Programação Científica Estruturada
Linguagem C, Paralelização, Cluster/Grid

Nilton Alves, Deyse Peixoto e *Alexandre Maia*

Centro Brasileiro de Pesquisas Físicas – CBPF/MCT
Rua Dr. Xavier Sigaud 150
22290-180 - Rio de Janeiro, RJ – Brasil

RESUMO

Este texto contém as notas de aula do curso de mesmo nome da Escola do CBPF - 2004. São apresentados de forma didática os fundamentos da programação científica estruturada, serial e paralela, utilizando a linguagem C e a biblioteca MPI - *Message Passing Interface*. Serão também apresentados os conceitos que formam a base da tecnologia Cluster/Grid. Os itens que serão abordados são: 1. HISTÓRIA da Linguagem C/C++, 2. Introdução aos 7 ELEMENTOS BÁSICOS DA PROGRAMAÇÃO, 3. Conceitos de REDES, 4. Sistema CLUSTER, 5. PROGRAMAÇÃO PARALELA e MPI, 6. Tecnologia GRID e por fim, 7. Sistema GLOBUS.

1 História da Linguagem C

A linguagem C é uma linguagem de programação de uso geral que foi originalmente desenvolvida por Dennis Ritche - Bell Las e implementada para o computador PDP-11 em 1972. Muitos dos seus princípios e características são originários das linguagens B, BCPL - Basic CPL e CPL - Combined Programming Language. Estas linguagens tinham como principal objetivo serem de alto nível, independentes do hardware e permitirem o controle primário da informação, os bits.

Foi primeiramente usada como linguagem de sistema para o sistema operacional UNIX. Em 1970, Ken Thompson, usou as linguagens ASSEMBLER e uma outra chamada B para produzir versões iniciais do UNIX. O desenvolvimento da linguagem C foi uma consequência das limitações da linguagem B que era baseada em uma linguagem anterior chamada BCPL. Esta linguagem, por sua vez, foi desenvolvida para sistemas que não utilizavam fitas por Martin Richards e tinha como tipo básico os códigos de máquina e os ponteiros de memória e endereços aritméticos. Posteriormente a linguagem C incorporou a manipulação de caracteres aos tipos básicos das suas linguagens de origem. A linguagem C possui duas características principais: a portabilidade e a facilidade de manipulação dos bits.

A linguagem C++ foi escrita por Bjarne Sroustrup - Bell Labs no período de 1983-85. C++ é uma extensão da linguagem C e que inicialmente combinava a capacidade de trabalhar com classes e aspectos orientados a objeto da linguagem SIMULA (Kristen Nygaard - NCC, 1962) e a eficiência e performance de C.

As várias implementações da linguagem C/C++ estão baseadas no padrão ANSI C/C++ e acrescentam recursos específicos e proprietários que não abordaremos aqui.

2 Introdução aos Sete Elementos Básicos da Programação

Programas computacionais de aplicações científicas normalmente resolvem problemas numéricos, escritos em linguagem FORTRAN e mais recentemente em C/C++. Estes programas na maioria dos casos manipulam dados ou informações de entrada, efetuam sequências de atribuições e operações, tomam decisões de acordo com certas condições, executam procedimentos e por fim emitem dados de saída que são a solução do problema. Todo este procedimento pode ser classificado em sete elementos básicos característicos de todas as linguagens, que possuem também

características adicionais e específicas de cada uma delas. Uma boa maneira de aprender rapidamente uma linguagem de programação é entender o funcionamento de cada um destes elementos básicos. A seguir é apresentada uma breve descrição de cada um deles.

Output: significa escrever a informação na tela, em discos ou em uma porta de I/O. As principais funções utilizadas são aquelas de escrever na tela do computador.

Tipos de Dados: são constantes, variáveis e estruturas contendo números inteiros ou reais, textos (caracteres únicos ou em sequência) ou ainda endereços de memória (ponteiros).

Operações: atribuir um valor a uma variável (ou uma função, um ponteiro, etc), valores combinados (adição, subtração, etc) e valores comparados (igual, não igual, maior, etc).

Input: significa ler a informação do teclado, discos ou de uma porta de I/O.

Condicionais: refere-se a execução ou não de uma sequência de instruções se uma condição específica for satisfeita.

Laços: executa uma sequência de instruções um número fixo de vezes ou de acordo com alguma condição.

Subrotinas: conjunto de instruções que recebem um nome dedicado e que pode ser executado em qualquer parte do programa um número qualquer de vezes.

Devido a grande variedade de documentação disponível em livros e na Internet, faremos aqui uma breve abordagem não usual, porém útil para o aprendizado da linguagem C/C++ que neste curso deve ser rápido e específico sem envolver detalhamentos e características de pouca utilização. Algumas boas sugestões de documentação são apresentadas no item Referências.

2.1 Programa Base

Nesta seção, será apresentado o código fonte de um programa que envolve as principais características da linguagem C/C++ usualmente utilizadas em cálculos computacionais envolvendo dados científicos. Serão apresentados ainda os sete elementos básicos da programação.

Primeiramente vamos definir o problema físico e entender o objetivo do programa. O problema principal é determinar a condutividade térmica k de uma amostra polimérica com 25% em massa de lignina utilizando o método do fio quente. Neste método, o aparato é constituído de uma amostra que possui em seu interior uma resistência elétrica ligada a uma fonte de energia e um sensor de temperatura. A energia térmica cedida pela resistência será dissipada na amostra através do efeito Joule. O experimento envolve medir a variação da temperatura no interior da amostra em função do tempo, a medida que a dissipação for ocorrendo.

As medidas do tempo t em segundos (s) estão no corpo do código fonte e da temperatura Θ em graus Celsius ($^{\circ}C$) em um arquivo externo que será lido. A dependência entre estas variáveis é dada pelas seguintes relações

$$\boxed{\Theta(t) = a_0 + a_1 \ln t \qquad a_1 = q'/(4\pi k)}$$

A condutividade térmica k é obtida através do coeficiente angular a_1 do ajuste linear utilizando o método dos mínimos quadrados. Este método envolve a resolução de um sistema linear que neste caso é resolvido pelo método de Cramer.

Abaixo é apresentado o código fonte do programa anteriormente citado, onde os seus diversos blocos serão explicados e os diversos aspectos da linguagem comentados.

```
1    // Centro Brasileiro de Pesquisas Fisicas - CBPF/MCT
2    // V Escola do CBPF
3    // Rio de Janeiro, 5 a 16 de julho de 2004
4    // Programacao Cientifica Estruturada
```

```
5   // Prof. Nilton Alves
6   // Metodo Minimos Quadrados - Ajuste Linear
7
8   //------------------------------- Includes
9   #include <iostream>
10   #include <fstream>
11   #include <math.h>
12   #include <string.h>
13
14   using namespace std;
15
16   //------------------------------- Defines
17   #define NPTS 20
18   #define QL 6.46
19   #define PI 3.14159265
20
21   //------------------------------- Variables
22   double t[]={60,62,64,66,68,70,72,74,76,78,80,82,84,86,88,90,92,94,96,98};
23   double teta[NPTS];
24   double x[NPTS];
25   double sumx, sumx2, sumy, sumxy;
26   double delta,a0,a1,k;
27   double erro2;
28   int i;
29   char tecla;
30
31   //------------------------------- Functions
32   void LeTeta (void);
33   void CalculaX (void);
34   void CalculaSum (void);
35   void CalculaVar (void);
36   double CalculaQui2 (void);
37   void Saida (void);
38
39   //------------------------------- main
40   int main()
41   {
42   LeTeta();
43   CalculaX();
44   CalculaSum();
45   CalculaVar();
46   erro2=CalculaQui2();
47   Saida();
48   tecla=getchar();
49   }
50
51   //------------------------------- LeTeta
52   void LeTeta()
53   {
54   ifstream arqin;
```

```
55
56      arqin.open("teta.txt");
57      if(arqin==NULL)
58          {
59          cout << "Problemas com abertura de arquivo!!";
60          exit (9999);
61          }
62      for(i=0;i<NPTS;i++)
63          arqin >> teta[i];
64      arqin.close();
65      }
66
67      //------------------------------ CalculaX
68      void CalculaX()
69      {
70      for(i=0;i<NPTS;i++)
71          x[i]=log(t[i]);
72      }
73
74      //------------------------------ CalculaSum
75      void CalculaSum()
76      {
77      int i=0;
78
79      sumx=sumx2=sumy=sumxy=0;
80      while(i<NPTS)
81          {
82          sumx += x[i];
83          sumx2 += x[i]*x[i];
84          sumy += teta[i];
85          sumxy += x[i++]*teta[i];
86          }
87      }
88
89      //------------------------------ CalculaVar
90      void CalculaVar()
91      {
92      delta=NPTS*sumx2-sumx*sumx;
93      a0=(sumy*sumx2-sumxy*sumx)/delta;
94      a1=(NPTS*sumxy-sumx*sumy)/delta;
95      k=QL/(4*PI*a1);
96      }
97
98      //------------------------------ CalculaQui2
99      double CalculaQui2()
100      {
101      double qui, qui2;
102
103      for(i=0;i<NPTS;i++)
104          {
```

```
105         qui=a0+a1*x[i]-teta[i];
106         qui2+=qui*qui;
107         }
108     return (qui2);
109     }
110
111     //————————————————- Saida
112     void Saida()
113     {
114     // saida para tela
115     cout << "\n \n Qui2= "<< erro2 << "A0= "<< a0 << "A1= "<< a1 << "k= "<< k
<< endl;
116
117     // saida para arquivo
118     ofstream arqout;
119     arqout.open("arquivo_saida");
120     if(arqout==NULL)
121         {
122         cout << "Problemas com abertura de arquivo!!";
123         exit (9999);
124         }arqout << "Qui2= "<< erro2 << "A0= "<< a0 << "A1= "<< a1 << "k= "<<
k << endl;
125     arqout.close();
126     }
```

2.2 Comentários

É importante observar que a numeração das linhas do programa na realidade foi colocada somente para ser referênciada e portanto não está relacionada com a sintaxe da linguagem.

