

# Parallel Virtual Machine

*Roberto de **Beauclair** Seixas*

tron@lncc.br

# *Processamento Paralelo*

Método de computação utilizado na resolução de grandes problemas, através da subdivisão de tarefas por vários processadores.

- alta performance
- baixo custo
- produtividade

# *Tipos de Processamento Paralelo*

- *Massively Parallel Processors (MPPs)*
  - dezenas a centenas (ou até milhares) de processadores interligados como um único.
  - centenas de Gigabytes de memória.
- *Distributed Computing*
  - conjunto de computadores interligados via rede local de alta velocidade.
  - arquiteturas heterogêneas.

# *Paradigmas*

- *Shared Memory*
- *Parallelizing Compilers*
- *Message Passing*

– PVM

“Tornar um conjunto de computadores em uma grande **Máquina Paralela Virtual** “

# *Formas de Heterogeneidade*

- Arquitetura
- Formato dos dados
- Velocidade Computacional
- Carga da máquina
- Carga na rede local

# *Heterogeneidade*

- Arquitetura
  - 386/486 PCs (UNIX)
  - Workstations
    - » SGI4, SGI5, SUN4 e RS6K
  - Processadores com memória partilhada
  - Computadores vetoriais
  - MPPs

# *Heterogeneidade*

- Formato dos dados
  - Em computação distribuída isto é muito importante, pois dados mandados de um computador para outro **tem** que ser entendidos.
  - Sistemas especificamente projetados para *MPPs* não podem ser usados em sistemas distribuídos, pois não contém informações necessárias.

# *Heterogeneidade*

- Velocidade Computacional
  - arquiteturas heterogêneas
    - » processadores mais rápidos não podem depender ou esperar processadores mais lentos.
- Carga da Máquina
  - mesma arquitetura
    - » os processadores podem estar executando tarefas de outros usuários, alterando drasticamente a velocidade de processamento.

