Paralelización de Programas Secuenciales: Parallel Virtual Machine (PVM)

http://www.csm.ornl.gov/pvm

Manuel Arenaz

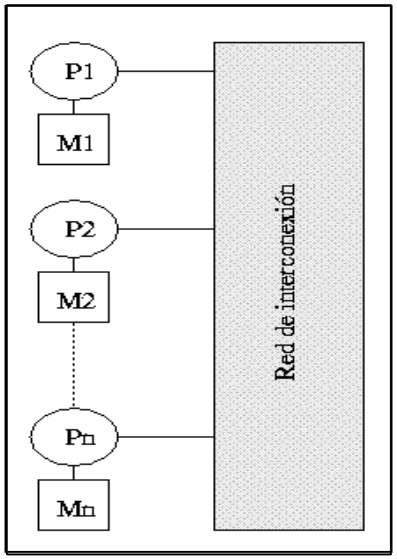
arenaz@udc.es

¿Qué es la Programación Paralela?

- Computador estándar secuencial:
  - Ejecuta las instrucciones de un programa en orden para producir un resultado
- Idea de la computación paralela:
  - Producir el mismo resultado utilizando múltiples procesadores
- Objetivos:
  - Reducir el tiempo de ejecución
  - Reducir los requerimientos de memoria

Tipos de arquitecturas paralelas

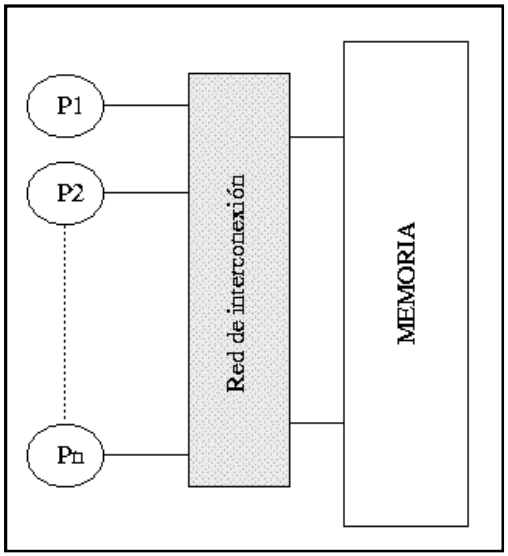
Memoria distribuida



IBM SP2, Clusters PCs, ...

Modelo de Programacion: Paso de mensajes

Memoria compartida

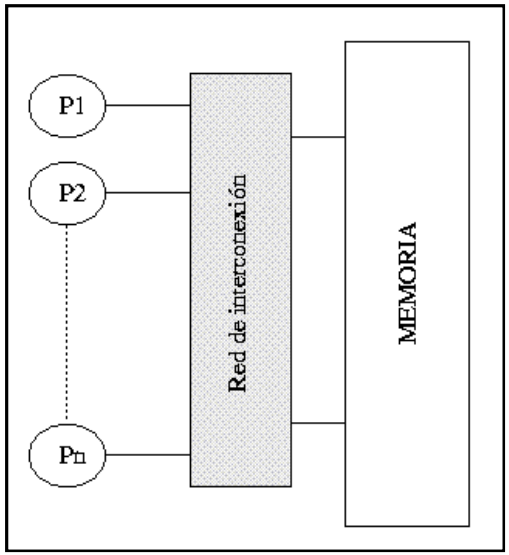


Sun HPC, IBM SMP, PCs multicore, ...

Modelo de Programacion: Memoria Compartida

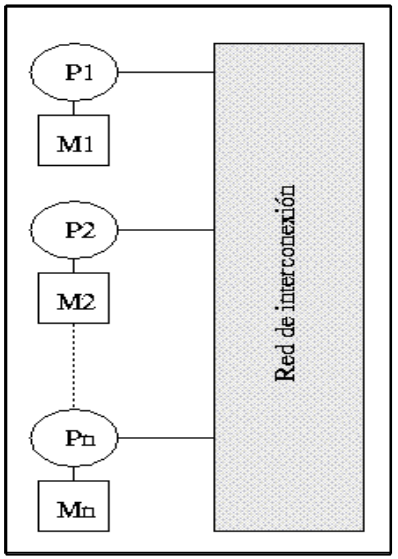
Modelos de Programación

- Memoria compartida:
  - Espacio de direcciones “global” (i.e. accesible por todos los procesadores)
  - Intercambio de información y sincronización mediante variables compartidas
- Ejemplos:
  - OpenMP
  - Threads

