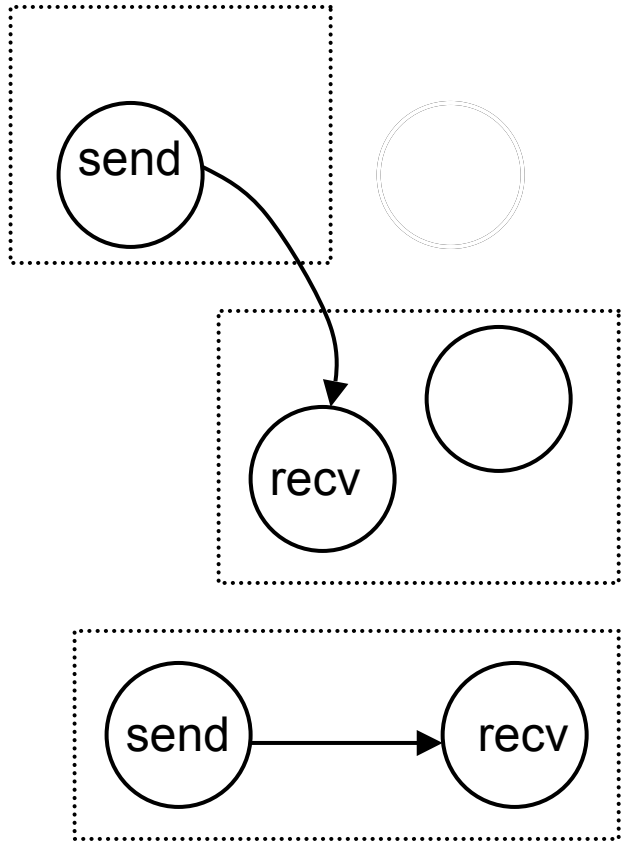
Programação Paralela por Passagem de Mensagens

Arquitecturas Paralelas 1

Luís Paulo Peixoto dos Santos

Processos Sequenciais Comunicantes

- A computação paralela consiste em vários processos sequenciais, com o seu espaço de endereçamento próprio, e que executam autonomamente;
- Os processos comunicam entre si através da passagem explícita de mensagens (send / receive)
- O número de processos pode variar durante a execução do programa;
- Os processos podem ser idênticos (SPMD) ou não;
- Os processos podem ser mapeados num número arbitrário de máquinas, que podem ser heterogéneas;



Processos Sequenciais Comunicantes

Funcionalidades

- Gestão de máquinas
  - Adicionar máquina Mi
  - Remover máquina Mi
- Gestão de processos
  - Criar processo `pid=cria_proc (<file>,<máquina>);`
  - Terminar processos `term_proc (pid);`
- Passagem de mensagens
  - Enviar mensagem
  - Receber mensagem

Passagem de Mensagens: Endereçamento

- Canais Virtuais
  - 2 processos criam entre si um canal, através do qual são posteriormente enviadas todas as mensagens
- Anónima
  - 1 processo envia uma mensagem anónima, que pode ser recebida por qualquer processo
- Nomeada
  - A mensagem é enviada para um processo identificado por um *nome (pid)*.

Passagem de Mensagems: etiqueta

- Cada mensagem contem uma etiqueta (tag) que a tipifica

  send (<destino>, <tag>, <msg>)

- Na recepção cada processo deve indicar quais as etiquetas que está interessado em receber

  recv (<origem>, <tag>, <msg>)

- Existem mecanismos para receber mensagens com qualquer etiqueta

  recv (<origem>, <any_tag>, <msg>)

  devendo depois o receptor inspeccionar a tag recebida e executar as acções apropriadas

Passagem de Mensagens: dados

- Na maioria dos sistemas de passagens de mensagens, cada mensagem pode conter vários campos de diferentes tipos de dados.

