CBPF-NT-001/97

Programação Paralela na Rede UNIX

Renato M. Silva & Marita Maestrelli

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

Prefácio

Afim de aumentar o poder de processamento das máquinas da Rede SUN-CBPF sem porém fazermos um *upgrade* das máquinas, resolvemos instalar nesta Rede o programa PVM (Parallel Virtual Machine), próprio para processamento paralelo em arquiteturas heterogêneas, o que permitirá que os usuários tenham também um grande ganho na performance de seus programas, se desenvolvido em paralelo.

O trabalho apresentado nesta nota técnica foi desenvolvido sob orientação do Doutor **Márcio Portes de Albuquerque** do corpo técnico do CAT e apresentado no Encontro Nacional de Física da Matéria Condensada de 1996, realizado em Águas de Lindóia.

Índice

1 Introdução **3**

2 Paralelismo **4**
2.1 Conceitos básicos 4
2.1.1 Arquitetura 4
2.1.2 Conceitos de programação paralela 4
2.1.3 Técnicas de programação paralela 5

3 Parallel Virtual Machine - PVM **6**
3.1 Princípios do PVM 7
3.2 Organização de Tarefas no PVM 7
3.3 Funções do PVM3 7

4 Exemplos **11**
4.1 O *hello.c* 11
4.2 Simulações numéricas no PVM 13

5 Conclusão **14**

Bibliografia **15**

1 Introdução

Com o recente avanço tecnológico nas diversas áreas do conhecimento, busca-se cada vez mais novos métodos de pesquisa, no sentido de reduzir custos e otimizar a obtenção de resultados. Assim atentou-se para o fato de que a computação, com suas máquinas cada vez mais poderosas e plataformas de alto desempenho torna cada vez mais viável a substituição, com vantagens, dos experimentos de bancada por simulações em modelos computacionais, seja de experimentos do próprio homem ou até mesmo de simulações de fenômenos naturais. O primeiro problema com o qual deparou-se foi o fato de que tais simulações requeriam uma enorme quantidade de cálculo e elevado número de interações o que, mesmo para a tecnologia atual, toma tempo proibitivo caso se queira resultados precisos.

Assim, surgiu inicialmente na forma de um computador capaz de resolver equações diferenciais em paralelo, projetado por Vanevaur Bush em 1920, a idéia do paralelismo. No entanto, o marco fundamental foi o surgimento de máquinas para processamento paralelo, como o ILLIAV IV, construído nos fins de 60 na universidade de Illinois composto de 64 processadores. Atualmente, pode-se contar com máquinas muito poderosas compostas de, até mesmo, milhares de processadores em um único computador.

Tais computadores porém tem custo muito elevado, sendo por vezes inviável a sua aquisição por instituições sem muitos recursos financeiros. Dessa forma, com a popularização do ambiente em rede local surgiram alternativas como a implantação de sistemas paralelos compostos de diversas máquinas, cada uma com CPU própria, realizando a comunicação entre si através de barramento *Ethernet*.

Seguindo a filosofia do paralelismo em redes locais, o CBPF, em conjunto com o FERMILAB, através do Lafex, desenvolveu o *ACP* [1] com a finalidade de processar a grande quantidade de dados gerada em experiências realizadas naquele instituto. O projeto do *ACP*, embora de grande valia na época foi desativado para poder dar lugar à processadores Paralelos IBM SP-2[2] de última geração .

Atualmente o modelo paralelo instalado no CBPF é baseado no **PVM** (Parallel Virtual Machine). Iniciado em 1989, no ORNL [3] o **PVM** é um projeto atualmente desenvolvido por *Vaidy Sunderman* na Universidade de Emory, *Al Geist* em ORNL, entre outros que visa, principalmente o progresso na Ciência, sendo usado em várias instituições no mundo.

O **PVM3** é um software que nos permite, a partir de uma rede heterogênea de computadores Unix, chamada de *Máquina Virtual*, a distribuição e comunicação entre as diversas tarefas que compõem um programa paralelo. Sob o **PVM** uma coleção de computadores seriais, paralelos e vetoriais, aparecem como um único computador de memória distribuída. Ele fornece as ferramentas para lançar tarefas na máquina virtual e permite que estas se comuniquem durante o processamento, afim de solucionar o problema.

