Curso de Informática – DCC-IM / UFRJ

Programação Paralela e Distribuída - PVM

Um curso prático

Adriano O. Cruz Gabriel P. Silva

Bibliografia

- PVM: Parallel Virtual Machine – A User´s Guide and Tutorial for Network Parallel Computing
  - Geist, A.; Beguelin, A.;Dongarra, J.; Jiang, W.; Mancheck, B.; Sunderam, V. The MIT Press, 1994
- http://www.netlib.org/pvm3/index.html

Introdução

- Ambiente de programação paralela segundo o modelo de troca de mensagens disponibilizado em domínio público.
- Inicialmente desenvolvido em 1989 no *Oak Ridge National Laboratory*, Estados Unidos.
- Permite que uma coleção heterogênea de computadores ligados em rede se apresente para o usuário como uma máquina paralela virtual única.
- Uma aplicação pode ser desenvolvida em C, C++, Fortran ou Java usando a biblioteca de funções do PVM.
- É uma ferramenta educacional que permite o ensino de programação paralela mesmo sem acesso a um computador paralelo.

Objetivos

- Aplicações:
  - Paradigmas simples e de uso corrente.
  - Facilidades para depuração.
  - Ferramentas gráficas de desenvolvimento.
- Sistemas:
  - Suporte para diversas arquiteturas.
  - Facilidade de instalação, operação e administração.
  - Algoritmos e protocolos eficientes e resistentes a falhas.

Características do PVM

- Pode ser instalado por qualquer usuário.
- Pode ser partilhado entre usuários.
- Fácil de configurar através do seu próprio arquivo host.
- Configurações de usuários podem se sobrepor sem conflitos.
- Fácil de escrever programas através de uma interface de troca de mensagens padrão.
- Suporta C, FORTRAN e Java.
- Múltiplas aplicações podem rodar em um único PVM.
- Pacote pequeno, requer apenas alguns Mbytes.

Características do PVM

- A biblioteca de funções do PVM oferece facilidades para:
  - Criar e terminar tarefas (processos Unix) a serem executadas em paralelo;
  - Configurar a máquina virtual, adicionando ou eliminando hosts (elementos de processamento);
  - Envio e recepção de mensagens entre tarefas em diversos formatos.
- A comunicação através de mensagens utiliza buffers e operações de codificação e decodificação (packing/unpacking) ➔ torna possível o uso de sistemas heterogêneos.

Heterogeneidade

- PVM suporta heterogeneidade em três níveis :
  - **Aplicação:** Subtarefas podem explorar a arquitetura que melhor se adapte ao seu problema.
  - **Máquina:** Computadores com diferentes formatos de dados, arquiteturas e sistemas operacionais.
  - **Redes:** Diferentes tipos de redes (FDDI, Ethernet, Token Ring).

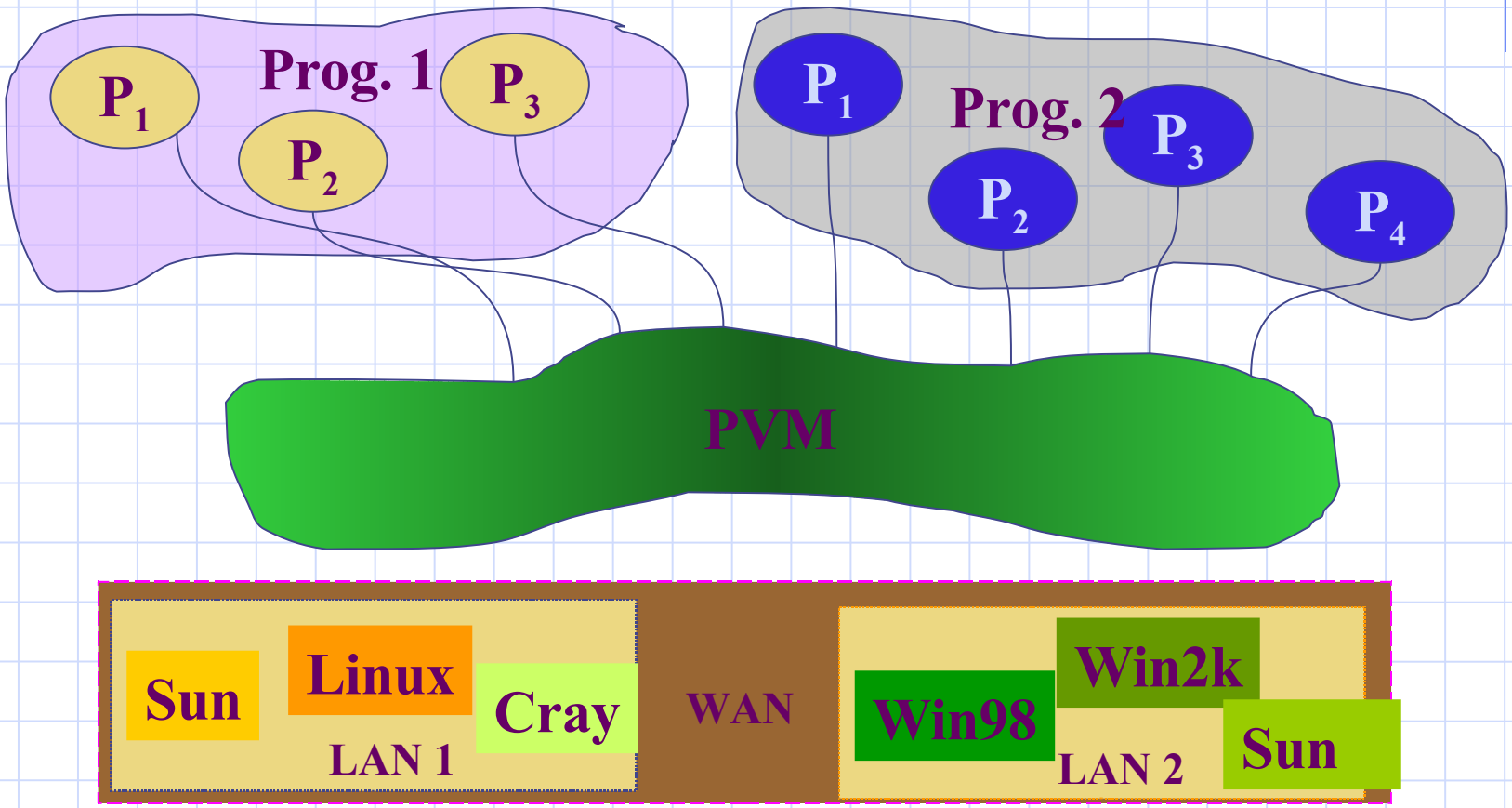
Onde o PVM roda?

PVM - ARCH	Máquina
AFX8	Alliant FX/8
ALPHA	DEC Alpha
BAL	Sequent Balance
BFLY	BBN Butterfly
BSD386	80386/486 PC
CM2	Thinking Machines
CNVX	Convex C-series
CRAY	C-90, YMP, T3D
HP300	HP-9000 model 300
HPPA	HP-9000 PA-RISC
I860	Intel iPSC/860
IPSC2	Intel iPSC/2 386
KSR1	Kendall Square KSR-1

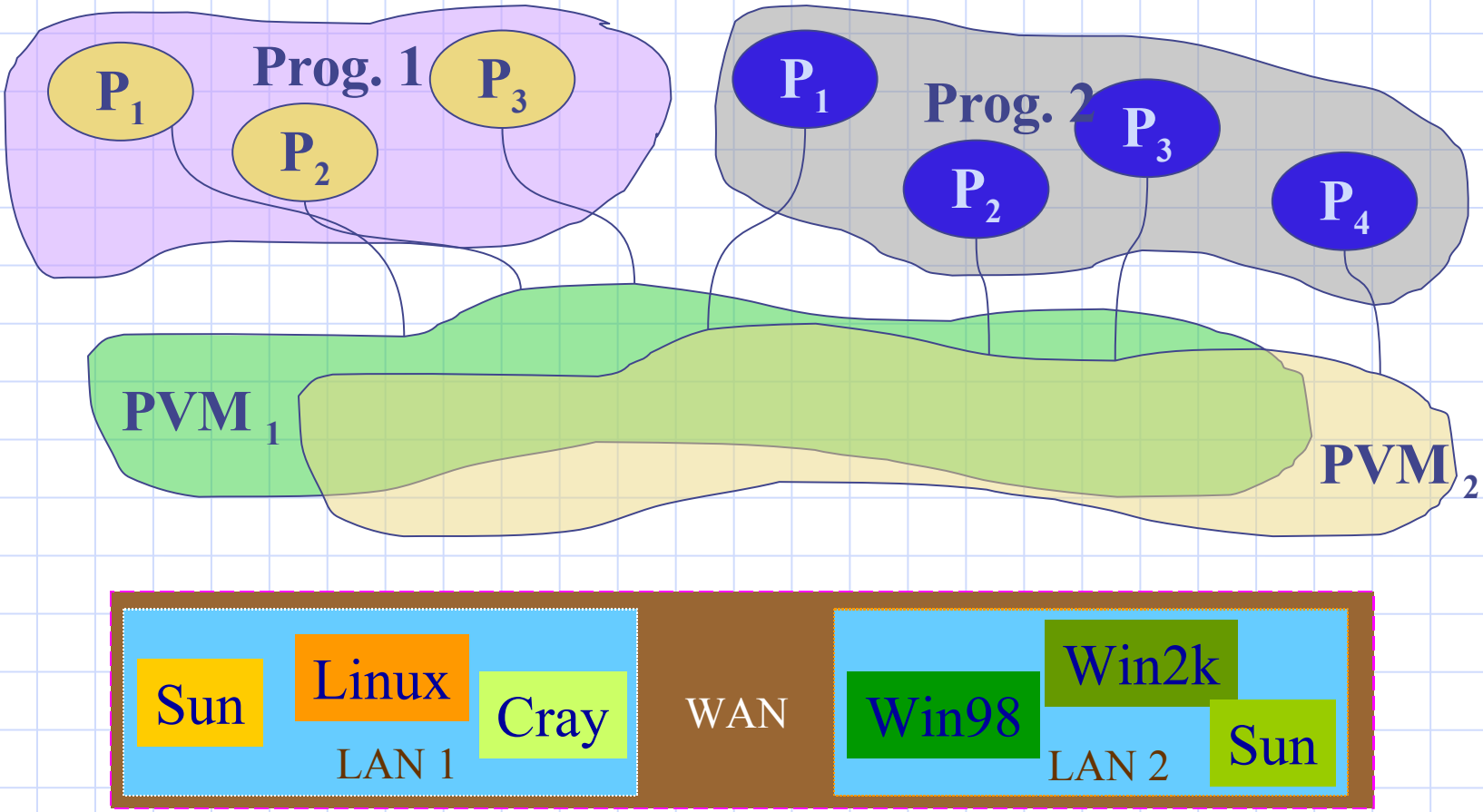
Onde o PVM roda?

PVM - ARCH	Máquina
LINUX	Intel x86 PC
NEXT	NeXT
PMAX	DECstation 3100, 5100
RS6K	IBM/RS6000
SGI	Silicon Graphics IRIS
SGI5	Silicon Graphics IRIS
SGIMP	SGI multiprocessor
SUN4	Sun 4, SPARCstation
SUN4SOL2	Sun 4, SPARCstation
SYMM	Sequent Symmetry
U370	IBM 370
UVAX	DEC MicroVAX
WINDOWS	Intel x86/PC