# *Heterogeneidade*

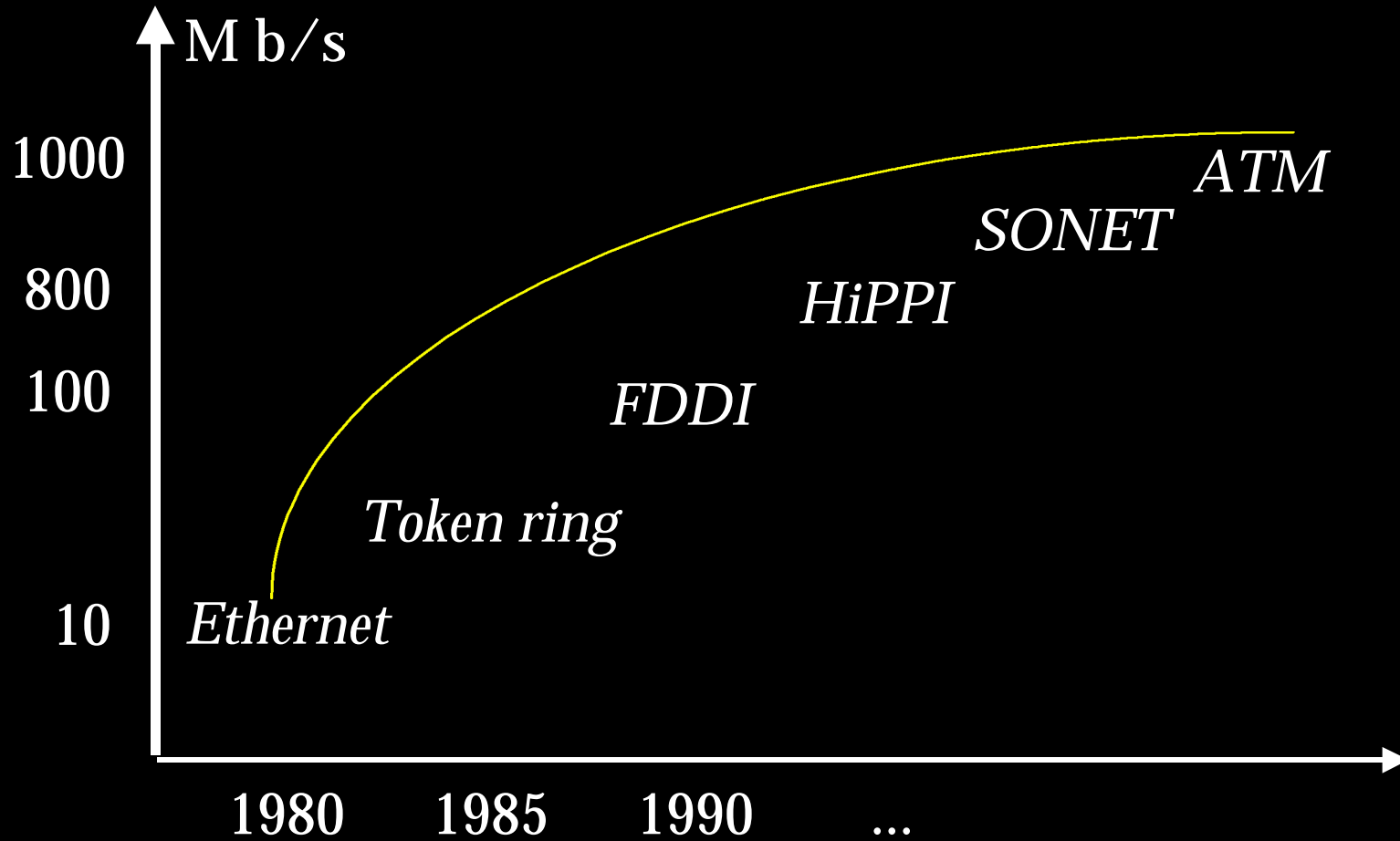
- Carga da Rede Local

Como o *PVM* usa o paradigma de *message-passing*, o que significa enviar, pela rede, mensagens de um computador para outro. Isto implica depender de todos os computadores da rede local, inclusive os que não pertencem a “máquina virtual”.

# *Computação Distribuída*

- Baixo custo pois utiliza os computadores já existentes na rede;
- A *performance* final pode ser melhorada pela alocação de tarefas específicas a computadores que executam melhor, ou mais rápido, aquela tarefa;
- Os recursos da “máquina virtual” pode ser expandidos a qualquer momento, seguindo as novas tecnologias;
- Os programas continuam sendo desenvolvidos nos seus próprios ambientes nativos, usando compiladores, depuradores e editores das próprias máquinas;
- Os computadores, isoladamente, são bastante estáveis.

# *Velocidade das Redes Locais*



# *Evolução*

1989 - Oak Ridge National Laboratory

– PVM 1.0

1991 - University of Tennessee

– PVM 2.0 até 2.4

1993 - Completamente reescrito e distribuído gratuitamente para ser usado em aplicações, não comerciais, paralelas e/ou distribuídas.

– PVM 3.0 (versão instalada: PVM 3.3.10)

# *PVM*

- PVM é um sistema que permite que uma rede de computadores heterogêneos (UNIX) seja usada como um grande computador paralelo.
  - fornece funções para, automaticamente, disparar tarefas na máquina virtual e permite a comunicação e sincronização entre elas. Uma tarefa é definida como uma “unidade de computação”, é análoga a um processo UNIX.