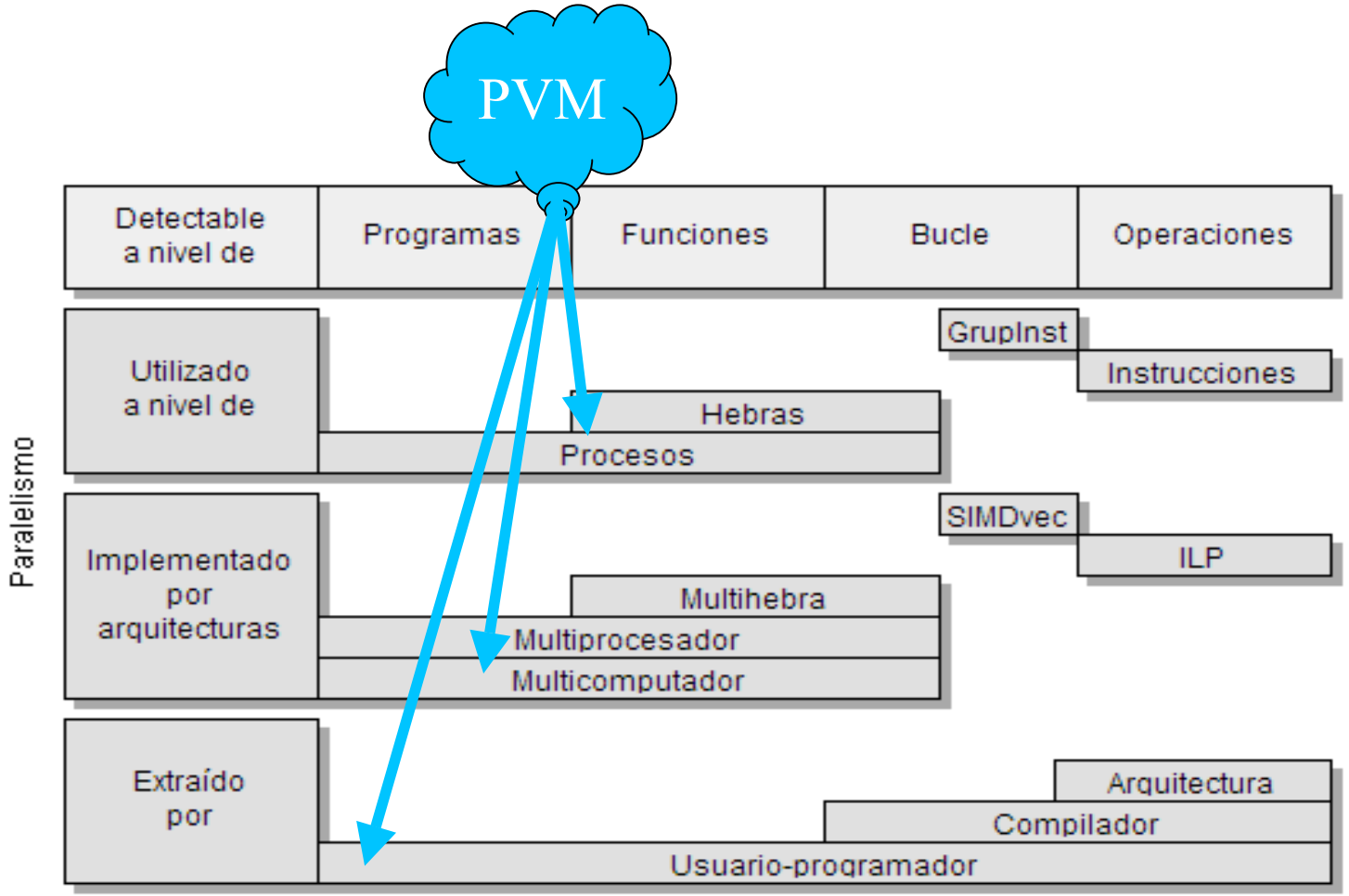


Modelos de Programación

- Paso de mensajes:
  - Espacio de direcciones “local” (i.e. accesible sólo por un procesador)
  - Intercambio de información y sincronización mediante mensajes
- Ejemplos:
  - PVM (Parallel Virtual Machine)
  - MPI (Message-Passing Interface)
  - Unix sockets



Programación Paralela



¿Qué es PVM?

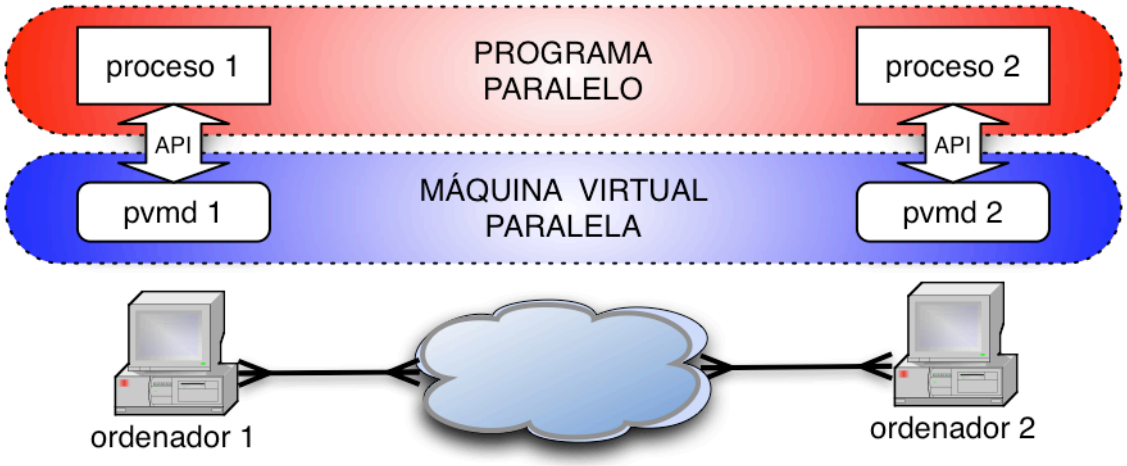
- Entorno de libre distribución (software libre) para el desarrollo y ejecución de programas paralelos utilizando sistemas distribuidos heterogéneos
  - Arquitectura de memoria distribuida
  - Paradigma de programación de paso de mensajes
- Útil en múltiples sistemas paralelos
  - Redes de estaciones de trabajo
  - Clusters de PCs (p.ej., Linux/Beowulf)
  - Supercomputadores de memoria distribuida (p.ej., Cray T3D)

¿Qué es PVM?