Funcionamento PVM



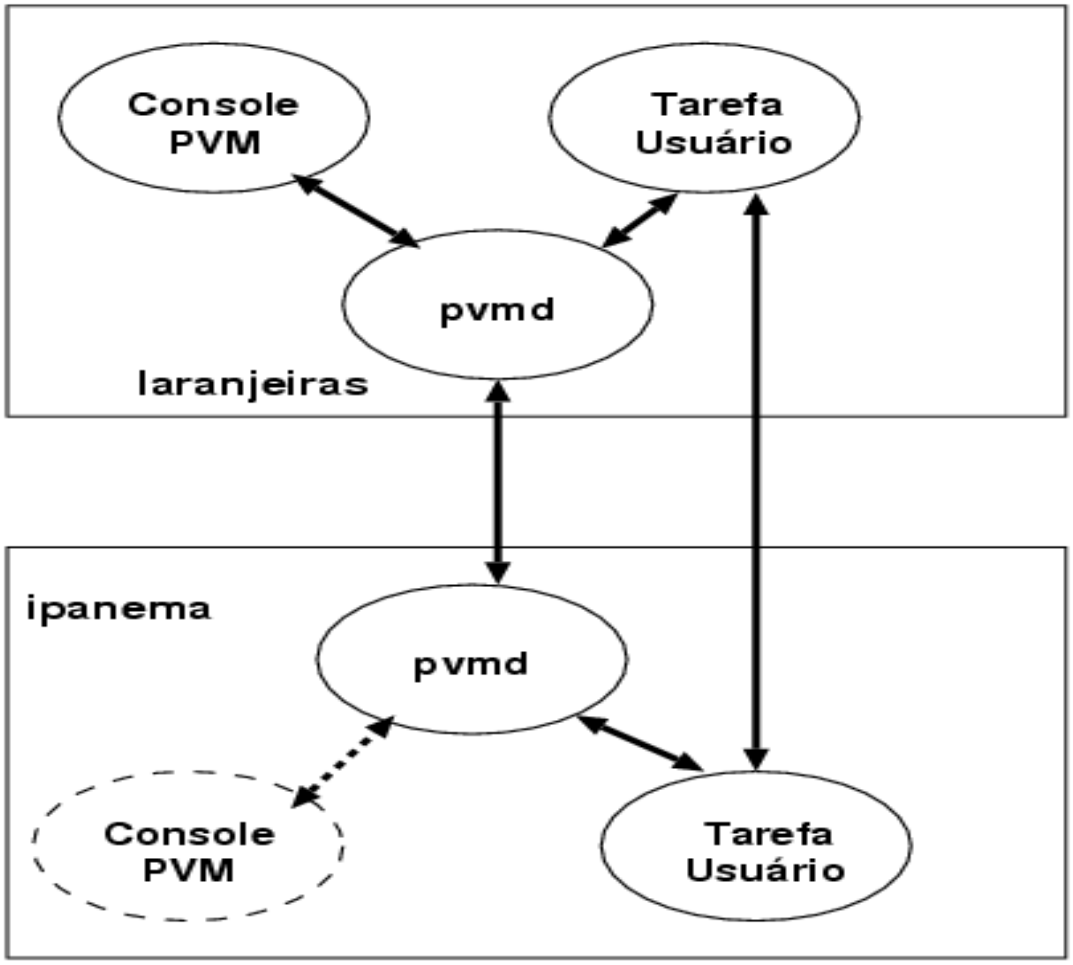
Funcionamento PVM



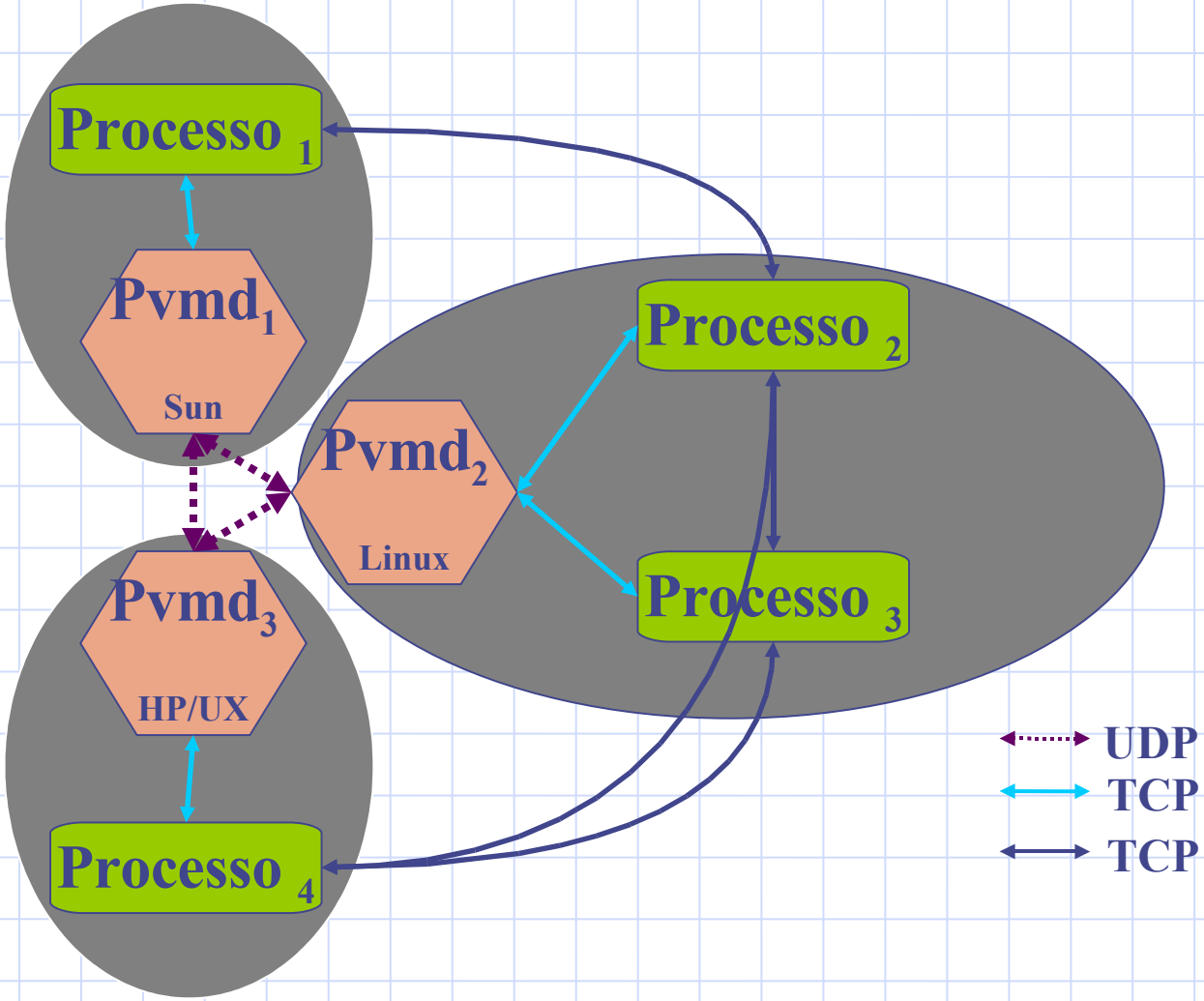
O Sistema PVM

- O sistema PVM é composto de:
  - pvmd - um daemon;
  - libpvm - a biblioteca de funções do pvm;
  - console - interface entre o usuário e o sistema;
  - aplicação - programa do usuário.

PVM em operação



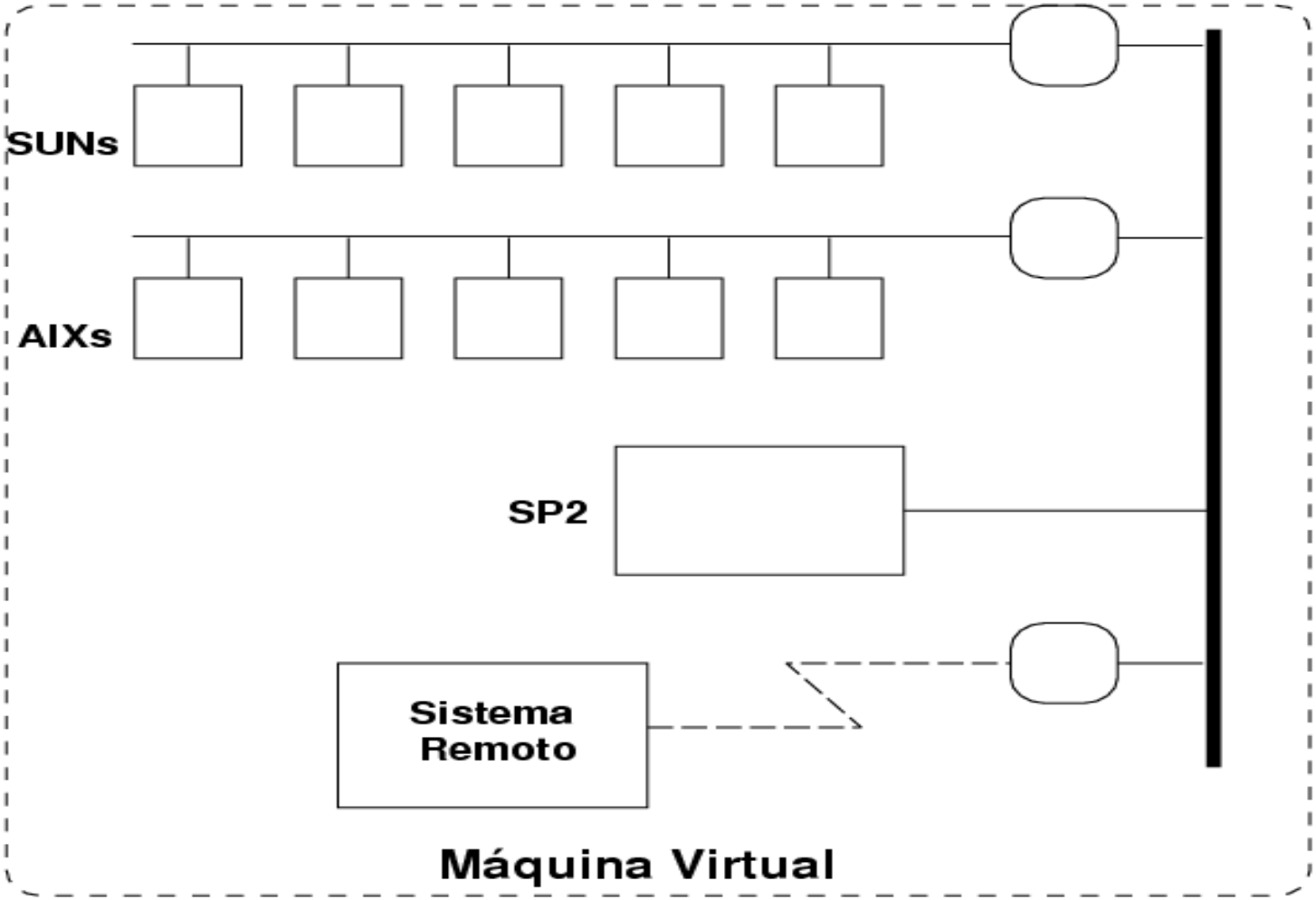
PVM em Operação



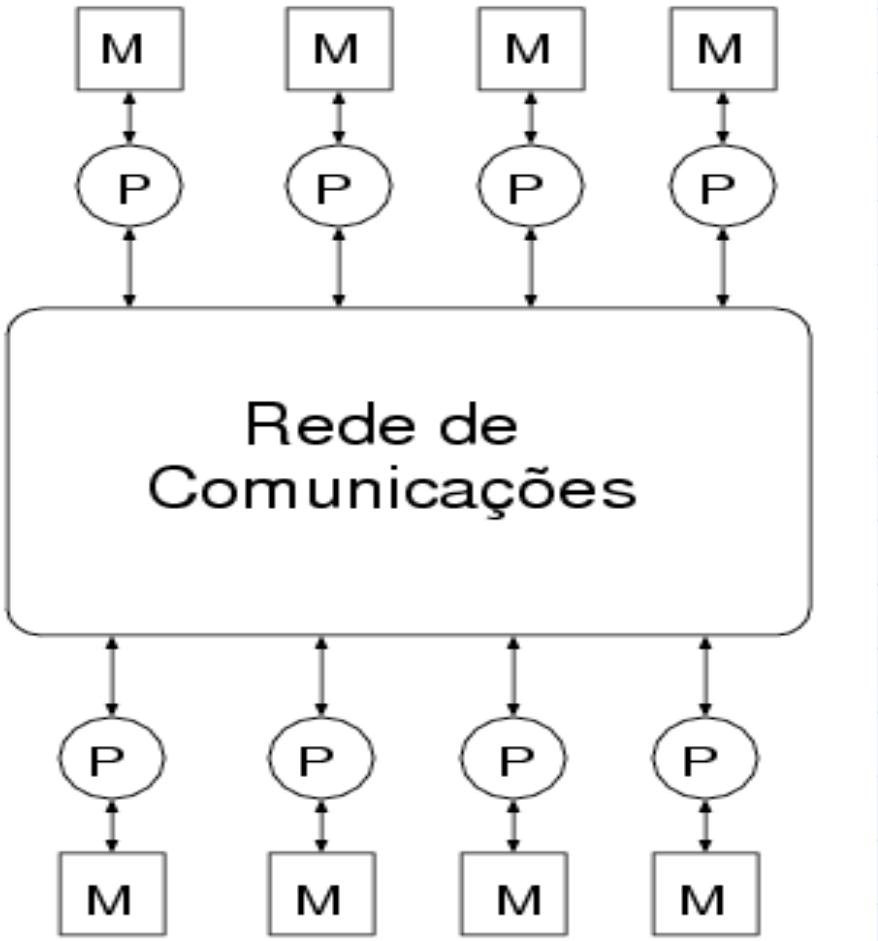
pvmd

- Pvmds são processos que executam em "background", no modo usuário, e são responsáveis pela construção da máquina virtual e pela realização da comunicação entre as tarefas nos diversos hospedeiros.
- Em cada hospedeiro da máquina virtual há um pvmd sendo executado, que é responsável por autenticar as tarefas e executar processos no hospedeiro.
- O pvmd também provê detecção de falhas e faz o roteamento das mensagens, e é mais robusto que as aplicações.

Arquitetuta do PVM



Modelo de Computação



Nomenclatura PVM

- **Hospedeiro:** O computador, também chamado de nó.
- **Máquina Virtual:** Uma meta-máquina composta de um ou mais hosts.
- **Processo:** Um programa em execução com dados, pilha, etc.
- **Tarefa:** Um processo no ambiente pvm.
- **pvmd:** O daemon do ambiente pvm.
- **Mensagem:** Uma lista ordenada de dados enviada de uma tarefa para outra.
- **Grupo:** Uma lista ordenada de tarefas que recebe um nome simbólico.

Conceitos de Programação

- Existem algumas estruturas que são comuns a todo programa PVM escrito em C.
- Todo programa PVM deve incluir o arquivo com os "headers" da biblioteca PVM.
- Isto pode ser feito colocando-se a seguinte diretiva no início do programa:

```
#include "pvm3.h"
```

- A primeira função PVM chamada por um programa deve ser:

```
info = pvm_mytid()
```

Conceitos de Programação

- Esta função serve para incorporar o processo no ambiente PVM e retorna um inteiro ou um valor negativo se ocorrer um erro.
- Ao final de um programa PVM deve ser chamada a seguinte rotina:

  pvm_exit()

- **Para se escrever um programa paralelo, as tarefas devem ser executadas em processadores distintos. Isso pode ser feito com o uso da rotina:**

  pvm_spawn()

Conceitos de Programação

- **Um exemplo típico de uma chamada para essa rotina é mostrado a seguir:**

  numt = pvm_spawn("my_task", NULL, PvmTaskDefault, "", n_task, tids)

- **Essa chamada dispara n_task cópias do programa "my_task" nos computadores que o PVM escolher.**
- **O número real de tarefas disparadas é retornado pela rotina para *numt*.**
- **O identificador de cada tarefa disparada é retornado no vetor de inteiros "tids".**
- Cada processo PVM tem um identificador único chamado "task id".

Troca de Mensagens

- **Para efetivamente programar em paralelo é necessário que as diversas tarefas possam se comunicar entre si.**
- **No PVM isso é feito através de troca de mensagens.**
- **Quando for necessário enviar uma mensagem da tarefa A para a tarefa B, a tarefa A deve antes chamar a rotina:**

```
pvm_initsend( )
```

- Esta chamada limpa o conteúdo do "buffer" e especifica a codificação da mensagem

Troca de Mensagens

- Um uso típico para essa rotina é:

  bufid=pvm_initsend(PvmDataDefault)

- Depois da iniciação, a tarefa que vai enviar a mensagem deve colocar os dados a serem enviados no "buffer" de envio de mensagens. Isso é feito com a família de rotinas:

  pvm_pack()

- Que é uma função do tipo "printf" para empacotar diversos tipos de dados.

Troca de Mensagens

- **Depois que os dados tiverem sido empacotados no buffer de envio, a mensagem está pronta para ser enviada.**
- **Isto pode ser feito com uma chamada para a rotina**

```
info=pvm_send(tid, msgtag)
```

- **Que enviará os dados para o processo com o "task id" igual a tid, com um "tag" para identificar a mensagem com um valor inteiro, no caso msgtag.**

Troca de Mensagens

- A tarefa que estiver recebendo a mensagem faz uma chamada para

  pvm_recv()

- Por exemplo:

  bufid=pvm_recv(tid, msgtag)

- Irá esperar por uma mensagem da tarefa "tid", com um identificador "msgtag".
- O valor de -1 pode ser especificado tanto para "tid" como "msgtag", significando qualquer um.