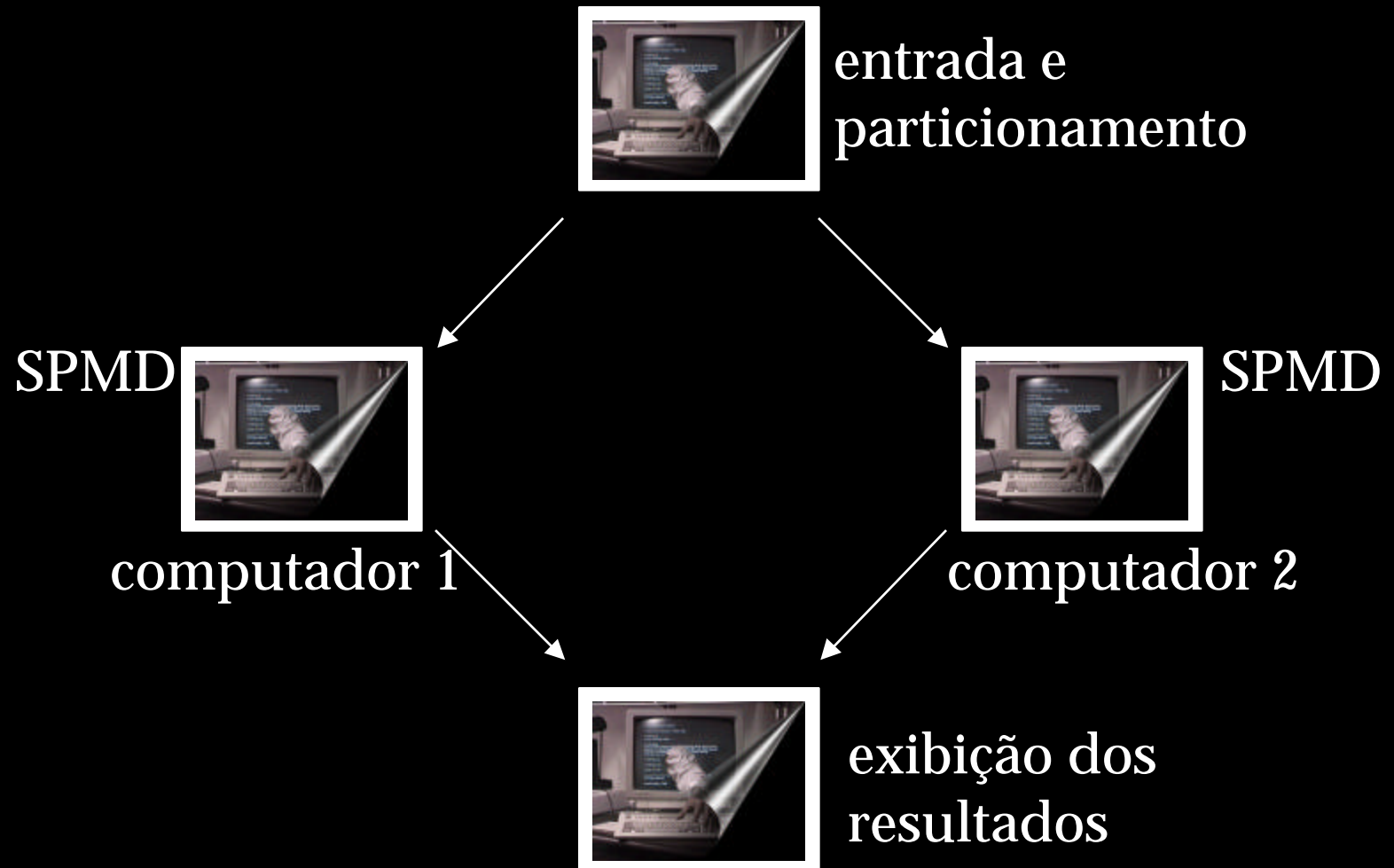
# *PVM*

- Ambiente da máquina virtual é definido pelo usuário;
- Acesso transparente ao *hardware*;
- Computação baseada em tarefas (*tasks*);
  - programas UNIX
- Modelo explícito de *message-passing*;
  - a decomposição é definida na aplicação
- Suporte à heterogeneidade;
- Suporte à multiprocessamento.