---

[1] Advanced Computer Program

[2] Super Parallel-2

[3] Oak Ridge National Laboratory

2 Paralelismo

2.1 Conceitos básicos

2.1.1 Arquitetura

Se divide, basicamente em dois tipos à saber:

- SIMD

  A arquitetura SIMD[4] consiste basicamente em um grande número de processadores que executam, simultaneamente uma mesma instrução em diferentes tipos de dados. Esta organização é bem aplicada quando no problema há um estrutura repetitiva de dados. As máquinas SIMD são antigas e sua utilização foi importante para o aparecimento de modelos de linguagens de programação e de algoritmos

- MIMD

  A segunda arquitetura, MIMD[5], consiste num conjunto de processadores, operando de modo independente, havendo assim uma maior possibilidade de paralelização de tarefas.

  As máquinas MIMD possuem duas subdivisões:

  1. Memória comum - As tarefas das diversas CPU's acessam a mesma área de memória. Nessa subdivisão o ponto mais crítico é o estrangulamento que ocorre quando as diversas CPU's tentam acessar uma mesma área. Isto faz com que algumas CPU's fiquem ociosas, durante o período de acesso de outras à memória.
  2. Memória distribuída - Neste tipo, em que cada CPU possui uma área própria, apesar de não trazer um ponto de estrangulamento, se faz necessária uma maior troca de dados entre as diversas CPU's para todas terem os dados atualizados a medida que se desenrola a execução do programa.

2.1.2 Conceitos de programação paralela

A programação paralela consiste, basicamente em se dividir um programa em vários módulos, a serem executados em diferentes estações paralelamente, visando a solução do problema. Cada módulo, comumente chamado de "tarefa" trabalha em cima da parte de dados que lhe é conferida e, de acordo com a estrutura do programa, troca os resultados com outras tarefas ou com uma tarefa mestre. Essa troca de resultados é feita através do que se conhece por *message passing*, que pode se dar entre CPU's de uma mesma máquina ou até mesmo entre máquinas independentes.

Ao criar um programa paralelo, é fundamental se ter em mente quais partes do código são paralelizáveis e como dividir as tarefas entre as CPU's. De maneira geral, as partes do código paralelizáveis consistem de funções do programa que não dependem umas das

[4]Single Instruction Multiple Data

[5]Multiple Instruction Multiple Data

outras e que podem ser lançadas simultaneamente. Um exemplo típico de código paralelizável seria o problema de calcular a integral de uma função pelo método de somação simples. A forma mais direta de resolver este programa em paralelo seria dividir a função em partições e passar para diferentes CPU's cada uma dessas partições de modo que cada CPU calcule a integral em sua pequena região. Ao final do cálculo cada CPU passaria o seu resultado para uma outra CPU que se encarregaria de somar os valores, obtendo assim a integral.

Um conceito importante na programação paralela é o *speedup*, que consiste na razão entre o tempo de execução do programa serial e a execução em paralelo.

2.1.3 Técnicas de programação paralela

A programação paralela de algoritmos envolve algumas considerações básicas importantes.

– Gerência de processos paralelos

Ao desenvolver um algoritmo paralelo devemos ter sempre em mente o que se quer paralelizar, o custo de processamento de cada tarefa (*i.e.* quanta carga cada tarefa vai exigir da CPU) e qual a arquitetura de máquinas que possuímos. Assim, ao se dividir um problema temos que notar a importância de balancear a divisão dos processos de modo que uma CPU mais poderosa não fique sub-utilizada ou que uma com menos poder de processamento não seja sobrecarregada. Este item envolve conceitos de granularidade e considerações sobre tempo de rede, que veremos em 2.1.3.