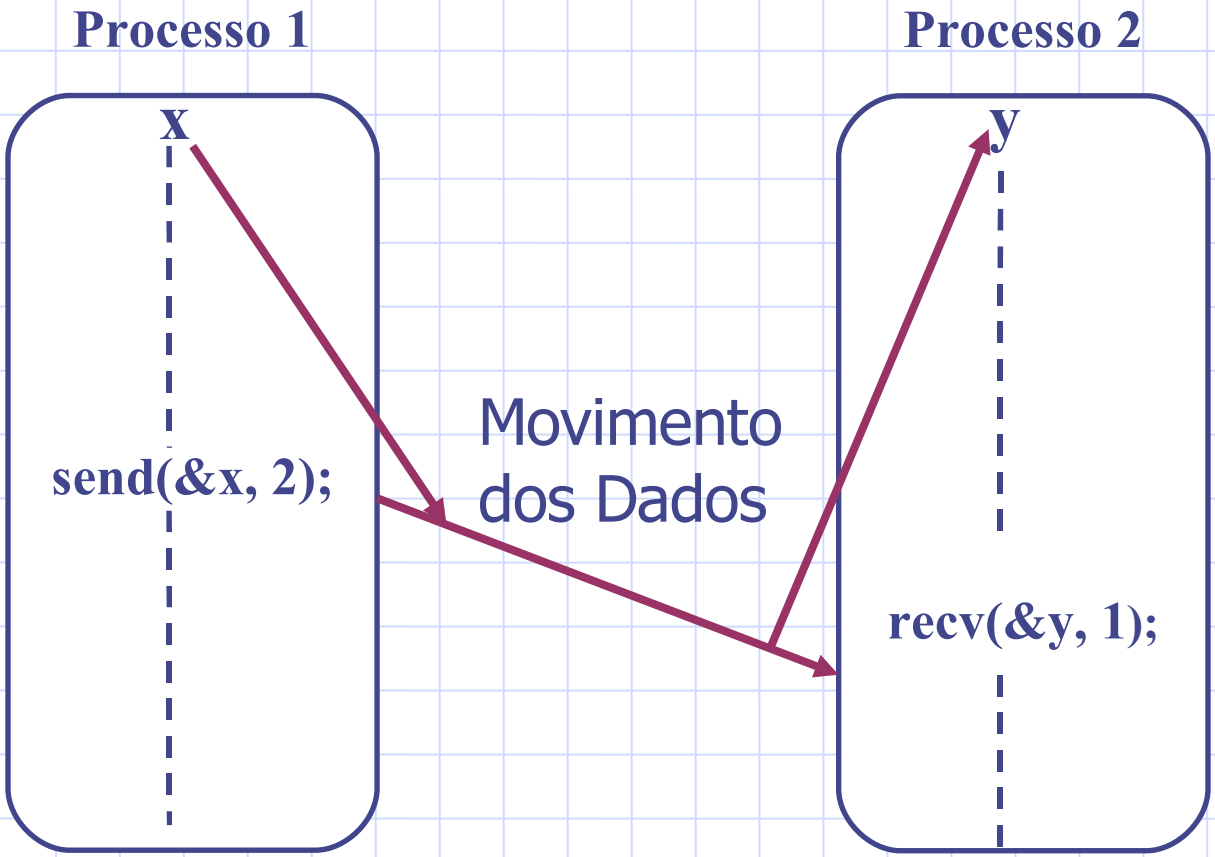
Troca de Mensagens

- A rotina

  pvm_unpack()

- Faz a função inversa de pvm_pack() e é utilizada para retirar os dados do buffer de recepção.
- Todos os dados devem ser desempacotados exatamente na mesma ordem em que foram empacotados.
- Note que as estruturas em C devem ser empacotadas elemento por elemento.

Troca de Mensagens - Modelo



Comunicação Síncrona

- Rotinas somente retornam quando a transferência da mensagem foi completada.
- Não necessita de buffer para armazenar a mensagem:
  - uma rotina síncrona de envio deve esperar até que a mensagem completa possa ser aceita pelo processo receptor antes de enviar a mensagem.
- Uma rotina síncrona de recebimento espera até que a mensagem que ela está esperando chegue.
- Rotinas síncronas realizam duas ações: transferem dados e sincronizam processos.
- Sugere a existência de alguma forma de protocolo de sinalização.

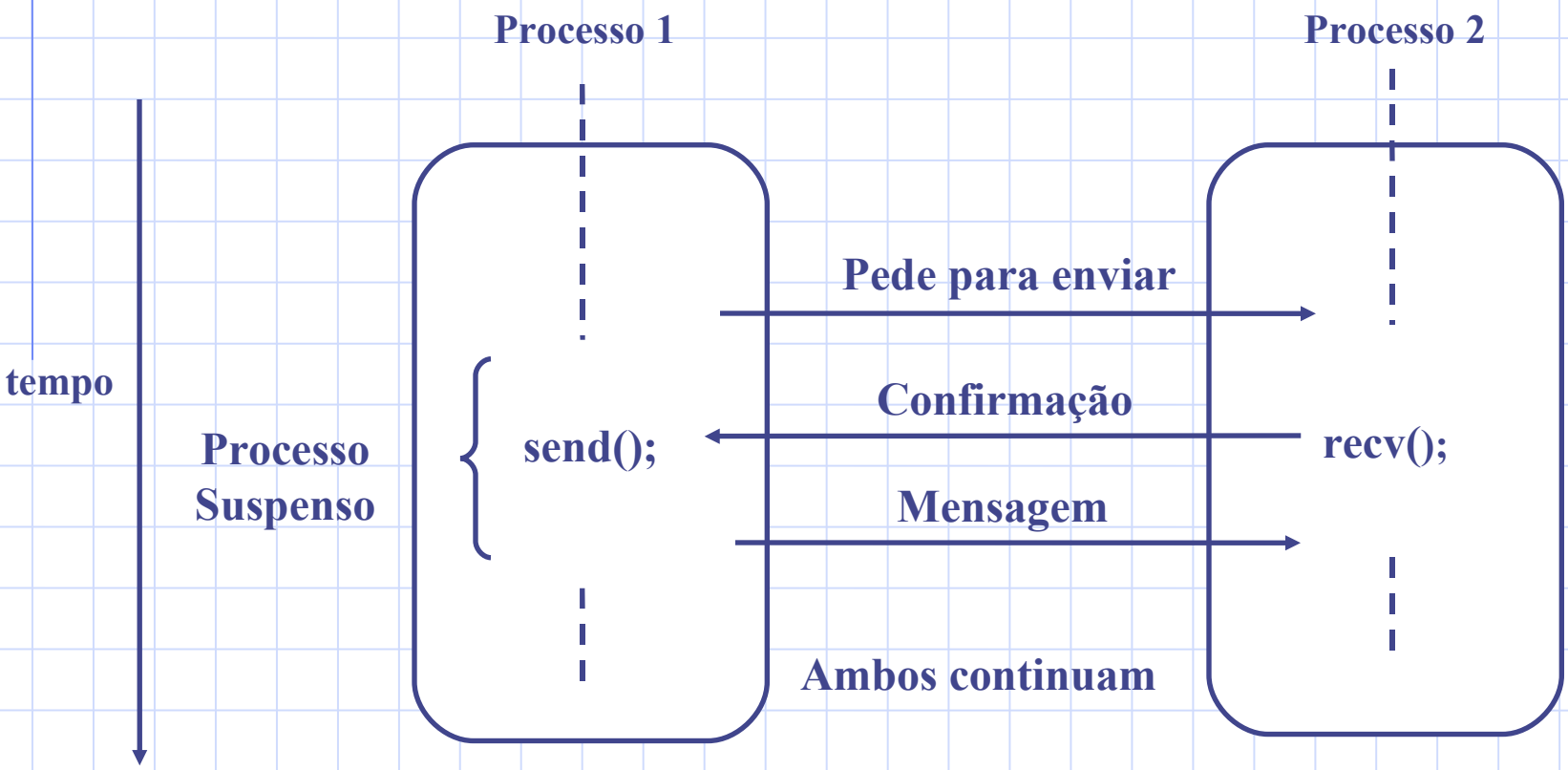
Comunicação Síncrona

- O processo que deseja enviar uma mensagem faz uma requisição inicial de autorização para enviar a mensagem ao processo destinatário.
- Depois de receber essa autorização, a operação de envio (*"send"*) é realizada.
- A operação de envio só se completa quando a operação de recepção (*"receive"*) correspondente retorna uma mensagem de *"acknowlegement"*.
- Portanto, ao se concluir a operação de envio, os *buffers* e estruturas de dados utilizados na comunicação podem ser reutilizados.

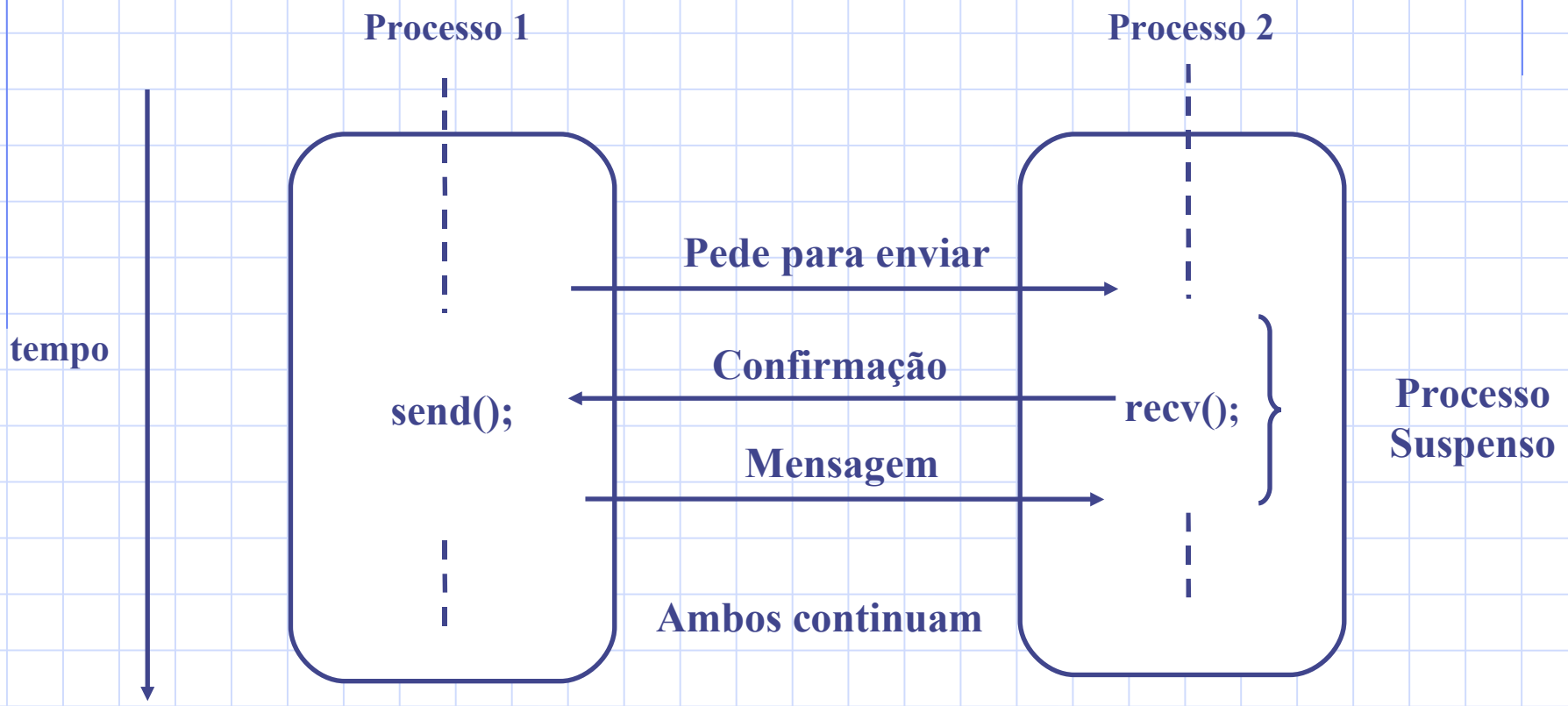
Comunicação Síncrona

- Este modo de comunicação é simples e seguro, contudo elimina a possibilidade de haver superposição entre o processamento da aplicação e o processamento da transmissão das mensagens.
- Requer um protocolo de quatro fases para a sua implementação do lado do transmissor: pedido de autorização para transmitir; recebimento da autorização; transmissão propriamente dita da mensagem; e recebimento da mensagem de *"acknowledgement"*.

Comunicação Síncrona



Comunicação Síncrona



Comunicação Assíncrona Bloqueante

- A operação de envio (*"send"*) só retorna o controle para o processo que a chamou após ter sido feita a cópia da mensagem a ser enviada de um *buffer* da aplicação para um *buffer* do sistema operacional.
- Portanto, ao haver o retorno da operação de envio, a aplicação está livre para reutilizar o seu *buffer*, embora não haja nenhuma garantia de que a transmissão da mensagem tenha completado ou vá completar satisfatoriamente.

Comunicação Assíncrona Bloqueante

- A operação de envio bloqueante difere da operação de envio síncrona, uma vez que não é implementado o protocolo de quatro fases entre os processos origem e destino.
- A operação de recepção (*"receive"*) bloqueante é semelhante à operação de recepção síncrona, só retornando para o processo que a chamou após ter concluído a transferência da mensagem do *buffer* do sistema para o *buffer* especificado pela aplicação. A diferença em relação a operação de recepção síncrona é que a mensagem de *"acknowledgement"* não é enviada.

Comunicação Assíncrona Não Bloqueante

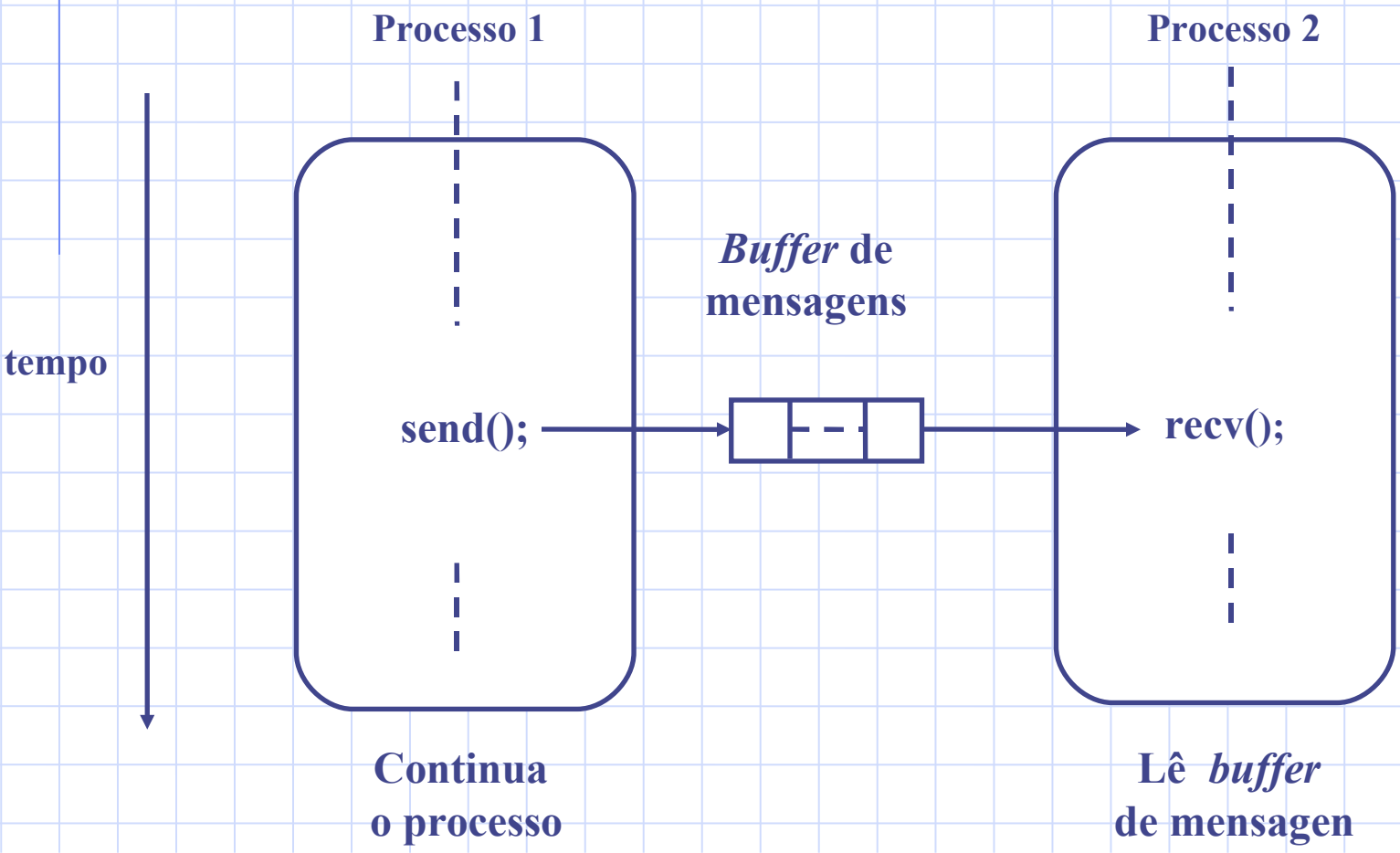
- Na primeira fase, a operação de *"send"* retorna imediatamente, tendo apenas dado início ao processo de transmissão da mensagem e a operação de *"receive"* retorna após notificar a intenção do processo de receber uma mensagem.
- As operações de envio e recepção propriamente ditas são realizadas de forma assíncrona pelo sistema.
- O retorno das operações de *"send"* e de *"receive"* nada garantem e não autoriza a reutilização das estruturas de dados para nova mensagem.

