Introdução ao Processamento Paralelo e ao Uso de Clusters de Workstacions em Sistemas GNU/LINUX Parte I: Filosofia*

Bueno A.D. (andre@lmpt.ufsc.br)
Laboratório de Meios Porosos e Propriedades Termofísicas - LMPT
Versão 0.5

12 de novembro de 2002

Resumo

O termo cluster de workstacions, refere-se a utilização de diversos computadores conectados em rede para realização de processamento paralelo. Este artigo introduz os termos e conceitos necessários ao entendimento e uso do processamento paralelo e de clusters em sistemas GNU/LINUX. Descreve-se os direntes tipos de processamento paralelo em máquina multi-processadas (SMP) e em cluster de workstacions. A seguir, apresenta-se as bibliotecas para implementação de multi-processos, multi-threads, e os sistemas de troca de mensagens como PVM e MPI. Da-se ênfase a linguagem C/C++ e inclue-se um conjunto de links e referências para documentos externos, facilmente obtidos na internet.

Sumário

 O artigo "Introdução ao Processamento Paralelo e ao Uso de Clusters de Workstacions em Sistemas GNU/LINUX Parte II: Processos e Threads", complementa este artigo abordando, mais profundamente o uso de processos e threads.

1 Introdução

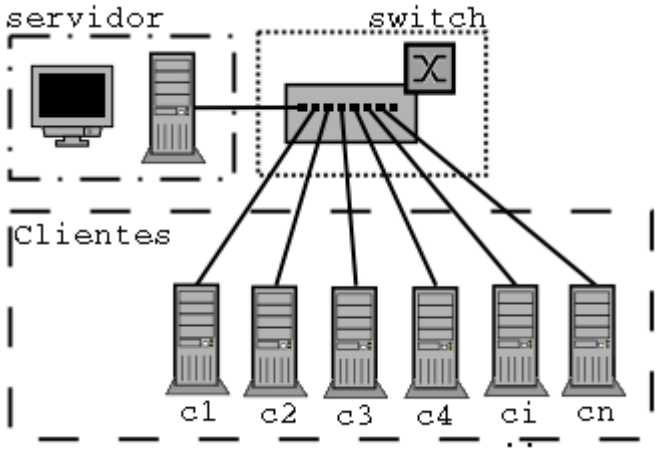
Um cluster de workstation é um conjunto de computadores (heterogêneos ou não) conectadas em rede para o desenvolvimento de processamento paralelo. Ou seja, as máquinas são conectadas via rede para formar um *único computador*, [13]. Embora poucas tarefas possam ser paralelizadas com facilidade, um cluster de computadores é útil para uma enorme quantidade de problemas científicos.

Deve-se considerar ainda que paralelizar algoritmos é um ramo bastante inexplorado e de importância vital para a indústria de informática nos próximos anos [13].

Como exemplo, pode-se citar a utilização de modelos de gás em rede para simulação de propriedades físicas dos materiais. Como estes modelos são massissamente paralelizáveis, pode-se, com o uso de cluster, aumentar o desempenho dos algoritmos e possibilitar o trabalho com representações tridimensionais de elevada dimensão, [10]. O mesmo ocorre com os métodos para determinação das configurações de equilíbrio e de permeabilidade relativa, [3, 2].

Veja diagrama de um cluster na Figura 1.

Figura 1: Diagrama de um cluster.



O projeto pioneiro em cluster de computadores foi desenvolvido no CESDIS (Center of Excellence in Space Data and Information Sciences) em 1994. Contava com 16 máquinas 486 rodando GNU/Linux. O nome dado ao primeiro cluster foi BEOWULF.

O princípio de funcionamento é simples. O servidor divide as tarefas em suas partes independentes (ditas concorrentes), a seguir, distribuí estas tarefas entre os vários computadores que fazem parte do cluster. As tarefas são processadas e então os resultados são enviados para o servidor.

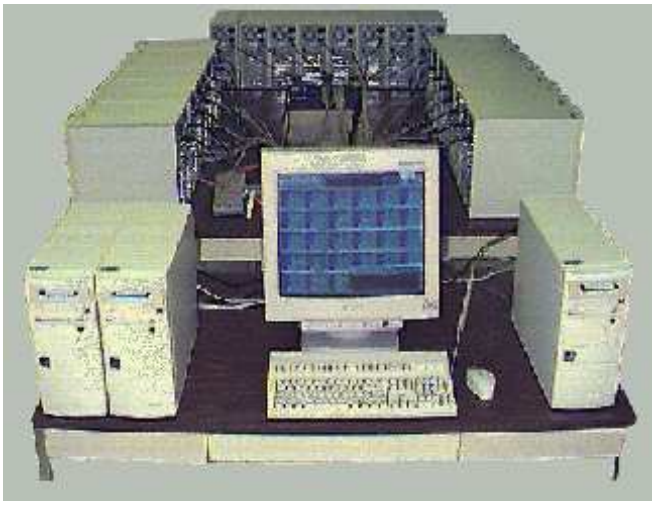
Para que o sistema funcione é necessário um servidor, vários clientes, uma biblioteca para troca de mensagens e o hardware para conexão via rede dos diversos computadores.

- O servidor distribue o processamento para as diversas máquinas clientes (nós). Observe na Figura 1 que o servidor envia e recebe mensagens (e dados) para os diversos clientes passando pelo switch.
- O hardware da rede (placas de rede, switch, cabos) deve ter qualidade e capacidade para transferir os dados do servidor para os clientes com a menor perda de tempo possível (latência).
- Cada cliente recebe as mensagens e um conjunto de dados a serem processados. Concluído o processamento, os resultados são enviados para o servidor.
- A biblioteca de troca de mensagens deve conter um conjunto de funções que serão utilizadas para distribuir o processamento entre as diversas máquinas do cluster.

Exemplos de cluster:

- O Avalon (http://swift.lanl.gov/avalon/) é um cluster de 140 máquinas Alpha 533 MHz, localizado no Laboratório Nacional de Los Alamos, nos Estados Unidos. Esse computador já foi o 113º no TOP500, a lista das 500 máquinas mais rápidas do mundo. O fato de o Avalon ter custado uma fração do preço de seus concorrentes próximos no ranking, mesmo usando máquinas de alta qualidade, é um testemunho da vantagem da solução de processamento pesado usando clusters. O avalon é formado com Switch 3com superstack II 3900 36portas-fast ethernet switch. Inicialmente contava com 70 máquinas Alpha 533Mhz, obtendo o mesmo desempenho de um computador paralelo com 64 processadores a 195Mhz (que custava na época 1.8 milhões de dolares). Usa o GNU/Linux da Red Hat, o pacote de compilação egcs-1.1b e a biblioteca MPICH. Os autores indicam a necessidade de um sistema de cooling (resfriamento) de melhor qualidade em cada máquina, além de uma rede elétrica dedicada [http://www.cnls.lanl.gov/avalon/FAQ], adaptado de [13]. Veja na Figura 2 um cluster.

