Instituto de Ciências Matemáticas de São Carlos

ISSN - 0103-2577

PVM
Parallel virtual machine
conceitos básicos e exemplo de aplicação

ROBERTA SPOLON
PAULO SÉRGIO LOPES DE SOUZA
MARCOS JOSÉ SANTANA
REGINA HELENA CARLUCCI SANTANA

Nº 36

RELATÓRIOS TÉCNICOS DO ICMSC

São Carlos
Out./1995

SYSNO 894517
DATA / /
ICMC - SBAB

PVM

Parallel Virtual Machine

Conceitos Básicos e Exemplo de Aplicação

Roberta Spolon

Departamento de Computação
Faculdade de Ciências
Universidade Estadual Paulista - UNESP
Av. Luis Edmundo Carrijo Coube, s/n
17044-360 Bauru - SP Brasil
fone: (0142) 302111 r. 124
e-mail: ros@icmsc.sc.usp.br

Paulo Sérgio Lopes de Souza

Departamento de Informática
Universidade Estadual de Ponta Grossa - UEPG
Praça Santos Andrada, s/n
Ponta Grossa - PR Brasil
fone: (042) 2239355 r. 34 / (016) 2710943
e-mail: pssouza@icmsc.sc.usp.br

Marcos José Santana

Regina Helena Carlucci Santana

Departamento de Computação e Estatística
Instituto de Ciências Matemáticas de São Carlos - Universidade de São Paulo
Av. Dr. Carlos Botelho, 1465
13560-950 São Carlos - SP Brasil
fone: (016) 2726222 r. 3859
e-mail: | mjs, rcs | @icmsc.sc.usp.br

São Carlos -SP
Outubro de 1995

Sumário

1. Introdução	**02**
2. Principais Características no PVM3	**02**
3. Obtendo e Instalando o PVM	**04**
3.1 Como Obter PVM	**04**
3.2 Como Instalar o PVM	**05**
3.3 O PVM Console	**05**
3.4 Estrutura de um Programa	**06**
3.5 Como Compilar e Executar Programas	**07**
4. Interface com o Usuário	**07**
4.1 Controle de Processos	**08**
4.2 Informação	**08**
4.3 Configuração Dinâmica	**09**
4.4 Sinalização	**09**
4.5 Troca de Mensagens	**10**
5. Exemplo	**12**
6. Considerações Finais	**17**
Apêndice A	**17**
Referências Bibliográficas	**20**

1. Introdução

PVM (Parallel Virtual Machine) é um sistema de *software* que permite que uma rede de computadores heterogêneos possa ser utilizada como um único computador paralelo. Dessa forma grandes problemas computacionais podem ser resolvidos utilizando os computadores que compõem a rede [GEI94a, GEI94b, SUN94]. O desenvolvimento do PVM iniciou-se no verão de 1989 no Oak Ridge National Laboratory (ORNL), e envolve hoje diversos institutos de pesquisas e universidades. A primeira versão do software (PVM 1.0), ficou disponível em 1991. A versão atualmente disponível é PVM 3.3.9 (setembro de 1995).

No PVM, uma coleção de computadores seriais, vetorias e paralelos funcionam como um único computador com memória distribuída, denominado máquina virtual. Os computadores que compõem essa máquina virtual são denominados *hosts*. Uma *task* é definida de maneira análoga a um processo no UNIX. Aplicações, escritas em C ou Fortran77, podem ser paralelizadas utilizando troca de mensagens. Enviando e recebendo mensagens, inúmeras *tasks* de uma aplicação podem cooperar para resolver um problema em paralelo. O sistema PVM é composto por duas partes:

- ***daemon*** (denominado *PVMD*): reside em todos os *hosts* e possibilita a formação da máquina virtual (um exemplo de programa daemon é o *sendmail* que manipula todas as mensagens em um sistema UNIX). Quando um usuário quer executar uma aplicação PVM, ele deve primeiro iniciar o mesmo para criar a máquina virtual, executando o *PVMD* (ou utilizando o console). As aplicações no PVM podem ser iniciadas em qualquer dos *hosts* que compõem a máquina virtual.

- **rotinas de interface do PVM** (denominada *libpvm3.a*): essa biblioteca contém rotinas para a troca de mensagens, criação de processos, sincronização de *tasks* e alterações na máquina virtual. Programas de aplicação devem ser linkeditados com essa biblioteca para utilizar o PVM.

2. Principais Características no PVM 3

Algumas características e inovações apresentadas pela versão 3 do PVM são discutidas a seguir [GEI94a, GEI94b].