Uma segunda observação a ser feita é com relação a identação. Ela pode ser qualquer uma das usuais porém única ao longo do programa. Visualmente deve ser agradável e passar informações relativas aos vários níveis de aninhamento.

O procedimento usual é a edição de um arquivo com terminação .c ou .cc para posterior compilação e se tudo estiver ok, a execução do programa.

As primeiras linhas começam com o identificador \\ que informa ao compilador que a referida linha não deve ser considerada pois contém comentários para simples documentação. Se quiser comentar mais de uma linha consecutivas utilize /* e */, antes e depois das linhas a serem comentadas respectivamente.

A sequência do programa pode ser descrita em blocos e funções definidas pelas linhas de comentário que antecede cada um deles.

Includes: as primeiras linhas do código fonte começam com o caracter # que indica a utilização de diretivas de preprocessamento. A diretiva *#include* instrui ao compilador para incluir um outro arquivo fonte que normalmente apresenta definições e cabeçalhos de funções que serão utilizadas ao longo do programa.

Defines: que define um identificador e uma string que será substituída pelo identificador cada vez que for utilizada no código fonte. Neste caso funciona como definição de constantes.

Variables: este bloco do programa define as variáveis globais que são conhecidas e que serão utilizadas pelas funções do programa. Existem as variáveis locais, definidas dentro das funções e portanto só conhecidas e utlizadas por elas. As variáveis podem ser de vários tipos e com intervalos

característicos definidos pelo número de byte específicos (vide tabela a seguir).

Type	N. bytes	Intervalo
int	2 ou 4	-2147483648 até 2147483647
short int	2	-32768 até 32767
long int	4	-2147483648 até 2147483647
float	4	3.4e+/–38 (7 digitos)
double	8	1.7e+/-308 (15 digitos)
char	1	-128 até 127
bool	1	True ou False

As primeiras variáveis, [22]-[24], na realidade são conjuntos de variáveis de mesmo tipo agrupadas e representadas pelo mesmo identificador (t, *teta*, x), também chamadas de vetores. Observe a possibilidade de definir o conteúdo (valor) de cada elemento do vetor no caso do vetor t ou de somente especificar o tamanho (NPTS) do vetor (*teta* e x) para posterior atribuição.

Observe ainda a possibilidade de definir mais de uma variável na mesma instrução desde que sejam separadas por vírgula. O espaço é somente um padrão a ser utilizado ou não ([25] e [26]). Vale ressaltar que cada instrução deve terminar com ; (ponto e vírgula).

O bloco a seguir, *Functions*, mostra o cabeçalho de cada uma das funções que foram desenvolvidas e serão utilizadas. O cabeçalho deve conter o tipo da função e o tipo dos parâmetros que podem ou não ser passados para as funções.

A parte principal do programa é a função *main* que assim como qualquer função possui cabeçalho e o conjunto de instruções deve estar contido entre chaves, {instrução1; instrução2; ...}. A leitura do corpo desta função deve informar as diversas etapas do algoritmo e portanto a escolha dos nomes das funções é fundamental. Descreveremos agora estas funções e as características da linguagem utilizadas.

LeTeta: a primeira linha define uma variável do tipo ifstream (*input file stream* que será utilizada com as funções (métodos na linguagem C++) *open* e *close*. A estrutura condicional *if(condição)declarações* verifica se houve algum problema com a abertura do arquivo de dados que contém os valores do vetor t. Na linha [59], a função *cout* junto com o operador << é uma das funções da linguagem C++ utilizadas para saída de dados. A principal característica desta função é a simplicidade de uso, pois não requer formatação ao contrário da função *printf()* da linguagem C. A penúltima parte desta função é a leitura de dados e atribuição aos elementos do vetor teta[] utilizando um laço do tipo determinístico onde é conhecido o número de repetições. Ao final, o fechamento do *stream* arquin.

CalculaX: contém um laço onde é atribuído aos elementos do vetor x[] os valores do algoritmo natural dos elementos do vetor t[] necessários na resolução do nosso problema. Observe nesta função e na anterior o operador aritmético de incremento ++ que aumenta de uma unidade a variável a qual é aplicado. Existe também o operador de decremento –.

CalculaSum: calcula todos os somatórios relativos ao método de Cramer. Inicialmente é definida uma variável local do tipo *int* e atribuído valor zero na mesma instrução. As variáveis do somatório são globais sendo definidas externamente. Nesta função é utilizado um outro tipo de laço, onde o número de repetições é desconhecido e dependente de uma condição ser falsa ou verdadeira conforme o caso. Também foi utilizada a forma contraída do operador aritmético de adição += que significa $a = a+b \rightarrow a+ = b$. A mesma interpretação vale para as formas contraídas -=, *= e /=.

CalculaVar: nesta função são calculados os coeficientes linear a_0 e angular a_1 assim como a condutividade térmica k. Observe a utilização dos identificadores definidos anteriormente.

CalculaQui2: esta função calcula o erro *qui2*, ou seja, a soma dos quadrados das diferenças *qui* entre os pontos experimentais *teta*[] e os pontos ajustados $a_0 + a_1 \ln t$. Foi utilizado um laço determinístico com operadores aritméticos de incremento na forma contraída, variáveis locais *qui* e *qui*2, globais *i*. Observe que esta função é do tipo double e portanto deve retornar um valor para ser atribuído a uma variável na função *main*.

Saida: são apresentadas duas formas de exibir os resultados, na tela utilizando o método *cout* << e na forma de arquivo onde também é feita a verificação de erro na sua abertura. Verifique a semelhança entre este procedimento de escrita em arquivos e aquele outro de leitura de arquivos na função LeTeta().

A última linha da função *main()* é somente um truque pouco elegante para que o programa fique parado esperando a entrada pelo teclado da variável *tecla* utilizando a função *getchar()* definida no arquivo string.h. O mesmo efeito poderia ser implementado pela instrução *cin* << *tecla;*

Após o detalhamento deste código fonte, foi possível observar os sete elementos básicos da programação, assim como alguns procedimentos e técnicas de programação. Programar é uma arte e por isto deve ser desenvolvida, pois seu conhecimento é acumulativo. A linguagem C/C++ é muito poderosa devido principalmente a sua portabilidade e a possibilidade de acesso direto a memória através do uso de variáveis ponteiro[1]. Este tipo de variável não foi abordado devido a sua complexidade visto que o tempo disponível é limitado.

3 Tópicos Básicos de Redes

O conceito básico de redes de computadores está no compartilhamento da informação por dispositivos conectados utilizando placas de rede (*hosts*). Sendo assim, cada troca de informação compreende um dispositivo fonte e outro dispositivo destino diferenciados por endereços. Esta informação é transmitida através de conjunto de bytes[2] que possuem cabeçalho (endereço e parâmetros) e dados (informação) sendo encaminhandos por dispositivos especiais, comutadores e roteadores, que analizam o cabeçalho e decidem para que outro dispositivo conectado deve enviar o pacote.

3.1 Modelo de Camadas ISO/OSI

Na década de 70, o cenário dos ambientes de redes exibia uma grande variedade de sistemas operacionais, CPUs, interfaces de rede, tecnologias e várias outras variáveis, que, além de apresentarem grandes complexidades para implementação, possuíam origens (fabricantes) variadas, independentes e concorrentes. O modelo de referência ISO/OSI surgiu da necessidade de interoperabilidade entre os diversos dispositivos de rede existentes neste cenário.

Em 1977, a *International Organization for Standartization* - ISO, criou um sub-comitê para o desenvolvimento de padrões de comunicação para promover a interoperabilidade entre as diversas plataformas. Foi então desenvolvido o modelo de referência *Open Systems Interconnection* - OSI. É importante observar que o modelo OSI é simplesmente um modelo que especifica as funções a serem implementadas pelos diversos fabricantes em suas redes. Este modelo não detalha como estas funções devem ser implementadas, deixando que cada empresa/organização tenha liberdade.