Figura 2: Cluster do Numerical Aerospace Simulation Facility NASA Ames Research Center.



2 Objetivo e conceito de processamento paralelo

O objetivo de um cluster de workstatcions é possibilitar o uso de computadores ligados em rede para execução de processamento com alto desempenho, permitindo a realização de simulações avançadas.

O **processamento paralelo** consiste em dividir uma tarefa em suas partes independentes e na execução de cada uma destas partes em diferentes processadores.

3 O que você precisa saber

Para que você possa desenvolver programas usando processamento paralelo em um cluster de computadores, você precisa ter o domínio de um conjunto de conceitos, são eles:

1. os diferentes tipos de processamento paralelo (seção 4).
2. as bibliotecas utilizadas para distribuição do processamento, processos (seção 5.1), threads (seção 5.2), PVM (seção 5.3) e MPI (seção 5.4).
3. como desenvolver algoritmos e códigos utilizando processmanto paralelo. Isto é, como desenvolver as rotinas dos programas usando processamento paralelo. Veja seções 6, 8, 9, 10.

É necessário ainda o conhecimento da terminologia utilizada (veja um pequeno glossário no apêndice B), e do hardware necessário (apêndice A).

Para maiores detalhes consulte a bibliografia [17, 7, 12, 13, 21, 11, 15, 4, 16, 14, 18, 20, 22].

4 Tipos de processamento paralelo

Descreve-se a seguir os diferentes tipos de estruturas utilizadas para implementar o processamento paralelo.

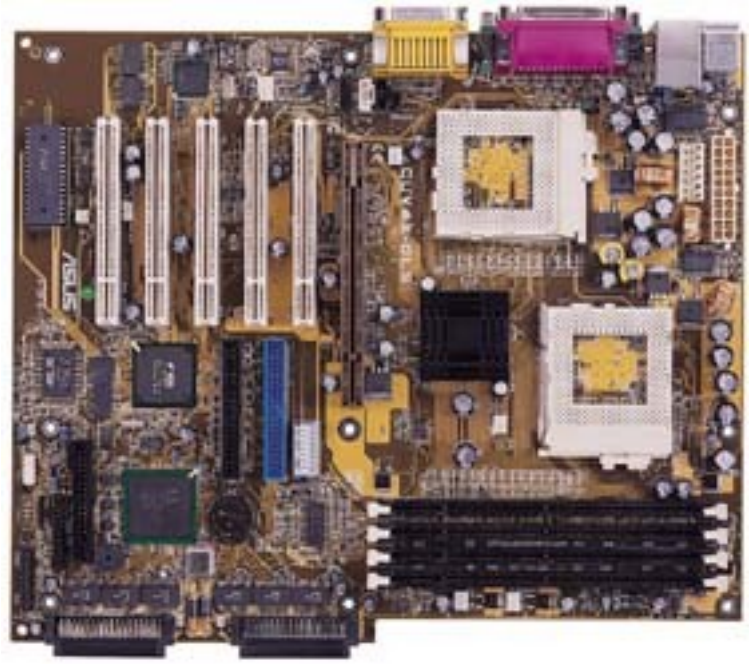
4.1 Processamento paralelo com Swar (Simd Withn a Register)

Consiste em utilizar as instruções MMX disponibilizadas nos novos processadores (Pentium MMX), para realizar tarefas em paralelo. Requer programação em baixo nível. Observe que com swar você pode fazer processamento paralelo em uma máquina com um único processador.

4.2 Processamento paralelo com SMP (Symetric Multi Processor)

SMP é uma sigla que designa computadores com mais de um processador com as mesmas características, daí o termo *Symetric Multi Processor*. Os processadores compartilham o mesmo BUS e a mesma memória. Veja na Figura 3 uma placa mãe modelo ASUS-CUV4X-DLS com dois slots para processadores PIII de 1000MHz.

Figura 3: Placa mãe ASUS-CUV4X-DLS com slot para 2 processadores.



Requisitos: Os programas devem ser desenvolvidos com o uso de múltiplas threads (multi-threadings) ou múltiplos-processos (multi-processing).

Vantagens: Relativamente fácil de programar.

Desvantagens: Requer máquinas com dois ou mais processadores (são máquinas caras).

4.3 Processamento paralelo com cluster Beowulf

Beowulf é uma tecnologia de cluster que agrupa computadores rodando GNU/Linux para formar um supercomputador virtual via processamento paralelo (distribuído). Veja maiores detalhes em [17, 9, 8, 5, 19, 6]. Veja nas Figuras 4 e 5 exemplos de clusters tipo Beowulf.

Requisitos: Conjunto de computadores (sem teclado, sem monitor e sem mouse) conectados em rede para processamento paralelo (uso exclusivo). Requer o uso de uma biblioteca de mensagens como PVM ou MPI, ou o uso de múltiplos processos com o Mosix.

Vantagens: Manutenção facilitada, redução do número de problemas ocasionados pela instalação de pacotes desnecessários. Menor custo das máquinas e de manutenção.

Desvantagens: As máquinas tem seu uso limitado ao processamento definido pelo servidor.

Figura 4: Exemplo de cluster estilo Beowulf (http://loki-www.lanl.gov).

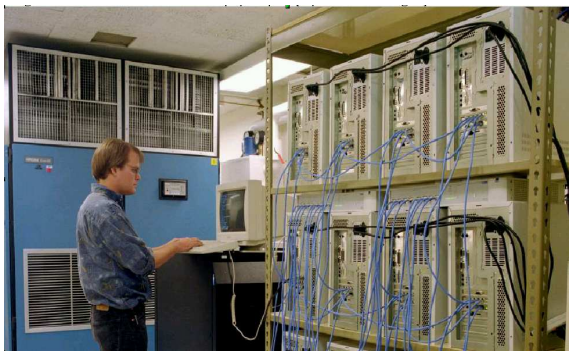
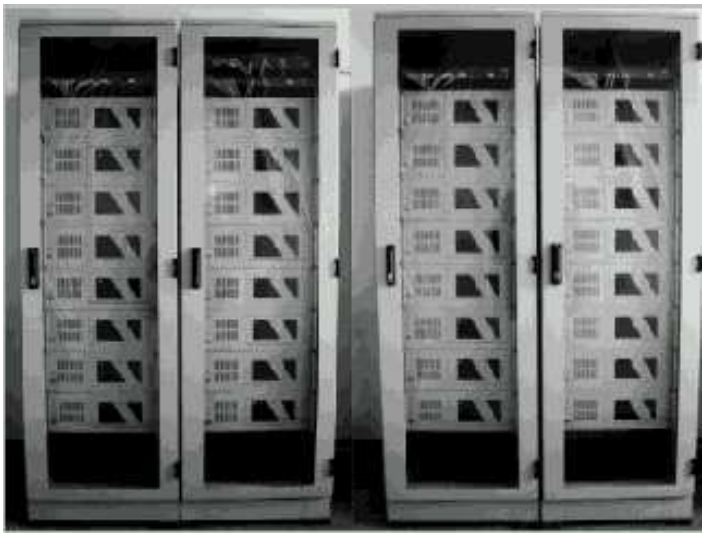


Figura 5: Cluster da University Koblenz-Landau Institute of Physics.