- **Interface com o usuário atualizada:**

  Novos argumentos e características foram adicionados, tornando a interface mais flexível ao usuário. A conversão entre aplicações desenvolvidas na versão 2.4 para a versão 3 do PVM é direta.

- **Identificador de *task* inteiro:**

  Todos as *tasks* no PVM 3 são identificadas por um número inteiro, denominado *tid*, que é único na máquina virtual e fornecido pelo *pvmd* local. Muitas rotinas no PVM têm como valor de retorno o *tid*, possibilitando a identificação de outros processos no sistema.

- **Controle de processos:**

  PVM fornece rotinas que possibilitam adicionar ou retirar *hosts* da máquina virtual, iniciar e encerrar *tasks*, enviar sinais para outras *tasks*, depurar programas, além de obter infomações sobre a configuração da máquina virtual e *tasks* ativas no sistema.

- **Tolerância a falhas:**

  O PVM detecta automaticamente a falha de um *host*, retirando-o da máquina virtual. O controle e recuperação das tasks "mortas" pela falha de um *host* é de responsabilidade do usuário.

- **Grupos de processos dinâmicos:**

  Processos podem pertencer a muitos grupos, e estes podem ser alterados dinamicamente em qualquer ponto de execução. Rotinas permitem que as *tasks* possam se unir e deixar qualquer grupo, e também obter informações sobre outros grupos.

- **Sinalização:**

  Existem dois métodos para a sinalização de *tasks* (notificar a ocorrência de um evento) no PVM:

  . utilizar o *signal* do UNIX;

  . enviar uma mensagem (com um identificador) para a aplicação.

- **Comunicação:**

  No PVM qualquer *task* pode enviar mensagens para qualquer outra *task*, e não há limite para o tamanho da mensagem a ser enviada (o *buffer* para enviar a mensagem é alocado dinamicamente, e limitado apenas pela quantidade de memória disponível no *host*). Existem primitivas para a comunicação nos seguintes modos:

. *send* bloqueante assíncrono;

. *receive* bloqueante assíncrono e

. *receive* não bloqueante.

3. Obtendo e Instalando o PVM

PVM não requer privilégios especiais para ser instalado, bastando ter login válido em um *host*. A versão atualmente instalada na rede de SUN's do ICMSC é 3.3.4.

3.1 Como Obter PVM

O PVM (versão 3.3.9) pode ser obtido através da WWW, no seguinte endereço:

http://www.netlib.org/pvm3/index.html

Também está disponível a *homepage* oficial do PVM, no endereço:

http://www.eece.ksu.edu/pvm3/pvm3.html

O PVM é composto pelos seguintes elementos:

- Código fonte, manuais e exemplos;
- Documentação: guia do usuário, guia de referência rápida e uma lista com respostas para alguns problemas relativos à instalação do PVM;
- XPVM: console gráfica para o PVM;
- Informações sobre grupos de usuários do PVM, slides e artigos.

A maior parte dos arquivos vem compactada, devendo-se adotar o seguinte procedimento para descompactá-los:

- para arquivos com extensão .z ou .Z:

  gunzip nome_do_arquivo.z (ou .Z)

- para arquivos com extensão .gz:

  gzip -d nome_do_arquivo.gz

- para arquivos com extensão .uu

  uudecode nome_do_arquivo.uu

- para arquivos com extensão .tar

  tar -xvf nome_do_arquivo.tar

3.2 Como Instalar o PVM

O PVM utiliza duas variáveis de ambiente durante a inicialização e execução: PVM_ROOT e PVM_ARCH.

- PVM_ROOT indica o diretório no qual o PVM foi instalado.
- PVM_ARCH indica a arquitetura do *host*.

Para se atribuir as informações corretas à essas variáveis incluiu-se o arquivo *pvm3/lib/cshrc.stub* no arquivo *.cshrc*, após a definição de *path*. O arquivo *cshrc.stub* foi obtido com o código do PVM, conforme descrito na seção 3.1. Dentro do *cshrc.stub*, atribuiu-se para a variável PVM_ROOT a indicação do diretório PVM3. A variável PVM_ARCH é determinada automaticamente pelo sistema.

Depois desse processo, basta executar o comando *make* no diretório PVM3. O *makefile* vai automaticamente determinar em qual arquitetura está sendo executado e instalar o PVM. Para executar o PVM e criar a máquina virtual, basta rodar o *PVMD* ou o PVM (console), descrito na próxima seção.

3.3 O PVM Console

O PVM console permite ao usuário interagir e modificar a máquina virtual. Ele pode ser chamado e encerrado inúmeras vezes em qualquer *host* da máquina virtual sem afetar a execução do PVM ou qualquer outra aplicação em execução.