O comitê ISO assumiu o método "dividir para conquistar", dividindo o processo complexo de comunicação em pequenas sub-tarefas (camadas), de maneira que os problemas passem a ser mais fáceis de tratar e as sub-tarefas melhor otimizadas. O modelo ISO/OSI é constituído por sete camadas, descritas sucintamente a seguir, de cima para baixo:

[1] Ponteiros guardam endereços de memória

[2] Um byte é constituído de 8 bits, que é a unidade mínima de informação

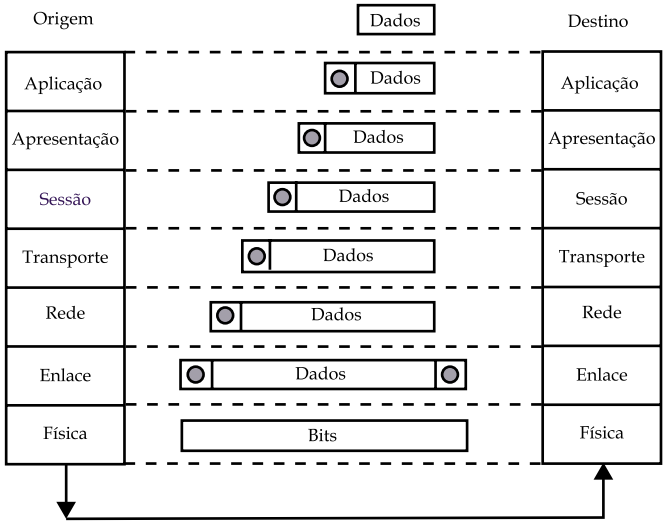

Figura 1: Modelo de Referência ISO/OSI

- **7. Aplicação**: Esta camada funciona como uma interface de ligação entre os processos de comunicação de rede e as aplicações utilizadas pelo usuário.
- **6. Apresentação**: Aqui os dados são convertidos e garantidos em um formato universal.
- **5. Sessão**: Estabelece e encerra os enlaces de comunicação.
- **4. Transporte**: Efetua os processos de sequenciamento e, em alguns casos, a confirmação de recebimento dos pacotes de dados.
- **3. Rede**: O roteamento dos dados através da rede é implementado aqui.
- **2. Enlace**: Aqui a informação é formatada em quadros (frames). Um quadro representa a exata estrutura dos dados fisicamente transmitidos através do fio ou outro meio.
- **1. Física**: Define a conexão física entre o sistema computacional e a rede. Especifica o conector, a pinagem, níveis de tensão, dimensões físicas, características mecânicas e elétricas, etc.

Cada camada se comunica com sua semelhante em outro computador. Considerando o *host* origem, quando a informação é passada de uma camada para outra inferior, um cabeçalho é adicionado aos dados. Cada cabeçalho contém campos que parametrizam as diversas funções de cada camada. O bloco de cabeçalho+dados de uma camada é o dado da próxima camada. Observe o esquema apresentado na Figura 1.

Considerando agora o *host* destino, cada camada interpreta e executa as funções relativas aos diversos campos do cabeçalho que foi colocado pela mesma camada no *host* origem. Após, o cabeçalho desta camada é retirado e a informação (dados) é passada para camada superior e assim sucessivamente até chegar na aplicação em questão.

3.2 Endereçamentos

Os *hosts* possuem dois endereços: um endereço físico MAC - *Media Access Control* e característico da placa de rede e um endereço lógico (IP - *Internet Protocol*) configurável no sistema operacional

do dispositivo. Os endereços MAC e IP possuem 6 bytes separados por espaços e expressos em hexadecimal (00 a FF) e 4 bytes separados por ponto e expressos em decimal (0 a 255), por exemplo 00-0C-76-4A-2B-76 e 152.84.59.250, respectivamente. Com estes endereços usados em situações diferentes, os *hosts* são identificados a assim os pacotes podem ser endereçados.

No endereço MAC os três primeiros bytes caracterizam o fabricante e o três últimos um número serial único, mas para os dispositivos este endereço é visto integralmente. No caso do endereço IP o significado é mais complexo.

O endereçamento IP é constituído de 4 bytes separados por ponto (.) e dividido em classes A, B e C. Existem ainda as classes D e E com utilização específica para transporte multicast e experimentos. Parte do endereço IP representa a rede e parte o endereço dentro da rede, que é o endereço do *hosts*. Sendo assim, as classes A, B e C possuem 1/3, 2/2 e 3/1 bytes de endereços de rede e de hosts, respectivamente. A caracterização do endereço de rede destas classes é feita através de uma máscara de bits: 255.0.0.0 (/8)- classe A, 255.255.0.0 (/16) - classe B e 255.255.255.0 (/24) classe C.

A tabela a seguir relaciona os diversos parâmetros relevantes na definição do endereço IP, como: o número de sistemas possíveis, os primeiros bits do primeiro octeto e os seus possíveis valores. Os demais octetos podem assumir livremente os valores entre 0 e 255.

Classe	Máscara	rede	host	Bits iniciais	Prim. Byte
A	255.0.0.0 - /8	128	$> 16.10^6$	0xxx	0-127
B	255.255.0.0 - /16	65.536	65.536	10xx	128-191
C	255.255.255.0 -24	$> 16.10^6$	256	110x	192-223
D	-	-	-	1110	224-239
E	-	-	-	1111	240-255

Existe a possibilidade de valores intermediários, por exemplo 255.255.255.128 (/25) que divide a classe C em duas redes, cada uma com 128 endereços. Os endereços 199.200.50.1-127/25 e 199.200.50.128-255/25 pertencem a redes diferentes. O endereço de rede e de *broadcast* são o primeiro e o último endereços disponíveis. Por exemplo no endereço 200.84.161.150/24 o endereço de rede é 200.84.161.0, o de *broadcast*[3] é 200.84.161.127 e o endereço do *host* é 200.84.161.150. Se a máscara for /25, o endereço de rede é 200.84.161.128, o de *broadcast* é 200.84.161.255 e o endereço do *host* é 200.84.161.150.

Analizando os aspectos topológicos e de equipamentos das redes de computadores, a figura que segue mostra que cada rede, LAN - *Local Area Network* é formada por diversos *hosts* (micros, servidores, impressoras, etc) que se comunicam através de um comutador que basicamente é um dispositivo com várias placas de rede. O comutador é o dispositivo encarregado de encaminhar os pacotes de informação dentro da própria LAN. Para conexão à Internet é necessário um roteador que faça a conexão com algum *backbone*[4], por exemplo RedeRio de Computadores - RR, Rede Nacional de Pesquisas - RNP, Embratel - EBT, etc. O roteador é um dispositivo que encaminha os pacotes para fora da LAN. Este dispositivo pode possuir também funções de *firewall*[5].

Cada rede deve possuir um endereço (convencionalmente o primeiro livre) neste dispositivo que encaminha os pacotes e que são chamados roteadores ou *gateways*. Este endereço assim como o endereço IP e sua máscara devem ser configurados em cada dispositivo participante da LAN (microcomputadores, servidores, impressoras, etc).

3.3 ARP - *Address Resolution Protocol*

Este protocolo é considerado a base da comunicação em redes baseadas no protocolo Ethernet. Na realidade, a troca de dados entre dispositivos IP é efetuada através do endereço MAC, endereço

[3] Endereço de *Broadcast* corresponde a todos os endereços do segmento de rede

[4] *Backbone* é uma rede específica para conexões entre LANs, Ex. RedeRio, RNP, Embratel

[5] *Firewall* é um sistema de controle de acesso da informação à uma rede ou dispositivo

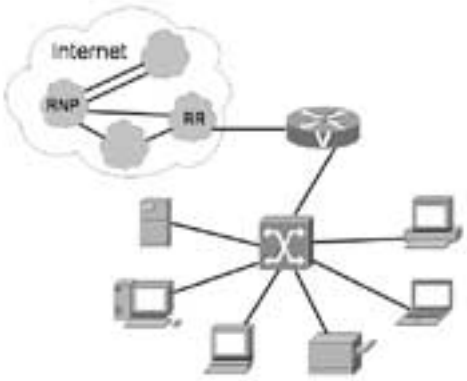

Figura 2: *Esquema de uma LAN com conexão Internet*

Ethernet ou ainda endereço Físico. De maneira bem simplificada, podemos considerar o protocolo ARP como sendo um broadcast no segmento de rede perguntando "qual é o endereço MAC do dispositivo que tem um certo IP?". Vamos considerar dois exemplos típicos na figura: comunicação no mesmo segmento de rede e em redes distintas.