- El hardware consta de un conjunto de ordenadores interconectados mediante una red de comunicaciones
- Cada ordenador tiene su propio procesador y su propia memoria

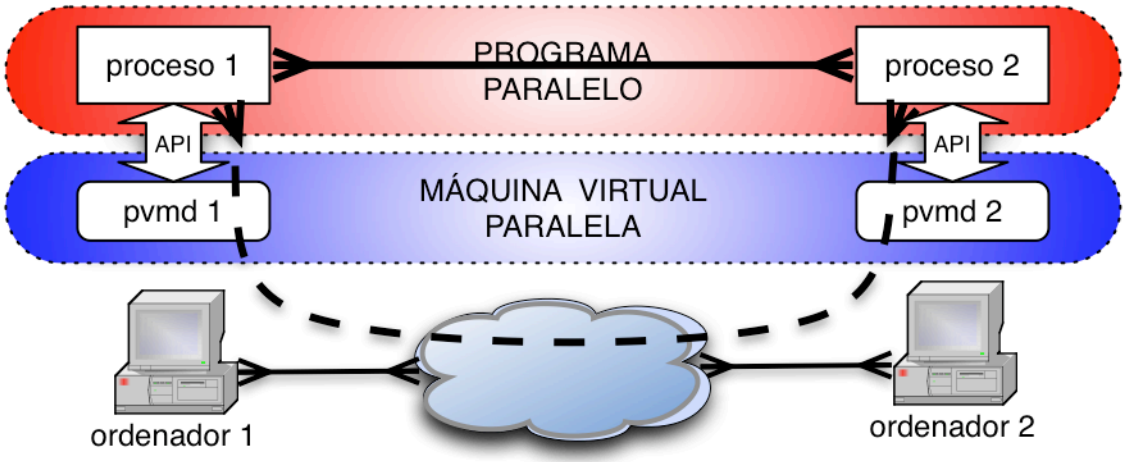
¿Qué es PVM?

- La máquina virtual paralela consta de un conjunto de procesos especiales (*pvmd*) con los que se comunican los procesos del programa paralelo (*proceso*).



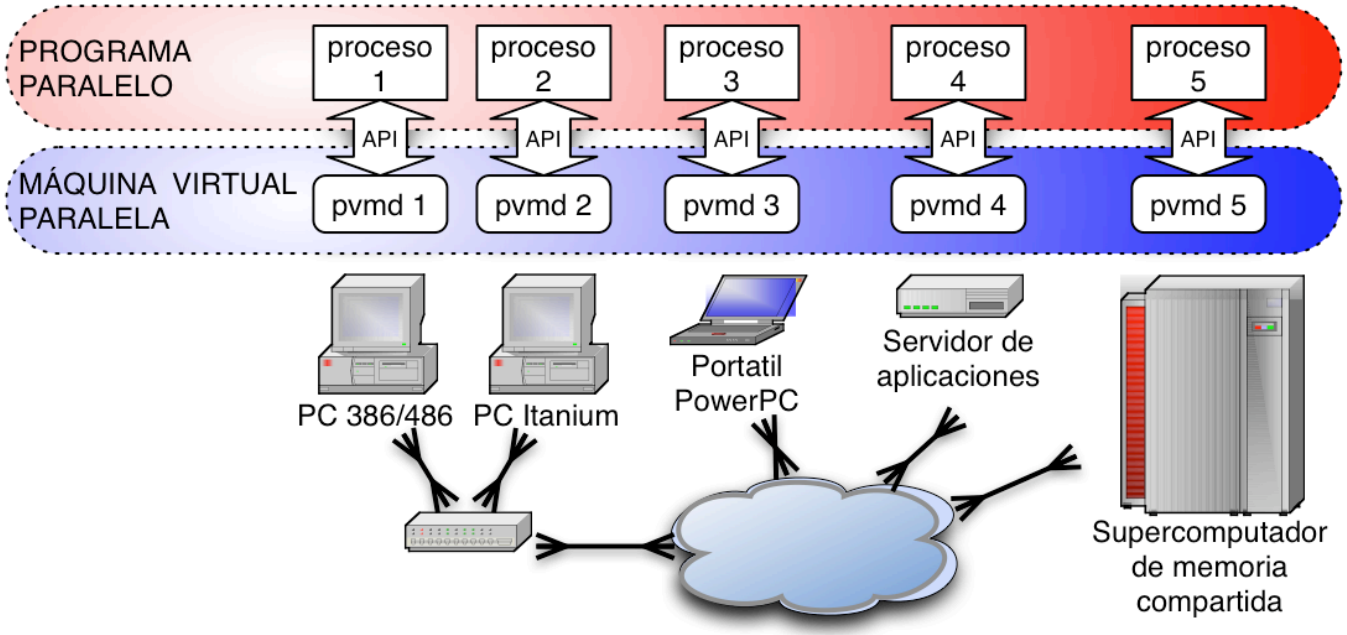
¿Qué es PVM?

- A nivel lógico, los procesos del programa paralelo se comunican entre si mediante el paso de mensajes
- A nivel físico, las comunicaciones de mensajes son gestionadas por los procesos especiales *pvmd*.



¿Qué es PVM?

- PVM resuelve los problemas de heterogeneidad:
  - ➔ Arquitectura, formato de datos, velocidad de procesamiento, carga de trabajo del computador (p.ej., planificación de tareas) y de la red de interconexión (p.ej., enrutamiento de datos)



¿Qué es PVM?

- Ventajas de la computación distribuida usando PVM
  - Reutilización de hardware existente
  - Aumento de los recursos de la máquina virtual paralela bajo demanda
  - Facilitar el desarrollo de programas paralelos para una colección de máquinas heterogéneas
  - Aumentar la productividad mediante el uso de un entorno de desarrollo familiar independiente del sistema paralelo subyacente
  - Herramienta que permite optimizar el rendimiento del programa paralelo asignando cada tarea individual a la arquitectura más apropiada
  - Herramienta para implementar tolerancia a fallos

Modo de trabajo en PVM

- Modelo de programación simple y general
  - API para creación y destrucción de procesos a través de la red de interconexión
  - API para comunicación y sincronización entre los procesos a través de la red
- Programas paralelo PVM
  - Conjunto de procesos que se comunican entre sí mediante paso de mensajes
  - Los procesos pueden acceder en cualquier momento a los recursos PVM a través de funciones del API

Modo de trabajo en PVM

- Paso 1: El desarrollador escribe uno o varios programas secuenciales que contienen llamadas al API de PVM