Quando chamado, o PVM console determina se o PVM já está em execução (através de chamadas anteriores ao PVM ou *PVMD*) ou não; em caso negativo, automaticamente o PVM é instalado.

Para executar o PVM console, basta digitar na linha de comando:

pvm <hostfile> onde:

hostfile = arquivo com o nome dos *hosts* que vão constituir a máquina virtual.

Exemplo:

pvm nomes_hosts

nomes_hosts

configuração da máquina virtual
ianomami
xavante
xapacura
caiapó
caiua

No exemplo anterior, a máquina virtual vai ser criada com 5 *hosts*, conforme especificado no arquivo nomes_hosts.

O usuário pode também adicionar os *hosts* através dos comandos do PVM console, da seguinte maneira, após ter iniciado o pvm console:

pvm> add ianomami xavante xapacura caiapó caiua

O comando abaixo permite verificar a configuração da máquina virtual:

pvm>conf

Além destes, muitos outros comandos estão disponíveis para o usuário, como comandos para retirar *hosts* da máquina virtual, verificar quais *jobs* estão em execução, enviar sinais e encerrar processos, dentre outros (ver [GEI94a], páginas 7 a 9).

Para encerrar a execução do PVM, deve-se utilizar o comando *halt* do PVM console, após a execução de todas as *tasks*.

3.4 Estrutura de um Programa

Programas utilizando o PVM podem ser desenvolvidos nas seguintes linguagens de programação:

- Fortran;
- C.

Não existe uma estrutura complexa para as aplicações desenvolvidas. Essas devem incluir a biblioteca que contém as primitivas do PVM (a seção 4 apresenta as classes de primitivas disponíveis):

- *fpvm3.h* para a linguagem Fortran;
- *pvm3.h* para a linguagem C.

Além da inclusão da biblioteca, toda aplicação deve fazer uma chamada à primitiva que encerra o processo em execução:

- *pvmfexit()* na linguagem Fortran;
- *pvm_exit();* na linguagem C.

Na seção 5 é apresentado um exemplo de aplicação desenvolvida em C. Geist [GEI94a, GEI94b] apresenta outras aplicações nas linguagens C e Fortran.

3.5 Como Compilar e Executar Programas

Para compilar os programas exemplos no PVM deve-se utilizar o seguinte comando, a partir do subdiretório *pmv3/examples* [GEI94b]:

../lib/aimk nome_programa

Os programas fontes desenvolvidos pelo usuário devem estar no diretório citado acima, ou então deve-se alterar o *path* no arquivo *Makefile.aimk*, no mesmo diretório (o código desse arquivo é apresentado no apêndice A). É necessário também incluir informações e o nome do programa a ser compilado, como mostra o trecho grifado no apêndice A. Basta substituir o nome quick.c pelo nome do programa a ser compilado.

O código executável é gerado no subdiretório *pvm3/bin/SUN4* (*default*). O usuário pode especificar outro diretório, bastando alterar o trecho em itálico no arquivo Makefile.aimk.

4. Interface com o Usuário

Nessa seção é apresentada uma breve descrição de algumas das principais rotinas oferecidas pelo PVM 3.3 ao usuário, nas versões para C e Fortran, que pode ser identificada pela presença do "call". A referência completa dessas rotinas pode ser encontrada em [GEI94a].

4.1 Controle de Processos

- *int tid = pvm_mytid (void)*

  call pvmfmytid (tid)

  Registra o processo UNIX no PVM em sua primeira chamada transformando-o em uma *task* do PVM (é gerado um identificador para a *task*).

- *int pvm_exit (void)*

  call pvmfexit (void)

  Permite que o processo deixe o PVM.

- *int numt = pvm_spawn (char *task, char **argv, int flag, char *where, int ntask, int *tids)*

  call pvmfspawn (task, flag, where, ntask, tids, numt)

  Inicia *ntask* cópias de um arquivo executável (*task*) na máquina virtual, uma em cada *host* definido. Uma série de opções pode ser especificada em *argv* e *flag*, como em qual *host* ou arquitetura criar o mesmo. Portanto esta rotina é responsável por ativar processos paralelos.

- *int info = pvm_kill (int tid)*

  call pvmfkill (tid, info)

  Essa rotina "mata" uma *task* identificada por *tid*.

Uma característica adicional no PVM 3.3 é a possibilidade de se utilizar uma interface gráfica, denominada XPVM. Essa interface combina as características do PVM Console e ferramentas de depuração permitindo a monitoração dos processos.