Comunicação Assíncrona Não Bloqueante

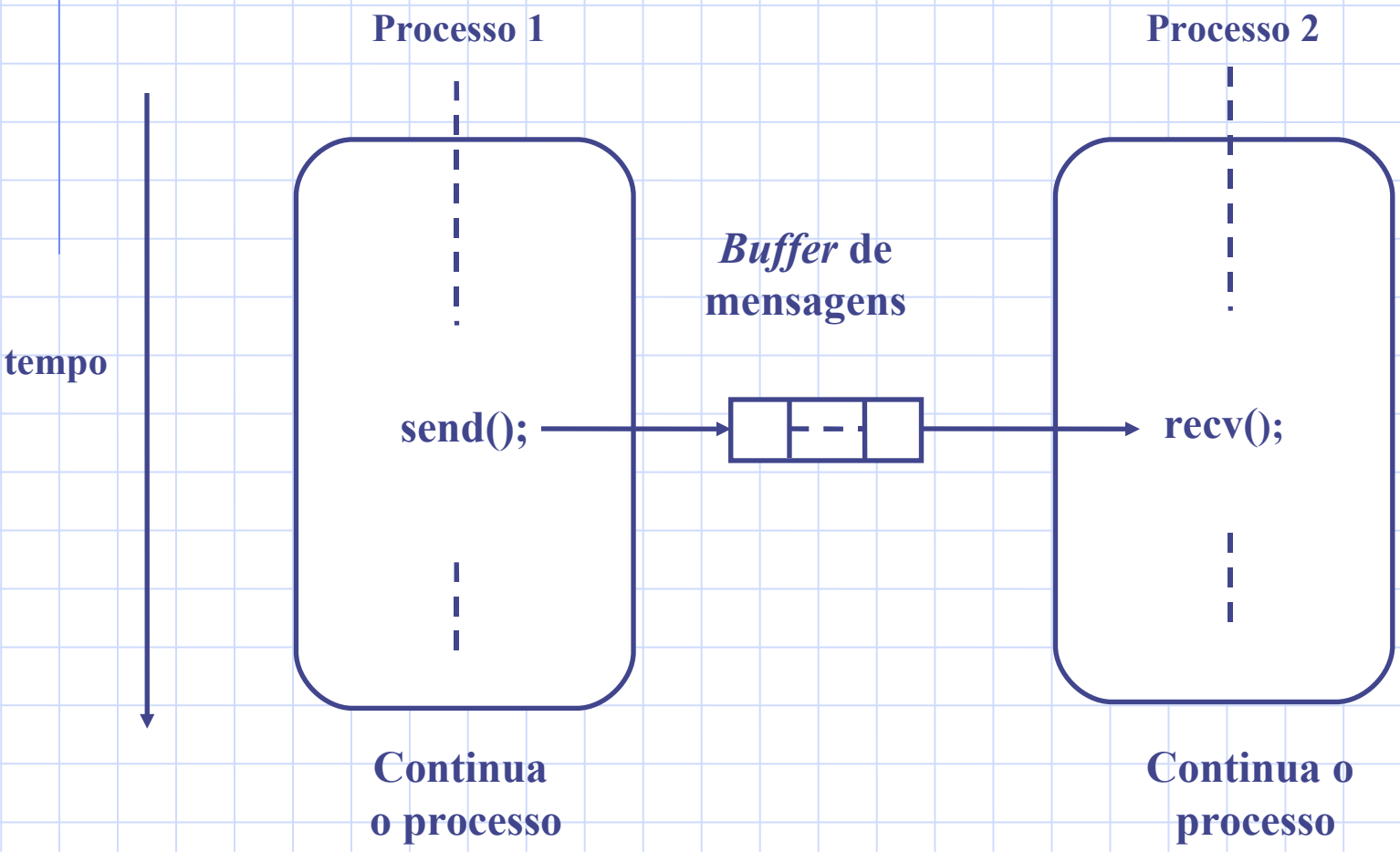
- Na segunda fase, é verificado se a operação de troca de mensagens iniciada anteriormente pelas primitivas *"send"* e *"receive"* já foi concluída.
- Somente após esta verificação é que o processo que realizou o envio de dados pode alterar com segurança sua área de dados original.
- Este modo de comunicação é o que permite maior superposição no tempo entre computações da aplicação e o processamento da transmissão das mensagens.

Comunicação Assíncrona

- Necessita de um *buffer* para guardar a mensagem



Comunicação Assíncrona



Programa Mestre - ola

```
main() {
   int cc, tid, msgtag;
   char buf[100];
   printf("Meu tid é  t%x\n", pvm_mytid());
   cc = pvm_spawn("outro_ola", (char**)0, 0, "", 1, &tid);
   if (cc == 1) {
      msgtag = 1;
      pvm_recv(tid, msgtag);
      pvm_upkstr(buf);
      printf("de t%x: %s\n", tid, buf); }
    else
      printf("Não consigo disparar o programa outro_ola\n");

   pvm_exit();  }
```

Programa Escravo – outro_ola

```
#include “pvm3.h”

main() {
   int ptid, msgtag;
   char buf[100];
   ptid = pvm_parent();
   strcpy(buf, “Olá mundo de ");
   gethostname(buf + strlen(buf), 64);
   msgtag = 1;
   pvm_initsend(PvmDataDefault);
   pvm_pkstr(buf);
   pvm_send(ptid, msgtag);

   pvm_exit();
}
```

Um Programa PVM - Makefile

```
DESTDIR = $(HOME)/pvm3/bin/$(PVM_ARCH)
CFLAGS = -I$(PVM_ROOT)/include
LDFLAGS = -L$(PVM_ROOT)/lib/$(PVM_ARCH)
LIBS = -lpvm3 -lsocket -lnsl

all: ola outro_ola
ola: hola.o
    $(CC) -o ola ola.o $(LDFLAGS) $(LIBS)
outro_ola: outro_ola.o
    $(CC) -o outro_ola outro_ola.o $(LDFLAGS) $( LIBS )
clean:
    rm -f ola outro_ola ola.o outro_ola.o
install:
    cp outro_ola $(DESTDIR)
    cp ola $(DESTDIR)
```

Manuais

- **pvm_spawn:**
  http://www.csm.ornl.gov/pvm/man/pvm_spawn.3PVM.html
- **pvm_initsend:**
  http://www.csm.ornl.gov/pvm/man/pvm_initsend.3PVM.html
- **pvm_recv:**
  http://www.csm.ornl.gov/pvm/man/pvm_recv.3PVM.html
- **pvm_send:**
  http://www.csm.ornl.gov/pvm/man/pvm_send.3PVM.html

Um Programa PVM – Fonte Único

```
#include <stdlib.h>
#include <stdio.h>
#include <pvm3.h>
void main() {
        int mytid, parent;

        mytid = pvm_mytid();
        parent = pvm_parent();;
        if (parent == PvmNoParent)
                master();
        else
                slave();
```

Um Programa PVM – Fonte Único

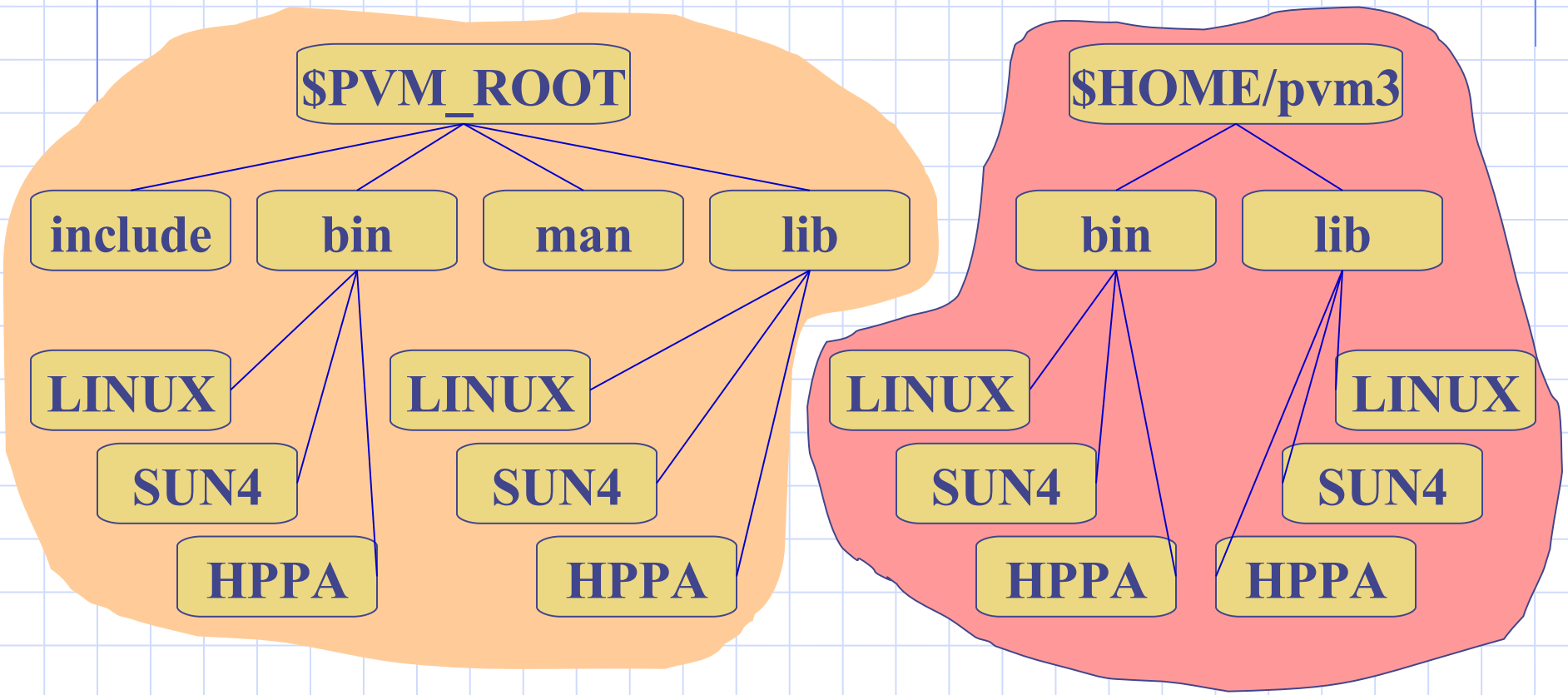
```
void master(){
        int tids[3], numt;
        pvm_catchout(stdout);
        numt = pvm_spawn("bobao", NULL, PvmTaskDefault,
                        "", 3, tids);
        printf("Abriu %d processos\n", numt);
        if (numt < 0)
                printf("Não abriu os processos\n");
        else
                printf("Eu sou o mestre.\n");
        pvm_exit(); }
void slave(){
        int mytid;
        mytid = pvm_mytid();
        printf("Eu sou o escravo %d\n", mytid); }
}
```

Iniciando o PVM

O console inicia o PVM caso seja necessário.

```
pvm> conf
1 host, 1 data format
                    HOST     DTID     ARCH   SPEED
                 primata    40000    LINUX    1000
pvm> add ipanema
1 successful
                    HOST     DTID
                 ipanema    80000
pvm> add joa
0 successful
                    HOST     DTID
                     joa Can't start pvmd
pvm> conf
2 hosts, 2 data formats
                    HOST     DTID     ARCH   SPEED
                 primata    40000    LINUX    1000
                 ipanema    80000     SUN4    1000
```

Estrutura de Diretórios do PVM



Comandos da Console

help	Apresenta uma lista semelhante a esta
add	Adicionar mais nós à maquina virtual
delete	Remover nós da máquina virtual
reset	Matar todos os processos da MV exceto a(s) consoles
quit, ^D	Sair da console deixando o PVM executando
halt	Sair da console terminando o PVM
conf	Listar os nós da máquina virtual
ps -a	Listar todos (-a) os processos executando na MV
…	…

Comandos da Console

- [add] seguido de um ou mais nomes de *hosts*, adiciona estes *hosts* à máquina virtual.
- [alias] define ou lista os apelidos (alias) dos comandos.
- [conf] lista a configuração da máquina virtual incluindo o hostname, pvmd task id, tipo de arquitetura e um índice de velocidade relativa.
- [delete] seguido de um mais nomes de *hosts*, apaga estes *hosts* da máquina virtual PVM. Os processos PVMs ainda executando nessas máquinas são perdidos.
- [echo] ecoa os argumentos.
- [halt] mata todos os processos PVM incluindo a console, e desliga a maquina virtual PVM. Todos os daemons terminam sua execução.

Comandos da Console

- [help] O comando *help* pode ser seguido de um nome de comando, quando então lista as opções disponíveis para este comando.
- [id] imprime o *task id* da console.
- [jobs] imprime os *jobs* em execução .
- [kill] pode ser utilizado para terminar qualquer processo PVM.
- [mstat] mostra o status dos *hosts* especificados.
- [ps -a] lista todos os processos atualmente na máquina virtual, sua localização , o *task-id* e o *task-id* do seu pai.
- [pstat] mostra o "status" de um único processo PVM.

Comandos da Console

- [quit] termina a console, deixando os daemons e tarefas PVM executando.
- [reset] mata todos os processos PVM exceto consoles e reseta todas as filas de mensagens e tabelas internas do PVM. Os daemons são deixados no estado "idle".
- [setenv] mostra ou seta variáveis de ambiente.
- [sig] seguido por um número e um TID, envia o sinal identificado pelo número para a tarefa TID.
- [trace] seta ou mostra a máscara de trace de eventos.
- [unalias] desfaz o comando alias

Comandos da Console

- [version] imprime a versão do PVM em uso.
- [spawn] inicia uma aplicação PVM. As opções que possui são as seguintes:
  - [-count] número de tarefas; padrão é 1.
  - [-host] dispare no host; o padrão é qualquer.
  - [-ARCH] dispare nos hosts com arquitetura do tipo ARCH.
  - [-?] habilita depuração.
  - [->] redireciona saída padrão para console.
  - [-> file] redireciona saída padrão para arquivo.
  - [->> file] redireciona saída padrão para anexar ao arquivo.
  - [-@] trace job, mostra saída na console.
  - [-@ file] trace job, saída para arquivo

Hostfile

- O arquivo de hospedeiros serve para especificar uma configuração ou parâmetros para cada um dos hospedeiros.
- Cada hospedeiro listado é automaticamente adicionado a menos que seja precedido por &.
- A seguir as opções que podem ser encontradas neste arquivo.