4.4 Processamento paralelo com cluster de workstation

Um cluster de workstation é um conjunto de computadores completos (com teclado, monitor, mouse), conectados em rede, e que cumprem duas funções: i) o uso diário, com diversos tipos de programas como processadores de texto e planilhas, ii) o uso para processamento paralelo pesado no final do dia e/ou nos fins de semana. A Figura 6, mostra um exemplo de cluster de workstation.

Requisitos: As máquinas devem ser completas e independentes. Requer o uso de uma biblioteca de troca de mensagens como PVM ou MPI.

Vantagens: Possibilita o uso das máquinas por diferentes usuários para realização de suas tarefas rotineiras.

Desvantagens: Como vários usuários estão utilizando os processadores para outras tarefas, o desempenho do sistema é reduzido. Na prática reduz o uso do cluster ao final do dia e nos fins de semana. Tem um custo maior por máquina e mais problemas com a manutenção do sistema.

Figura 6: Exemplo de um *"cluster de workstation"* (Distributed Computing Laboratory, Department of Mathematical Sciences, Cameron University, Lawton).

4.5 Processamento paralelo em um cluster com MOSIX

Segundo o site do MOSIX, http://www.mosix.com, o MOSIX é um adendo ao kernel do GNU/LINUX que adiciona ao mesmo capacidades de computação com cluster. Possibilitando que as estações do cluster, baseadas em X86/Pentium/AMD, trabalhem de forma cooperativa, como sendo um único sistema.

A migração dos processos entre as várias máquinas do cluster é automática. Isso permite que programas antigos funcionem num ambiente de cluster com pouquíssimas alterações, [13].

Características do MOSIX: Balanceamento dinâmico e inteligente de carga, uso com cluster heterogêneo, transparência, escabilidade, descentralização e autonomia dos nós. Migração preemptiva dos processos com uso de algoritmos probabilísticos. Comunicação entre núcleos eficiente, controle descentralizado, [http://www.ppgia.pucpr.br/~almendes/DIPC_MOSIX/ppframe.htm].

Requisitos: Requer a recompilação do kernel com a inclusão do Mosix ou instalação de kernel em pacote (como os pacotes rpm do GNU/Linux/RedHat). O site do Mosix apresenta informações detalhadas de como instalar o Mosix.

Vantagens: O trabalho de programação é reduzido, exigindo apenas a implementação dos mecanismos de troca de mensagens entre os diferentes processos. Otimização do uso das máquinas do cluster com a migração automática dos processos.

Desvantagens: Exige a recompilação do kernel.

5 Bibliotecas para desenvolvimento de programas utilizando processamento paralelo[1]

Os programas podem ser desenvolvidos utilizando-se processos (5.1), threads (5.2), ou sistemas de troca de mensagens PVM (5.3), MPI(5.4).

Os dois grandes padrões para troca de mensagens em clusters, são o *Paralel Virtual Machine* (PVM) e o *Message Passing Interface* (MPI). O PVM é o padrão mais antigo, é realmente o nome de uma biblioteca. O MPI é a novidade na área, é um padrão com várias implementações criadas principalmente por universidades e algumas suportadas comercialmente por empresas (adaptado de [13]).

5.1 Processos

De um modo geral, os computadores com sistemas operacionais multi-tarefa disponibilizam um conjunto de funções para divisão e compartilhamento do processador e da memória. Estes sistemas costumam disponibilizar chamadas ao kernel que possibilitam a criação de múltiplos processos. Se a máquina tem mais de um processador, o sistema operacional distribui os processos pelos processadores.

No GNU/Linux e nas variantes do Unix, um processo pode ser clonado com a função fork(). A comunicação entre os processos é feita de forma simplificada com o uso de pipes.

Requisitos: Requer o aprendizado do uso das instruções fork (para clonar processos) e pipe (para comunicação entre os processos).

Vantagens: Pode ser utilizado com Mosix, não sendo necessário acrescentar mecanismos de distribuição dos processos.

Desvantagens: O Mosix só é disponível na plataforma GNU/LINUX.

5.2 Threads

Threads são múltiplos caminhos de execução que rodam concorrentemente na memória compartilhada e que compartilham os mesmos recursos e sinais do processo pai. Uma thread é um processo simplificado, mais leve ou "light", custa pouco para o sistema operacional, sendo fácil de criar, manter e gerenciar.

O padrão de implementação de threads é o *POSIX 1003.1c threads standard*.

Requisitos: Requer o conhecimento da biblioteca de programação com threads (também conhecida como *PThreads*).

Vantagens: Em poucas palavras é o pacote definitivo para o desenvolvimento de programação em larga escala no Linux, [7]. Relativamente fácil de programar. O GNU/Linux tem total suporte ao *Pthread*s.

Desvantagens: Não pode ser utilizado com MOSIX.

Veja uma introdução sobre threads em `http://centaurus.cs.umass.edu/~wagner/threads_html/tutorial.html`, um conjunto de links em `http://pauillac.inria.fr/~xleroy/linuxthreads/`, descrições adicionais nas referências [12, 4, 16, 14, 20, 18]. Veja um exemplo de uso de threads na seção 8.

5.3 PVM (Parallel Virtual Machine)

É a biblioteca mais utilizada para processamento distribuído. É o padrão de fato da indústria de software.

O PVM se baseia em duas primitivas básicas: i) envie mensagem e ii) receba mensagem. É de fácil utilização, mas não é tão poderoso quando comparado com o MPI.

O usuário deve configurar as máquinas para que sejam o mais idênticas possível, facilitando a manutenção e estabelecendo uma relação de confiança entre elas. Usar rhosts e rsh é a forma mais simples de conseguir isso. O usuário roda o gerenciador do PVM, adiciona máquinas ao cluster e depois simplesmente executa o programa feito usando as bibliotecas PVM. Veja mais detalhes em [17, 7, 12, 21] e exemplo de código na seção 9.

Requisitos: Para o desenvolvimento dos programas é necessário conhecer a biblioteca PVM. É um sistema explicito, ou seja, cabe ao programador dividir as tarefas através da troca de mensagens.

Vantagens: Possibilita o uso do processamento distribuído. É o mais utilizado. Alguns programas de engenharia e matemática geram código automaticamente para o PVM.

Desvantagens: Não é mais o padrão. O desenvolvimento dos programas fica bem mais complicado quando comparado com threads.