– Comunicação entre os processos

Sem dúvida a comunicação é o maior gargalo da execução de programas em paralelo. Assim, devemos procurar minimizá-lo ao máximo, afim de evitarmos uma queda de performance na execuçãoÀ maneira mais óbvia de minimizar a comunicação é diminuir o número de processadores envolvidos na solução do problema. Mais adiante introduzimeros o conceito de granularidade, que está intimamente relacionado com a divisão das tarefas pelas CPU's envolvidas.

– Sincronização

Uma última consideração importante é a sincronização entres as tarefas. Uma vez que as CPU's geralmente poderão possuir carga e poder de processamento diversos, há a possibilidade de que uma CPU termine a sua tarefa antes de outra e tenha que passar dados para aquela que ainda está trabalhando. Assim é importante que o programador tenha uma boa noção da ordem de execução dos processos para que não deixe um processo esperando por muito tempo a resposta de outra CPU, o que certamente atrasa a execução.

Dessa forma vemos que a programação paralela não envolve somente o desenvolvimento em si, mas também várias considerações importantes no sentido de otimizar o código com finalidade do obter o maior ganho possível em tempo.

A estruturação de um programa paralelo pode ser segundo alguns algoritmos básicos:

- Paralelismo de dados. Nesta técnica, uma grande quantidade de dados é submetida a um processamento idêntico ou similar em diferentes CPU's paralelamente. O paralelismo de dados é de fato a denominação de diversos tipos de paralelismo.

- Algoritmo relaxado. Neste algoritmo, os processos paralelos rodam de maneira auto suficiente, cada um com a sua porção de dados sem qualquer sincronização ou comunicação entre eles.

- Interação síncrona. Cada processador realiza o mesmo trabalho em diferentes porções de dados. Neste algoritmo, entretanto, é necessária uma sincronização e atualização dos dados afim de usar-se o resultado de uma interação nas interações subseqüentes. Muitas simulações quando convertidas para o ambiente paralelo, tomam a forma deste algoritmo.

– Granularidade

Um conceito muito importante em paralelismo é o de granularidade, que consiste no "grau de divisão" do dado de um problema entre as CPU's que participarão da solução. Assim, a granularidade está diretamente relacionada com o tempo de processamento total (quanto maior ela for, menor será o tempo de processamento) e com o tempo de comunicação entre as CPU's (uma vez que quanto mais CPU's participarem da tarefa, maior será o tempo de comunicação).

3 Parallel Virtual Machine - PVM

É importante, antes de falarmos de PVM em si, vermos algumas definições e conceitos intrínsecos à este software. Um dos primeiros conceitos é o de ***máquina virtual*** que na realidade é o conjunto de máquinas, cada uma com sua CPU operando, através do PVM como uma máquina única. Esta ***máquinar virtual*** é criada pelo programador que então dividirá o programa em processos para rodarem nas CPU's que a compõem.

O PVM, software de paralelismo instalado na rede do CBPF, consiste de duas partes: a primeira parte, é um *daemon*[6], chamdo ***pvmd3***, que deve estar instalado em todas as máquinas com as quais se pretende compor a máquina virtual. A segunda parte consiste de uma library - ***pvm3.h*** no caso do C e ***fpvm3.h*** no caso do FORTRAN - à ser incluída pelo usuário em seu programa, a qual implementa as funções do PVM que serão chamadas para lançar outros processos, trocar mensagens entre os processos e terminá-los ao fim do programa.

O primeiro passo, no PVM é definir que máquinas comporão a máquina virtual. Assim, através do programa *pvm* adiciona-se as máquinas uma a uma, com o comando ***add hostname***, onde ***hostname*** será o nome das estações que participarão do problema (atomo, lepton, pion, etc..). Uma maneira mais simples consiste em criar-se um arquivo de configuração na conta do usuário que conterá as máquinas à serem usadas no programa paralelo.

Uma vez criada a máquina virtual, o programador lança o seu processo, o qual, por sua vez lançará outros processos à medida do necessário em outra máquinas que compõem a máquina virtual.

No PVM cada processo, ao se iniciar, recebe um Task ID (TID) que será único e terá como função identificar diversos processos do programa afim de permitir a troca de

[6]Programa que permanece rodando na memória

informações entre eles. Assim um processo enviará uma mensagem para um TID e não para outro processo em determinada máquina.