Ejemplo “Hello world”

```
#include “pvm3.h”
main()
{
  int tid, msgtag, cc;
  char buf[100];

  printf("i'm t%x\n", pvm_mytid());

  cc = pvm_spawn(”slave", (char**)0, 0, "", 1, &tid);
  if (cc == 1) {
    msgtag = 1;
    pvm_recv(tid, msgtag);
    pvm_upkstr(buf);
    printf("from t%x: %s\n", tid, buf);
  } else
    printf("can't start hello_other\n");
  pvm_exit();
}
```

Programa PVM *hello.c*

```
#include "pvm3.h"
main()
{
  int ptid, msgtag;
  char buf[100];

  ptid = pvm_parent();

  strcpy(buf, "hello, world from ");
  gethostname(buf + strlen(buf), 64);
   msgtag = 1;
  pvm_initsend(PvmDataDefault);
  pvm_pkstr(buf);
  pvm_send(ptid, msgtag);

  pvm_exit();
}
```

Programa PVM *slave.c*

Ejemplo “Hello world”

```
#include “pvm3.h”
main()
{
  int tid, msg
  char buf[10

  printf("i'm

  cc = pvm_spawn(”slave", (char**)0, 0, "", 1, &tid);
  if (cc == 1) {
    msgtag = 1;
    pvm_recv(tid, msgtag);
    pvm_upkstr(buf);
    printf("from t%x: %s\n", tid, buf);
  } else
    printf("can't start hello_other\n");
  pvm_exit();
}
```

Programa PVM *hello.c*

```
#include "pvm3.h"
main()
{

  strcpy(buf, "hello, world from ");
  gethostname(buf + strlen(buf), 64);
   msgtag = 1;
  pvm_initsend(PvmDataDefault);
  pvm_pkstr(buf);
  pvm_send(ptid, msgtag);

  pvm_exit();
}
```

Programa PVM *slave.c*

Incluir fichero de cabecera “pvm3.h” que contiene las declaraciones de las funciones del API de PVM.

Ejemplo “Hello world”

```
#include “pvm3.h”
main()
{
  int tid, msgtag, cc;

  cc = pvm_spawn(”slave", (char**)0, 0, "", 1, &tid);
  if (cc == 1) {
    msgtag = 1;
    pvm_recv(tid, msgtag);
    pvm_upkstr(buf);
    printf("from t%x: %s\n", tid, buf);
  } else
    printf("can't start hello_other\n");
  pvm_exit();
}
```

Programa PVM *hello.c*

```
#include "pvm3.h"
main()
{
  int ptid, msgtag;

  strcpy(buf, "hello, world from ");
  gethostname(buf + strlen(buf), 64);
   msgtag = 1;
  pvm_initsend(PvmDataDefault);
  pvm_pkstr(buf);
  pvm_send(ptid, msgtag);

  pvm_exit();
}
```

Programa PVM *slave.c*

Creación e inicialización de los procesos PVM que forman el programa paralelo (e.g., pvm_spawn()).

Ejemplo “Hello world”

```
#include “pvm3.h”
main()
{
  int tid, msgtag, cc;
  char buf[100];

  printf("i'm t%x\n", pvm_mytid() );

  cc = pv
  if (cc ==
    msgta
    pvm_
    pvm_
    printf("from t%x: %s\n", tid, buf);
  } else
    printf("can't start hello_other\n");
  pvm_exit();
}
```

Programa PVM *hello.c*

```
#include "pvm3.h"
main()
{
  int ptid, msgtag;
  char buf[100];

  ptid = pvm_parent();

                                    ;
  pvm_send(ptid, msgtag);

  pvm_exit();
}
```

Programa PVM *slave.c*

Identificación unívoca de los procesos PVM mediante enteros TIDs (Task IDentifiers):
* Proporcionados por PVM (e.g., pvm_mytid())

Ejemplo “Hello world”

```
#include “pvm3.h”
main()
{
  int tid,
  char bu

  printf(

  cc = pvm_spawn(”slave", (char**)0, 0, "", 1, &tid);
  if (cc == 1) {
    msgtag = 1;
    pvm_recv(tid, msgtag);
    pvm_upkstr(buf);
    printf("from t%x: %s\n", tid, buf);
  } else
    printf("can't start hello_other\n");
  pvm_exit();
}
```

Programa PVM *hello.c*

```
#include "pvm3.h"

  strcpy(buf, "hello, world from ");
  gethostname(buf + strlen(buf), 64);
   msgtag = 1;
  pvm_initsend(PvmDataDefault);
  pvm_pkstr(buf);
  pvm_send(ptid, msgtag);

  pvm_exit();
}
```

Programa PVM *slave.c*

Identificación unívoca de los procesos PVM mediante enteros TIDs (Task IDentifiers):
* Proporcionados por PVM (e.g., pvm_mytid())
* Usados en las comunicaciones (e.g., pvm_send())

Ejemplo “Hello world”

```
#include “pvm3.h”
main()
{
  int tid, msgtag, cc;
  char buf[100];

  printf("i'm t%x\n", pvm_mytid());

  cc = pvm_spawn(”slave", (char**)0, 0, "", 1, &tid);
  if (cc == 1) {
    msgtag = 1;
    printf("can't start hello_other\n");
  pvm_exit();
}
```