4.2 Informação

- *int tid = pvm_parent (void)*

  call pvmfparent (tid)

  Retorna o *tid* do processo que criou a *task*.

- *int pstat = pvm_pstat (int tid)*

  call pvmfpstat (tid, pstat)

  *int info = mstat = pvm_mstat (char *host)*

  call pvmfmstat (host, mstat)

  Essas rotinas retornam o estado de uma *task* identificada por *tid*.

- *int info = pvm_config (int *nhost, int *narch, struct pvmhostinfo **hostp)*
  call pvmfconfig (nhost, narch, dtid, name, arch, speed, info)
  Retorna informações sobre a máquina virtual.

- *int info = pvm_tasks (int wich, int *ntask, struct pvmtaskinfo **taskp)*
  call pvmftasks (which, ntask, tid, ptid, dtid, flag, aout, info)
  Retorna informações sobre as *tasks* em execução sobre a máquina virtual.

- *int dtid = pvm_tidtohost (int tid)*
  call pvmftidtohost (tid, dtid)
  Determina em qual *host* a *task* está sendo executada.

4.3 Configuração Dinâmica

- *int info = pvm_addhosts (char ** hosts, int nhost, int *infos)*
  call pvmfaddhosts (host, info)

  *int info = pvm_delhosts (char ** hosts, int nhost, int *infos)*
  call pvmdelhost (host, info)
  Permitem ao usuário adicionar ou retirar *hosts* da máquina virtual.

4.4 Sinalização

- *int info = pvm_sendsig (int tid, int signum)*
  call pvmfsendsig (tid, signum, info)
  Cria um sinal "*signum*" para outra *task* identificada por *tid*.

- *int info = pvm_notify (int what, int msgtag, int cnt, int tids)*
  call pvmfnotify (what, msgtag, cnt, tids, info)
  Informa às *taks* a ocorrência de certos eventos, como a retirada ou inclusão de um *host* à máquina virtual ou o término da execução de um processo.

4.5 Troca de Mensagens

O envio de uma mensagem no PVM é realizada em três passos (para exemplificar a utilização das primitivas, veja o código do exemplo apresentado na seção 5):

1. um *buffer* deve ser preparado através da chamada às primitivas *pvm_initsend()* ou *pvm_mkbuf()* (no exemplo utilizou-se *pvm_initsend()*);

   Linha 55 - Programa Ordena.C

   Linha 67 - Programa Quick.C

2. a mensagem deve ser compactada no *buffer* utilizando uma das rotinas disponíveis (no exemplo utilizou-se *pvm_pkint*);

   Linha 56 - Programa Ordena.C

   Linha 68 - Programa Quick.C

3. a mensagem completa é enviada para outro processo através da chamada a primitiva *pvm_send()* ou *pvm_mcast()* (no exemplo utilizou-se *pvm_send*).

   Linha 57 - Programa Ordena.C

   Linha 69 - Programa Quick.C

Uma mensagem é recebida através de rotinas bloqueantes ou não bloqueantes. Através dessas rotinas, pode-se receber qualquer mensagem, mensagens de uma determinada origem ou mensagens com um determinado identificador. Cada um dos itens recebidos deve ser decodificado a partir do *buffer* do receptor.

Buffers de Mensagens

Essas rotinas são necessárias somente se o usuário deseja gerenciar múltiplos *buffers* em sua aplicação. No PVM 3.3 há somente um *buffer* para transmissão e um *buffer* para recepção ativos por processo em um determinado instante. As rotinas para codificar, enviar, receber e decodificar afetam apenas os *buffers* ativos.

- *int bufid = pvm_mkbuf (int encoding)*

  call pvmfmkbuf (encoding, bufid)

  Essa rotina cira um *buffer* vazio para enviar dados e especifica o método de codificação a ser utilizado.

- *int bufid = pvm_initsend (encoding, bufid)*

  call pvmfinitsend (encoding, bufid)

  Prepara o *buffer* para enviar dados.

- *int bufid = pvm_getsbuf (void)*
  call pvmgetsbuf (bufid)
  Retorna o idenficador do *buffer* ativo para a transmissão.

- *int bufid = pvmgetrbuf (void)*
  call pvmfgetrbuf (bufid)
  Retorna o identicador do *buffer* ativo para a recepção.

- *int oldbuf = pvm_setsbuf (int bufid)*
  call pvmfretsbuf (bufid, oldbuf)

  int oldbuf = pvm_setrbuf (int bufid)
  call pvmfsetebuf (bufid, oldbuf)
  Determinam o *buffer* ativo para transmissão e recepção, respectivamente.