3.1 Princípios do PVM

1. Máquina Virtual configurável pelo usuário: A aplicação paralela é executada em um conjunto de máquinas definidas pelo próprio usuário

2. Acesso seletivo ao *hardware*: O programa paralelo pode "ver" a máquina virtual como uma coleção de CPU's sem características especiais, ou pode aproveitar as vantagens da arquitetura de cada CPU da máquina virtual.

3. Computação baseada em processos: A unidade de paralelismo no PVM é uma tarefa (*task*), que consiste em um código seqüencial à ser executado em uma CPU da máquina virtual.

4. Modelo de "message passing" explícito: Os tasks que formam o programa paralelo dividem o trabalho através da troca de mensagens determinada pelo usuário à ser realizada pelos mesmos.

3.2 Organização de Tarefas no PVM

A organização das tarefas no PVM se dá segundo alguns paradigmas fundamentais à saber:

- Paradigma "Crowd Computing" - Consiste numa coleção de processos intimamente relacionados, tipicamente executando o mesmo código, que trabalharão em porções diferentes do código, trocando resultados intermediários entre si.

  - Modelo Master Slave - Neste um processo é lançado pelo usuário, se encarregando de lançar outras tarefas que executarão o trabalho computacional, se encarregando de coletar os dados e apresentar os resultados.
  - Modelo Nó á Nó - Neste modelo múltiplas instâncias de um único programa são executadas, com um processo cuidando do trabalho não computacional, além de contribuir eventualmente com o cálculo também.

- Paradigma de computação em árvore - Com este paradigma, os processos são lançados (usualmente no decorrer do cálculo) em uma estrutura em árvore estabelecendo portanto uma relação pai - filho entre os processos, em oposição ao paradigma "Crowd", no qual há uma relação em forma de "estrela". Este algoritmo é muito útil quando não se sabe *a priori* a carga total de trabalho à ser executado.

3.3 Funções do PVM3

A *library* do PVM3 consta de várias rotinas que permitem o lançamento de novos tasks, comunicação entre os tasks entre outras operações inerentes à programação paralela. À seguir veremos as algumas das funções bàsicas do PVM para linguagem C.

- **pvm_addhosts()** - Adiciona um ou mais hosts na máquina virtual.

<u>**Sintaxe**</u> int info = pvm_addhosts(char **hosts, int nhost, int *infos)

Este comando permite adicionar hosts à máquina virtual durante a execução do programa

<u>**Parâmetros**</u>

- hosts - *Array* de ponteiros para *strings* de caracteres contendo os nomes das máquinas a serem adicionadas.
- nhost - Inteiro especificando o número de hosts a serem adicionados.
- infos - *Array* de inteiros que contêm o *status* retornado pelos hosts individuais. Um valor menor que zero indica um erro.

Exemplo:

```
static char *hosts[] = {"atomo","lepton","spin","muon"};
info = pvm_addhosts( hosts, 3, infos );
```

- **pvm_bufinfo()** - Retorna informação sobre o *buffer* de mensagens solicitado.

<u>**Sintaxe**</u> int info = pvm_bufinfo(int bufid, int *bytes, int *msgtag, int *tid)

<u>**Parâmetros**</u>

- bufid - Inteiro contendo o identificador do *buffer* da mensagem.
- bytes - Inteiro retornando o comprimento em *bytes* da mensagem inteira.
- msgtag - Retorna um inteiro contendo o *label* da mensagem.
- tid - Inteiro retornando o task ID do qual vem a mensagem.

Exemplo:

```
buifd = pvm_recv( -1, -1);
info = pvm_bufinfo(bufid, &bytes, &type, &source);
```

- **pvm_delhosts()** - Remove um ou mais *hosts* da máquina virtual.

<u>**Sintaxe**</u> int info = pvm_delhosts(char **hosts, int nhost, int *infos)

<u>**Parâmetros**</u>

- hosts - *Array* de ponteiros para *strings* de caracteres contendo os nomes dos hosts à serem removidos.
- nhost - Inteiro especificando o número de *hosts* à serem deletados.