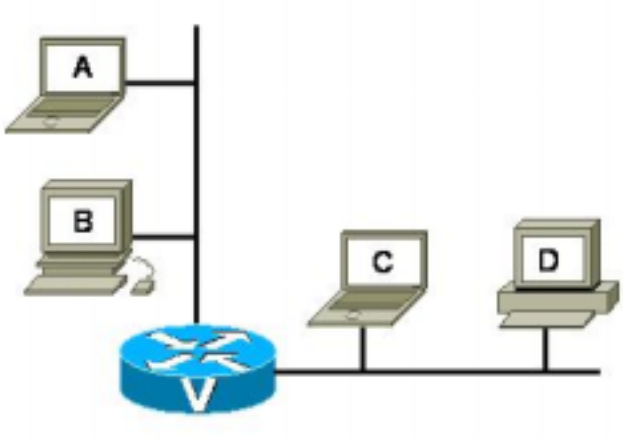

Figura 3: *Comunicação dentro da LAN e entre LANs.*

Primeiramente considere uma aplicação no computador A enviando dados para o computador B; considere, por simplicidade, uma transferência de arquivo de A para B. O primeiro passo é determinar se A e B pertencem ao mesmo segmento de rede. Isto é feito através de um algoritmo que compara o resultado de uma operação AND lógica entre os IP e a sua respectiva máscara. Se o resultado for o mesmo, então os dispositivos pertencem a mesma rede, caso contrário, pertencem a redes diferentes. Neste caso, A e B são vizinhos de um mesmo segmento.

Na construção do datagrama, a aplicação sabe os endereços MAC e IP da fonte A e somente o endereço IP do destino B. Para descobrir o endereço MAC de B, o protocolo ARP envia uma mensagem *broadcast* a todos os dispositivos do segmento, perguntando ao dono do IP B o seu endereço MAC. Por sua vez, o dispositivo dono do IP, envia também por *broadcast*, ou seja, para todos, o seu endereço MAC. Todos os dispositivos do segmento acrescentam na sua tabela ARP (IP-MAC), também chamada de *proxycache* ARP, este registro relativo ao B, que permanece durante

um certo tempo. Finalmente, o dispositivo A envia o quadro (frame) destinado ao dispositivo B. Neste exemplo, o mesmo quadro é enviado para B e a interface do roteador deste segmento, porém somente o dispositivo B irá abrir o quadro até a última camada, pois somente ele tem o endereço MAC destino. Observe que se houvesse outros dispositivos no segmento, eles passariam a conhecer também o endereço MAC de B, de maneira que se quisessem enviar algo à B posteriormente, não seria necessário um *broadcast* ARP novamente.

Vamos agora considerar que a comunicação seja entre os dispositivos A e C. Primeiramente, o dispositivo A determina que C pertence a outro segmento de rede, através do algoritmo comparativo de operações AND comentado anteriormente. O dispositivo A então envia os dados para o *gateways*, que é a interface do roteador. O protocolo ARP é utilizado para descobrir o endereço MAC da interface, da mesma maneira que no caso anterior. Observe que o endereço MAC destino é do roteador, porém o IP destino continua sendo do dispositivo C. Quando o roteador recebe os dados, ele procura pela rede à qual pertence o IP destino na sua tabela de roteamento e assim roteia para interface deste segmento. O roteador irá utilizar o protocolo ARP para determinar o endereço MAC do dispositivo C, que será anexado ao cabeçalho da camada de enlace, como MAC destino e o seu próprio, como MAC origem. É importante observar que os IPs origem (A) e destino (C) permanecem inalterados durante todo o processo. Quando o dispositivo C finalmente recebe a mensagem oriunda de A, o processo de volta é simplificado, pois os diversos endereços MAC continuam nas tabelas dos dispositivos envolvidos (C, roteador e A).

Estes dois exemplos simples mostram o funcionamento e a importância do protocolo ARP que, na realidade, é usado para manter a tabela IP/MAC de cada dispositivo atualizada.

4 Sistema *Cluster - Projeto SSOLAR*

Muitas aplicações computacionais da atualidade necessitam de mais potência computacional que os computadores tradicionais (seriais) podem oferecer, por exemplo, aplicações no campo da meteorologia, prospecção de petróleo, cálculos científicos que envolvam grandes matrizes ou muitas iterações, etc. Para solucionar este problema existem quatro abordagens: i. aumento de velocidade dos procesadores, discos, memórias e barramentos; ii. optimização dos algorítmos; iii. adoção de um ambiente computacional comcorrente ou paralelo e iv. escolha de recursos (mais) ociosos para utilização de uma tarefa computacional.

Um ambiente *cluster* agrega estas quatro abordagens e deve ser projetado de forma que sempre que um novo computador com mais performance estiver disponível, este seja empregado para melhorar o desempenho da aplicação optimizada e se possível implementada usando linguagens com suporte à programação paralela.

Atualmente a maioria dos ambientes *clusters* são compostos por computadores fisicamente independentes, porém logicamente configurados para trabalharem coordenadamente em conjunto. Em geral, existe um computador específico (*master*) para gerenciar a distribuição de tarefas computacionais (filas) entre os demais (nós).

O sistema operacional destes computadores pode ser Linux, Windows, Solaris, etc, assim como são vários os sistemas gerenciadores de filas, proprietários ou não, gratuitos ou não e por fim, existem várias formas de configuração operacional. Falaremos um pouco sobre o projeto SSOLAR[6] que é um SCAD - Sistema Computacional de Alto Desempenho do CBPF.

O projeto SSOLAR atua em duas áreas: a implantação de um *cluster* de computadores e de um ambiente em grade (Projeto Grid).

A fim de atender a demanda de computação científica o CBPF adquiriu um *cluster* de computadores do tipo Beowulf. Dessa forma, o CBPF conta atualmentecom uma infra-estrutura central

[6]http://mesonpi.cat.cbpf.br/ssolar

de processamento baseado no *cluster* comercial Microway com 39+10 processadores AMD Athlon XP 1800+ e 2400+, respectivamente, 50GBytes de memória RAM e 1,3TBytes de disco rígido, interligadospor uma rede em tecnologia Gigabit Ethernet. O sistema operacional utilizado é o Linux RedHat com compiladores C/C++, FORTRAN77 e FORTRAN90, biblioteca MPI, sistema de gerência de filas OpenPBS - *Portable Batch System* e de monitoramento Ganglia.

Atualmente. é utilizada a política de não compartilhamento, ou seja, cada CPU é dedicada para um job apenas. Desta forma é possível aumentar a performance, porém em alguns casos de muita submissão podem ocorrer estados de *queue* onde alguns jobs ficam na fila de espera aguardando uma CPU disponível.

O procedimento usual para submeter um processo envolvendo cálculo computacional no *cluster* do CBPF é: i. ter em uma conta no *master*; ii. escrever e compilar o código fonte compatível com os compiladores disponíveis (linguagens C/C++ e FORTRAN 77/90); iii. construir um *script* PBS que defina um nome de referência, o executável, o arquivo de saída e o número de nós (se o processamento for paralelo) e por fim iv. submeter este *script* com o comando adequado.

O usuário pode acompanhar ou apagar o estado do *script* bem como o(s) nó(s) envolvido(s) utilizando comandos específicos. As possibilidades de configuração do *script* são variadas e podem envolver situações de espera, cancelamento, escolha de procedimento, etc.

4.1 Processamento Paralelo

Paralelismo é uma técnica usada em tarefas grandes e complexas para obter resultados mais rápidos, dividindo-as em tarefas pequenas que serão distribuídas em vários processadores para serem executadas simultaneamente. Esses processadores se comunicam entre si para que haja coordenação (sincronização) na execução das diversas tarefas em paralelo.

Alguns objetivos do paralelismo são:

- Aumentar o desempenho (reduzindo o tempo) no processamento;
- Fazer uso de um sistema distribuído para resolução de tarefas;
- Auxiliar cálculos que necessitam de grande capacidade de processamento;

É importante ressaltar que o programador é diretamente responsável no caso de ferramentas não automáticas de paralelismo. Um ambiente paralelo possui:

- Vários processadores interligados em rede;
- Plataforma para manipulação de processos paralelos;
- Linguagem de Programação

Message Passing é o método de comunicação baseada no envio e recebimento de mensagens através da rede. Neste tipo de programação, deve existir a sincronização entre os processos. As tarefas executadas em paralelo, num determinado instante, aguardam a finalização mútua para coordenar os resultados ou trocar dados e reiniciar novas tarefas em paralelo. É necessário que haja a coordenação dos processos e da comunicação entre eles para evitar que a comunicação seja maior do que o processamento e que consequentemente haja uma queda no desempenho dos processos em execução.