Codificar/Decodificar Dados

Essas rotinas podem ser utilizadas múltiplas vezes para compactar uma única mensagem, permitindo que a mesma seja consituída por muitos *arrays* de diferentes tipos de dados. Não há limites para a complexidade das mensagens, mas estas devem ser decodificadas exatamente da mesma maneira como foram codificadas.

Enviar e Receber Dados

- *int info = pvm_send (int tid, int msgtag)*
  call pvmfsend (tid, msgtag, info)
  Identifica a mensagem (*msgtag*) e envia para o processo *tid*.

- *int info = pvm_mcast (int *tids, int ntask, int msgtag)*
  call pvmfmcast (ntask, tids, msgtag, info)
  Identifica a mensagem, enviando-a para todas as *tasks* especificadas em *tids*.

- int *bufid = pvm_recv (int tid, int msgtag)*
  call pvmfrecv (tid, msgtag, bufid)
  Rotina para recepção bloqueante.

Muitas outras rotinas estão disponíveis, como rotinas para recepção com *timeout*, verificação de chegada de mensagens, etc.

5. Exemplo

É apresentado a seguir o código fonte (na linguagem C) de uma aplicação desenvolvida utilizando-se o PVM para ordenação de vetores em paralelo (uma versão paralela do QuickSort). Após o código dos programas, tem-se uma tabela com os tempos de execução do programa paralelo e sequencial para diversos tamanhos de vetores e número variado de *hosts* na máquina virtual. As linhas do código foram numeradas para facilitar a compreensão do mesmo.

```
/* ************************************************************************
```

Paulo Sérgio Lopes de Souza
Roberta Spolon

```
*****************************************
**          ordena.c                   **
*****************************************

************************************************************************
```

Permite a ordenação de um vetor em paralelo: processos filhos recebem e ordenam um subvetor (uma parte do vetor principal); este é retornado ao processo principal que então faz o merge com todos os subvetores ordenados.

```
*/

 1    #include <stdio.h>
 2    #include <limits.h>
 3    #include "pvm3.h"
 4    #include <time.h>
 5
 6    #define TRUE  1
 7    #define FALSE 0
 8    #define MaxVet  60000
 9    #define NUM_PROC  5
10
11    #define SORT    "quick"
12
13    main()
14    {
15            int  t, ind, ind2, info;
16            int  tid_sort[NUM_PROC], indices[NUM_PROC];
17            int  tot_subvet, pos_min;
18            int vetor[MaxVet],vet_ordenado[MaxVet];
19            unsigned long int val_min;
```

```
20
21          struct timeval tv1, tv2;  /* capturar o tempo                    */
22          int dt1;                  /* tempo gasto para uma ordenacao */
23
24          static char *hosts[]= {"xavante",
25                                 "caiapo",
26                                "caiua",
27                                "xapacura",  };
28
29          tot_subvet = MaxVet / NUM_PROC; /* tamanho de cada subvetor */
30          /* gera elementos do vetor aleatoriamente */
31          for (t = 0; t < MaxVet; t++)
32                  vetor[t] = rand();
33          /* mostra os 200 primeiros elementos do vetor
34          for (t = 0; t < (MaxVet - (MaxVet - 200 )); t++)
35                  printf("vetor gerado [%d] = %d  \n", t, vetor[t]);
36          */
37          /* determinar tempo de ordenação */
38          gettimeofday(&tv1, (struct timezone*)0);
39
40          /* cria os processos paralelos para ordenacao dos subvetores*/
41          info = TRUE;
42          for (t = 0; t < NUM_PROC; t++)
43          {
44                  if ((pvm_spawn(SORT, (char **)0, 1, hosts[t], 1, &tid_sort[t])) < 1)
45                          info = FALSE;
46          }
47
48          if (! info)
49                  printf("ERRO : pvm_spawn nao gerou todas as tasks... \n");
50          else
51          {
52                  /* envia subvetores para os processos */
53                  for(t = 0; t < NUM_PROC; t++)
54                  {
55                          pvm_initsend(PvmDataRaw);
56                          pvm_pkint(&vetor[ (NUM_PROC * t) ], tot_subvet, 1);
57                          pvm_send(tid_sort[t], 1);
58                  }
59                  /* recebe os subvetores ordenados */
60                  for(t = 0; t < NUM_PROC; t++)
61                  {
62                          pvm_recv(-1, 2);
63                          pvm_upkint(&vetor[ (NUM_PROC * t) ], tot_subvet , 1);
64
65                          indices[t] = (NUM_PROC * t);
66                  }
67
68                  /* merge - busca o menor elemento dos subvetores */
69                  for(t = 0; t < MaxVet; t++)
70                  {
71                          val_min = ULONG_MAX;
72                          pos_min = -1;
73                          for(ind = 0; ind < NUM_PROC; ind++)
74                          {
75                                  if( (indices[ind] < (tot_subvet * (ind+1) )) &&
```

```
76                                      (vetor[indices[ind]] <= val_min) )
77                                  {
78                                      val_min = vetor[indices[ind]];
79                                      pos_min = ind;
80                                  }
81                          }
82
83                          if (pos_min > -1)
84                          {
85                                  vet_ordenado[t] = val_min;
86                                  (indices[pos_min])++;
87                          }
88                          else
89                                  printf("Nao conseguiu achar o menor... \n");
90                  }
91                  gettimeofday(&tv2, (struct timezone*)0);
92
93                  /* calcula o tempo em microsegundos */
94                  dt1 = (tv2.tv_sec - tv1.tv_sec) * 1000000 + tv2.tv_usec - tv1.tv_usec;
95
96                  printf("uSeg %8d\n", dt1);
97
98                  /* mostra os 200 primeiros elementos do vetor ordenado
99                  for (t = 0; t < (MaxVet - (MaxVet - 200)); t++)
100                         printf("vet_ordenado[%d] = %d\n", t, vet_ordenado[t]);
101
102                 /* mostra os 200 ultimos elementos do vetor ordenado
103                 for (t = (MaxVet - 200); t < MaxVet; t++)
104                         printf("vet_ordenado[%d] = %d\n", t, vet_ordenado[t]);
105                 */
106
107                 /* envia sinal para matar os processos paralelos */
108                 for(t=0; t<NUM_PROC; t++)
109                         pvm_kill(tid_sort[t]);
110         }
111         pvm_exit();
112         exit();
113 }
```

```
/* ************************************************************************
```