5.4 MPI (Message Passing Interface)

É um método que inclui conceitos novos como rank (cada processo tem uma identificação única, crescente), group (conjunto ordenado de processos) e communicator (uma coleção de grupos), que permitem um gerenciamento mais complexo (e inteligente) do uso de cada máquina do cluster.

O MPI tem opções mais avançadas (que o PVM), como envio de mensagens broadcast (para todas as máquinas do cluster) e multicast (para um grupo específico de máquinas), assim como um melhor controle sobre o tratamento que cada mensagem terá ao ser recebida por outro ponto do cluster. A configuração do MPI depende da implementação utilizada e algumas delas chegam a instalar front-ends para compiladores em C e Fortran, mas a forma geral de uso é semelhante.

Veja maiores detalhes e exemplo na seção 10.

Requisitos: Requer o conhecimento de um sistema bastante complexo de troca de mensagens, o MPI. É um método explicito.

[1] Existem várias outras opções de gerenciamento e controle da memória e dos processos (como o System V Shared Memory). De uma maneira geral, quanto maior a eficiência desejada, maior a complexidade dos modelos a serem utilizados. Você vai ter de aprender conceitos como atomicidade, volatilidade, travamento de memória, gerenciamento de cache. A dica é iniciar o desenvolvimento de softwares com processamento paralelo utilizando threads, posteriormente, pode-se adotar um sistema mais complexo e com maior desempenho.

Vantagens: É o novo padrão para processamento distribuído, embora ainda seja menos utilizado que o PVM.

Desvantagens: Na prática significa aprender uma nova linguagem de programação. É um padrão da indústria com várias implementações individuais. É complicado.

Descritas as diferentes formas de se implementar processamento paralelo, apresenta-se a seguir alguns exemplos.

6 Exemplo de código comum[2]

Apresenta-se a seguir um pequeno código em C, que será desenvolvido posteriormente utilizando processos, threads, PVM e MPI. O programa recebe como parâmetro um inteiro com o número de intervalos que serão utilizados na estimação do valor de π (pi).

Listing 1: Determinação de pi.

```
//Inclue bibliotecas de C
#include <stdlib.h>
#include <stdio.h>

//Função main
int main(int argc, char *argv[])
{
register double width, sum;
register int intervals, i;   //Número de intervalos
intervals = atoi(argv[1]);
width = 1.0 / intervals;     //largura
sum = 0;
for (i=0; i<intervals; ++i)
{
  register double x = (i + 0.5) * width;
  sum += 4.0 / (1.0 + x * x);
}

sum *= width;

//Mostra o resultado
printf ( "Estimation␣of␣pi␣is␣%f\n", sum);
return(0);
}
```

7 Exemplo de código usando múltiplos processos

Apresenta-se a seguir um pequeno código que cria um processo filho usando a função fork[3]. A seguir troca dados entre os dois processos usando streans e pipes. Observe que a grande vantagem é a simplicidade, o uso de strans ja é conhecido por programadores de C++, a única novidade é a função fork.

Apresenta-se a seguir um pequeno código em C++, que determina o valor de pi usando multiplos processos.

Listing 2: Determinação de pi usando multiplos processos.

```
#include <unistd.h>
#include <cstdlib>
#include <cstdio>
#include <iostream>

#include <fstream>

using namespace std;

double process(int iproc,double intervalos,int nproc);

//----------------------Função main
int main (int argc, char* argv[])
{
  //Obtém o intervalo
  if( argc < 3 )
    {
      cout <<"Uso:␣"<< argv[0] << "␣
          numero_de_intervalos␣numero_de_processos"
           << endl ;
      cout <<"Exemplo:␣"<<argv[0] << "␣1000␣4␣" << endl
           ;
      return -1;
    }
  double intervalos = atof (argv[1]);
  int nproc = atoi (argv[2]);
  cout << argv[0]<< "␣intervalos␣="<< intervalos <<"␣"
      << "nproc=" << nproc << endl;

  //----------------
  //Criando o pipe
  //----------------
  int Fd[2];     //identificador
  pipe (Fd);     //cria pipe

  //----------------
  //Criando streans
  //----------------
  ifstream In;
  ofstream Out;

  //-----------------
  //Criando processos
  //-----------------
  int iproc = 0 ;       //id do processo
  //O processo pai ( Pid = 0 ) vai criar (nproc-1)
     processos
  int Pid = 0;          //id do processo
  for (int n = 0; n < (nproc - 1); n++)
    {
      if( Pid == 0)     //se for o processo pai
        {
        iproc++;        //incrementa o iproc
        Pid = fork ();  //cria processo filho
        }
    }

  //------------------------------
  //se for o processo pai Pid == 0
  //------------------------------
   if (Pid == 0)
    {
      double pi = 0;
      double extern_pi = 0;
      iproc = 0;
      pi = process ( iproc, intervalos, nproc);
      cout << "Pid␣pai␣=␣" << Pid << "␣iproc␣=␣"<<
          iproc << endl;
      cout << "pi␣pai␣=␣" << pi << endl;

      close (Fd[1]);     //fecha pipe
      In.attach (Fd[0]); //conecta para leitura
      for (int n = 0; n < (nproc - 1); n++)
        {
          In >> extern_pi;   //lê valores
          pi += extern_pi;
        }
```

[2] O material que segue é uma cópia parcial da referencia [7].

[3] Testado no RedHat 7.3, gcc 2.96.

```
      In.close ();       //fecha conexão

      cout << "Valor␣estimado␣de␣pi␣=␣" << pi << endl;
    }
  //-----------------------
  //se for o processo filho
  //-----------------------
  else
    {
      double pi = 0;
      cout << "Pid␣filho␣=␣" << Pid  << "␣iproc␣=␣"<<
          iproc << endl;
      pi = process ( iproc, intervalos, nproc);
      cout << "pi␣filho␣=␣" << pi << endl;

      close (Fd[0]);      //fecha o pipe
      Out.attach (Fd[1]); //conecta para escrita
      Out << pi << endl;  //escreve
      Out.close ();       //fecha conexão
    }
  return 0;
  }

//-----------------------Função process
double process (int iproc, double intervalos, int nproc
    )
{
  register double width  , localsum;
  width = 1.0 / intervalos;
  localsum = 0;
  for ( int i = iproc; i < intervalos; i += nproc )
    {
      register double x = (i + 0.5) * width;
      localsum += 4.0 / (1.0 + x * x);
    }

  localsum *= width;
  return (*(new double(localsum)));
}

/*
Informações:
===========
O processo pai tem Pid=0, setado na saída da função
    fork.
Cada processo filho tem um Pid != 0.

Para identificar de forma mais clara os processos, foi
    criada
a variavel iproc, que assume o valor 0 para o processo
    pai
e 1,2,3... sucessivamente para os filhos.

Saída:
======
[andre@mercurio Processos]$ ./a.out 1000 4
./a.out intervalos =1000 nproc=4
Fd[0]= 3
Fd[1]= 4
Pid filho = 12545 iproc = 1
pi filho = 0.785648
Pid filho = 12546 iproc = 2
Pid filho = 12547 iproc = 3
pi filho = 0.784648
[andre@mercurio Processos]$ Pid pai = 0 iproc = 0
pi pai = 0.786148
pi filho = 0.785148
Valor estimado de pi = 3.1416

Para compilar:
==============
g++ processos4.cpp -o processos4
*/
```

8 Exemplo de código usando múltiplas threads

Apresenta-se nesta seção um exemplo[4] de programa usando threads em C.