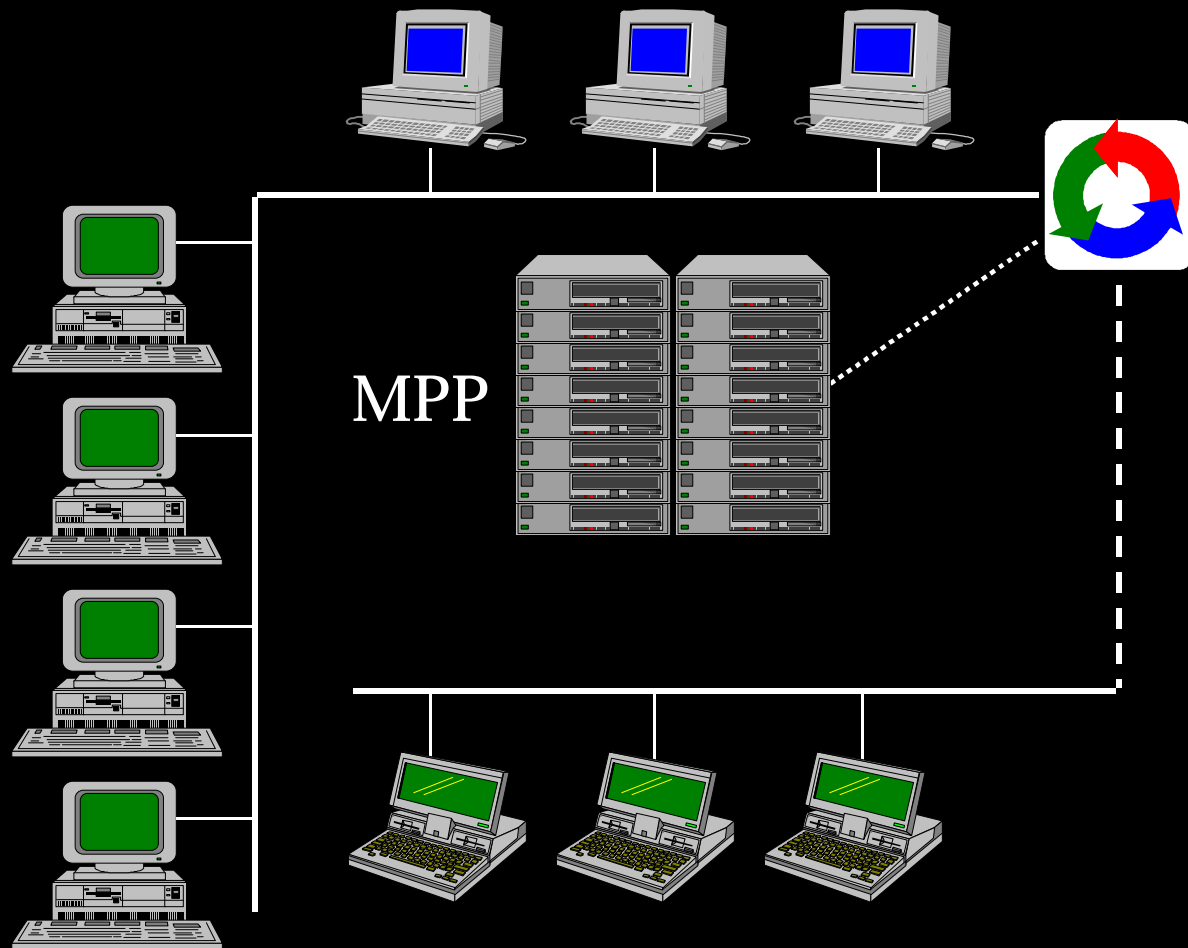
# PVM

- É composto de duas partes:
  - *daemon*
  - biblioteca (C, C++ e Fortran)
- Aplicação é um conjunto de tarefas
  - *Functional Parallelism*
    - » input, initialization, solution, output and display.
  - *Data Parallelism*
    - » uma única tarefa atuando em conjuntos diferentes de dados.

# PVM



# PVM



## *TIDs e Groups*

- Todas as tarefas PVM são identificadas por um número inteiro: *task identifier* - TID;
- As mensagens são recebidas e enviadas pelos TIDs;
- Os TIDs são únicos em toda a máquina virtual;
- Pode-se agrupar e numerar tarefas;
- Cada tarefa pode “entrar” e “sair” dos grupos sem interferir nas outras tarefas.

# *Desenvolvendo uma Aplicação*

1. Criam-se programas sequenciais, contendo chamadas à biblioteca PVM.
  - cada programa corresponde a uma tarefa da aplicação
  - os programa são compilados para cada arquitetura disponível
2. A aplicação é executada por um dos programas, responsável pela execução dos demais, contendo tarefas de inicialização e particionamento.
  - esta *master task* dispara e controla, via mensagens, as *slave tasks* e, normalmente, coleta os resultados

# Protocolo PVM

## *master*

- dispara *slave tasks*
  - » parâmetros (*argv*)
- transmite perguntas
  - » aloca *buffer*
  - » empacota dados
  - » qualifica mensagem
  - » envia *buffer*
- recebe respostas
  - » recebe *buffer*
  - » desempacota dados

## *slave*

- espera perguntas
  - » testa qualificador
  - » recebe *buffer*
  - » desempacota dados
- processa os dados
- transmite respostas
  - » aloca *buffer*
  - » empacota dados
  - » qualifica mensagem
  - » envia *buffer*

# *Hello, PVM !*

```
#include "pvm3.h"
void main ()
{
int nt, tid, msgtag = 1;
char buf[100];
printf ("Hello, world!\n");
nt = pvm_spawn ("hello_cnt", NULL, PvmTaskDefault, NULL, 1, &tid);
pvm_recv (tid, msgtag);
pvm_upkstr (buf);
printf ("%s da TID=%x\n", buf, tid);
pvm_exit();
}
```

# *Hello, PVM !*

```
#include "pvm3.h"
void main ()
{
int p_tid, msgtag = 1;
char buf[100] = "Hello, PVM! da ";
p_tid = pvm_parent ();
strcat (buf, getenv("HOST")); /* nome da máquina */
pvm_initsend (PvmDataDefault);
pvm_pkstr (buf);
pvm_send (p_tid, msgtag);
pvm_exit ();
}
```

# *Executando o PVM*

- PVM usa duas variáveis de ambiente:
  - PVM\_ROOT (  $\{\text{HOME}\}$ /pvm3 )
  - PVM\_ARCH (  $\{\text{PVM\_ROOT}\}$ /lib/cshrc.stub )
- $\{\text{PVM\_ROOT}\}$ /lib/PVM\_ARCH
  - pvm e pvmd3
  - libpvm3.a, libfpvm3.a e libgpvm3.a
- $\{\text{PVM\_ROOT}\}$ /bin/PVM\_ARCH
  - pvmgs
  - executáveis gerados pelo usuário

# *Executando o PVM*

`% pvm ↵`

dispara o *console* PVM

`pvm> add hostname ↵`

insere na VM

`pvm> delete hostname ↵`

retira da VM

`pvm> conf ↵`

lista as máquinas

`pvm> t a ↵`

lista as *tasks*

ativas

`pvm> quit ↵`

sai do *console* PVM

`pvm> halt ↵`

encerra o PVM

`% pvm hostfile ↵`

dispara & insere

# *Compilando a Aplicação*

```
PVM_ARCH = SGI
PVMDIR = /usr/local/pvm
PVMLIB = $(PVMDIR)/lib/$(PVM_ARCH)/libpvm3.a \
         $(PVMDIR)/lib/$(PVM_ARCH)/libgpvm3.a
BDIR = $(HOME)/pvm3/bin
XDIR = $(BDIR)/$(PVM_ARCH)

CC = gcc
CFLAGS = -g
INCS = -I$(PVMDIR)/include -I$(XABDIR)/include
LIBS = $(PVMLIB) -lm
```