Paulo Sérgio Lopes de Souza
Roberta Spolon

```
                ****************************************
                **          quick.c                   **
                ****************************************

************************************************************************
```

Recebe um subvetor (parte do vetor principal) , ordena utilizando o método quicksort e retorna ao processo principal, que faz o merge dos subvetores ordenados.

```
*/
```

```
1   #include <stdio.h>
2   #include "pvm3.h"
3
4   #define MaxVet   60000
5   #define NUM_PROC  5
6
7   typedef int vetor[ (MaxVet / NUM_PROC) ];
8   typedef int indice;
9
10  void particao(a, esq, dir, i, j)
11  vetor a;
12  indice esq, dir, *i, *j;
13  {
14          int x, w;
15
16          *i = esq;
17          *j = dir;
18          x = a[ ( *i + *j ) / 2];
19
20          do
21          {
22                  while (a[*i] < x)
23                          (*i)++;
24                  while (a[*j] > x)
25                          (*j)--;
26                  if (*i <= *j)
27                  {
28                          w = a[*i];
29                          a[*i] = a[*j];
30                          a[*j] = w;
31                          (*i)++;
32                          (*j)--;
33                  }
34          } while (*i <= *j);
35  }
36
37
38  void ordena(a, esq, dir)
39  vetor a;
40  indice esq;
41  indice dir;
42  {
43          indice i, j;
44
45          particao(a, esq, dir, &i, &j);
46          if(esq < j)
47                  ordena(a, esq, j);
48          if(i < dir)
49                  ordena(a, i, dir);
50  }
51
52  void main()
53  {
54          int tid_parent;
55          int tot_subvet;
56          vetor a;
```

```
57
58              tid_parent = pvm_parent();
59
60              tot_subvet = (MaxVet / NUM_PROC);
61
62              pvm_recv(tid_parent, 1);
63              pvm_upkint(a, tot_subvet, 1);
64
65              ordena(a, 0, (tot_subvet-1) );
66
67              pvm_initsend(PvmDataRaw);
68              pvm_pkint(a, tot_subvet, 1);
69              pvm_send(tid_parent, 2);
70
71              pvm_exit();
72              exit();
73      }
```

Tempos de execução: Quicksort sequencial e paralelo

VETOR	SEQ	1 HOST	2 HOSTS	3 HOSTS	4 HOSTS
60.000	1.73	3.69	4.00	3.14	2.92
120.000	3.55	7.15	6.70	5.47	5.47
240.000	7.87	14.28	13.26	10.72	9.88
480.000	16.26	29.67	24.76	22.06	20.49

Obs: Tempo médio em segundos, para 10 execuções

Para 1 *host*: ianomami e xavante

Para 2 *hosts*: ianomami, xavante e caiapó

Para 3 *hosts*: ianomami, xavante, caipó e caiua

Para 4 *hosts*: ianomami, xavante, caiapó, caiua e xapacura

A estação ianomami executou sempre o programa *ordena*; as demais estações executaram o programa *quick*, responsável pela ordenação dos sub-vetores.