A conversão de programas seqüenciais em programas em paralelo exige um considerável tempo gasto do programador em analisar o código fonte para paralelizar e recodificar. Basicamente é necessário rever o programa sequencial, avaliar como será particionado, quais os pontos de sincronização, quais as partições que poderão ser executadas em paralelo e qual a forma de comunicação que será empregada entre as tarefas paralelas. É necessário que haja uma análise do algoritmo a

ser paralelizado para que seja possível a paralelização nos dados ou nas funções do mesmo, levando sempre em consideração a quantidade de comunicação em relação ao processamento.

No processamento paralelo, o tempo de processamento, tempo de CPU, quase sempre aumenta, no entanto pode-se reduzir o tempo total de processamento.

4.2 Acesso a Memória

A comunicação entre os processadores depende da arquitetura de memória utilizada no ambiente. Para entendermos como os processadores se comunicam é fundamental compreender os tipos de arquitetura de memória, compartilhada e distríbuida.

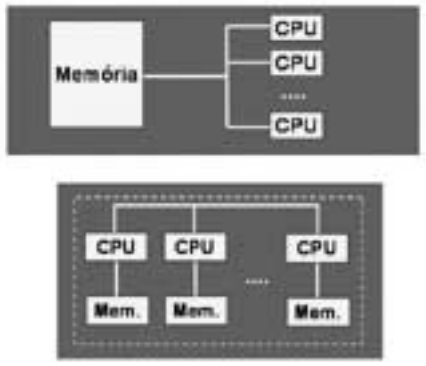

Figura 4: *Modos de acesso à memória: compartilhada e distribuída.*

Memória Compartilhada: as principais características deste tipo de memória são:

- Os processadores operam independentemente mas compartilham o recurso de uma única memória;
- Somente um processador acessa um endereço na memória por vez;
- Implementado em hardware.

Podemos citar como principal vantagem o fato do compartilhamento de dados entre os processos ser mais rápido e como desvantagens o alto custo dos computadores, a limitação no número de processadores e a necessidade de técnicas de sincronização para a leitura e gravação.

Memória Distribuída: As principais características deste tipo de memória são:

- Os processadores operam independentemente onde cada um possui sua própria memória;
- Os dados são compartilhados através da rede usando um mecanismo de troca de mensagens (Message Passing).

Neste tipo de memória podemos citar como vantagens o acesso à memória local sem interferência e o limite teórico para o número de processadores ser livre e as desvantagens são a dificuldade para mapear as informações e a responsabilidade do usuário pelo sincronismo e recebimento de dados.

4.3 Considerações de Performance

A Lei de Amdahl determina o potencial de aumento de velocidade a partir da porcentagem paralelizável do programa.

$$\boxed{Speedup = \frac{1}{1 - paralelizável}}$$

Consideremos uma aplicação que leva T unidades de tempo quando executado em modo serial em um único processador. Quando a aplicação é paralelizável, assumimos que o parâmetro S constitui o percentual serial e P o percentual que pode ser paralelizado. Assim, Tempo Seqüencial: T=Ts

Tempo Paralelo: Assumindo N processadores em implementação paralela, o tempo de execução é dado pela seguinte expressão (assumindo *Speedup* perfeito na parte da aplicação que pode ser paralelizada).

$$\boxed{T_p = (S + P/N) * T}$$

O *Speedup* que é obtido por N processadores é:

$$\boxed{S_p = \frac{T_s}{T_p} = \frac{1}{S+\frac{P}{N}}}$$

Logo,

$$\boxed{Speedup = \frac{1}{S+\frac{P}{N}}}$$

Onde, P é o percentual paralelizável, N é o número de processadores e S é o percentual serial. Se assumimos que a parte serial é fixa, então o *Speedup* para infinitos processadores é limitado em: $\alpha = \frac{1}{S}$ Por exemplo, se $\alpha = 10\%$, então o *Speedup* máximo é 10, até se usarmos um número infinito de processadores.

Em um programa que seja sequencial, o valor do % paralelizável deste programa é igual a zero e logo o valor do *Speedup* é 1, ou seja, não existe aumento na velocidade de processamento.

Em um programa no qual ocorra paralelização total, o valor de % paralelizável é igual a 1, o valor do *Speedup* teoricamente tende a valores infinitos.

Duas propriedades relevantes são a granulosidade e a escalabilidade. A granulosidade é uma medida usada para indicar a relação entre o tamanho de cada tarefa e o tamanho total do programa, ou seja, é a razão entre computação e comunicação. Este parâmetro depende do grau de dependência entre os processos, é definida pelo grau de controle (sincronização) necessário e a própria natureza do problema pode limitar o seu grau possível. A escalabilidade é a propriedade de um sistema paralelo de aumentar o *Speedup* a medida que aumentamos o número de processadores. Um dos maiores gargalos é o tráfego de informação na rede que interliga os processadores. O aumento do número de processadores pode degradar a capacidade de comunicação da rede, diminuindo a performance do programa. Alguns algoritmos trabalham melhor em certas escalas de paralelismo. O aumento do número de processadores pode acarretar uma queda na eficiência durante a execução do programa.

4.4 *Message Passing*

O modelo de *Message Passing* é um conjunto de processos que possuem acesso à memória local. As informações são enviadas da memória local do processo para a memória local do processo remoto. A comunicação dos processos é baseada no envio e recebimento de mensagens. A transferência dos dados entre os processos requer operações de cooperação entre cada processo de forma que operação de envio deve casar com uma operação de recebimento.

O modelo computacional *Message Passing* não inclui sintaxe de linguagem nem biblioteca sendo completamente independente do hardware. O paradigma de passagem de mensagem apresenta-se apenas como uma das alternativas mais viáveis, devido a muitos pontos fortes como por exemplo a generalidade pois pode-se construir um mecanismo de passagem de mensagens para qualquer linguagem, como ferramentas de extensão das mesmas (bibliotecas) e a adequação à ambientes distribuídos.

A seguir, algumas características do padrão MPI:

- Eficiência - Foi cuidadosamente projetado para executar eficientemente em máquinas diferentes. Especifica somente o funcionamento lógico das operações. Deixa em aberto a implementação. Os desenvolvedores otimizam o código usando características específicas de cada máquina.

- Facilidade - Define uma interface não muito diferente dos padrões PVM, NX, Express, P4, etc. e acrescenta algumas extensões que permitem maior flexibilidade.

- Portabilidade - É compatível para sistemas de memória distribuída, NOWs (network of workstations) e uma combinação deles.

- Transparência - Permite que um programa seja executado em sistemas heterogêneos sem mudanças significativas.

- Segurança - Provê uma interface de comunicação confiável. O usuário não precisa preocupar-se com falhas na comunicação.

- Escalabilidade - O MPI suporta escalabilidade sob diversas formas, por exemplo: uma aplicação pode criar subgrupos de processos que permitem operações de comunicação coletiva para melhorar o alcance dos processos.

As bibliotecas de *Message Passing* possuem rotinas com finalidades bem específicas, como: i. Rotinas de Gerência de Processos; ii. Rotinas de comunicação Ponto a Ponto onde a comunicação é feita entre dois processos e iii. Rotinas de comunicação de grupos.

As duas principais bibliotecas de *Message Passing* da atualidade são:

- PVM - (*Parallel Virtual Machine*): Possui como característica, o conceito de "máquina virtual paralela". Neste caso os processadores recebem e enviam mensagens, com finalidades de obter um processamento global.

- MPI - (*Message Passing Interface*): Amplamente conhecido e utilizado, está se tornando um padrão na indústria.

5 Introdução ao MPI

A biblioteca de *Message Passing* foi desenvolvida para ambientes de memória distribuída, máquinas paralelas massivas, network of workstations e redes heterogêneas.

MPI define um conjunto de rotinas para facilitar a comunicação (troca de dados e sincronização) entre processos paralelos. A biblioteca MPI é portável para qualquer arquitetura, tem aproximadamente 125 funções para programação e ferramentas para se analisar a performance.

A biblioteca MPI possui rotinas para programas em Fortran 77 e ANSI C, portanto pode ser usada também para Fortran 90 e C++. Os programas são compilados e linkados a biblioteca MPI.Todo paralelismo é explícito, ou seja, o programador é responsável por identificar o paralelismo e implementar o algoritmo utilizando chamadas aos comandos da biblioteca MPI.

O MPI básico contém 6 funções básicas indispensáveis para o uso do programador, permitindo escrever um vasto número de programas em MPI. O MPI avançado contém cerca de 125 funções adicionais que acrescentam às funções básicas a flexibilidade (permitindo tipos diferentes de dados), robustez (modo de comunicação non-blocking), eficiência (modo ready), modularidade através de grupos e comunicadores (group e communicator) e conveniência (comunicações coletivas, topologias).

5.1 Processos, *Rank* e Comunicação

Por definição, cada programa em execução constitui um processo. Considerando um ambiente multiprocessado, podemos ter processos em inúmeros processadores. De acordo com o MPI Padrão a quantidade de processos deve corresponder à quantidade de processadores disponíveis.