- infos - *Array* de inteiro de comprimento *nhost* contendo o status retornado por cada um dos *hosts* especificados.

Exemplo:

```
static char *hosts[] = {"atomo","lepton","spin","muon"};
info = pvm_delhosts(hosts, 3, infos);
```

- **pvm_exit()** - Informa ao *daemon* local que o processo está deixando a máquina virtual.

Sintaxe int info = pvm_exit(void)

Parâmetros

- info - Código de erro. Valores menores que zero indicam erro na execução do comando.

Exemplo:

```
pvm_exit();
exit();
```

- **pvm_initsend()** - Limpa o *buffer* de mensagens e especifica o tipo de codificação.

Sintaxe int bufid = pvm_initsend(int enconding)

Parâmetros

- encoding - Inteiro especificando a maneira de codificaçãopara envio da mensagem. Este pode ter 3 valores diferentes:
  1. **PvmDataDefault**
  2. **PvmDataRaw**
  3. **PvmDataInPlace**
- bufid - Identificador do *buffer* de mensagem.

Exemplo:

```
bufid = pvm_initsend(PvmDataDefault);
info = pvm_pkint(array, 10, 1);
msgtag = 3;
info = pvm_send(tid, msgtag);
```

- **pvm_kill()** - Termina o processo especificado

Sintaxe int info = pvm_kill(int tid)

Parâmetros

- tid - Identificador do task à ser terminado.
- info - Código de erro retornado pela rotina. Valor menor que zero indica que ocorreu algum erro.

Exemplo:

```
info=pvm_kill(tid);
```

- **pvm_mytid()** - Retorna o *Task ID* (TID) do processo, incluindo-o na máquina virtual.

<u>**Sintaxe**</u> inf tid = pvm_mytid(void)

<u>**Parâmetros**</u>

- tid - Inteiro retornando o TID do processo. Um valor menor que zero indica um erro.

Exemplo:

```
tid = pvm_mytid();
```

- **pvm_pk*()** - Prepara o *buffer* de mensagens ativo com uma mensagem à ser enviada, do tipo int.

<u>**Sintaxe**</u> int info = pvm_pk*(tipo *dp, int nitem, int stride)

<u>**Parâmetros**</u>

- tipo - Tipo de variável que se deseja enviar (pode ser double, int, float, etc..).
- nitem - Número de "itens" à serem enviados

- dp - *Array* de variáveis tipo *double* com pelo menos nitem × stride itens

Exemplo:

```
info = pvm_initsend( PvmDataDefault ); info = pvm_pkstr( "initial data" );
```

- **pvm_spawn()** - Lança um novo processo.

<u>**Sintaxe**</u> int numt = pvm_spawn(char *task, char **argv, int flag, char *where, int ntask, int *tids)

<u>**Parâmetros**</u>

- task - String de caracteres contendo o nome do arquivo executável à ser lançado.

- argv - Matriz de argumentos ao processo à ser lançado.
- flag - Inteiro especificando as opções para o lançamento do processo. Esse parâmetro pode assumir um dos seguintes valores:

  PvmTaskDefault - O PVM escolhe em que máquina, do *host pool* lançar o task.

  PvmTaskHost - O parâmetro **where** especifica em que máquina lançar o task.

  PvmTaskArch O - parâmetro **where** especifica em que arquitetura lançar o task.

  PvmTaskDebug - Lança o processo debaixo de um *debugger*

- where - String contendo o nome da máquina na qual o PVM deve lançar o task.
- ntask - Inteiro especificando o número de cópias do processo à serem lançadas.
- tids - Matriz de inteiros que receberá os task IDs dos processos lançados.
- numt - Inteiro retornando o número de tasks lançados.

4 Exemplos

4.1 O *hello.c*

À seguir temos um exemplo simples de um programa em paralelo afim de ilustrar o lançamento de processos e a comunicação básica entre os tasks.