# *Compilando a Aplicação*

default: raycast main

main: main.c

```
$(CC) $(CFLAGS) -o main main.c $(INCS) $(LIBS)  
cp main $(XDIR)
```

raycast: raycast.c

```
$(CC) $(CFLAGS) -o raycast raycast.c $(INCS) $(LIBS)  
cp raycast $(XDIR)
```

# *Tipos de Aplicações*

- *Crowd*

- É uma coleção de processos (relacionando-se), tipicamente executando um mesmo código, realizando cálculos em partes diferentes do problema, normalmente envolvendo troca de resultados intermediários.

- » *Host-Node Model (Star Computational Model)*

- domain decompositions and number crunching algorithms

# *Tipos de Aplicações*

- *Tree*

- Os processos são “disparados”, dinamicamente, de acordo com o decorrer da execução, estabelecendo uma hierarquia *parent-child*, em uma configuração de árvore.

- » *Node-Only Model (Tree Computational Model)*

- branch-and-bound, divide-and-conquer and recursive algorithms

# *Técnicas de Programação*

- Fase 1: *Crowd Model*
  - Inicialização do grupo de processos
- Fase 1: *Tree Model*
  - Disseminação das informações do grupo e dos parâmetros do problema e particionamento.
- Fase 2
  - Computação propriamente dita.
- Fase 3
  - Coleta dos resultados, exibição e encerramento do grupo de processos.