Mensagem é o conteúdo de uma comunicação composto duas partes: envelope e dados. O envelope é o endereço (origem ou destino) e rota dos dados e é composto de três parâmetros: i. identificação dos processos (transmissor e receptor); ii. Rótulo da mensagem e iii. o Comunicator.

A parte Dados contém a informação que se deseja enviar ou receber. É representada por três argumentos: i. o endereço onde o dado se localiza; ii. o número de elementos do dado na mensagem e iii. o tipo do dado.

Todo o processo tem uma identificação única atribuída pelo sistema quando é inicializado. Essa identificação é contínua representada por um número inteiro, começando de zero até N-1, onde N é o número de processos. O identificador Rank é único do processo e utilizado para identificar o processo no envio (*send*) ou recebimento (*receive*) de uma mensagem.

A comunicação entre os processos pode ser de dois tipos: Ponto a Ponto e Coletiva. Na comunicação Ponto a Ponto são utilizadas as rotinas básicas do MPI de comunicação ponto a ponto que executam a transferência de dados entre dois processos. Na comunicação Coletiva, a comunicação padrão invoca todos os processos em um grupo (*group*) que é uma coleção de processos que podem se comunicar entre si. Normalmente a comunicação coletiva envolve mais de dois processos. As rotinas de comunicação coletiva são voltadas para a comunicação/coordenação de grupos de processos.

5.2 Utilizando MPI

No desenvolvimento de programas devemos adicionnar os arquivos mpi.h para programas em C e mpif.h para programas em FORTRAN. Na fase de compilação e linkedição devemos usar o compilador com a sintaxe adequada, incluindo os parâmetros desejados.

- Para linguagem C/C++: mpicc [fonte.c] -o [executável] [parâmetros]
- Para Fortran: mpif77 [fonte.f] -o [executável] [parâmetros]

Na fase de execução use o comando mpirun -[argumentos] [executável], por exemplo para executar em 5 processadores o programa teste utilize mpirun -np 5 teste, onde os argumentos podem ser:
h - Mostra todas as opções disponíveis
arch - Especifica a arquitetura da(s) máquina(s)
machine - Especifica a(s) máquina(s)
machinefile - Especifica o arquivo que contém o nome das máquinas
np - Especifica o número de processadores
leave pg - Registra onde os processos estão sendo executados
nolocal - Não executa na máquina local
t - Testa sem executar o programa, apenas imprime o que será executado
dbx - Inicializa o primeiro processo sobre o dbx

A seguir abordaremos as funções básicas do MPI, sua sintaxe (em C/C++ e FORTRAN), parâmetros e códigos de erro. Em seguida será mostrado um programa exemplo.

- **MPI_INIT**: Inicializa um processo MPI e portanto, deve ser a primeira rotina a ser chamada por cada processo, pois estabelece o ambiente necessário para executar o MPI. Ela também sincroniza todos os processos na inicialização de uma aplicação MPI.
  Sintaxe:

int MPI_Init (int *argc, char *argv[])
call MPI_INIT (mpierr)
Parâmetros:
argc - Apontador para a qtde. de parametros da linha de comando
argv - Apontador para um vetor de strings
Erro:
MPI-ERR-OTHER - Ocorreu mais de uma chamada dessa rotina no mesmo código.

- **MPI_COMM_RANK**: Identifica um processo MPI dentro de um determinado grupo. Retorna sempre um valor inteiro entre 0 e n-1, onde n é o número de processos.
  Sintaxe:
  int MPI_Comm_rank (MPI_Comm comm, int *rank)
  call MPI_COMM_RANK (comm, rank, mpierr)
  Parâmetros:
  comm - Comunicador do MPI;
  rank - Variável inteira com o numero de identificação do processo.
  Erro:
  MPI_ERR_COMM - Communicator inválido.

- **MPI_COMM_SIZE**: Retorna o número de processos dentro de um grupo.
  Sintaxe:
  int MPI_Comm_size (MPI_Comm comm, int *size)
  call MPI_COMM_SIZE (comm, size, mpierr)
  Parâmetros:
  comm - Comunicador do MPI;
  size - Variável interna que retorna o numero de processos iniciados pelo MPI.
  Erros:
  MPI_ERR_COMM - Communicator inválido.
  MPI_ERR_ARG - Argumento inválido.
  MPI_ERR_RANK - Identificação inválida do processo.

- **MPI_SEND** : Rotina básica para envio de mensagens no MPI, utiliza o modo de comunicação "blocking send "(envio bloqueante), o que traz maior segurança na transmissão da mensagem. Após o retorno, libera o "system buffer"e permite o acesso ao "application buffer".
  Sintaxe:
  int MPI_Send (void *sndbuf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm)
  call MPI_SEND (sndbuf, count, dtype, dest, tag, comm, mpierr)
  Parâmetros:
  Sndbuf - Identificação do buffer (endereço inicial do "application buffer- de onde os dados serão enviados);
  count - Número de elementos a serem enviados;
  dtype - Tipo de dado;
  dest - Identificação do processo destino;
  tag - Rótulo (label) da mensagem;
  comm - MPI Communicator.
  Erros:

MPI_ERR_COMM - Communicator inválido.
MPI_ERR_COUNT - Argumento numérico inválido.
MPI_ERR_RANK - Identificação inválida do processo.
MPI_ERR_TYPE - Tipo de dado inválido.
MPI_ERR_TAG - Declaração inválida do label.
MPI_ERR_RANK - Identificação inválida do processo.

- **MPI_RECV**: É uma rotina básica para recepção de mensagens no MPI, que utiliza o modo de comunicação "blocking receive"(recepção bloqueante), de forma que o processo espera até que a mensagem tenha sido recebida. Após o retorno, libera o "system buffer", que pode ser então, novamente utilizado. Sintaxe:
  int MPI_Recv (void *recvbuf, int count, MPI_Datatype dtype, int source, int tag, MPI_Comm comm, MPI_Status status)
  call MPI_RECV (recvbuf, count, dtype, source, tag, comm, status, mpierr)
  Parâmetros:
  sndbuf - Identificação do buffer (endereço inicial do "application buffer- de onde os dados estão sendo enviados);
  count - Número de elementos a serem recebidos;
  dtype - Tipo de dado;
  source - Identificação do processo emissor;
  tag - Rótulo (label) da mensagem;
  comm - MPI Communicator;
  status - Vetor de informações envolvendo os parâmetros source e tag.
  Erros:
  MPI_ERR_COMM - Communicator inválido.
  MPI_ERR_COUNT - Argumento numérico inválido.
  MPI_ERR_RANK - Identificação inválida do processo.
  MPI_ERR_TYPE - Tipo de dado inválido.
  MPI_ERR_TAG - Declaração inválida do label.
  MPI_ERR_RANK - Identificação inválida do processo.

- **MPI_FINALIZE**: Finaliza um processo MPI. Portanto deve ser a última rotina a ser chamada por cada processo. Sincroniza todos os processos na finalização de uma aplicação MPI. Sintaxe:
  int MPI_Finalize (void)
  call MPI_FINALIZE (mpierr)
  Parâmetros: Não Possui
  Erro: Não Possui

5.3 Programa Exemplo

A seguir, será mostrado o código fonte de um programa em C simples, onde são utilizadas as rotinas MPI descritas acima.

Como já foi dito anteriormente, um cluster possui nós escravos (em número variável) e 1 nó mestre (que envia tarefas para os nós escravos). O programa apresentado utiliza a técnica de paralelismo e foi executado no cluster do CBPF para 4 processadores (no caso do CBPF, podem ser usados até 8 processadores). Da maneira como foi implementado este programa, enquanto 3 processadores estão calculando, 1 processador fica responsável apenas em receber as tarefas dos

outros.

Após o código fonte do programa, será mostrado o arquivo de envio de tarefas para o cluster (.pbs), assim como a saída após a execução do programa. O código do programa abaixo já está comentado (utilizando o recurso da linguagem /* e */).