Para este problema compilamos dois executáveis. Um *master (hello.c)* e um *slave (hello_other.c)* cujos fontes vemos à seguir.

hello.c

```
#include <stdio.h>
#include "pvm3.h"

main()
{
        int i, cc, tid;
        char buf[100];

        printf("i'm t%x\n", pvm_mytid());
        cc = pvm_spawn("hello_other", (char**)0, 0, "", 1, &tid);

        if (cc == 1) {
                cc = pvm_recv(-1, -1);
                pvm_bufinfo(cc, (int*)0, (int*)0, &tid);
                pvm_upkstr(buf);
                printf("from t%x: %s\n", tid, buf);

        } else
                printf("can't start hello_other\n");
        pvm_exit();
        exit(0);
}
```

hello_other.c

```
#include "pvm3.h"

main()
{
        int ptid;
        char buf[100];

        ptid = pvm_parent();

        strcpy(buf, "hello, world from ");
        gethostname(buf + strlen(buf), 64);

        pvm_initsend(PvmDataDefault);
        pvm_pkstr(buf);
        pvm_send(ptid, 1);
```

```
        pvm_exit();
        exit(0);
}
```

Neste exemplo vimos como lançar um processo em PVM (*pvm_spawn*) e como enviar e receber mensagens (*pvm_send e pvm_recv*). À seguir veremos um exemplo de simulação numérica em PVM.

4.2 Simulações numéricas no PVM

Uma excelente aplicação do PVM aparece para a realização de simulações numéricas em sistemas magnéticos através do Algoritmo de Monte Carlo.

O problema consiste em simular em computador a evolução de um sistema de domínios magnéticos em filmes finos, no sentido do estado de menor energia possível. Para se calcular essa configuração de energia mínima, representa-se o filme magnético por uma matriz $L \times L$ onde cada elemento (que representará um *spin* magnético) pode ter valor de +1 ou -1.

Assim, o programa, partindo de uma configuração inicial, irá "virar" um *spin* qualquer, calculando a variação de energia resultante desta mudança de estado, dada por

(1)
$$\Delta E = \underbrace{-\sum_{<i,j>} \sigma_i \sigma_j}_{a} + \underbrace{\alpha \sum_{<i,j>} \frac{\sigma_i \sigma_j}{r_{ij}^3}}_{b} - h \sum_i \sigma_i$$

Com esta variação de energia em mãos, usa-se o algoritmo de Metropolis que nos dará a probabilidade P_1 do sistema ficar neste novo estado descrita à seguir:

(2)
$$P_1 \begin{cases} 1 & \text{se} \Delta E \leq 0 \\ p = e^{-\Delta E/kT} & \text{se} \Delta E \geq 0 \end{cases}$$

Este cálculo é feito L^2 vezes, de maneira que todos os spins da rede tenham a possibilidade de virar pelo menos um vez. Isto é o Passo Monte Carlo.

Dessa forma, sendo t_1 o tempo para o cálculo de um interação dipolar, N_{pmc} o número de Passos de Monte Carlo a serem realizados, temos que o tempo de execução do programa em serial será: $T_{exec} = N_{pmc} L^2 (t_1 L^2)$

A aplicação do PVM à este problema consiste em dividir-se a matriz entre N_{cpu} máquinas de maneira que cada CPU receba uma porção da matriz com L colunas e L/N_{cpu} linhas. A seguir, a CPU que dividiu a matriz (*master*) e lançou os tasks nas outras CPUs (*slave*), sorteia um *spin* aleatoriamente e comunica aos *slaves* para que estes calculem a energia da interação à longa distância, dada por "b". Enquanto isto, o master calcula a energia de interação próxima dada por "b". Para simplificação supomos $h = 0$.