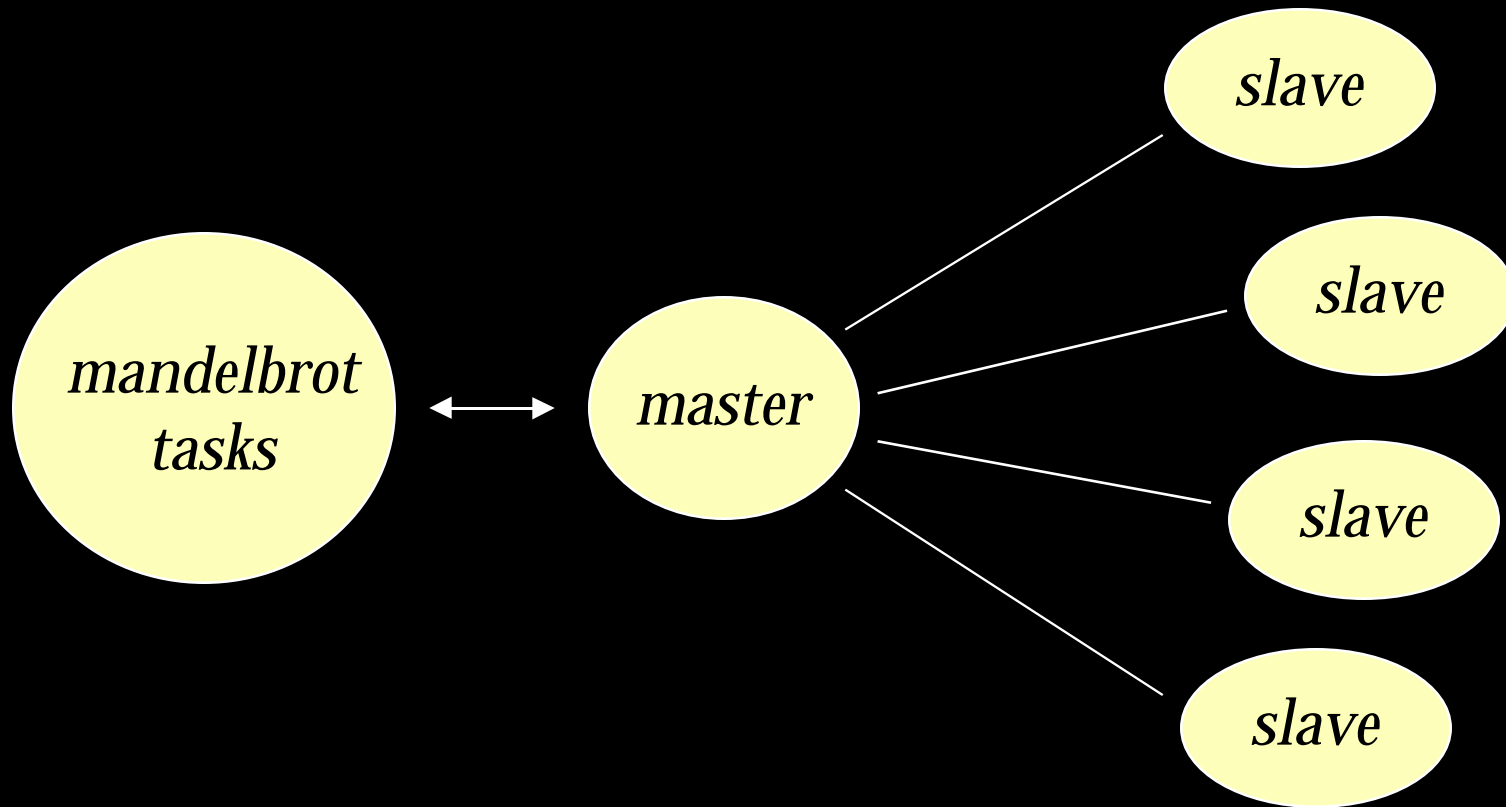
# *Particionamento do Problema*

- Particionamento Estático (*Static Data Decomposition*)
  - Assumindo-se que um problema de  $N$  elementos será executado em  $P$  processadores, determina-se um particionamento de  $N/P$  elementos por processador.
- Particionamento Dinâmico (*Dynamic Data Decomposition*)
  - O *master* subdivide o problema em de  $N$  elementos em  $K$  partes e cada processador recebe  $N/K$  elementos. De acordo com a disponibilização dos processadores, estes recebem mais  $N/K$  elementos até o término do problema.

# *Exemplos*

- *Crowd Model*
  - Mandelbrot algorithm
    - » sem comunicação entre clientes