Programa PVM *hello.c*

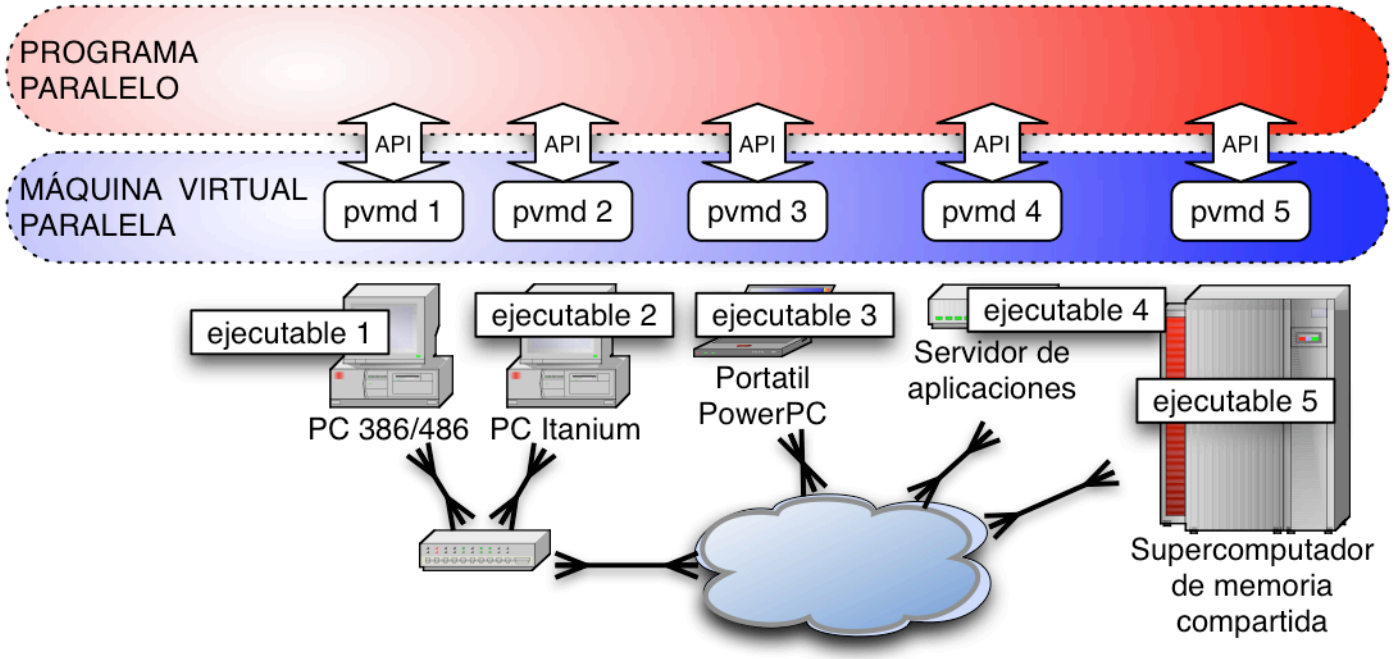
```
#include "pvm3.h"
main()
{
  int ptid, msgtag;
  char buf[100];

  ptid = pvm_parent();

  strcpy(buf, "hello, world from ");
  gethostname(buf + strlen(buf), 64);
   msgtag = 1;
  pvm_exit();
}
```

Programa PVM *slave.c*

Eliminación de los procesos PVM de la máquina virtual (e.g., pvm_exit())

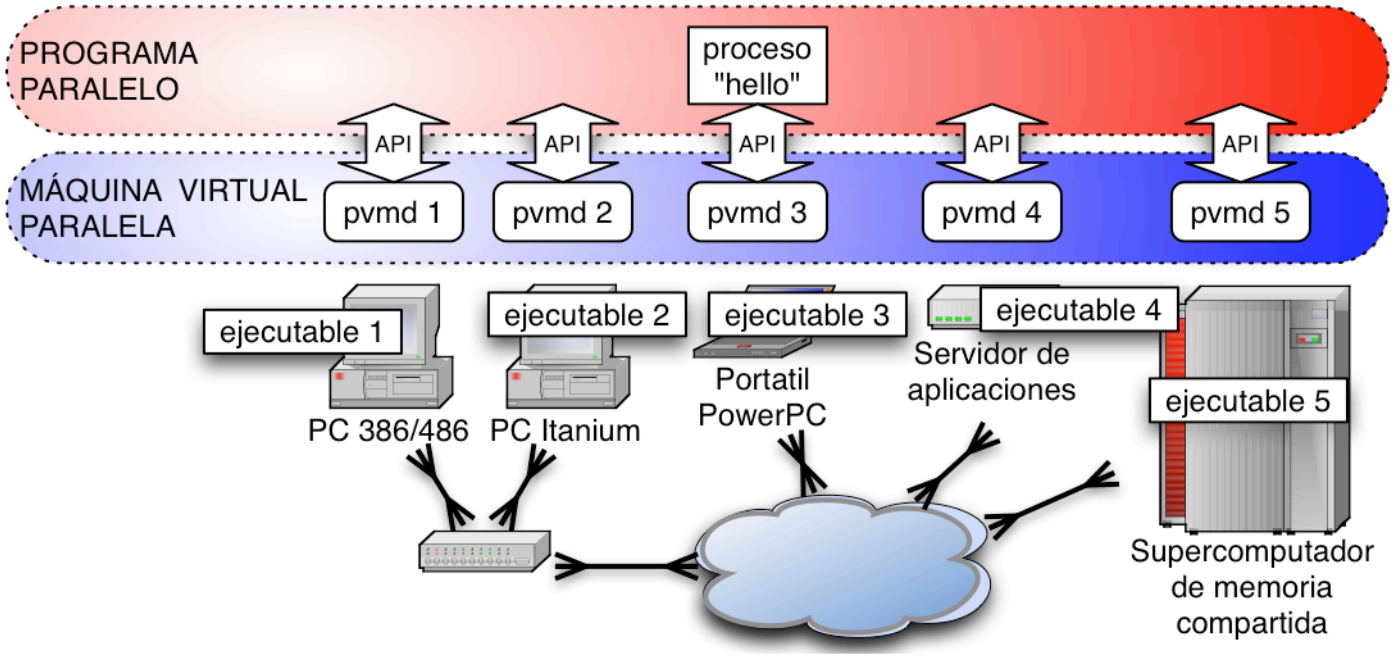
Modo de trabajo en PVM

- Paso 2: Compilación de los programas para cada tipo de arquitectura existente en la máquina virtual
  - ➔ Almacenamiento de los ficheros objeto en las máquinas