Para compilar este exemplo use:
g++ -v exemplo-thread.cpp -lpthread -D_REENTRANT.

O programa inicia incluindo as bibliotecas de C e criando algumas variáveis globais. A variável que deve ser compartilhadas é pi, a mesma foi declarada com volatile. Cria uma variável mutex (**pthread_mutex pi_lock;**), que será utilizada para bloquear a parte do código que não pode ser executada ao mesmo tempo pelas duas (ou mais) threads. A seguir, monta uma função separada para a seção que tem código concorrente (paralelizado). Por padrão, a função deve retornar void* e receber void* (**void* FunçãoThread(void*arg)**). Dentro de main(), Inicializa a variável mutex (**pthread_mutex_init(&pi_lock,val)** e cria as diferentes threads (**pthread_t t1,t2,...,tn;**). Dispara a execução de cada thread usando chamadas a **pthread_create(&t1,NULL,process,&arg)**. A função **pthread_join(thread,&retval)** é usada para esperar o encerramento da thread e obter o valor de retorno da função process executada pela thread. Observe que você deve usar pthread_join para evitar que o programa principal encerre sua execução antes das threads terem sido concluídas.

Observe dentro da função process, o uso do bloquador de acesso (o mutex) usando **pthread_mutex_lock(&pi_lock)** e **pthread_mutex_unlock(&pi_lock).**

Listing 3: Determinação de pi usando múltiplas threads.

```
#include <stdio.h>
#include <stdlib.h>
#include "pthread.h"

volatile double pi = 0.0;
volatile double intervals;
pthread_mutex_t pi_lock;

//Função thread
void * process (void *arg)
{
  register double width, localsum;
  register int i;
  register int iproc = (*((char *) arg) - '0');

  width = 1.0 / intervals;
  localsum = 0;
  for (i = iproc; i < intervals; i += 2)
    {
      register double x = (i + 0.5) * width;
      localsum += 4.0 / (1.0 + x * x);
    }

  localsum *= width;

  /*trava acesso a esta parte do código, altera pi, e
      destrava*/
  pthread_mutex_lock (&pi_lock);
   pi += localsum;
```

[4] Este exemplo obedece o padrão POSIX, podendo ser utilizado em outras plataformas (como Windows).

```
  pthread_mutex_unlock (&pi_lock);
  return (NULL);
}

int main (int argc, char *argv[])
{
  pthread_t thread0, thread1;
  void *retval;

  intervals = atoi (argv[1]);

  /* Inicializa a variável mutex*/
  pthread_mutex_init (&pi_lock, NULL);

  /* Executa duas threads */
  if (pthread_create (&thread0, NULL, process, (void*)
      "0") ||
      pthread_create (&thread1, NULL, process, (void*)
          "1"))
    {
      fprintf (stderr, "%s:␣cannot␣make␣thread\n", argv
          [0]);
      exit (1);
    }


  /* Join espera as threads terminarem, o retorno é
      armazenado em retval */
  if (pthread_join (thread0, &retval) || pthread_join (
      thread1, &retval))
    {
      fprintf (stderr, "%s:␣thread␣join␣failed\n", argv
          [0]);
      exit (1);
    }

  printf ("Estimation␣of␣pi␣is␣%f\n", pi);
  return 0;
}

//g++ -v exemplo-thread.cpp -lpthread -o exemplo-thread
```

9 Exemplo de código usando PVM[5]

PVM (Parallel Virtual Machine) is a freely-available, portable, message-passing library generally implemented on top of sockets. It is clearly established as the de-facto standard for message-passing cluster parallel computing. PVM supports single-processor and SMP Linux machines, as well as clusters of Linux machines linked by socket-capable networks (e.g., SLIP, PLIP, Ethernet, ATM). In fact, PVM will even work across groups of machines in which a variety of different types of processors, configurations, and physical networks are used - Heterogeneous Clusters - even to the scale of treating machines linked by the Internet as a parallel cluster.

PVM also provides facilities for parallel job control across a cluster. Best of all, PVM has long been freely available (currently from http://www.epm.ornl.gov/pvm/pvm_home.html), which has led to many programming language compilers, application libraries, programming and debugging tools, etc., using it as their "portable message-passing target library." There is also a network newsgroup, comp.parallel.pvm.

It is important to note, however, that PVM message-passing calls generally add significant overhead to standard socket operations, which already had high latency. **Further, the message handling calls themselves do not constitute a particularly "friendly" programming model**.

Using the same Pi computation example, the version using C with PVM library calls is:

Listing 4: Determinação de pi usando PVM.

```
#include <stdlib.h>
#include <stdio.h>
#include <pvm3.h>
#define NPROC   4

main(int argc, char *argv[])
{
  register double lsum, width;
  double sum;
  register int intervals, i;
  int mytid, iproc, msgtag = 4;
  int tids[NPROC];  /* array of task ids */

  /* enroll in pvm */
  mytid = pvm_mytid();
  /* Join a group and, if I am the first instance,
     iproc=0, spawn more copies of myself
  */
  iproc = pvm_joingroup("pi");

  if (iproc == 0)
    {
    tids[0] = pvm_mytid();
    pvm_spawn("pvm_pi", &argv[1], 0, NULL, NPROC-1, &
        tids[1]);
  }

  /* make sure all processes are here */
  pvm_barrier("pi", NPROC);

  /* get the number of intervals */
  intervals = atoi(argv[1]);
  width = 1.0 / intervals;
  lsum = 0.0;
  for (i = iproc; i<intervals; i+=NPROC)
    {
    register double x = (i + 0.5) * width;
    lsum += 4.0 / (1.0 + x * x);
    }

  /* sum across the local results & scale by width */
  sum = lsum * width;
  pvm_reduce(PvmSum, &sum, 1, PVM_DOUBLE, msgtag, "pi"
      , 0);

  /* have only the console PE print the result */
  if (iproc == 0)
    {
    printf("Estimation␣of␣pi␣is␣%f\n", sum);
    }
  /* Check program finished, leave group, exit pvm */
  pvm_barrier("pi", NPROC);
  pvm_lvgroup("pi");
  pvm_exit();
  return(0);
}
```

[5] O material que segue é uma cópia parcial da referencia [7].