  - Cannon's algorithm (multiplicação de matrizes)
    - » com comunicação entre clientes
- *Tree Model*
  - Split-Sort-Merge algorithm

# *Mandelbrot*



# *Mandelbrot: master*

```
for (i = 0; i < NumWorkers; i++)  
    pvm_spawn (worker name);    { start up worker i }  
    pvm_send (worker tid, 999); { send task to worker i }
```

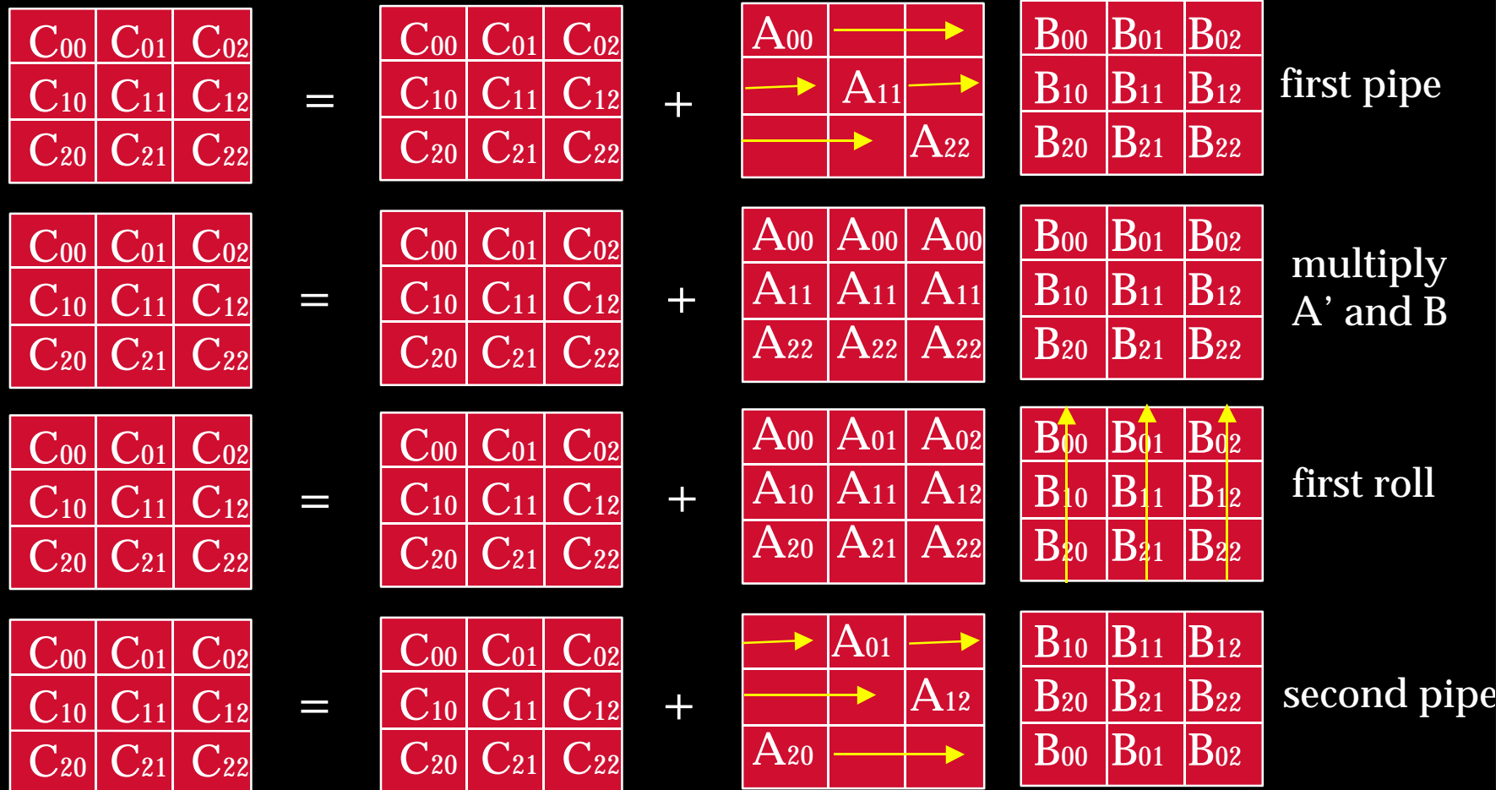
```
while (“work to do”)  
    pvm_rcv (888);                { receive results }  
    pvm_send (available worker id, 999);  
    display_result ();
```

```
for (i = 0; i < NumWorkers; i++)  
    pvm_rcv (888);                { receive results }  
    pvm_kill (worker tid i);    { terminate worker i }  
    display_result ();
```