Para os resultados obtidos, o que mais se aproxima do esperado são os tempos de ordenação para vetores com 480.000 elementos (o desempenho melhora a medida em se que aumenta o número de *hosts*).

Para os demais resultados salienta-se a diferença existente entre as estações utilizadas (SparcStation 2 - xavante e ianomami -, SparcStation1 - xapacura - e estações SLC - caiapó e caiua), que compromete a obtenção de resultados precisos. A alteração

na seqüência das estações (utilizando uma estação SLC nos testes com 1 *host*) por exemplo, altera também os tempos obtidos.

6. Considerações Finais

Com o desenvolvimento deste trabalho alguns pontos devem ser ressaltados:

- a documentação que acompanha o *software* é de boa qualidade e suficiente para a instalação e utilização do mesmo; o PVM apresenta um manual com todas as primitivas implementadas, parâmetros e explicações sobre as mesmas, que ajudam no desenvolvimento dos programas;
- os exemplos que acompanham o PVM são simples e de fácil compreensão (os exemplos apresentados no manual são diferentes dos exemplos que acompanham o PVM);
- a compilação dos programas desenvolvidos pelo usuário exige apenas uma pequena alteração no arquivo *Makefile.aimk*.
- não ocorreram problemas na instalação do PVM e na compilação dos programas exemplos;
- o usuário desenvolve sua aplicação como uma coleção de *tasks* que cooperam entre si para realizar determinada atividade. Essas *tasks* têm acesso às rotinas do PVM através de uma biblioteca que deve ser incluída na aplicação do usuário. O PVM manipula de modo transparente ao usuário todo o roteamento de mensagens, conversão de dados e escalonamento de *tasks* em uma rede de computadores.

Dessa forma, e levando-se em consideração os pontos ressaltados acima, conclui-se que o PVM é adequado para o desenvolvimento de aplicações paralelas de uma maneira eficiente e direta utilizando o *hardware* disponível.