- Antes de ser enviada o corpo da mensagem é criado, num *buffer*, usando funções de empacotamento

```
empacotar_int (msg, int *);
empacotar_char (msg, char *);
send (<destino>, <tag>, <msg>);
```

- Na recepção a mensagem deve ser desempacotada

```
recv (<origem>, <tag>, <msg>);
desempacotar_char (msg, char *);
desempacotar_int (msg, int *);
```

Passagem de mensagens: sincronismo e bloqueio

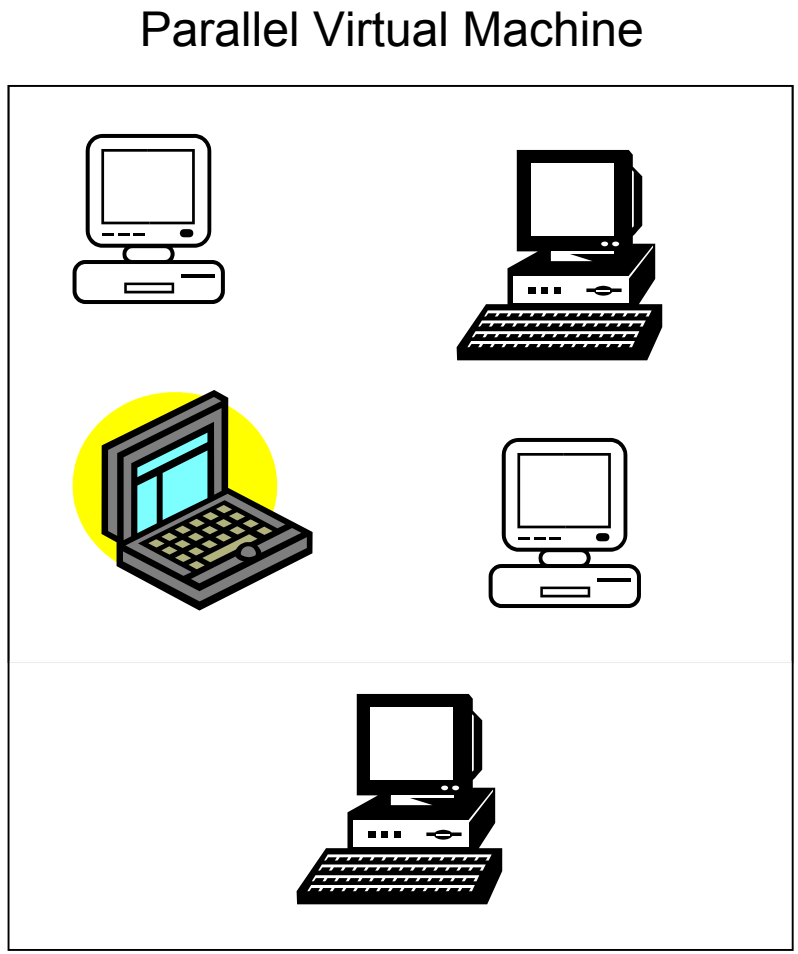
- A comunicação diz-se síncrona se os interlocutores esperam uns pelos outros, garantindo que estão simultaneamente envolvidos na operação
- A comunicação diz-se assíncrona se os interlocutores não estão simultaneamente envolvidos na operação
- A comunicação diz-se bloqueante se o processo espera que alguma fase da comunicação esteja completa
- A comunicação diz-se não bloqueante se o processo não pára à espera que alguma fase da comunicação se complete

Passagem de Mensagens vs. Invocação Remota de Métodos

	Passagem de Mensagens	Invocação Remota de Métodos
Dados a transmitir	Empacotamento de dados	Lista de parâmetros
Envio de pedidos/ informação	Envio explícito de mensagens etiquetadas	Invocação de um método específico
Recepção de pedidos/ informação	Recepção explícita de mensagens	Execução implícita do método invocado
Reacção do receptor	A acção a executar é determinada pela etiqueta (tag) da mensagem	A acção a executar é determinada pelo método invocado
Identificação do receptor	Canal, nome ou anónima	Apontador para objecto remoto (proxy)

PVM – Parallel Virtual Machine

- Ambiente de execução e desenvolvimento de programas paralelos baseados em passagem de mensagens
- Agrupa um conjunto de máquinas numa única máquina virtual
- A constituição da máquina virtual pode ser heterogénea e dinâmica



PVM

O PVM é constituído por 2 componentes:

1. O daemon, residente em todas as máquinas que constituem a máquina virtual, responsável pela gestão de processos e máquinas e que assegura a entrega de mensagens entre processos.