# *Mandelbrot: slave*

```
while (TRUE)
  pvm_recv (999);           { receive task}
  result = Mandelbrot_Calculations (task);      { compute
  result }
  pvm_send (master tid, 888);           { send result to master
  }
```

# Matrix Multiplication



# *Matrix Multiplication: SPMD*

```
{ Processor 0 starts up other processes }
```

```
if (my processor number >= 0)
```

```
    for (i = 1; i <= MeshDimension*MeshDimension; i++)
```

```
        pvm_spawn (slave name, ...)
```

# *Matrix Multiplication: SPM*

forall processors Pij,  $0 \leq i, j < \text{MeshDimension}$

for k = 0 to MeshDimension-1

**{ pipe }** if (myrow = (mycolumn+k) mod MeshDimension  
          {send A to all Pxy, x = myrow, y <> mycolumn }  
          pvm\_mcast ((Pxy, x = myrow, y <> mycolumn), 99)

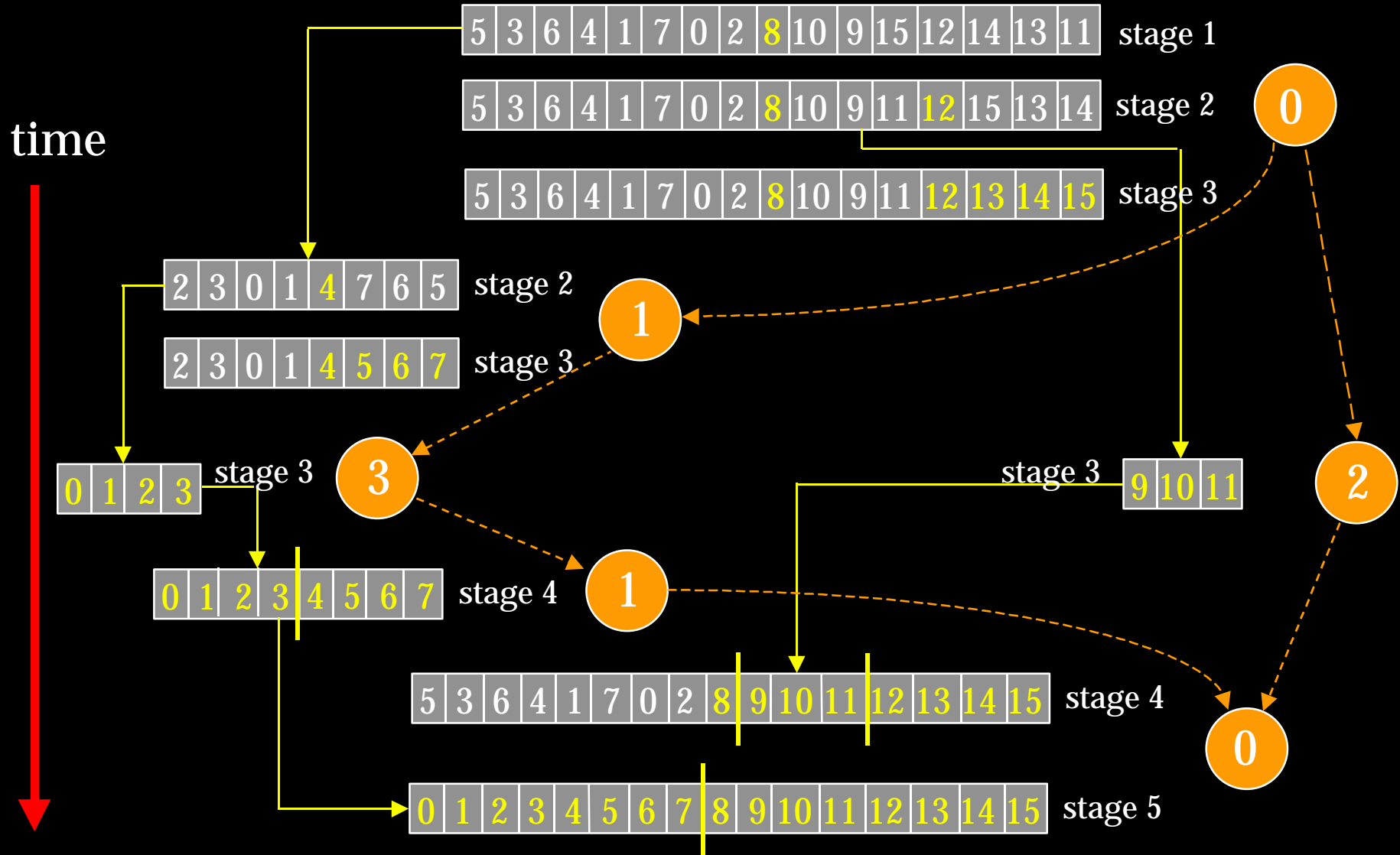
else

pvm\_recv (999) { receive A }

**{ multiply }** Multiply (A, B, C)

**{ roll }** { send B to Pxy, x = myrow -1, y = mycolumn }  
pvm\_send ((Pxy, x = myrow -1, y = mycolumn), 888)  
pvm\_recv (888) { receive B }

# Split-Sort-Merge Algorithm



# *Split-Sort-Merge Algorithm*

```
{ Spawn and partition list based on a broadcast tree pattern. }
for (i = 1; i <=N; i++)    { such that  $2^N = Num\_Procs$  }
  forall processors P such that  $P < 2^i$ 
    pvm_spawn (. . .)      { process id  $P \text{ xor } 2^i$  }
    if ( $P < 2^{(i - 1)}$ )
      midpt = PartitionList (list)
      { send list [0..midpt] to  $P \text{ xor } 2^i$  }
      pvm_send (( $P \text{ xor } 2^i$ ), 999)
      list = list [midpt + 1 .. MAXSIZE]
    else
      pvm_recv(999)
```

# *Split-Sort-Merge Algorithm*

```
{ Sort remaining list }
Quicksort (list [midpt + 1 .. MAXSIZE])
{ Gather/Merge sorted sub-lists. }
for (i = N; i >=1; i--)      { such that  $2^N = Num\_Procs$  }
    forall processors P such that  $P < 2^i$ 
        if ( $P > 2^{(i - 1)}$ )
            { Send list to  $P \text{ xor } 2^i$  }
            pvm_send (( $P \text{ xor } 2^i$ ), 888)
        else
            pvm_recv (888)
            merge templist into list
```

# *API do PVM (funções principais)*

- *Process Control*

- int tid = pvm\_mytid ( void )
- int info = pvm\_exit ( void )

- *Information*

- int tid = pvm\_parent ( void )

- *Message Buffer*

- int bufid = pvm\_initsend (int encoding)

- *Sending / Receiving Data*

- int info = pvm\_send (int tid, int msgtag)
- int info = pvm\_mcast (int \*tids, int ntask, int msgtag)
- int bufid = pvm\_recv (int tid, int msgtag)
- int bufid = pvm\_nrecv (int tid, int msgtag)

# *API do PVM (funções principais)*

- *Packing Data (pvm\_pk...) / Unpacking Data (pvm\_upk...)*
  - int info = pvm\_pkbyte (char \*cp, int nitem, int stride)
  - int info = pvm\_pkint (int \*ip, int nitem, int stride)
  - int info = pvm\_pkfloat (float \*fp, int nitem, int stride)
  - int info = pvm\_pkdouble (double \*dp, int nitem, int stride)
  - int info = pvm\_pkstr (char \*cp)
  - int info = pvm\_pkcplx (float \*xp, int nitem, int stride)
  - int info = pvm\_pkdcplx (double \*zp, int nitem, int stride)
  - int info = pvm\_pklong (long \*lp, int nitem, int stride)
  - int info = pvm\_pkshort (short \*sp, int nitem, int stride)

# *Bibliografia*

*Parallel Virtual Machine  
Quick Reference Guide Release 3.3*

*Parallel Virtual Machine: A Users' Guide and Tutorial for  
Networked Parallel Computing*  
*Al Geist, Adam Beguelin, Jack Dongarra, et al*  
*The MIT Press; 1994*

*PVM3 Users' Guide and Reference Manual*  
*Al Geist, Adam Beguelin, Jack Dongarra, et al*  
*ORNL/TM-12187; 1994*