Hostfile

Opção	Comentário	Default
lo=userid	nome do login se diferente	o mesmo do atual
so=senha	pedirá senha	mesma senha
dx=	localização do pvmd	/pvm3/lib
ep=	localização dos executáveis	$HOME/pvm3/bin/PVM_ARCH
sp=	velocidade relativa (1 até 1000000)	1000
bx=	localização do depurador	pvm3/lib/debugger
wd=	diretório de trabalho	$HOME
ip=nome	nome alternativo para IP	
so=ms	requer partida manual do pvmd escravo	

Hostfile

```
# Comment lines start with # (blank lines ignored)
gstws
ipsc dx=/usr/gaist/pvm3/lib/I860/pvmd3
ibmd.scri.fsu.edu lo=gst so=pw

# set default options for following hosts with *
* ep=$sun/problemi:~/nla/mathlib
sparky
#azure.epm.ornl.gov
midnight.epm.ornl.gov

# replace default options with new values
* lo=gageist so=pw ep=problemi
thud.cs.utk.edu
speedy.cs.utk.edu

# machines for adding later are specified with &
# these only need listing if options are required
&sun4    ep=problemi
&castor dx=/usr/local/bin/pvmd3
&dasher.cs.utk.edu lo=gageist
&elvis   dx=~/pvm3/lib/SUN4/pvmd3
```

Controle de Processos

- **Identificando-se:**

  int tid = pvm_mytid (void)

  - Devolve o *tid* do processo e o inscreve no PVM.

- **Saindo:**

  int info = pvm_exit (void)

  - Avisa ao pvm local que este processo está saindo do ambiente.

- **Terminando:**

  int info = pvm_kill (int tid)

  - Termina a tarefa identificada por *tid*.

Controle de Processos

- **Criando Processos:**

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

  - Cria *ntask* cópias de *task* e devolve o número de tarefas iniciadas em *numt* e os seus identificadores em *tids*.
  - *argv* é um ponteiro para uma lista de argumentos para *task* terminando por NULL.

Controle de Processos

- O valor de *flag* é uma soma de:

Valor	Opção	Comentário
0	PvmTaskDefault	PVM escolhe onde iniciar os processos.
1	PvmTaskHost	"where" indica o nome do host onde bvai ser executada a tarefa.
2	PvmTaskArch	"where" é o nome de uma PVM_ARCH onde vai ser executada a tarefa.
4	PvmTaskDebug	Começa a tarefa sob o depurador.
8	PvmTaskTrace	Gera dados para conferência.
16	PvmMppFront	Ccomeça a tarefa no MPP front-end.
32	PvmHostComplement	Complementa o conjunto de hospedeiros em "where".

Controle de Processos

int info = pvm_catchout (FILE *ff)

- O padrão é que PVM escreva em *stderr* e *stdout* das tarefas criadas no arquivo de registro /tmp/pvml.<uid>
- Esta rotina faz com que as tarefas filhas chamadas após esta chamada tenham suas saídas redirecionadas para o arquivo correspondente ao descritor de arquivos *ff*.
- A saída das tarefas "netas" também são redirecionadas, desde que estas não modifiquem o valor de *PvmOutputTid*.

Informações

int tid = pvm_parent (void)

- Retorna o *tid* do processo que deu partida nesta tarefa ou o valor PvmNoParent se a tarefa não foi criada por pvm_spawn().

int dtid = pvm_tidtohost (int tid)

- Retorna o *tid* do *daemon* que está rodando no mesmo hospedeiro que *tid*.

Informações

int info = pvm_config (int *nhost, int *narch, struct pvmhostinfo **hostp)

- Retorna informações sobre a máquina virtual:
  - [nhost] número de hospedeiros
  - [narch] número de arquiteturas
  - [hostp] ponteiro para um vetor de estruturas do tipo pvmhostinfo.

Exemplo de pvm_conf

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

main() {
    struct pvmhostinfo*hostp;
    int nhost, narch, info,  i;

    info= pvm_config(&nhost,&narch,&hostp);
    if( info==PvmSysErr )
        printf("O pvm não está respondendo");
```

Exemplo de pvm_conf

```
else{
        printf("Hosts na maquina virtual.\t%d\n",nhost);
        printf("Formatos de dados em uso.\t%d\n\n",
                narch);
        printf("Hosts em operacao\n");
        printf("Nome\t\tArquitetura\t\tVelocidade\n");
        for(i= 0;i<nhost;i++)
           printf("%s\t\t%s\t\t\t%d\n",
                  hostp[i].hi_name, hostp[i].hi_arch,
                  hostp[i].hi_speed);
           printf("\n");
```

Informações

int tid = pvm_tasks (int which, int *ntask, struct pvmtaskinfo **taskp)

- Retorna informações sobre as tarefas PVM rodando na máquina virtual:
  - [which] especifica sobre que tarefas retornam informações, as opções são :
    - [0] todas as tarefas
    - [dtid] todas as tarefas rodando no hospedeiro do *daemon dtid*
    - [tid] a tarefa com identificação *tid*
  - [ntask] número de tarefas
  - [taskp] ponteiro para um vetor de estruturas *pvmtaskinfo*.

Configuração Dinâmica

```
int info = pvm_addhosts (char **hosts,
                         int nhost, int *infos)

int info = pvm_delhosts (char **hosts,
                         int nhost, int *infos)
```

- Estas rotinas adicionam ou retiram os hospedeiros em *hosts* da máquina virtual.
- *info* é o número de hospedeiros adicionados ou retirados com sucesso.
- O argumento *infos* é um vetor de comprimento *nhost* que contém o estado para cada hospedeiro sendo adicionado ou retirado.

Sinalizando

int info = pvm_sendsig (int tid, int signum)

- Esta rotina envia um sinal de número *signum* para o processo de número *tid.*

int info = pvm_notify (int what, int msgtag, int cnt, int *tids)

- Pede que o PVM notifique a rotina quando os seguintes eventos ocorrerem:
  - [PvmTaskExit] tarefa saiu;
  - [PvmHostDelete] hospedeiro foi retirado ou falhou;
  - [PvmHostAdd] hospedeiro foi adicionado.
- As mensagens para a rotina que chamou são enviadas com a etiqueta *msgtag*. O vetor de *tids* diz que tarefas monitorar quando usando TaskExit ou HostDelete

Mensagens

- A ordem das mensagens enviadas é sempre preservada.
- Qualquer tarefa pode mandar mensagens para qualquer tarefa na máquina virtual.
- Mensagens podem ser recebidas em modo bloqueante ou não.
- Suporte a formatos heterogêneos.

Ordem das Mensagens

- A ordem das mensagens enviadas é sempre preservada.
- Se tarefa 1 envia mensagem A para tarefa 2, e em seguida envia mensagem B para tarefa 2. A mensagem A chegará na tarefa 2 antes da mensagem B.
- Se ambas as mensagens chegam antes da tarefa 2 executar um "receive", então uma recepção com "tag" livre sempre retorna primeiro a mensagem A.

Envio das Mensagens

- Enviar mensagens requer três passos:
  - Estabelecer um buffer
  - Empacotar os dados
  - Enviar as mensagens

Buffer de Mensagens

int bufid = pvm_initsend (int encoding)

- Se o usuário está usando um único *buffer* esta é a rotina necessária para estabelecer um *buffer*.
- As opções para *encoding* são:
  - [PvmDataDefault] - codificação XDR é usada (máquinas heterogêneas).
  - [PvmDataRaw] - nenhuma codificação é usada (máquinas homogêneas).
  - [PvmDataInPlace] - dados são transferidos a partir da área de dados do usuário. O usuário deve evitar alterar os dados antes que eles sejam enviados.

Empacotando os dados

- As rotinas de empacotamento podem ser chamadas múltiplas vezes em uma única mensagem.
- Estruturas podem ser passadas empacotando-se individualmente seus elementos.
- Não há limite para a complexidade das mensagens, mas o usuário deve desempacotar os dados na mesma ordem do empacotamento

Empacotando os dados

int info = pvm_pkbyte (char *cp, int nitem, int stride)
int info = pvm_pkcplx (float *xp, int nitem, int stride)
int info = pvm_pkdcplx (float *zp, int nitem, int stride)
int info = pvm_pkdouble (float *dp, int nitem, int stride)
int info = pvm_pkfloat (float *fp, int nitem, int stride)
int info = pvm_pkint (int *np, int nitem, int stride)
int info = pvm_pklong (int *np, int nitem, int stride)
int info = pvm_pkshort (int *np, int nitem, int stride)
int info = pvm_pkstring (char *cp)
int info = pvm_packf (const char *fmt)

Envio de Mensagens

int info = pvm_send (int tid, int msgtag)

- Coloca a etiqueta *msgtag* na mensagem que está no "buffer" ativo e a envia para a tarefa *tid*.

int info = pvm_mcast (int *tids, int ntask, int msgtag)

- Coloca na mensagem, que está no buffer ativo, a etiqueta *msgtag* e a envia para as tarefas identificadas no vetor *tids*. O vetor tem o tamanho *ntask*.

Envio de Mensagens

int info = pvm_psend (int tid, int msgtag, void *vp, int len, int type)

- A rotina empacota e envia os dados em uma só chamada.
- pvm_psend retira *len* dados de tipo *type* do buffer apontado por *vp*.
- Após colocar na mensagem a etiqueta *msgtag* a envia para a tarefa identificada por *tid*.
- pvm_psend não afeta o estado do buffer ativo e que está sendo usado por pvm_send.

Envio de Mensagens

- PVM_STR
- PVM_BYTE
- PVM_SHORT
- PVM_INT
- PVM_FLOAT
- PVM_CPLX
- PVM_DOUBLE
- PVM_DCPLX
- PVM_LONG
- PVM_USHORT
- PVM_UINT
- PVM_ULONG

Recebendo Mensagens

- Receber mensagens requer dois passos:
  - Chamando uma rotina de recebimento, que pode ser bloqueante ou não.
  - Desempacotar os dados
- A rotina de recebimento pode escolher entre:
  - Aceitar qualquer mensagem;
  - Aceitar qualquer mensagem de uma determinada tarefa;
  - Aceitar qualquer mensagem com uma determinada etiqueta;
  - Aceitar somente mensagens de uma determinada tarefa com uma determinada etiqueta.
- Qualquer das rotinas de recebimento pode ser usada independentemente do método utilizado para enviar.

Recebendo Mensagens

int bufid = pvm_recv (int tid, int msgtag)

- A rotina é bloqueante.
- Esta rotina espera uma mensagem da tarefa *tid* com a etiqueta *msgtag*.
- Uma etiqueta igual a -1 indica que qualquer etiqueta serve e um *tid* igual a -1 indica que serão recebidas mensagens de qualquer tarefa.

int bufid = pvm_nrecv (int tid, int msgtag)

- Esta rotina não é bloqueante, caso a mensagem não tenha chegado a tarefa continua.
- Se a mensagem não chegou ainda o valor 0 é colocado em *bufid*.
- Caso a mensagem tenha chegado ela é colocada no *buffer* ativo.

Recebendo Mensagens

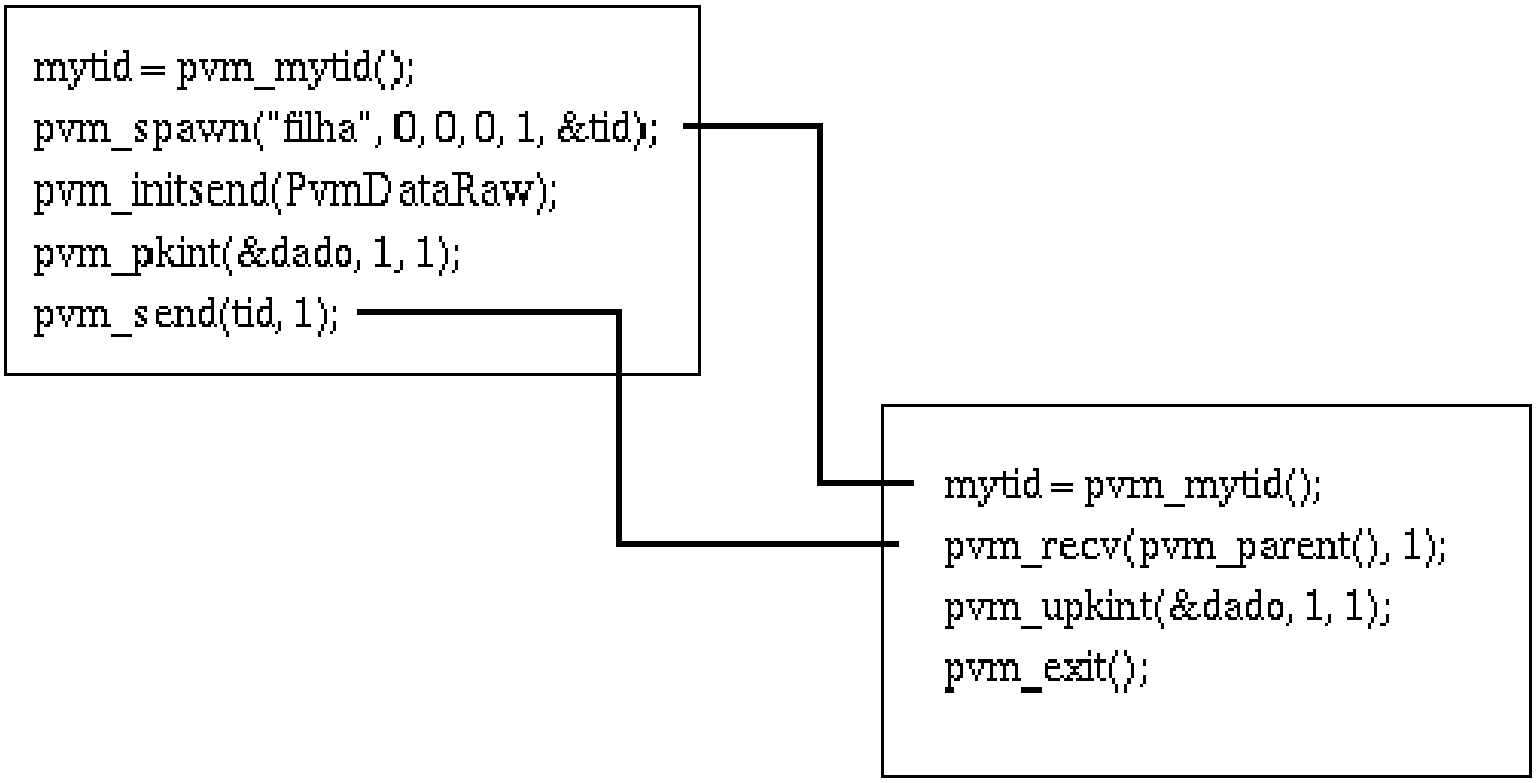
int bufid = pvm_trecv (int tid, int msgtag, struct timeval *tmout)

- Esta rotina espera a chegada de uma mensagem por um período de tempo definido pela estrutura *tmout*.
- Os campos *tv_sec* e *tv_usec* da estrutura *tmout* servem para definir quanto tempo a tarefa deve esperar pela mensagem.

int bufid = pvm_probe (int tid, int msgtag)

- Testa se existe alguma mensagem disponível compatível com os argumentos especificados (*tid* e *msgtag*).

Sincronismo na Comunicação



Sincronismo na Comunicação

- A chamada a pvm_mytid() inscreve o programa no PVM.
- pvm_spawn() cria o processo escravo.
- O processo mestre prepara o buffer de transmissão e o envia para o escravo.
- O processo escravo recebe a mensagem especificando o *tid* do mestre e o mesmo *tag* usado no envio da mensagem.

Mensagens - Exemplo