10 Exemplo de código usando MPI[6]

Although PVM is the de-facto standard message-passing library, MPI (Message Passing Interface) is the relatively new official standard. The home page for the MPI standard is http://www.mcs.anl.gov:80/mpi/ and the newsgroup is comp.parallel.mpi.

However, before discussing MPI, I feel compelled to say a little bit about the PVM vs. MPI religious war that has been going on for the past few years. I'm not really on either side. Here's my attempt at a relatively unbiased summary of the differences:

1. Execution control environment. Put simply, PVM has one and MPI doesn't specify how/if one is implemented. Thus, things like starting a PVM program executing are done identically everywhere, while for MPI it depends on which implementation is being used.

2. Support for heterogeneous clusters. PVM grew-up in the workstation cycle-scavenging world, and thus directly manages heterogeneous mixes of machines and operating systems. In contrast, MPI largely assumes that the target is an MPP (Massively Parallel Processor) or a dedicated cluster of nearly identical workstations.

3. Kitchen sink syndrome. PVM evidences a unity of purpose that MPI 2.0 doesn't. The new MPI 2.0 standard includes a lot of features that go way beyond the basic message passing model - things like RMA (Remote Memory Access) and parallel file I/O. Are these things useful? Of course they are... but learning MPI 2.0 is a lot like learning a complete new programming language.

4. User interface design. MPI was designed after PVM, and clearly learned from it. MPI offers simpler, more efficient, buffer handling and higher-level abstractions allowing user-defined data structures to be transmitted in messages.

5. The force of law. By my count, there are still significantly more things designed to use PVM than there are to use MPI; however, porting them to MPI is easy, and the fact that MPI is backed by a widely-supported formal standard means that using MPI is, for many institutions, a matter of policy.

Conclusion? Well, there are at least three independently developed, freely available, versions of MPI that can run on clusters of Linux systems (and I wrote one of them):

LAM (Local Area Multicomputer) is a full implementation of the MPI 1.1 standard. It allows MPI programs to be executed within an individual Linux system or across a cluster of Linux systems using UDP/TCP socket communication. The system includes simple execution control facilities, as well as a variety of program development and debugging aids. It is freely available from http://www.osc.edu/lam.html.

MPICH (MPI CHameleon) is designed as a highly portable full implementation of the MPI 1.1 standard. Like LAM, it allows MPI programs to be executed within an individual Linux system or across a cluster of Linux systems using UDP/TCP socket communication. However, the emphasis is definitely on promoting MPI by providing an efficient, easily retargetable, implementation. To port this MPI implementation, one implements either the five functions of the "channel interface" or, for better performance, the full MPICH

[6] O material que segue é uma cópia parcial da referencia [7].

10.1 Exemplo de código usando MPI_1.0

Listing 5: Determinação de pi usando MPI 1.0.

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>

main(int argc, char *argv[])
{
register double width;
double sum, lsum;
register int intervals, i;
int nproc, iproc;
MPI_Status status;

//Inicializa o MPI
if (MPI_Init(&argc, &argv) != MPI_SUCCESS) exit(1);
//Dispara n processos
MPI_Comm_size(MPI_COMM_WORLD, &nproc);
//Obtém id deste processo (cada processo um id
    diferente)
MPI_Comm_rank(MPI_COMM_WORLD, &iproc);

//***inicio do código paralelo
intervals = atoi(argv[1]);
width = 1.0 / intervals;
lsum = 0;

for (i=iproc; i<intervals; i+=nproc)
  {
    register double x = (i + 0.5) * width;
    lsum += 4.0 / (1.0 + x * x);
  }
lsum *= width;
//***fim do código paralelo
 if (iproc != 0) //se for processo secundário (slave)
   {             //envia mensagem para processo
       primário
    MPI_Send(&lbuf, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD
        );
  }
 else
   {            //se for processo master (pid==0)
     sum = lsum;//acumula valores de lsum de cada
         processo
    for (i=1; i<nproc; ++i)
      {
        MPI_Recv(&lbuf, 1, MPI_DOUBLE, MPI_ANY_SOURCE,
                 MPI_ANY_TAG, MPI_COMM_WORLD, &status)
                     ;
        sum += lsum;
   }
printf("Estimation␣of␣pi␣is␣%f\n", sum);
}
 //Finaliza o MPI
MPI_Finalize();
return(0);
}
```

10.2 Exemplo de código usando MPI_1.1

The second MPI version uses collective communication (which, for this particular application, is clearly the most appropriate):

Listing 6: Determinação de pi usando MPI 1.1.

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>

main(int argc, char *argv[])
{
register double width;
double sum, lsum;
register int intervals, i;
int nproc, iproc;
if (MPI_Init(&argc, &argv) != MPI_SUCCESS) exit(1);
MPI_Comm_size(MPI_COMM_WORLD, &nproc);
MPI_Comm_rank(MPI_COMM_WORLD, &iproc);

intervals = atoi(argv[1]);
width = 1.0 / intervals;
lsum = 0;

for (i=iproc; i<intervals; i+=nproc)
  {
    register double x = (i + 0.5) * width;
    lsum += 4.0 / (1.0 + x * x);
  }
lsum *= width;
MPI_Reduce(&lsum, &sum, 1, MPI_DOUBLE,
MPI_SUM, 0, MPI_COMM_WORLD);

if (iproc == 0)
  {
    printf("Estimation␣of␣pi␣is␣%f\n", sum);
  }
MPI_Finalize();
return(0);
}
```

10.3 Exemplo de código usando MPI_2

The third MPI version uses the MPI 2.0 RMA mechanism for each processor to add its local lsum into sum on
processor 0:

Listing 7: Determinação de pi usando MPI 2.0.

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>

main(int argc, char *argv[])
{
register double width;
double sum = 0, lsum;
register int intervals, i;
int nproc, iproc;
MPI_Win sum_win;

if (MPI_Init(&argc, &argv) != MPI_SUCCESS) exit(1);
MPI_Comm_size(MPI_COMM_WORLD, &nproc);
MPI_Comm_rank(MPI_COMM_WORLD, &iproc);
MPI_Win_create(&sum, sizeof(sum), sizeof(sum),0,
    MPI_COMM_WORLD, &sum_win);
MPI_Win_fence(0, sum_win);

intervals = atoi(argv[1]);
width = 1.0 / intervals;
lsum = 0;

for (i=iproc; i<intervals; i+=nproc)
  {
    register double x = (i + 0.5) * width;
    lsum += 4.0 / (1.0 + x * x);
  }
lsum *= width;
MPI_Accumulate(&lsum, 1, MPI_DOUBLE, 0, 0,1, MPI_DOUBLE
    , MPI_SUM, sum_win);
MPI_Win_fence(0, sum_win);