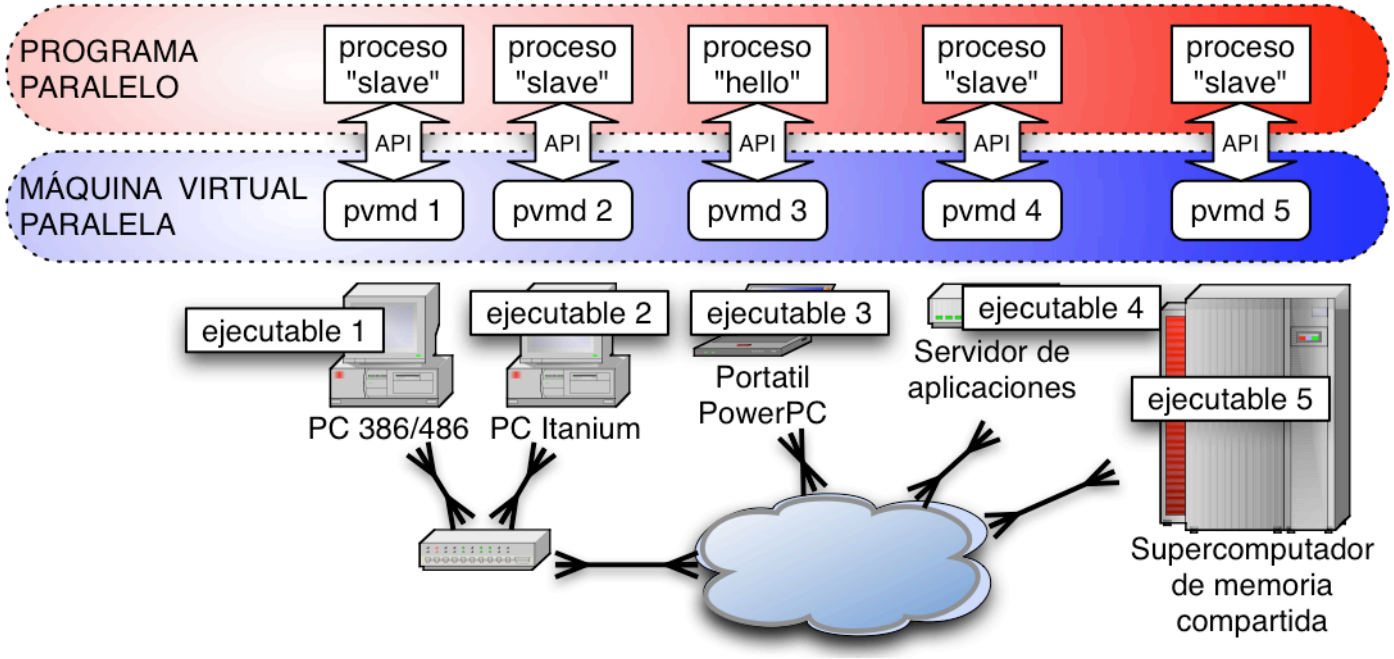
Modo de trabajo en PVM

- Paso 3: Ejecucion del proceso "*maestro*" del programa PVM desde una de las máquinas del máquina virtual



Modo de trabajo en PVM

- Paso 3: Ejecucion del proceso "*maestro*" del programa PVM desde una de las máquinas del máquina virtual
  - ➔ Este proceso "maestro" crea/inicializa otros procesos PVM



Funciones del API de PVM

- **Control de procesos**
- Información
- Paso de mensajes
  - Manipulación de buffers
  - Empaquetamiento de datos
  - Envío
  - Recepción
  - Desempaquetamiento de datos
- Grupos de procesos dinámicos

Control de procesos

int tid = pvm_mytid(void)

- Enrola al proceso llamante en la máquina virtual PVM
- Devuelve el TID que identifica al proceso llamante

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

- Crea e inicia múltiples copias de un fichero ejecutable en la máquina virtual
- Devuelve bien el número de procesos creados bien un código de error

int info = pvm_exit(void)

- Elimina al proceso llamante de la máquina virtual PVM
- El proceso sigue ejecutándose como cualquier otro proceso UNIX
- Se usa típicamente justo antes de la rutina *exit()* del programa C

Ejemplo "Hello world"

```
#include "pvm3.h"
main()
{
  int tid, msgtag, cc;
  char buf[100];

  printf("i'm t%x\n", pvm_mytid() );

  cc = pvm_spawn("hello_other", (char**)0, 0, "", 1, &tid);
  if ( cc == 1 ) {
    msgtag = 1;
    pvm_recv(tid, msgtag);
    pvm_upkstr(buf);
    printf("from t%x: %s\n", tid, buf);
  } else
    printf("can't start hello_other\n");
  pvm_exit();
}
```

Argumentos de pvm_spawn():
"**hello_other**", ruta del fichero ejecutable
"**(char **)0**", sin argumentos (NULL)
"**0**", PVM elige la máquina que ejecutará el proceso (PvmTaskDefault)
"", no se indica máquina para ejecución
"**1**", creación de una copia
"**&tid**", TIDs de los procesos creados

Programa PVM *hello.c*

Funciones del API de PVM

- Control de procesos
- **Información**
- Paso de mensajes
  - Manipulación de buffers
  - Empaquetamiento de datos
  - Envío
  - Recepción
  - Desempaquetamiento de datos
- Grupos de procesos dinámicos

Información

int tid = pvm_parent(void)

- Devuelve bien el TID del proceso que creó al proceso llamante ejecutando pvm_spawn(), bien PvmNoParent si el proceso no fue creado por pvm_spawn()