```
#define PROCS 3
#include "pvm3.h"
#include <stdio.h>
int main(int argc, char **argv)  {
        void master(), slave ();
        int mytid, parent, nprocs;
        nprocs = argc > 1 ? nprocs = atoi(argv[1]) : PROCS;
        mytid = pvm_mytid();
        parent = pvm_parent();
        if (parent == PvmNoParent)
                master(nprocs);
        else
                slave();
        pvm_exit();
```

Mensagens - Exemplo

```
void slave()
{
        int mytid, parenttid;
        char buf[100], hostName[64];
        mytid = pvm_mytid();
        parenttid= pvm_parent();
        gethostname(hostName, 63);
        sprintf(buf, "Eu sou o escravo t%x, falando de %s.",
    mytid,hostName);
        pvm_initsend(PvmDataDefault);
        pvm_pkstr(buf);
        pvm_send(parenttid, 1);
        printf("Eu sou o escravo t%x avisando que já enviei a
    mensagem para t%x.
}
```

Mensagens - Exemplo

```
id master(int nprocs) {
        int *tids, numt;
        char buf[100];
        tids = (int *) malloc (nprocs * sizeof(int));
        if (!tids) pvm_exit();
        pvm_catchout(stdout);
        numt = pvm_spawn("bobao1", NULL, PvmTaskDefault, "",
      nprocs, tids);
        if (numt < 0)
                printf("[Mestre] Nao abriu os processos\n");
        else
                printf("[Mestre] Abriu %d processos\n", numt);

      printf("[Mestre] Tenho de esperar %d mensagens.\n", numt);
        }
}
```

Mensagens - Exemplo

```
void slave()
        int mytid, parenttid;
        char buf[100], hostName[64];
        mytid = pvm_mytid();
        parenttid= pvm_parent();
        gethostname(hostName, 63);
        sprintf(buf, "Eu sou o escravo t%x, falando de %s.",
                        mytid, hostName);
        pvm_initsend(PvmDataDefault);
        pvm_pkstr(buf);
        pvm_send(parenttid, 1);
         printf("Eu sou o escravo t%x avisando que já enviei a
    mensagem.\n", mytid)
```

Mensagens - Exemplo

```
void master(int nprocs) {
    int *tids, numt;
    char buf[100];
    tids = (int *) malloc (nprocs * sizeof(int));
    if (!tids) pvm_exit();
    pvm_catchout(stdout);
    numt = pvm_spawn("mensagem", NULL, PvmTaskDefault,
"", nprocs, tids);
    if (numt < 0)
        printf("[Mestre] Nao abriu os processos\n");
    else
        printf("[Mestre] Abriu %d processos\n", numt);

    printf("[Mestre] Tenho de esperar %d mensagens.\n",
numt);
```

Mensagens - Exemplo

```
        do {
                pvm_recv(-1,1);
                pvm_upkstr(buf);
                printf("[Mestre] Mensagem %d: %s\n", numt, buf);}
         while (--numt);

        do {
            pvm_tasks (0, &numt, NULL);
         while (numt>1);}

        printf("[Mestre] Finalmente todas as tarefas filhas
    terminaram. \n");
}
}
```

Rotinas para gerenciar vários *buffers*

int bufid = pvm_mkbuf (int encoding)

- A rotina cria um novo buffer e especifica o método de codificação usado.

int info = pvm_freebuf (int bufid)

- Libera a área ocupada pelo buffer definido por bufid.

Rotinas para gerenciar vários *buffers*

int bufid = pvm_getsbuf (void)

int bufid = pvm_getrbuf (void)

- pvm_getsbuf (pvm_getrbuf) retorna o buffer de envio (recepção) ativo.

int oldbuf = pvm_setsbuf (int bufid)

int oldbuf = pvm_setrbuf (int bufid)

- pvm_setsbuf (pvm_setrbuf) estabelece o buffer de envio (recepção), salva o estado do buffer ativo e retorna o buffer ativo anterior identificado por oldbuf.

Controle de buffers - Exemplo

```
#include <stdlib.h>
#include <stdio.h>
#include “pvm3.h”

void master(char *nome, int mytid){
        int tid, numt, token = 11, bufid, buffer;

        pvm_catchout(stdout);
        /* Gera uma copia do programa */
        pvm_spawn(nome, NULL, PvmTaskDefault, "", 1, &tid);

        /* Prepara mensagem com o buffer padrao */
        bufid = pvm_initsend(PvmDataDefault);
        pvm_pkint(&mytid, 1, 1);
```

Controle de buffers - Exemplo

```
/* Gera um novo buffer para transmissao */
  buffer = pvm_mkbuf(PvmDataDefault);
  pvm_setsbuf(buffer);
  pvm_pkint(&token, 1, 1);
  printf("Eu sou o mestre e estou enviando %d.\n", token);

  /* Envia mensagem do novo buffer */
  pvm_send(tid, 1);
  pvm_freebuf(buffer);

  /* Volta ao buffer padrao */
  pvm_setsbuf(bufid);
  printf("Eu sou o mestre e estou enviando %x.\n", mytid);
  pvm_send(tid, 1);
```

Controle de buffers - Exemplo

```
void slave(int parent){
  int mytid, token, bufid;

  mytid = pvm_mytid();
  /* Recebe primeira mensagem */
  bufid = pvm_recv(parent, 1);
  pvm_upkint(&token, 1, 1);
  printf("Recebi o valor %d.\n", token);

  /* Recebe segunda mensagem */
  bufid = pvm_recv(parent, 1);
  pvm_upkint(&token, 1, 1);
  printf("Recebi o valor %x.\n", token);
```

Controle de buffers - Exemplo

```
void main(int argc, char **argv) {

    int mytid, parent;
    mytid = pvm_mytid();
    parent = pvm_parent();
    if (parent == PvmNoParent)
        master(argv[0], mytid);
    else
        slave(parent);

    pvm_exit();
```

Buffer – Mais Detalhes

int bufid = pvm_probe (int tid, int msgtag)

- Caso a mensagem pedida ainda não tenha chegado então pvm_probe retorna *bufid* = 0, caso contrário retorna o *bufid* da mensagem.

int info= pvm_bufinfo (int bufid , int *bytes,
int *msgtag, int *tid)

- Retorna *msgtag, tid* e número de bytes da mensagem identificado por *bufid*.

Buffer – Mais Detalhes

```
for ( ; msgtag != TAGKILL ; )
    if ((bufid=pvm_probe(paitid,-1)))
    {
        pvm_bufinfo(bufid, &bytes, &msgtag, &tid);
        switch(msgtag)
        {
        case TAGDADOS:
            @<Espera dados do pai@>@;
            @<Calcula produto escalar@>@;
            @<Envia resultado para o pai@>@;
            break;
        case TAGKILL:
            printf("Acabei, mytid e %x.\n", mytid);
            break;
```

Roteamento de Mensagens

- Normalmente as mensagens são roteadas de uma tarefa para outra através do ”pvm daemon”.
- O pvmd roteia a mensagem para o pvmd da máquina que contém a tarefa destino da mensagem.
- Esse, por sua vez, repassa a mensagem para a tarefa local adequada.
- O PVM, contudo, oferece rotinas para permitir a comunicação direta entre tarefas.
- Neste caso o protocolo utilizado é o TCP.

Roteamento de Mensagens

int oldval = pvm_setopt (PvmRoute, int val)

- A função pvm_setopt serve para estabelecer valores para uma série de parâmetros da biblioteca PVM.
- A primeira opção sendo igual a *PvmRoute* diz para o PVM como uma tarefa deve estabelecer comunicação com outras tarefas.
- Neste caso, a opção *val* indica que forma deve ser estabelecida esta comunicação.
- Não há meios de especificar a forma de roteamento específica para uma única mensagem.

Roteamento de Mensagens

- Os valores que *val* pode assumir são os seguintes:

Val	Significado
PvmRouteDirect	Tenta estabelecer ligações diretas e aceita pedidos de lilgações diretas.
PvmDontRoute	Não solicita e não permite ligações diretas.
PvmAllowDirect (default)	Não tenta estabelecer, mas aceita pedidos de ligações diretas.

Roteamento de Mensagens

- Uma vez estabelecida uma ligação direta entre tarefas ela permanece aberta enquanto as tarefas estiverem ativas.
- Uma ligação direta é usada por ambas as tarefas.
- Se por algum motivo uma ligação direta não puder ser estabelecida entre tarefas, então a ligação padrão entre os *daemons* é utilizada.
- Existe um custo para estabelecer ligações diretas, e o número delas é limitado.

Grupos Dinâmicos

- Grupos são coleções de tarefas que colaboram de uma forma mais integrada na solução de problemas.
- As funções de grupo foram criadas por cima do PVM.
- As funções estão em um biblioteca (libgpvm3.h) separada que dever ser ligada com o programa.
- O pvmd não executa funções de grupo, que são executadas por um servidor de grupos.

Grupos Dinâmicos

- Qualquer tarefa PVM pode entrar e sair de grupos sem avisar as outros membros do grupo.
- Tarefas podem enviar mensagens para grupos dos quais não fazem parte.
- Qualquer tarefa pode chamar funções de grupo a qualquer hora, com exceção das funções pvm_lvgroup, pvm_barrier e pvm_reduce, que requerem que a tarefa seja membro daquele grupo.

Grupos Dinâmicos

int inum = pvm_joingroup (char *nome)

- A primeira chamada para a rotina cria um grupo com o nome apontado pela variável nome e põe a tarefa no grupo.
- *inum* é o número de entrada da tarefa no grupo. Estes números começam em 0 e são incrementados para cada nova tarefa que entra no grupo.
- Se o número de tarefas que deixam um grupo é maior do que as que entram, pode haver "buracos" nessa numeração.
- Uma tarefa pode entrar em vários grupos.

int info = pvm_lvgroup (char *nome)

- Esta função retira a tarefa do grupo indicado pela variável nome.

Informações sobre Grupos

int tid = pvm_gettid (char *nome, int inum)

- Retorna o *tid* do processo que pertence ao nome e tem número de entrada *inum*.
- Com esta função é possível a duas tarefas descobrirem os *tids* respectivos simplesmente se alistando no mesmo grupo.

int inum = pvm_getinst (char *nome, int tid)

- Retorna o número de entrada da tarefa com tid igual a *tid* no grupo *nome* .

int size = pvm_gsize (char *nome)

- Retorna o número de membros do grupo *nome*.

Informações sobre Grupos

```
int tid, inum;
 ...
tid = 135;
 inum = pvm_getinst("work", tid);
 if (inum > 0) printf("\nInstance number of
    the task identified by %d: %d", tid,inum);
    ...
```

Barreiras em Grupos

int inum = pvm_barrier (char *nome, int count)

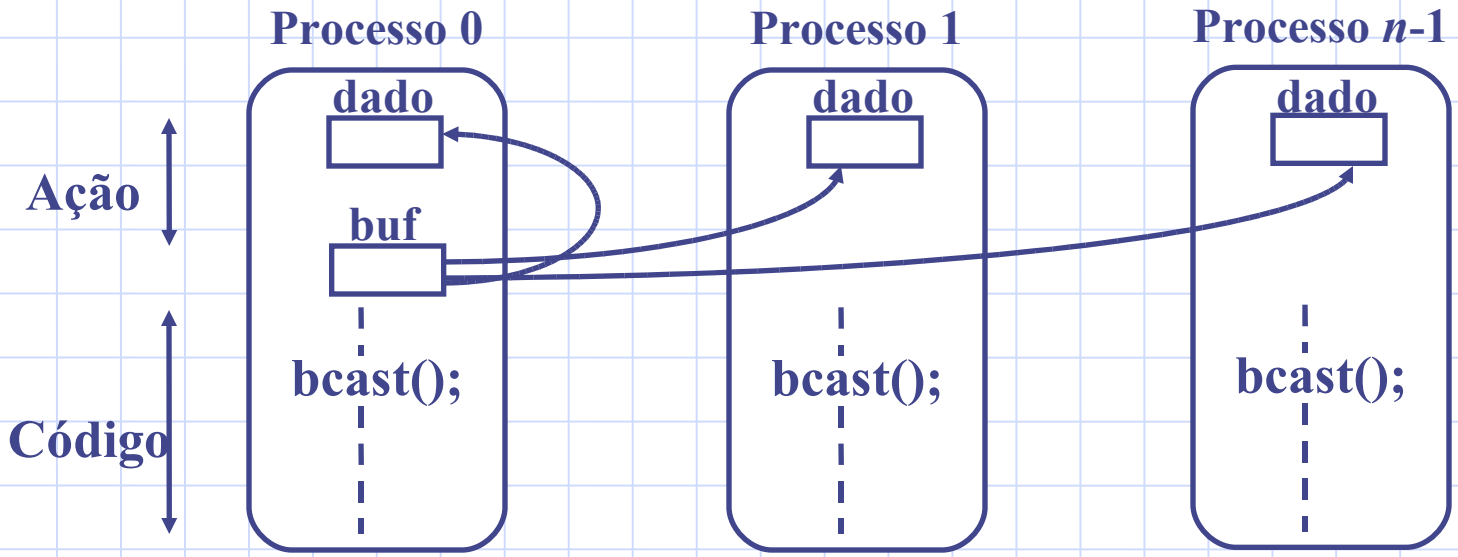
- Ao chamar pvm_barrier() a tarefa é bloqueada até que *count* membros do grupo tenham chamado pvm_barrier().
- É um erro uma tarefa que não pertence a um grupo chamar essa função.
- É um erro se os argumentos *count* não são idênticos em todas as chamadas.
- Um uso comum é sua utilização após a chamada da função "joingroup" por cada tarefa e antes de serem utilizadas as operações de comunicação coletivas.
- Isso garante que todas as tarefas tiveram oportunidade de se juntar ao grupo e que as operações coletivas terão resultado correto.

Barreiras em Grupos