if (iproc == 0)
  {
    printf("Estimation␣of␣pi␣is␣%f\n", sum);
  }
MPI_Finalize();
return(0);
}
```

It is useful to note that the MPI 2.0 RMA mechanism very neatly overcomes any potential problems with the corresponding data structure on various processors residing at different memory locations. This is done by referencing a "window" that implies the base address, protection against out-of-bound accesses, and even address scaling. Efficient implementation is aided by the fact that RMA processing may be delayed until the next MPI_Win_fence. In summary, the RMA mechanism may be a strange cross between distributed shared memory and message passing, but it is a very clean interface that potentially generates very efficient communication.

11 Como se aprofundar (leituras aconselhadas)[7]

11.0.1 Iniciar com os HOWTOS

- **Beowulf Howto**: Um como fazer, que descreve uma série de conceitos e técnicas para implementação de um cluster (http://www.sci.usq.edu.au/staff/jacek/beowulf/BDP).
- **Linux Parallel Processing HOWTO**: Um como fazer, que descreve três tipos de processamento paralelo (swar, SMP, cluster) e conceitos básicos de cluster (http://yara.ecn.purdue.edu/~pplinux/PPHOWTO/pphowto.html).
- **C++ Programming Howto**: Um como fazer que descreve algumas características da linguagem C++, exemplos de uso da STL.

11.0.2 Verificar a Home Page do Beowulf

- Beowulf Homepage (http://www.beowulf.org).
- **Beowulf Installation and Administration HOWTO** (http://www.sci.usq.edu.au/staff/jacek/beowulf/BDP).
- Beowulf Underground (http://beowulf-underground.org).

[7] Dica: Dê preferencia para material em PDF, é fácil de ler no micro, imprimir com qualidade e possibilita a abertura de links da internet. É universal (Windows/-Mac/Unix/Linux).

- Beowulf mailing list. {beowulf-request@cesdis.gsfc.nasa.gov}.
- Extreme Linux (http://www.extremelinux.org).
- Extreme Linux Software from Red Hat (http://www.redhat.com/extreme).
- **Building a Beowulf System** (http://www.cacr.caltech.edu/beowulf/tutorial/building.html).
- Programas na página do Beowulf {http://beowulf.gsfc.nasa.gov/software/software.html}
- Jacek's Beowulf-utils {ftp://ftp.sci.usq.edu.au/pub/jacek/beowulf-utils}
- bWatch - cluster monitoring tool {http://www.sci.usq.edu.au/staff/jacek/bWatch}

11.0.3 Bibliotecas PVM, MPI e Threads

- **PVM - Parallel Virtual Machine** {http://www.epm.ornl.gov/pvm/pvm_home.html}
- **LAM/MPI (Local Area Multicomputer / Message Passing Interface** {http://www.mpi.nd.edu/lam}
- **Threads** (http://www.humanfactor.com/pthreads/).

12 Conclusões

O uso do processamento paralelo em um cluster de workstation é uma ferramenta extremamente poderosa, possibilitando o desenvolvimento de simulações avançadas em sistemas de baixo custo. Como visto, os computadores podem ser utilizados para processamento comum de dia e para processamento pesado a noite e nos finais de semana, aproveitando melhor o parque de máquinas instaladas.

Os programadores precisam aprender os conceitos básicos de processamento paralelo e as diferentes formas de distribuição do processsamento (processos, threads, PVM, MPI). Os conceitos básicos destes sistemas, um pequeno exemplo e referências externas foram apresentados.

O mecanismo mais fácil de desenvolver processamento paralelo envolve a utilização de multiplas threads, e a seguir multiplos-processos sendo aconselhável iniciar com estes mecanismos. Posteriormente, pode-se trabalhar com bibliotecas especializadas como PVM e MPI. Em ambos os casos procure utilizar uma biblioteca padrão e multi-plataforma.

Referências

[1] André Duarte Bueno. *Introdução ao Processamento Paralelo e ao Uso de Clusters, Parte I: Filosofia*, 2002.

[2] André Duarte Bueno, Fabio Santana Magnani, and Paulo Cesar Philippi. Método para determinação da permeabilidade relativa de rochas reservatório de petróleo através da análise de imagens reconstruídas. page 12, Caxambú - MG - Brasil, 2002. CIT02-0672.

[3] André Duarte Bueno and Paulo Cesar Philippi. Modelo do grafo de conexão serial para determinação da permeabilidade de rochas reservatório de petróleo. page 12, Caxambú - MG - Brasil, 2002. CIT02-0668.

[4] David R. Butenhof. *Programming with POSIX(R) Threads*. Addison-Wesley, 1987.

[5] Daniel Ridge Daniel Savarese Donald Becker Chance Reschke, Thomas Sterling and Phillip Merkey. A design study of alternative network topologies for the beowulf parallel workstation. *Fifth IEEE International Symposium on High Performance Distributed Computing*, 1996.

[6] Phillip Merkey Thomas Sterling Becker Daniel Ridge, Donald Becker and Phillip Merkey. Harnessing the power of parallelism in a pile-of-pcs. *IEEE Aerospace*, 1997.

[7] Hank Dietz. *Linux Parallel Processing HOWTO*. http://yara.ecn.purdue.edu/ pplinux/PPHOWTO/pphowto.html, 1998.

[8] Daniel Savarese Bruce Fryxell Kevin Olson Donald J. Becker, Thomas Sterling. Communication overhead for space science applications on the beowulf parallel workstation. *High Performance and Distributed Computing*, 1995.

[9] Daniel Savarese John E. Dorband Udaya A. Ranawak Charles V. Packer Donald J. Becker, Thomas Sterling. Beowulf: A parallel workstation for scientific computation. *International Conference on Parallel Processing*, 1995.

[10] Luis Orlando Emerich dos Santos, Paulo Cesar Philippi, and M C.Damiani. A boolean lattice gas method for predicting intrinsic permeability of porous medias. Puerto Iguazu, May 08-12-2000.

[11] Al Geist, Adam Beguelin, and Jack Dongarra. *PVM: Parallel Virtual Machine*. MIT Press, 1994.

[12] Cameron Hughs and Tracey Hughes. *Object Oriented Multithreading using C++: architectures and components*, volume 1. John Wiley Sons, 2 edition, 1997.

[13] Guilherme Wunsch Manika. *Super-Computador a Preço de Banana*, volume 2. Revista do Linux, 1999.

[14] Brian Masney. Introdutcion to multi-thread programming. Linux Journal, april 1999.

[15] Peter Pacheco. *Parallel Programming With MPI*. Morgan Kaufmann Publishers, 1996.

[16] LinuxThreads Programming. Matteo dell omodarme.

[17] Jacek Radajewski and Douglas Eadline. *Beowulf HOWTO*. http://www.sci.usq.edu.au/staff/jacek/beowulf/BDP, 1998.

[18] Bryan Sullivan. Faq - threads - http://www.serpentine.com/ bos/threads-faq/, 1996.

[19] Daniel Savarese Michael R. Berry Thomas Sterling, Donald J. Becker and Chance Res. Achieving a balanced low-cost architecture for mass storage management through multiple fast ethernet channels on the beowulf parallel workstation. *International Parallel Processing Symposium*, 1996.

[20] Tom Wagner and Don Towsley. Getting started with posix threads. University of Massachusetts at Amherst, july 1995.

[21] Kurt Wall. *Linux Programming Unleashed*, volume 1. SAMS, 2 edition, 2001.

[22] Barry Wilkinson and C. Michael Allen. *Parallel Programming: Techniques and Applications Using Workstation and Parallel Computers*. Prentice Hall, 1999.

A Apêndice A- Hardware para processamento paralelo

Descreve-se nesta seção o hardware básico, necessário, para dar suporte ao processamento paralelo (usando SMP ou cluster).

Processadores: Com relação aos processadores, pode-se utilizar as especificações de multi-processamento da Intel para MPS, ou da SUN (Sun4m).

Cache: Uso de cache nível 2.

Memória: edo dram, sdram, hdram.

BUS: Barramentos: ISA/EIDE, SCSI, PCI

A Tabela 1 apresenta as principais características relacionadas a troca de dados utilizando-se as interfaces disponíveis.

Tabela 1: Hardware de rede.

Tipo	PLIP	Eth10	Et100	SCSI	ATM	FC
largura banda[mb/s]	1.2	10	100	4000	155	1062
latência[μs]	1000	100	80	2.7	120	?
custo[U$]	2	50	400	1000	3000	?

B Apêndice B- Glossário

Descreve-se a seguir alguns termos usuais da área de processamento paralelo.

SIMD: Single Instruction Stream, Multiple Data. Todos os processadores executam a mesma operação no mesmo tempo. Muito usado no processamento de vetores e matrizes, fácil de implementar. Implementado usando swar ou smp.

MIMD: Multiple Instruction Stream, Multiple Data Stream. Cada processador atua de forma independente. Implementado usando cluster.

SPMD: Single Program, Multiple Data. Todos os processadores rodam o mesmo programa. Implementado usando cluster.

Largura de banda: é o número de *canos* por onde os dados *fluem*. Quanto maior a largura de banda, maior a velocidade com que os dados fluem do processador para a memória e para os dspositivos (placa de rede, monitor).

Latência: Tempo perdido com a comunicação entre os diversos dispositivos. Veja tabela 1.

Troca de mensagens: Sistema pelo qual o processamento é distribuído pelas várias máquinas do cluster. Os dados são enviados de uma máquina para outra, processados e devolvidos. As bibliotecas para troca de mensagem mais usados são o MPI e o PVM.

Memória compartilhada: Em máquinas com mais de um processador (SMP), a memória costuma ser compartilhada (os dois processadores podem acessar a mesma memória).

RAID: Redundanty Array of Inexpensive Disks. Consiste em conectar ao computador um conjunto de discos rígidos de alto desempenho para armazenamento dos dados em paralelo.
Ex: cada elemento de uma matriz seria armazenado em um disco rígido diferente.

DSP: Digital Signal Processing. Dispositivos conhecidos, como placas de som e de vídeo, costumam ter chips para realização de processamento de sinais (DSP). Estes mecanismos podem ser acessados pelo seu programa, para realização de tarefas específicas. Ex: Processamento da transformada de Fourier FFT.

Single tasking operating system: Sistema operacional que roda somente uma tarefa de cada vez.
Ex: DOS.

Multi tasking operating system: Sistema operacional que roda mais de uma tarefa de cada vez. Cada tarefa usa o processador por um determinado tempo. Ex: Sun-Solaris rodando na enterprise.

Multi tasking operating system em sistemas multi-processados: O sistema operacional é multi-tarefa e o computador tem mais de um processador.

Multi-session: Um sistema operacional multi-session admite que mais de um usuário esteja logado, rodando mais de uma seção.

Multi-processing: Um sistema operacional que possibilita a execução de mais de um processo por seção.

Multi-threading: Um sistema operacional que possibilita que um processo tenha mais de uma thread.

C Apêndice C - Roteiro para paralelização e otimização de algoritmos

Apresenta-se a seguir um roteiro básico para paralelização e otimização de programas.

- Identificação de erros lógicos nos algoritmos que impliquem em processamento desnecessário.
- Avaliação geral da performance do software (com uso de um profiler[8]). Avaliação das partes do programa em que vale a pena o processamento paralelo.
- Uso de opções de otimização do compilador (-O1,-O2,-O3).
- Identificação das partes do programa que são concorrentes.
- Implementação dos algoritmos usando threads ou um dos métodos para troca de mensagens (MPI/PVM).

D Apêndice - HOWTO INSTALL MOSIX

No artigo [1], falou-se do Mosix. Apresenta-se aqui, as instruções para instalação do Mosix em sistemas GNU/Linux.

Listing 8: Roteiro para instalação do MOSIX.