2. Uma biblioteca de funções que permitem o desenvolvimento de programas para a máquina virtual, providenciando:
   - Comunicação entre processos
   - Criação / término de programas
   - Adição / remoção de máquinas à máquina virtual

PVM – A Consola

A invocação do pvm numa máquina inicia a máquina virtual e dá acesso a uma consola.

Esta consola aceita vários comandos para monitorização e configuração das máquinas e processos da máquina virtual.

O pvm pode ser invocado com um argumento (hostfile) que permite a inicialização da máquina virtual com várias máquinas reais:

`pvm cluster3` onde cluster3 =

```
pe1
pe2
pe3
```

PVM – Comandos da Consola

`conf` – informação sobre a configuração da máquina virtual

`ps` – informação sobre os processos existentes na máquina virtual

`quit` – abandonar a consola, mantendo a máquina virtual activa

`halt` – terminar a máquina virtual

`add` – adicionar uma máquina à máquina virtual

`delete` – remover uma máquina da máquina virtual

`spawn -> <file>` - iniciar um processo na máquina virtual

`kill <tid>` - terminar processo `<tid>` na máquina virtual

PVM – Modelo de Programação

Um programa típico em PVM é:

- iniciado num único nodo
- se é o primeiro processo então
  - lança processos noutras máquinas
  - comunica identificadores de processos (tids) aos seus filhos
  - inicia a sua actividade
- Senão
  - inicia a sua actividade

PVM – biblioteca de funções

- Configuração da máquina virtual

```
int pvm_config (int *nhosts, int *narch, struct pvmhostinfo **hostp);
```

- Gestão de processos

```
int pvm_mytid (void);
int pvm_parent (void);
int pvm_spawn (char *task, char **argv, int flag, char *where,
               int ntask,int *tids);
int pvm_kill (int tid);
```

- Comunicação

```
int pvm_initsend (int encoding);
int pvm_pk* (data type *, int nitem, int stride);
int pvm_send (int tid, int msgtag);
int pvm_recv (int tid, int msgtag);
int pvm_upk* (data type *, int nitem, int stride);
```

- Término

```
int pvm_exit (void);
```

PVM - Configuração

```
int pvm_config (int *nhosts, int *narch, struct pvmhostinfo **hostp);

struct pvmhostinfo {
   int hi_tid;    // pvm daemon tid on this host
   int *hi_name;  // this host's name
   int *hi_arch;  // this host's architecture
   int hi_speed;  // this host relative speed }
```

Determina a configuração da máquina virtual, devolvendo:

- Número de máquinas
- Número de arquitecturas diferentes
- Informação sobre cada máquina

PVM – Gestão de Processos

```
int pvm_mytid (void);
```

Devolve identificador (`tid`) deste processo.

```
int pvm_parent (void);
```

Devolve identificador (`tid`) do pai deste processo; `PvmNoParent` se o processo não foi criado com `pvm_spawn ()`.

```
int pvm_spawn (char *task, char **argv, int flag, char *where,
               int ntask,int *tids);
```

Permite criar processos nas máquinas que constituem a máquina virtual, devolvendo os sues identificadores (`tids`).

```
int pvm_kill (int tid);
```

Permite terminar em qualquer instante o processo com identificador `tid`.