```
int inum, count;
 ...
inum = pvm_joingroup("work");
 ...                          /* total number of members */

count = pvm_gsize("work");

info = pvm_barrier("work", count);

 if (info < 0) pvm_perror(); ...
```

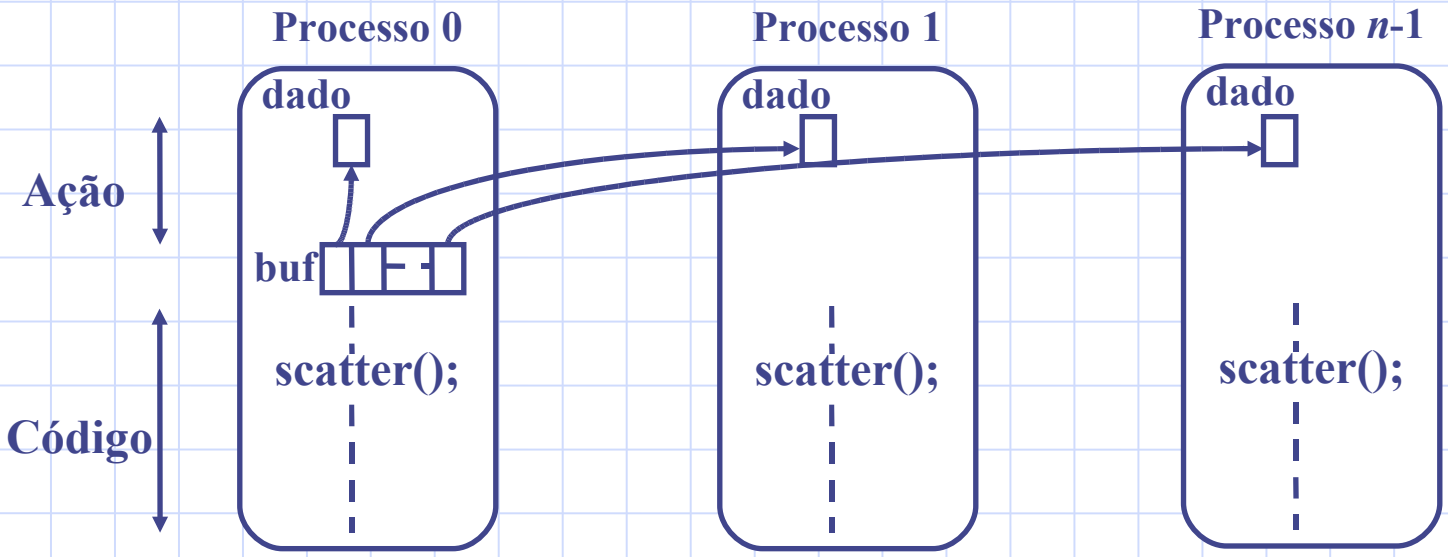
Broadcast

- Envio da mesma mensagem para todos os processos.
- Multicast: envio da mesma mensagem para um grupo de processos.



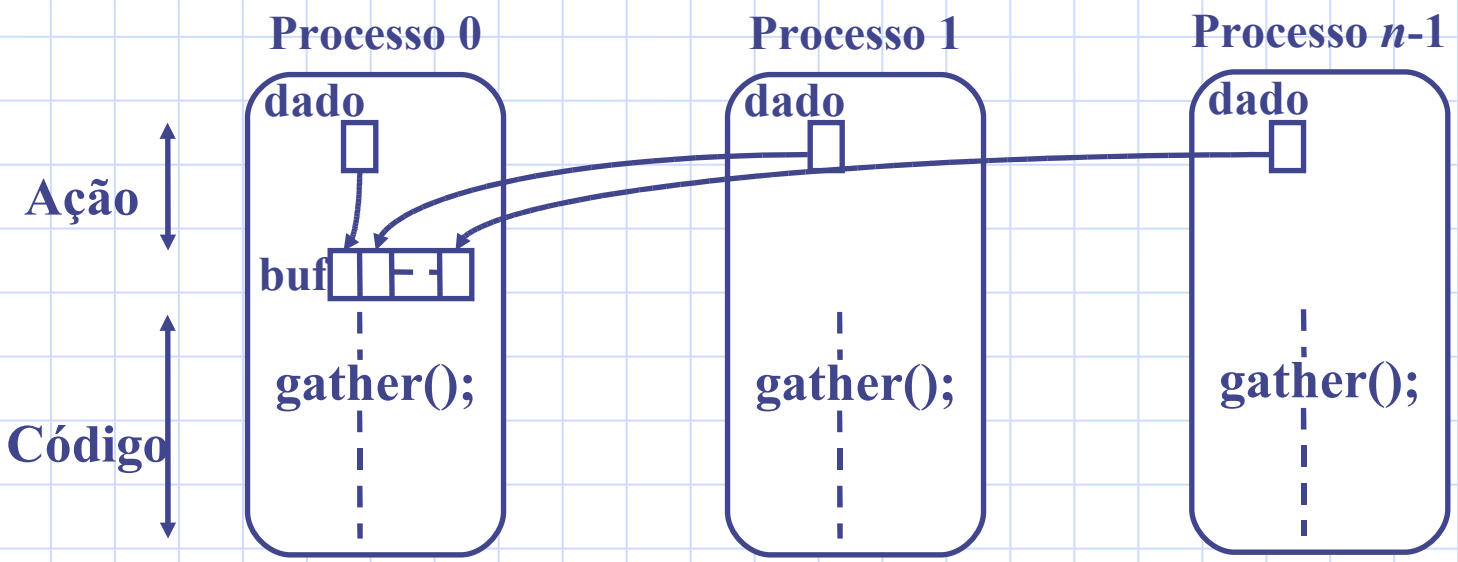
Scatter

- Envio de cada elemento de uma matriz de dados do processo raiz para um processo separado; o conteúdo da *i-ésima* localização da matriz é enviado para o *i-ésimo* processo.



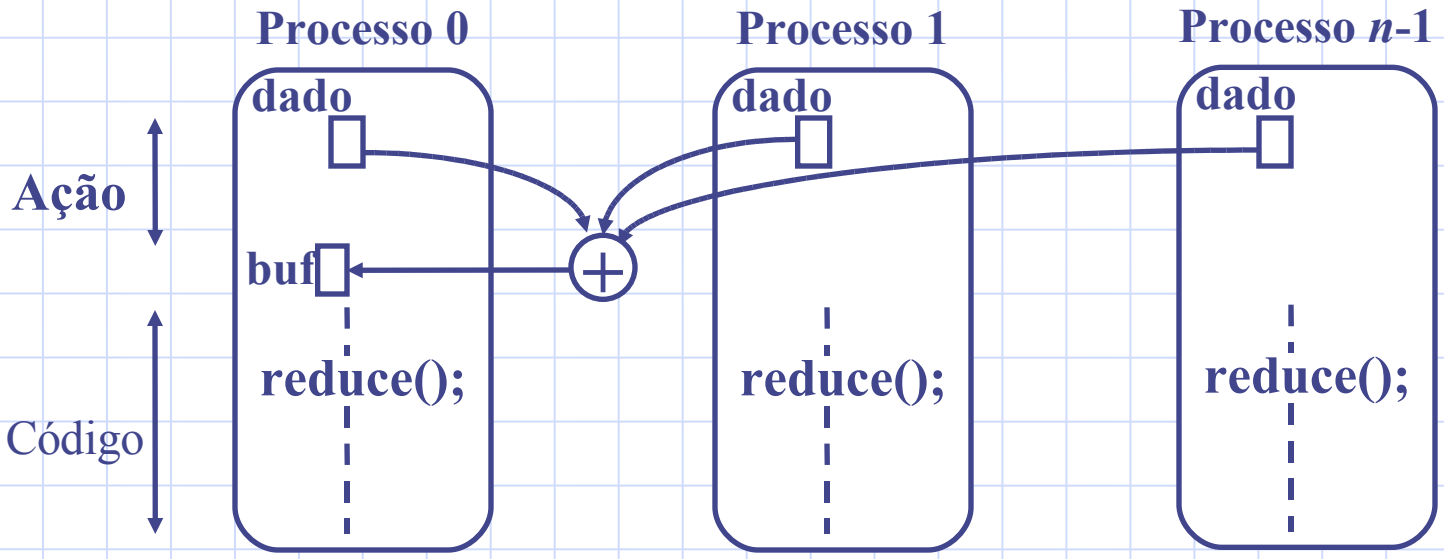
Gather

- Um processo coleta dados de um conjunto de processos.



Reduce

- Operação de "gather" combinada com uma operação lógica ou aritmética específica:
  - Ex: valores coletados e somados pelo processo raiz.



Mensagens para Grupos

int info = pvm_bcast (char *nome, int msgtag)

- pvm_bcast() coloca a etiqueta *msgtag* na mensagem e a envia para todos os membros do grupo excetuando a tarefa que está enviando, se ela é um membro do grupo.

- Membros do grupo são todas as tarefas que pertencem ao grupo no momento da chamada da função.

Mensagens para Grupos

```
int bufid, info, inum;
char buf[50]="Broadcast"; ... /* Join a named group */
 inum = pvm_joingroup("work");
 ... /* Initialize the send buffer */
bufid = pvm_initsend(PvmDataDefault);
/* Pack message into the buffer */
info = pvm_pkstr(buf);
/* Send the message */
info = pvm_bcast("work", 1);
 if (info < 0) { pvm_perror(); pvm_exit(); return -1; } ...
```

Scatter

int info = pvm_scatter(void *result, void *data, int count, int datatype, int msgtag, char *group, int rootginst)

- *pvm_scatter()* realiza uma distribuição de dados a partir da tarefa raiz para cada um dos membros do grupo, incluindo ela mesma.
- Todos os membros do grupo devem chamar pvm_scatter(), cada um recebe uma porção do vetor *data* vindo da tarefa raiz para o seu vetor local *result*.
- Se uma tarefa que não pertence ao grupo chamar a função, resultará em erro.
- Q quando for enviar uma matriz lembre que na linguagem C as matrizes são armazenadas linha a linha na memória.

Scatter

- result:
  - Ponteiro do endereço inicial do vetor de comprimento *count* de *datatype* que receberá a porção local dos dados.
- data:
  - Se *n* é o número de membros do grupo, então este vetor deve ter um comprimento *n*count* do tipo datatype. Este argumento só tem significado na tarefa raiz.
- count:
  - Inteiro especificando o númer de elementos de *datatype* para ser enviados para cada membro do grupo.
- msgtag:
  - Etiqueta da mensagem.
- group:
  - Nome do grupo.
- Rootginst:
  - Inteiro que indica a instância do membro que é a tarefa raiz.

Scatter

```
int info, rootginst;
 int matrix;
int val[3];


...
 rootginst = pvm_getinst("work", 100);
info = pvm_scatter(&val, &matrix, 3, PVM_INT, 19, "work",
     rootginst);

 if (info < 0) pvm_perror(); ...
```

Gather

int info = pvm_gather(void *result, void *data, int count, int datatype, int msgtag, char *group, int rootginst)

- *pvm_gather()* realiza uma coleta de dados para a tarefa raiz a partir de cada um dos membros do grupo, incluindo ela mesma.
- Todos os membros do grupo devem chamar pvm_gather(), cada um envia uma porção do vetor *data* para a tarefa raiz a partir do seu vetor local *result*.
- Se uma tarefa que não pertence ao grupo chamar a função, resultará em erro.
- Ao enviar uma matriz note que o C armazena as matrizes linha a linha na memória.

Gather

- result:
  - Se *n* é o número de membros do grupo, então este vetor deve ter um comprimento *n*count* do tipo datatype. Este argumento só tem significado na tarefa raiz.
- data:
  - Ponteiro do endereço inicial do vetor de comprimento *count* de *datatype* que receberá a porção local dos dados.
- count:
  - Inteiro especificando o número de elementos de *datatype* para ser enviados por cada membro do grupo para a raiz.
- msgtag:
  - Etiqueta da mensagem.
- group:
  - Nome do grupo.
- Rootginst:
  - Inteiro que indica a instância do membro que é a tarefa raiz.

Gather

```
int info, rootginst;
int matrix;
 int val[3]={5, 4, 3};
 ...
rootginst = pvm_getinst("work", 100);
info = pvm_gather(&matrix, &val, 3, PVM_INT, 19, "work",
    rootginst);
if (info < 0) pvm_perror(); ...
```

Redução em Grupos

int info = pvm_reduce (void (*func)(), void *data, int nitem, int datatype, int msgtag, char *nome, int root)

- pvm_reduce() executa uma operação aritmética global sobre um grupo, por exemplo soma global ou máximo global.
- Há quatro operações pré-definidas:
  - PvmMax
  - PvmMin
  - PvmSum
  - PvmProduct
- O resultado da operação aparece na tarefa com instância *root*. A redução será realizada elemento a elemento do vetor *data* fornecido como entrada.
- pvm_reduce não é bloqueante. Se a tarefa chama a função e sai do grupo antes que o *root* também chame, pode ocorrer um erro.

Redução em Grupos

data **Pointer to the local values. The data array on the root task will be overwritten with the result.**
count **Specifies the number of elements in the data array. The result will contain *count* values.**

datatype **Specifies the type of the data in the data array. Defined datatypes: PVM_BYTE (except for PvmSum and PvmProduct) PVM_SHORT PVM_CPLX PVM_FLOAT PVM_INT PVM_DOUBLE PVM_DCPLX**
msgtag **Label of the message.**
group **The name of the group.**
root **Instance number of a member where the result appears.**
Returned value
Info **is an integer status code. If *pvm_reduce* is not**

Redução em Grupos

```
int info;
int val[3]={5, 4, 3};
 ...
info = pvm_reduce(PvmMax, &val, 3,
   PVM_INT, 1, "work", 100)