No entanto, vale ressaltar o uso das bibliotecas:

- stdio.h - Esta biblioteca engloba o conjunto de funções que são normalmente usadas na linguagem C para I/O;
- string.h - Esta biblioteca possui uma série de funções para manipular cadeias de caracteres;
- time.h - Esta biblioteca possui funções de temporização que são para saber data e a hora correntes, medir o tempo de execução de uma operação, inicializar geradores de números aleatórios, etc.
- mpi.h - Biblioteca necessária para programas em C que utilizam o MPI

```
01  /*Os processos escravos enviam mensagens ao processo mestre!*/
02  #include <stdio.h>
03  #include <string.h>
04  #include <time.h>
05  #include "mpi.h"
06
07  int main(int argc, char* argv[])
08  {
09  int my_rank;            /* identificador do processo*/
10  int p;          /* quantidade total de processos*/
11  int source;           /* identificador do processo emissor*/
12  int dest;           /* identificador do processo receptor*/
13
14  char message[200];          /* mensagem enviada ao processo mestre*/
15  char machinename[20];           /* nome da maquina onde o processo está sendo executado*/
16  int sizemachinename;          /* tamanho da string machinename*/
17  time_t tm;          /* variavel para armazenar uma data/hora*/
18  MPI_Status status;          /* status de retorno para processo receptor*/
19
20  /* Inicia o MPI*/
21  MPI_Init(&argc, &argv);
22
23  /* Descobre o identificador (rank) do processo*/
24  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
25
26  /* Descobre a quantidade total de processos*/
27  MPI_Comm_size(MPI_COMM_WORLD, &p);
28
29
30
31  /* Processos escravos. Descobre a data/hora atual, o nome da maquina em que esta executando o processo e envia mensagem ao processo mestre*/
32
33  if(my_rank != 0)
34
```

```
35      tm = time(NULL);
36      MPI_Get_processor_name(machinename, &sizemachinename);
37      sprintf(message, "O processo %d esta sendo executado na maquina %s,Hello CBPF!
.\nData e hora em %s: %s",my_rank, machinename, machinename, ctime(&tm));
38      dest = 0;           /* identificador do processo mestre*/
39      MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, 0, MPI_COMM_WORLD);
40      }
41
42  /* Processo mestre. Recebe as mensagens dos processos escravos e exibe-as na tela*/
43  else
44      {
45      for(source=1;source¡p;source++)
46          {
47          MPI_Recv(message, 200, MPI_CHAR, source, 0, MPI_COMM_WORLD, &status);
48          printf("%s\n", message);
49          }
50      }
51
52  /* Encerra o MPI*/
53  MPI_Finalize();
54  return 0;
55  }
```

O arquivo (.pbs) necessário para enviar a tarefa para o cluster, tem o seguinte formato:

```
01  # nome do job
02  #PBS -N helloworld_mpi
03  # numero de nos e cpus (sendo um programa serial: 1 1)
04  #PBS -l nodes=4:ppn=1
05  # redireciona saida e erros em um unico arquivo
06  #PBS -j oe
07  # nome do arquivo de saida
08  #PBS -o saida_hello
09  #posiciona o ponteiro na area do usuario
10  cd $PBS_O_WORKDIR
11  echo "Job started at ‘date‘"
12  mpiexec ./helloworld_mpi
13  echo "Job ended at ‘date‘"
14  exit 0
```

O arquivo de saída deste programa, conterá:

```
01  Job started at Wed Jun 30 15:03:59 BRST 2004
02   processo 1 esta sendo executado na maquina node17.ssolar.cbpf.br,Hello CBPF! .
03   e hora em node17.ssolar.cbpf.br: Wed Jun 30 15:03:30 2004
04
05   processo 2 esta sendo executado na maquina node19.ssolar.cbpf.br,Hello CBPF! .
06   e hora em node19.ssolar.cbpf.br: Wed Jun 30 15:03:30 2004
07
08   processo 3 esta sendo executado na maquina node14.ssolar.cbpf.br,Hello CBPF! .
09   e hora em node14.ssolar.cbpf.br: Wed Jun 30 15:03:30 2004
10
```

```
11      ended at Wed Jun 30 15:03:59 BRST 2004
```

Repare que cada processo foi executado em nós diferentes do cluster e praticamente ao mesmo tempo.

6 Projeto GRID do CBPF

Os Grids são desenvolvidos para aplicações que precisam de grande capacidade de cálculos, enormes quantidades de dados transmitidos ou ambas as situações. O nome Grid foi idealizado baseado nas malhas de interligação dos sistemas de energia elétrica (power-grids), onde para o usuário não importa a origem da energia nem a complexidade da malha de transmissão e distribuição.

Uma boa definição de Grid dada por Ian Forster, considerado o criador do Grid, é "Recursos coordenados que não se sujeitam a um controle centralizado - (Sistemas em Grid integram e coordenam recursos e usuários que vivem em diferentes domínios administrativos)".

O Grid permite o uso de técnicas de programação paralela por passagem de mensagens. A biblioteca MPI, citada anteriormente, está disponível para ambiente Grid na versão MPICH-G2 . Devido a alta latência provocada na comunicação entre processos, as aplicações devem ser desenvolvidas com uma granularidade bem projetada de tal forma que se comuniquem o mínimo possível. Desta forma, as aplicações mais adequadas ao Grid são as que possuem tarefas independentes (*bag of tasks*), pois as tarefas não dependem da execução de outras e podem ser executadas em qualquer ordem.

6.1 Escalonamento em um Grid Computacional

Escalonador de Recurso - recebe solicitações de vários usuários e decide, entre estes usuários, o uso dos recursos que controlam. Um exemplo de escalonador de recurso é o sistema operacional que controla o computador no qual ele está instalado.

Devido à grande escala, ampla distribuição e existência de múltiplos domínios administrativos, não é possível construir um escalonador de recursos para Grids, pois seria muito difícil convencer os administradores dos recursos que compõem o Grid a abrir mão do controle de seus recursos.

Para utilizar recursos controlados por vários escalonadores de recursos diferentes, é necessário:

- escolher quais recursos serão utilizados na execução da aplicação;
- estabelecer quais tarefas cada um destes recursos realizará;
- submeter solicitações aos escalonadores de recurso apropriados para que estas tarefas sejam executadas.

Estas tarefas são do escalonador de aplicação. Escalonadores de aplicação não controlam os recursos que usam, apenas envia solicitações para os escalonadores que controlam os recursos.

Escalonador de Aplicação - têm que conhecer detalhadamente as aplicações que escalonam, como por exemplo, o tempo que cada recurso vai levar para processar uma dada tarefa, afim de escolher os melhores recursos a utilizar e determinar qual tarefa alocar a cada um desses recursos.

Neste sentido, foram desenvolvidos sistemas que monitoram e prevêem o comportamento futuro de diversos tipos de recursos. No entanto há uma questão relevante com relação ao comportamento dos escalonadores de aplicação: qual o impacto no sistema de ter vários escalonadores de aplicação, buscando melhor performance para sua aplicação?

6.2 Segurança

O acesso aos recursos que compõem um Grid não é tão simples quanto em outras plataformas, devido à existência de múltiplos domínios administrativos. Será necessária uma forma de acesso

para cada recurso que compõe o Grid, desde que ofereça garantias sobre autenticação dos usuários, caso contrário os administradores do sistema se oporão ao seu uso.

Atualmente, existem vários sistemas para computação em Grid. O Globus é um dos mais utilizados devido à independência razoável oferecida pelos seus serviços. A computação em Grid é uma área em expansão e extremamente dinâmica e grande parte do interesse neste tipo de computação se deve ao fato de executar aplicações que necessitem de grande poder computacional e capacidade de armazenamento.

7 Sistema Globus

Atualmente, existem vários sistemas para computação em Grid. O Globus é amplamente utilizado, pois é composto por um conjunto de serviços afim de facilitar a computação em Grid. Os serviços Globus podem ser usados para submissão e controle de aplicações, descoberta de recursos, movimentação de dados e segurança no Grid.

A solução proposta pelo Globus para computação em Grid é composta de vários serviços distintos, porém integrados. Os serviços Globus podem ser usados para submissão e controle de aplicações, descoberta de recursos, movimentação de dados e segurança no Grid.

Foram selecionados três serviços importantes oferecido pelo Globus, para ser abordado neste documento: o GRAM - *Globus Resource Alocation Manager* responsável pelo gerenciamento dos recursos, o MDS - *Meta-computing Directory Service* que disponibiliza informações do Grid e o GSI - *Globus Security Infrastructure* responsável pela segurança e autenticação no Grid.

O GRAM fornece uma interface uniforme para envio e controle das tarefas e informa sobre o status do recurso ao MDS. A grande vantagem de usar GRAM é a manipulação uniforme de tarefas não importando qual o escalonador de recurso está sendo usado para controlar a máquina. Isto se deve ao fato das requisições enviadas ao GRAM serem sempre escritas em RSL - *Resource Specification Language*. O *Job Manager* é o responsável por converter as requisições em RSL em um formato que o escalonador de recurso entenda.

As requisições enviadas pelo cliente são recebidas pelo *Gatekeeper*, que consulta o GSI - *Globus Security Infrastructure* para identificar o usuário e verificar se ele pode utilizar a máquina. Caso o usuário tenha permissão, é criado um Job Manager responsável por iniciar e monitorar a tarefa. Requisições sobre o estado da tarefa serão encaminhadas diretamente ao Job Manager. O GRAM Repórter obtém informações de status e carga da máquina junto ao escalonador de recursos e as repassa para o MDS.

O Globus disponibiliza o MDS que é o serviço de informação do Grid. O MDS contém informações sobre os recursos que formam o Grid sendo composto por dois serviços internos, o GIIS - *Grid Index Information Service* e o GRIS - *Grid Resource Information Service*. O GRIS reúne informações locais, como: espaço em disco, sistema operacional, memória, etc. enquanto o GIIS reúne em um único servidor as informações de vários GRISs.