PVM – Comunicação
(envio de mensagens)

```
int pvm_initsend (int encoding);
```

Inicia o buffer de envio e especifica o modo de codificação: `PvmDataDefault`, `PvmDataRaw`.

```
int pvm_pk* (data type *, int nitem, int stride);
```

Cria o corpo de dados da mensagem com campos de diferentes tipos.

```
int pvm_send (int tid, int msgtag);
```

Envia a mensagem para o processo `tid` com a etiqueta `msgtag`.

Este envio é bloqueante (espera que a mensagem seja copiada para o *buffer* do `pvmd` e assíncrono.

Exemplo:

```
pvm_initsend (PvmDataRaw);
pvm_pkint (arrint, 5, 1);
pvm_pkchar (&caracter, 1, 1);
pvm_send (dest_tid, MSG_TAG);
```

PVM – Comunicação
(recepção de mensagens)

```
int pvm_recv (int tid, int msgtag);
```

Recebe uma mensagem com etiqueta `msgtag` do emissor `tid`.

Esta recepção é bloqueante e assíncrona.

Se `tid==-1` qualquer emissor é aceite;

Se `msgtag==-1` qualquer etiqueta é aceite;

```
int pvm_upk* (data type *, int nitem, int stride);
```

Lê os dados da mensagem (desempacotamento).

Exemplo:

```
pvm_recv (orig_tid, MSG_TAG);
pvm_upkint (&nvalues, 1, 1);
pvm_upkint (arrint, nvalues, 1);
pvm_upkchar (&caracter, 1, 1);
```

Caso de Estudo – Crivo de Erastótenes

Objectivo: Determinar todos os primos até um número máximo designado por **alvo**;

Algoritmo:

1. Determinar todos os primos desde 3 até $\sqrt{alvo}$
   Estes são designados por filtros.

2. Para cada candidato ímpar > $\sqrt{alvo}$
   1. Dividir por cada filtro até o resto da divisão ser 0
   2. Se o resto da divisão nunca é 0, então o candidato é primo

Crivo de Erastótenes
(paralelismo total)

Cada processo divide um candidato por um filtro.

Exemplo: Calcular todos os primos até 400.

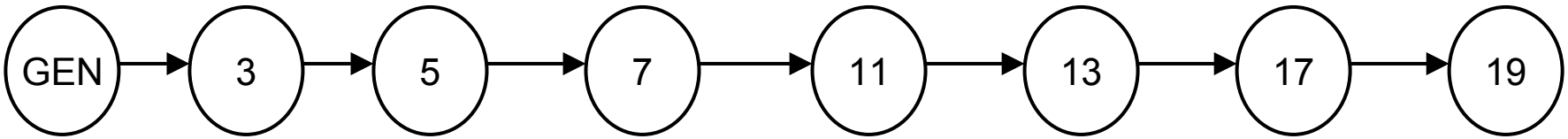
	3	5	7	11	13	17	19	Primo?
21	0	1	0	10	8	4	2	Não
23	2	3	2	1	10	6	4	Sim
25	1	0	4	3	12	8	4	Não
27	0	2	6	5	1	10	8	Não
...	...	...	...	...	...	...	...	...

Inconvenientes: Demasiados processos (nº filtros * nº candidatos);
Realiza-se trabalho desnecessário;

Crivo de Erastótenes
(pipeline)

Cada processo divide o candidato por um filtro.

Só envia o candidato ao próximo filtro se o resto da divisão é diferente de zero.



Inconvenientes: Demasiados processos (nº filtros);

Alvo	Raiz(alvo)	Nº filtros
1.000	32	9
10.000	100	24
100.000	317	64
1.000.000	1.000	167
5.000.000	2.237	330

Crivo de Erastótenes

(pipeline – múltiplos filtros por processo)

Cada processo é responsável por uma sequência de n filtros.

n = nºfiltros / nºprocessos

Só passa o candidato ao próximo processo se o resto da divisão por todos os filtros é diferente de 0.

Crivo de Erastótenes

(pipeline – múltiplos filtros por processo)