```
HOWTO - COMO INSTALAR O KERNEL COM O MOSIX:
===========================================

1- Fazer o download do kernel 2.4.19 do site
   ftp://ftp.kernel.org

2- Mover o arquivo para o diretório /usr/src
   mv linux-2.4.19.tar.bz2  /usr/src

3- Descompactar o kernel
   cd /usr/src
   tar -xvjf linux-2.4.19.tar.bz2

   Remover o link antigo e aplicar novo
   rm linux-2.4
   ln -s linux-2.4.19 linux-2.4

4- Fazer o download da patch do kernel do mosix
   MOSKRN-1.8.0.tgz
   tar -xvzf MOSKRN-1.8.0.tgz
   mv MOSKRN-1.8.0/MOSKRN-1.8.0/patches.2.4.19 /usr/src
       /linux-2.4.19

5- Aplicar a patch
   cd /usr/src/linux-2.4.19
   cat patches.2.4.19 | patch -p0

6-  Executar os passos de compilação do kernel.
    make mrproper
    make xconfig
    ...configura o kernel...
    ...salva configuração com nome: kernel-2-4-19-
        comMOSIX-v1
    ...abre o arquivo Makefile e inclue informacao
        MOSADB
    make dep
    make clean
    make modules
    make modules_install
    make bzImage

    cp /usr/src/linux-2.4/arch/i386/boot/bzImage /boot/
        vmlinuz-versao
    cp /usr/src/linux-2.4/System.map /boot/System.map-
        versao

7-  Inclue o novo kernel no programa de boot
    emacs /boot/grub/grub.conf

    ...adiciona informações do novo kernel..
    ...salva..

8- O Mosix tem um conjunto de pacotes que são usados
    para gerenciamento
   do sistema. Faça o download do arquivo rpm destes
       pacotes (ou tar.gz),
   e instale os mesmos em sua máquina.

9- Reinicializa o sistema e testa o novo kernel.
```

[8] Um profiler é um programa utilizado para identificar os gargalos de desempenho de seu programa. Você executa seu programa dentro do profiler e ele fornece uma estatística das funções e seus tempos de processamento.