistic Algorithm for Scheduling Workflows on Grids", to appear In: *Proceedings of VECPAR'06*, Rio de Janeiro,Brazil,July.

SNIR, M., et al., 1996, "*MPI: The Complete Reference*", MIT Press.

THAIN, D., TANNENBAUM, T., LIVNY, M., 2005, "Distributed computing in practice: the Condor experience ", *Concurrency and Computation: Practice and Experience*, v. 17, n. 2-4, pp. 323-356, February-April