Dessa forma, o tempo para a execução do programa em paralelo será dado por

$$T_{exec} = N_{pmc} L^2 \left\{ t_1 \overbrace{\frac{L^2}{N_{cpu}}}^{G} + 2 N_{cpu} t_{net} \right\}$$

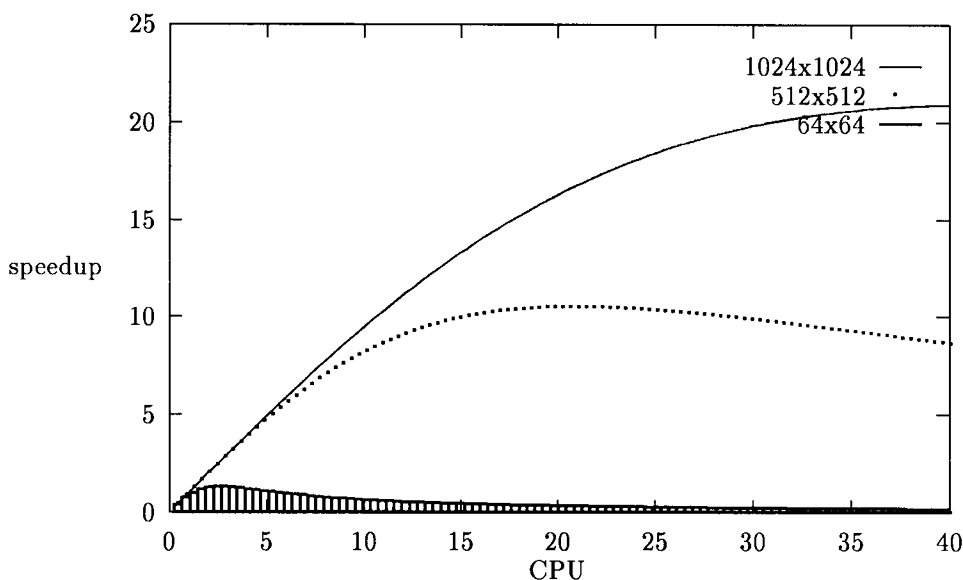

Figure 1: Gráfico de Ganho em função do número de CPU's

Assim, fixando-se o número de passos Monte Carlo, o tamanho da Matriz e conhecendo-se T_{net} e T_1, obtemos um valor ótimo para o número de CPU's à participar do problema paralelo, dado por

$$N_{optimum} = \frac{L}{\sqrt{2\frac{T_{net}}{T_1}}}$$

A seguir plotamos um gráfico do *speedup* em função do número de CPU's do problema para matrizes de tamanhos de 64, 512 e 1024. Notamos que para matrizes de 64x64 o *speedup* é menor que 1. Ou seja, em função do reduzido tempo de cálculo em comparação com o tempo de comunicação não é vantagem usar o PVM para matrizes deste tamanho. Podemos confirmar também, com erro razoável a nossa previsão do valor de 32 CPU's para o ponto ótimo em uma matriz de 1024x1024.

5 Conclusão

O tempo de comunicação obtido na rede *ethernet* do CBPF, independente da plataforma foi de $T_{net} = 2$ms. Para a simulação rodando em máquinas RISC6000 no CBPF, obtemos um valor de $T_1 = 6.76\mu$s. Dessa forma, vemos que, para matrizes de 1024 linhas apesar de haver um ganho em tempo de processamento, o ponto ótimo só seria atingido usando-se 32.19 (valor sem arrendondar o cálculo) CPU's iguais. Assim, para tornarmos este valor mais viável, necessitamos reduzir o valor de T_{net}, o que será possível brevemente com a instalação de uma rede *fast ethernet* com uma largura de banda em torno de 100Mbits.

Apesar de não ser considerado ideal para computação de alto desempenho, o PVM consiste em uma ótima alternativa para a programação paralela aonde não se dispõe, ou não se pode dispor de arquitetura própria para o paralelismo.

Assim, lançando-se mão de diversas CPU's em uma rede heterogênea, podemos aproveitar as características de cada processador afim de obter um melhor resultado sem haver necessidade de adquirir máquinas mais poderosas (e consequentemente mais caras).

Bibliografia

[1] Al Geist and Adam Beguelin, Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing, 1994, The MIT Press.

[2] Al Geist and Adam Beguelin, PVM 3 User's Guide and Reference Manual, 1993, Oak Ridge National Laboratory.