Um aspecto importante no uso de Grids é a autenticação de usuários em diferentes domínios administrativos. Naturalmente, este usuário deveria se autenticar para cada domínio administrativo (identificação de usuário e senha). O Globus disponibiliza o GSI - *Globus Security Infrastructure* a fim de viabilizar o login único no Grid. O serviço GSI utiliza criptografia de chave pública, certificados X.509 e comunicação SSL - *Secure Sockets Layer* para estabelecer a identidade Globus do usuário.

Depois de o usuário ter se identificado junto ao GSI, todos os demais serviços Globus saberão, de forma segura, que o usuário é realmente quem ele diz ser. Uma vez que um serviço sabe a identidade Globus do usuário, resta estabelecer quais operações tal usuário pode realizar. Isto é feito mapeando a identidade Globus para um usuário local.

Grids não têm um escalonador que controla todo o sistema. Assim sendo, quando um usuário que submeter uma aplicação para execução no Grid, o usuário utiliza um escalonador de aplicação

que escolhe os recursos a utilizar, particiona o trabalho entre tais recursos, e envia tarefas para os escalonadores dos recursos. Em Globus, os escalonadores de recurso são acessados através do serviço GRAM que fornece uma interface única que permite submeter, monitorar e controlar tarefas de forma independente do escalonador de recursos. Assim sendo,escalonadores de aplicação não precisam entender dos detalhes particulares de cada escalonador de recurso. Para facilitar ainda mais a tarefa dos escalonadores de aplicação, Globus também disponibiliza MDS (Metacomputing Directory Service), um serviço de informação sobre o Grid. MDS contém informações sobre os recursos que formam o Grid e também sobre seu estado (carga, disponibilidade, etc).

Uma idéia bastante interessante em Globus é que escalonadores de aplicação podem usar os serviços de outros escalonadores de aplicação. O escalonador que recebe a solicitação do cliente lida com a especificação em mais alto nível. Ele refina tal especificação e, para implementá-la, submete novas solicitações aos escalonadores de recurso e/ou escalonadores de aplicação.

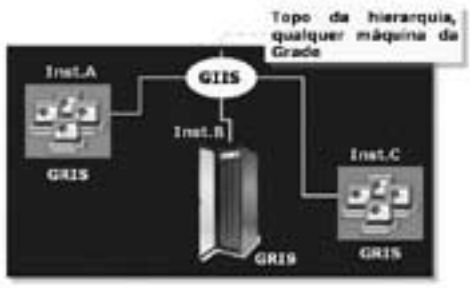

Figura 5: *Serviço de Informação do Globus envolvendo clusters de várias instituições.*

Globus suporta bem a hierarquia de escalonadores através da linguagem RSL (Resource Specification Language). RSL é capaz de expressar tanto solicitação de alto nível, como também solicitações concretas. Portanto, o trabalho de um escalonador de aplicação em Globus pode ser descrito como sendo o de refinar solicitações RSL. O Globus usa Broker para o que chamamos de escalonador de aplicação. Já o coalocador (Co-allocator) é um escalonador de aplicação especializado em garantir que tarefas localizadas em máquinas distintas executem simultaneamente. O co-alocador é fundamental para execução em Grids de aplicações fortemente acopladas. Em aplicações fortemente acopladas, as tarefas precisam se comunicar para que a aplicação faça progresso. Portanto, todas as tarefas da aplicação têm que ser executadas simultaneamente. É importante ressaltar que uma boa implementação de co-alocação depende da implementação, por parte dos escalonadores de recurso, do serviço de reserva adiantadas (advance reservation). Reservas adiantadas permitem a escalonadores de aplicação obter garantias de escalonadores de recurso que determinados recursos estarão disponíveis para aplicação em um intervalo de tempo preestabelecido.

O problema de comunicação no Grid pode ser visto como uma instância do eterno conflito entre generalidade e performance. Caso utilizemos um mecanismo de comunicação genérico que viabilize a comunicação entre quaisquer duas tarefas no Grid, perdemos performance em casos especiais. Por outro lado, gostaríamos de usar um mecanismo genérico para não ter que programar para cada uma das várias tecnologias de comunicação existentes. Globus resolve este problema com o Nexus que fornece uma interface de baixo nível, mas uma implementação adaptável que escolhe, dentre as tecnologias de comunicação disponíveis, a que vai oferecer melhor performance. Por exemplo, se ambas as tarefas estão em uma máquina de memória compartilhada, Nexus utilizará a memória para efetuar a comunicação. Caso as tarefas estejam em máquinas geograficamente

distantes, Nexus utilizará TCP/IP. Nexus fornece uma interface de relativo baixo nível: invocação remota de procedimento, mas sem retorno de resultado. Portanto, programar diretamente em Nexus não é das tarefas mais agradáveis. Entretanto, a idéia da equipe Globus é que Nexus seja usado por desenvolvedores de ferramentas e mecanismos de comunicação, não diretamente pelo desenvolvedor de aplicações.

A necessidade de acesso remoto e transferência de dados é frequente na computação em Grid. Na verdade, várias das aplicações aptas a executar no Grid necessitam de paralelismo exatamente porque processam enormes quantidades de dados. O Globus logo disponibiliza GASS (Global Access to Secundary Storage), um serviço para acesso remoto a arquivos sob a tutela de um servidor GASS. O cliente GASS é uma biblioteca C que é link-editada à aplicação usuária do serviço.

Apesar de ser um bom serviço, o GASS encontrou problemas de implantação. A dificuldade encontrada foi de interoperabilidade. A maioria das fontes de dados onde se instalaria um servidor GASS já executa algum serviço de transferência e/ou acesso remoto a arquivos. Essa realidade motivou a introdução do GridFTP por parte da equipe Globus. GridFTP estende o popular protocolo FTP para torná-lo mais adequado para as necessidades da Computação em Grid.

Uma vez que GridFTP é uma extensão do FTP, o problema de interoperabilidade fica resolvido, pois FTP é amplamente suportado pelos servidores de dados. Obviamente, se as extensões GridFTP não estiverem implementadas em um dado servidor, os benefícios adicionais do protocolo não estarão disponível.

Um aspecto importante para grande aceitação do Globus é que de certa forma os serviços oferecidos são independentes, possibilitando que se utilize apenas parte dos serviços Globus em uma dada solução. Essa possibilidade do uso parcial de Globus ajuda na adaptação de aplicações paralelas existentes para o Grid. Pode-se começar usando serviços mais básicos e ir, aos poucos, incorporando funcionalidades mais avançadas. O design oposto à abordagem "conjunto de serviços independentes"do Globus é exemplificado pelo Legion. Legion fornece um modelo orientado a objetos poderoso e flexível. Entretanto, o usuário tem abraçar a solução Legion integralmente, sem estágios intermediários. Esta diferença de abordagem talvez tenha contribuído para prevalência do Globus como padrão para Computação em Grid.

Vale ressaltar que a decisão de estruturar Globus como um conjunto de serviços independentes deixa claro que Globus não é uma solução pronta e completa para construção de Grids. Globus certamente fornece serviços úteis, porém os desenvolvedores, administradores e usuários precisam despender certo esforço para finalizar seu Grid. Por exemplo, administradores precisam decidir como quais usuários terão acesso a quais recursos que compõem o Grid e em quais condições este acesso se dará. Em outro exemplo, freqüentemente é necessário desenvolver escalonadores de aplicação que tenham conhecimento sobre as aplicações que serão executadas e eventualmente também sobre a estrutura do Grid a ser usado.

A versão do Globus citada neste documento utiliza vários protocolos diferentes (LDAP no MDS, FTP e GridFTP para transferência de arquivos, etc) para implementação dos diversos serviços oferecidos. Na versão mais atual todos os serviços Globus já são baseados em Web Services, em um esforço para criar uma série de padrões para computação distribuída denominados OGSA - *Open Grid Services Architecture*. Esta evolução na direção de Web Services e OGSA mostra o interesse de seus desenvolvedores de utilizar Globus em contextos mais amplos que Processamento de Alto Desempenho.

8 Bibliografia

- "C History", D. M. Ritchie, http://cm.bell-labs.com/cm/cs/who/dmr/chist.html
- "Brian W. Kernighan: Programming in C: A Tutorial", http://www.lysator.liu.se/c/bwk-tutor.html

- "Bjarne Stroustrup's Homepage", http://www.research.att.com/ bs/homepage.html
- "Curso MPI", Conselho Nacional de Processamento de Alto Desempenho - CENAPAD-NE
- "Projeto SSOLAR", http://mesonpi.cat.cbpf.br/ssolar
- "Projeto GRID CBPF", http://www.cbpf.br/cat/grid
- "Beowulf Cluster Computing with Linux (Scientific and Engineering Computation)", T. Sterling, J. S. Kowalik, G. Bell, MIT Press, 2001.