if (info < 0) pvm_perror(); ...
```

Balanceamento de Carga

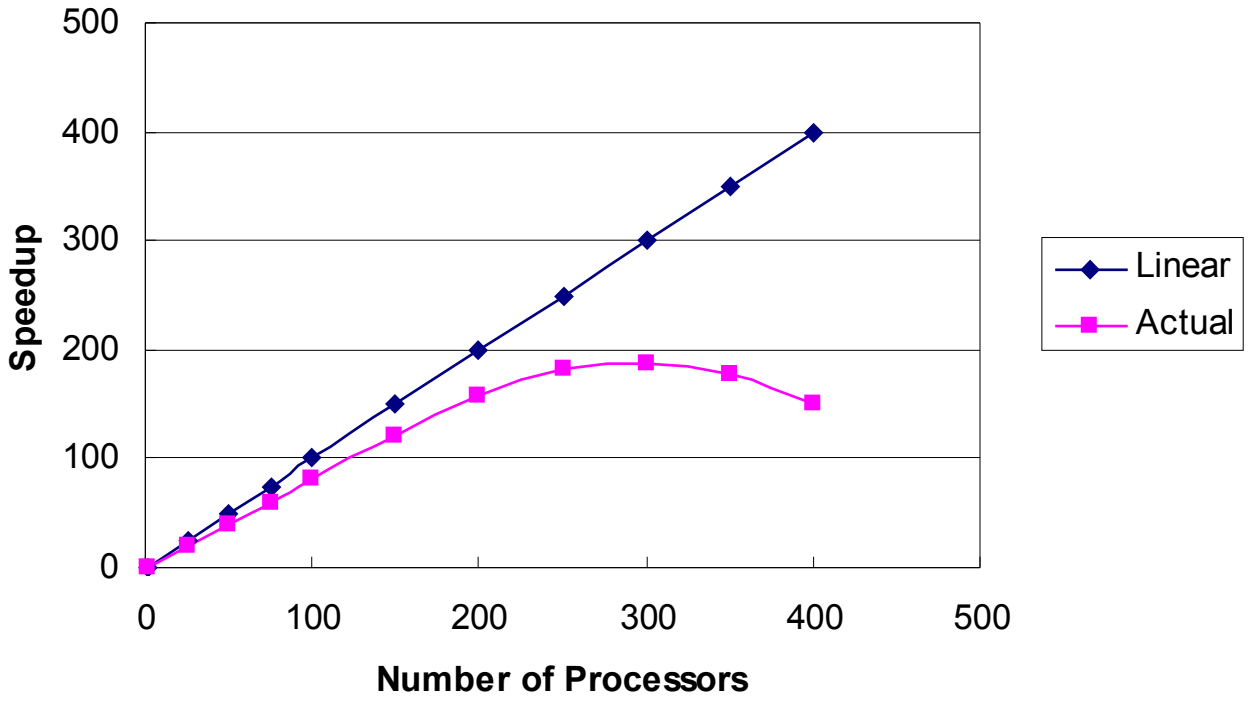
Balanceamento de Carga

- O Balanceamento de Carga é muito importante para o desempenho das aplicações.
- Assegurar que a parcela de atividade de cada máquina seja justa pode representar um grande incremento no desempenho.
- O método mais simples de balanceamento de carga é o estático.
- Neste método, o problema é dividido e tarefas são atribuídas a cada máquina apenas uma vez.

Balanceamento de Carga

- O objetivo é manter todos os processadores ocupados todo o tempo:
  - Mesmo que seja por motivos econômicos, você vai gostar de manter o uso de cada processador o mais alto possível.
  - A qualquer tempo que você tenha um processador ocioso, você terá uma evidência que sua carga de trabalho não está igualmente distribuída por todas as máquinas.
  - Existe a possibilidade que sua aplicação não esteja rodando tão rápido ou tão eficientemente como poderia.
  - Você poderá então descobrir uma distribuição mais balanceada da carga de trabalho.

Balanceamento de Carga



- Speed-up sublinear devido a:
  - Gargalos de comunicação
  - Sobrecarga de comunicaçãp
  - Limitações de memória
  - **Decomposição de tarefas mal feita.**

Balanceamento Estático

- O particionamento pode acontecer antes do trabalho iniciar, ou como o primeiro passo da aplicação.
- O tamanho e o número de tarefas pode ser ajustado dependendo da capacidade computacional de cada máquina.
- Se a única informação utilizada para alocar o trabalho aos processadores for unicamente a sua dimensão, então o método é dito estático por natureza, já que ele não muda independente do conteúdo dos dados, mas apenas se sua dimensão muda.
- Em uma rede com baixa carga, este esquema pode ser bastante eficiente.

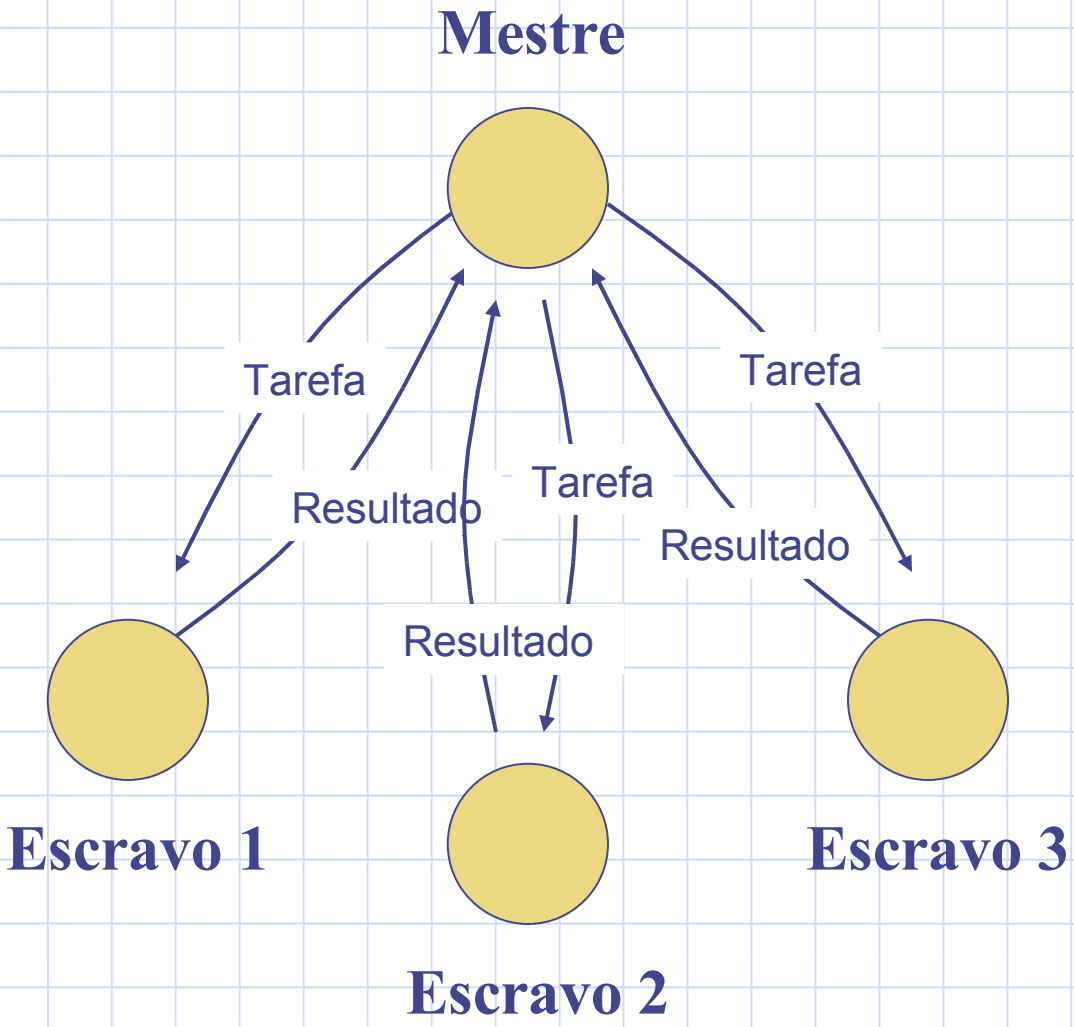
Balanceamento Estático

- Decomposição de Matriz:
  - Muitas operações de matrizes são muito facilmente decompostas algoritmicamente, em termos de subseções regulares da matriz original.
  - Por exemplo, o cálculo do determinante usa aspectos da estrutura da matriz bastante longe do conteúdo dos dados que possam ocupar um elemento em particular.
  - A computação pode ser facilmente dividida entre os processadores sem levar em conta os dados propriamente ditos, usando apenas conhecimentos a cerca da dimensão da matriz.
- Particionamento do Conjunto do Índice:
  - De uma forma mais geral, se você tem "n" coisas para computar e "m" processadores disponíveis para o trabalho, um processo muito simples é dividir "n/m" para cada processador.

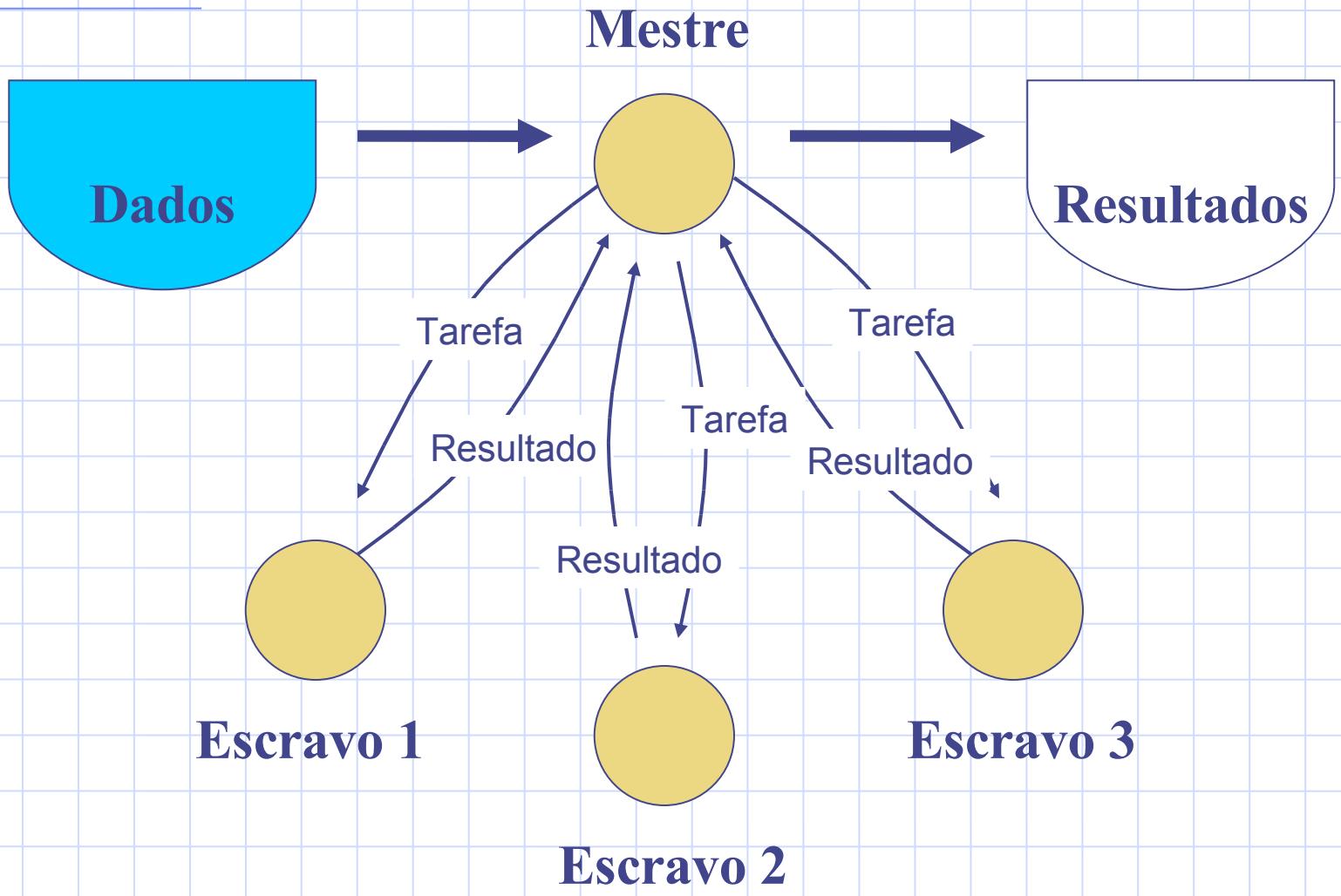
Balanceamento Dinâmico

- Quando a carga computacional varia, um método dinâmico mais sofisticado de balanceamento de carga é necessário.
- O método mais popular é chamado do paradigma do “saco de tarefas”.
- Ele é tipicamente implementado como um programa mestre/escravo, onde o mestre gerencia um conjunto de tarefas.
- Ele envia trabalhos para os escravos assim que eles ficam ociosos.
- Este método é utilizado no programa xep encontrado na distribuição do pvm.

Decomposição Mestre - Escravo



Decomposição Mestre-Escravo



Balanceamento Dinâmico

- O método mestre-escravo não é adequado para aplicações que requerem comunicação de tarefa para tarefa, já que as tarefas iniciarão e terminarão em tempos arbitrários.
- Neste caso, um terceiro método pode ser utilizado. Em algum tempo pré-determinado todos os processos param; as cargas de trabalho são re-examinadas e re-distribuídas se necessário.
- Variações desses métodos são possíveis para aplicações específicas.

Balanceamento Dinâmico

- Por mais fáceis e intuitivos que os método estáticos possam ser, eles apenas focam na estrutura dos dados e não nas características dos dados em si mesmos.
- Por exemplo: se a matriz cujo determinante está sendo calculado for esparsa (ou seja, com maioria dos elementos iguais a 0), um método estático não levará isto em consideração, enquanto que um projeto mais inteligente deve estar apto a reconhecer esta informação como significativa e processar a matriz mais eficientemente.

Balanceamento Dinâmico

- **Métodos de Grade Adaptativos:**
  1. Há uma classe inteira de problemas que podem ser caracterizados pelo uso de "grades" nos dados. Quanto mais fina a grade, mais acurado o resultado, contudo, maior será o tempo de computação necessário.
  2. Com freqüência você não sabe quão fina deve ser a grade que você necessita até que você inicie os cálculos e descubra, por exemplo, que o erro está muito grande, requerendo uma grade mais fina e cálculos mais precisos.

Balanceamento Dinâmico

- **Métodos de Grade Adaptativos:**
  1. Um algoritmo seqüencial apenas recalcularia os pontos da grade e forçaria no sentido de uma grade mais fina.
  2. Uma aproximação distribuída permitiria o uso de métodos de grade adaptativos, que poderia tratar a grade mais fina com um elemento de dados global, dividido entre os processadores disponíveis, e portanto alcançando um balanceamento de carga muito melhor do aquele que haveria com o processador trabalhando na sua partição inicial de dados e que a seção de refinamento por si mesmo.

Balanceamento Dinâmico

- Simulações N-body
  1. Uma outra classe de problemas (em realidade há alguma sobreposição entre as duas) lida com o caso onde N partículas unicamente identificadas (mas geralmente chamadas de "bodies"), as quais pode ou não ter interação uma sobre as outras, deve ter alguns cálculos efetuados em cada uma delas.
  2. Em alguns casos, especialmente naqueles onde não há interação intra-partículas, isto evolui para uma situação de particionamento estático de conjunto de índice.

Balanceamento Dinâmico

- Simulações N-body
  1. Mais freqüentemente, contudo, cada partícula exerce alguma forma de influência sobre seus pares, algumas vezes muito localizadamente (i.e., os efeitos tem curto alcance, como a Força Atômica Forte), algumas vezes não (como a gravidade).
  2. Nos casos onde há áreas bem definidas de interação, algumas vezes é melhor atribuir todas as partículas que interagem a um mesmo processador, e lidar com todas as iterações restantes (i.e., interações envolvendo partículas fora da área local) como se todas as partículas fossem apenas uma única grande partícula.

Balanceamento Dinâmico

- Mestre/Trabalhadores
  1. Algumas vezes é possível projetar sua aplicação de modo que o trabalho seja criado em unidades que possam ser colocadas em uma fila e distribuídas para os processadores que estão em outra fila, onde vão sendo adicionados assim que terminem as tarefas em que estejam trabalhando.
  2. Em realidade esta é uma estratégia geral que pode ser aplicada para a situação acima ou sempre que não houver dependências de dados entre as partes em que você dividir os dados.
  3. Lembre-se , contudo, que este esquema não costuma ser bem escalável para um grande número de processadores.

Considerações de Desempenho

- O PVM não coloca limites no paradigma de programação que um usuário pode escolher.
- Contudo, há algumas considerações de desempenho que devem ser levadas em conta. A primeira é a granulosidade da tarefa.
- Isto tipicamente se refere à taxa entre os números de bytes recebidos por um processo e o número de operações de ponto flutuante que ele realiza.
- Aumentando a granulosidade irá aumentar a velocidade de execução da aplicação, mas o compromisso é uma redução no paralelismo disponível.

Considerações de Desempenho

- O número total de mensagens enviadas também é um fator a ser considerado.
- Como uma regra geral, enviando um pequeno número de mensagens grandes leva menos tempo do que enviar um grande número de pequenas mensagens.
- Isto nem sempre se aplica, contudo. Algumas aplicações podem sobrepor computação com o envio de pequenas mensagens. O número ideal de mensagens é específico para cada aplicação.

Considerações de Desempenho

- Algumas aplicações são bem adequadas ao paralelismo funcional, enquanto que outras se aplicam bem ao paralelismo de dados.
- No paralelismo funcional diferentes máquinas fazem diferentes tarefas baseadas nas suas capacidades específicas.
- Por exemplo, um supercomputador pode resolver parte de um problema adequado ao uso de vetorização, um multiprocessador pode resolver outra parte com uso de paralelização e estações gráficas podem ser utilizadas para visualizar os dados em tempo real.

Considerações de Desempenho

- No paralelismo de dados, os dados são distribuídos para todas a tarefas na máquina virtual. Operações (freqüentemente bastante similares) são então realizadas em cada conjunto de dados e a informação é transmitida entre os processadores até que o problema seja resolvido.
- Programas em PVM podem também usar um mistura de ambos os modelos para explorar totalmente as potencialidades de cada máquina.
- Computadores diferentes terão capacidades de processamento diferentes.

Considerações de Desempenho

- Mesmo que todas máquinas sejam do mesmo tipo e modelo, eles podem apresentar diferenças de desempenho devido à carga dos outros usuários.
- Considerações a respeito da rede também são importantes se você estiver utilizando um conjunto de máquinas.
- As latências de rede podem causar problemas e também a capacidade de processamento disponível pode variar dinamicamente dependendo da carga de cada máquina.
- Para combater esses problemas, alguma forma de balanceamento de carga deve ser utilizada.