Información

int tid = pvm_parent(void)

- Devuelve bien el TID del proceso que creó al proceso llamante ejecutando pvm_spawn(), bien PvmNoParent si el proceso no fue creado por pvm_spawn()

El valor devuelto por *pvm_parent()* en la variable ***ptid*** es el TID del proceso *hello* que ejecuto el *pvm_spawn()*.

```
#include "pvm3.h"
main()
{
  int ptid, msgtag;
  char buf[100];

  ptid = pvm_parent();

  strcpy(buf, "hello, world from ");
  gethostname(buf + strlen(buf), 64);
  msgtag = 1;
  pvm_initsend(PvmDataDefault);
  pvm_pkstr(buf);
  pvm_send(ptid, msgtag);

  pvm_exit();
}
```

Programa PVM *slave.c*

Funciones del API de PVM

- Control de procesos
- Información
- **Paso de mensajes**
  - **Manipulación de buffers**
  - **Empaquetamiento de datos**
  - **Envío**
  - **Recepción**
  - **Desempaquetamiento de datos**
- Grupos de procesos dinámicos

Manipulación de buffers

int bufid = pvm_initsend(int encoding)

- En todo momento, cualquier proceso PVM tiene un buffer de envío activo y un buffer de recepción activo
- Limpia el buffer de envío y crea un nuevo buffer para empaquetar un mensaje mediante el método de codificación indicado en *encoding*:

  PvmDataDefault: Codificación XDR porque PVM no puede saber si el usuario va a añadir máquinas heterogéneas antes de enviar el mensaje.

  PvmDataRaw: No se realiza ninguna codificación
  ⇒ menor sobrecarga computacional.

  PvmDataInPlace: No se copian los datos del mensaje en el buffer, sino que éste almacena punteros a los mensajes y sus longitudes.
  ⇒ menor sobrecarga computacional.
  ⇒ no se puede modificar el mensaje entre que se empaqueta y se envía
- Devuelve un identificador *bufid* del buffer creado

Ejemplo “Hello world”

```
#include “pvm3.h”
main()
{
  int tid, msgtag, cc;
  char buf[100];

  printf("i'm t%x\n", pvm_mytid());

  cc = pvm_spawn("hello_other", (char**)0, 0, "", 1, &tid);
  if (cc == 1) {
    msgtag = 1;
    pvm_recv(tid, msgtag);
    pvm_upkstr(buf);
    printf("from t%x: %s\n", tid, buf);
  } else
    printf("can't start hello_other\n");
  pvm_exit();
}
```

Programa PVM *hello.c*

```
#include "pvm3.h"
main()
{
  int ptid, msgtag;
  char buf[100];

  ptid = pvm_parent();

  strcpy(buf, "hello, world from ");
  gethostname(buf + strlen(buf), 64);
  msgtag = 1;
  pvm_initsend(PvmDataDefault);
  pvm_pkstr(buf);
  pvm_send(ptid, msgtag);

  pvm_exit();
}
```

Programa PVM *slave.c*

Empaquetamiento de datos

int info = pvm_pk*type*(*type* *p, int nitem, int stride)

- Empaquetamiento de un array de elementos de tipo de datos ***type*** en el buffer de envío activo.

  "***type *p***" es un puntero al primer elemento que se va a empaquetar

  "***int nitem***" es el número de elementos que se va a empaquetar

  "***int stride***" es la distancia entre dos elementos consecutivos a empaquetar

- Un mensaje puede contener múltiples arrays con diferentes ***type*** (e.g., "byte", "int", "float", "short", "long").

int info = pvm_pkstr(char *s)

- Empaquetar un string acabado en NULL.

Envío de datos

int info = pvm_send(int tid, int mstag)
int info = pvm_mcast(int *tids, int ntask, int mstag)

- Pone la etiqueta *mstag* al mensaje y lo envía al proceso de identificador *tid* (o a un conjunto de *ntask* procesos indicados en el array *tids*)
- Tipo de ***envío asíncrono no bloqueante*** (i.e., el proceso emisor sigue ejecutándose una vez enviado el mensaje)

Ejemplo “Hello world”

```
#include “pvm3.h”
main()
{
  int tid, msgtag, cc;
  char buf[100];

  printf("i'm t%x\n", pvm_mytid());

  cc = pvm_spawn("hello_other", (char**)0, 0, "", 1, &tid);
  if (cc == 1) {
    msgtag = 1;
    pvm_recv(tid, msgtag);
    pvm_upkstr(buf);
    printf("from t%x: %s\n", tid, buf);
  } else
    printf("can't start hello_other\n");
  pvm_exit();
}
```

Programa PVM *hello.c*

```
#include "pvm3.h"
main()
{
  int ptid, msgtag;
  char buf[100];

  ptid = pvm_parent();

  strcpy(buf, "hello, world from ");
  gethostname(buf + strlen(buf), 64);
  msgtag = 1;
  pvm_initsend(PvmDataDefault);
  pvm_pkstr(buf);
  pvm_send(ptid, msgtag);

  pvm_exit();
}
```

Programa PVM *slave.c*

Recepción

int bufid = pvm_recv(int tid, int mstag)

- Recepción del mensaje con etiqueta *mstag* procedente del proceso de identificador *tid*

  "mstag=-1": mensaje con cualquier etiqueta

  "tid=-1": mensaje procedente de cualquier proceso

- Tipo de ***recepción asíncrona bloqueante*** (i.e., el proceso receptor detiene su ejecución hasta que se ha recibido el mensaje)

Desempaquetamiento

int info = pvm_upk*type*(*type* *p, int nitem, int stride)

- Desempaquetamiento de un array de elementos de tipo de datos ***type*** desde el buffer de recepción activo.