Apêndice A - Makefile.aimk

```
#
# Makefile.aimk for PVM example programs.
#
# Set PVM_ROOT to the path where PVM includes and libraries are installed.
# Set PVM_ARCH to your architecture type (SUN4, HP9K, RS6K, SGI, etc.)
# Set ARCHLIB to any special libs needed on PVM_ARCH (-lrpc, -lsocket, etc.)
# otherwise leave ARCHLIB blank
#
# PVM_ARCH and ARCHLIB are set for you if you use "$PVM_ROOT/lib/aimk"
# instead of "make".
```

```
#
# aimk also creates a $PVM_ARCH directory below this one and will cd to it
# before invoking make - this allows building in parallel on different arches.
#

SDIR    =       ..
BDIR    =       $(HOME)/pvm3/bin
XDIR    =       $(BDIR)/$(PVM_ARCH)

CC      =       cc
CFLAGS  =       -g -I$(PVM_ROOT)/include
LIBS    =       -lpvm3 $(ARCHLIB)
GLIBS   =       -lgpvm3

F77     =       f77
FFLAGS  =       -g
FLIBS   =       -lfpvm3

LFLAGS  =       -L$(PVM_ROOT)/lib/$(PVM_ARCH)

default: hello hello_other

all: c-all f-all

c-all:  gexample hello hello_other master slave spmd timing timing_slave

f-all:  fgexample fmaster fslave fspmd hitc hitc_slave testall

clean:
        rm -f *.o gexample hello hello_other hitc hitc_slave master1 slave1 \
                spmd testall timing timing_slave

$(XDIR):
        - mkdir $(BDIR)
        - mkdir $(XDIR)

quick:  $(SDIR)/quick.c $(XDIR)
        $(CC) $(CFLAGS) -o quick $(SDIR)/quick.c $(LFLAGS) $(LIBS)
        mv quick $(XDIR)

ordena: $(SDIR)/ordena.c $(XDIR)
        $(CC) $(CFLAGS) -o ordena $(SDIR)/ordena.c $(LFLAGS) $(LIBS)
        mv ordena $(XDIR)

ordendin: $(SDIR)/ordendin.c $(XDIR)
        $(CC) $(CFLAGS) -o ordendin $(SDIR)/ordendin.c $(LFLAGS) $(LIBS)
        mv ordendin $(XDIR)

hello: $(SDIR)/hello.c $(XDIR)
        $(CC) $(CFLAGS) -o hello $(SDIR)/hello.c $(LFLAGS) $(LIBS)
        mv hello $(XDIR)

hello_other: $(SDIR)/hello_other.c $(XDIR)
        $(CC) $(CFLAGS) -o hello_other $(SDIR)/hello_other.c $(LFLAGS) $(LIBS)
        mv hello_other $(XDIR)
```

```
timing_slave: $(SDIR)/timing_slave.c $(XDIR)
	$(CC) $(CFLAGS) -o timing_slave $(SDIR)/timing_slave.c $(LFLAGS) $(LIBS)
	mv timing_slave $(XDIR)

timing: $(SDIR)/timing.c $(XDIR)
	$(CC) $(CFLAGS) -o timing $(SDIR)/timing.c $(LFLAGS) $(LIBS)
	mv timing $(XDIR)

spmd: $(SDIR)/spmd.c $(XDIR)
	$(CC) $(CFLAGS) -o spmd $(SDIR)/spmd.c $(LFLAGS) $(GLIBS) $(LIBS)
	mv spmd $(XDIR)

fspmd: $(SDIR)/spmd.f $(XDIR)
	$(F77) $(FFLAGS) -o spmd $(SDIR)/spmd.f $(LFLAGS) $(FLIBS) $(GLIBS)
$(LIBS)
	mv spmd $(XDIR)

master: $(SDIR)/master1.c $(XDIR)
	$(CC) $(CFLAGS) -o master1 $(SDIR)/master1.c $(LFLAGS) $(LIBS)
	mv master1 $(XDIR)

slave: $(SDIR)/slave1.c $(XDIR)
	$(CC) $(CFLAGS) -o slave1 $(SDIR)/slave1.c $(LFLAGS) $(LIBS)
	mv slave1 $(XDIR)

fmaster: $(SDIR)/master1.f $(XDIR)
	$(F77) $(FFLAGS) -o master1 $(SDIR)/master1.f $(LFLAGS) $(FLIBS) $(LIBS)
	mv master1 $(XDIR)

fslave: $(SDIR)/slave1.f $(XDIR)
	$(F77) $(FFLAGS) -o slave1 $(SDIR)/slave1.f $(LFLAGS) $(FLIBS) $(LIBS)
	mv slave1 $(XDIR)

testall: $(SDIR)/testall.f $(XDIR)
	$(F77) $(FFLAGS) -o testall $(SDIR)/testall.f $(LFLAGS) $(FLIBS) $(LIBS)
	mv testall $(XDIR)

hitc: $(SDIR)/hitc.f $(XDIR)
	$(F77) $(FFLAGS) -o hitc $(SDIR)/hitc.f $(LFLAGS) $(FLIBS) $(LIBS)
	mv hitc $(XDIR)

hitc_slave: $(SDIR)/hitc_slave.f $(XDIR)
	$(F77) $(FFLAGS) -o hitc_slave $(SDIR)/hitc_slave.f $(LFLAGS) $(FLIBS)
$(LIBS)
	mv hitc_slave $(XDIR)

gexample: $(SDIR)/gexample.c $(XDIR)
	$(CC) $(CFLAGS) -o gexample $(SDIR)/gexample.c $(LFLAGS) $(GLIBS)
$(LIBS)
	mv gexample $(XDIR)

fgexample: $(SDIR)/gexample.f $(XDIR)
	$(F77) $(FFLAGS) -o gexample $(SDIR)/gexample.f $(LFLAGS) $(FLIBS)
$(GLIBS) $(LIBS)
	mv gexample $(XDIR)
```

Referências Bibliográficas

GEI94a GEIST, A.; BEGUELIN, A.; DONGARRA, J.; JIANG, W.; MANCHEK, R.; SUNDERAN, V.. PVM 3 User's Guide and Reference Manual Oak Ridge National Laboratory, 1994.

GEI94b GEIST, A.; BEGUELIN, A.; DONGARRA, J.; JIANG, W.; MANCHEK, R.; SUNDERAN, V.. PVM: Parallel Virtual Machine - A User's Guide and Tutorial for Networked Parallel Computing. The MIT Press, 1994.

SUN94 SUNDERAN, V. S.; GEIST, G. A.; DONGARRA, J.; MANCHEK, R.. The PVM concurrent computing system: Evolution, experiences and trends. Parallel Coputing, v. 20, n. 04, 1994.