  "***type *p***" es un puntero al primer elemento que se va a empaquetar

  "***int nitem***" es el número de elementos que se va a empaquetar

  "***int stride***" es la distancia entre dos elementos consecutivos a empaquetar
- Un mensaje puede contener múltiples arrays con diferentes ***type*** (e.g., "byte", "int", "float", "short", "long").
- Desempaquetamiento en el mismo orden en que se empaquetaron!!

int info = pvm_upkstr(char *s)

- Desempaquetar un string acabado en NULL.

Ejemplo “Hello world”

```
#include “pvm3.h”
main()
{
  int tid, msgtag, cc;
  char buf[100];

  printf("i'm t%x\n", pvm_mytid());

  cc = pvm_spawn("hello_other", (char**)0, 0, "", 1, &tid);
  if (cc == 1) {
    msgtag = 1;
    pvm_recv(tid, msgtag);
    pvm_upkstr(buf);
    printf("from t%x: %s\n", tid, buf);
  } else
    printf("can't start hello_other\n");
  pvm_exit();
}
```

Programa PVM *hello.c*

```
#include "pvm3.h"
main()
{
  int ptid, msgtag;
  char buf[100];

  ptid = pvm_parent();

  strcpy(buf, "hello, world from ");
  gethostname(buf + strlen(buf), 64);
  msgtag = 1;
  pvm_initsend(PvmDataDefault);
  pvm_pkstr(buf);
  pvm_send(ptid, msgtag);

  pvm_exit();
}
```

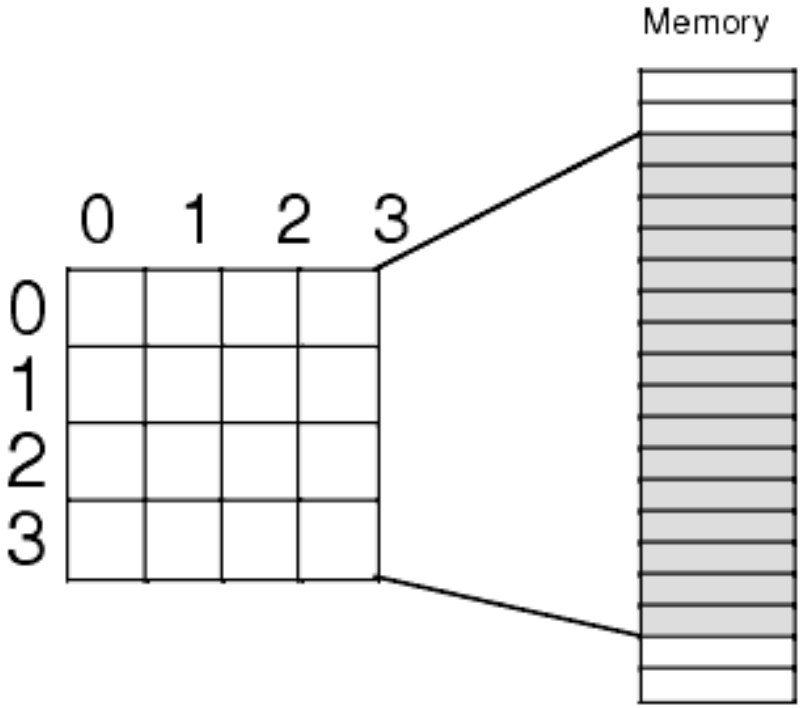
Programa PVM *slave.c*

(Des)Empaquetamiento

- Problema:

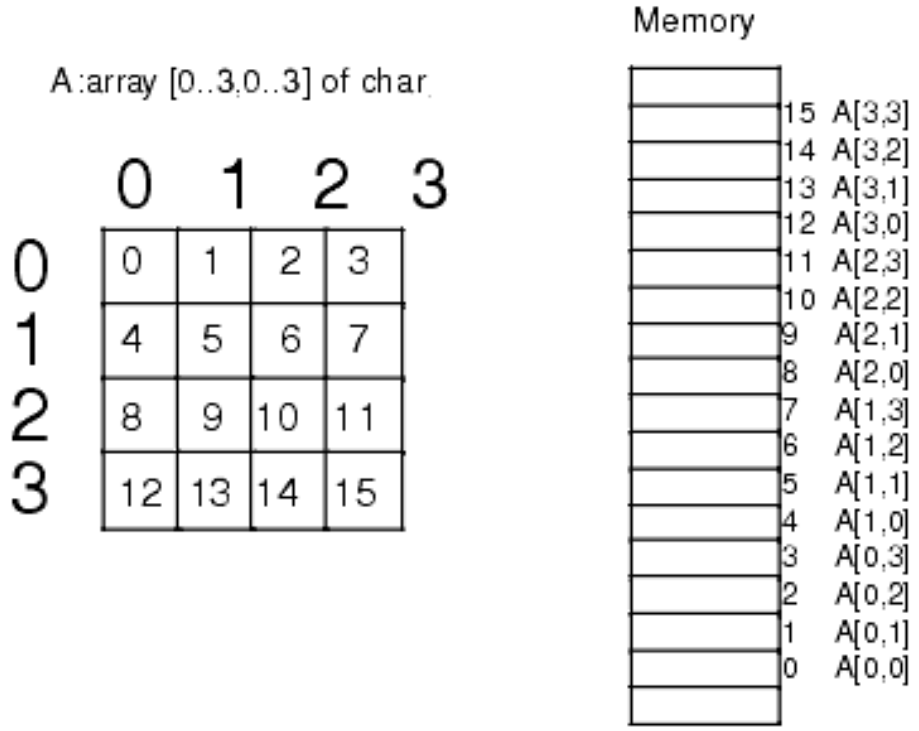
  Manejo de arrays multidimensionales
- Solución:

  Política de almacenamiento del lenguaje de programación
  - "*Row-major*" (e.g., C/C++, Java)
  - "*Coloumn-major*" (e.g., Fortran)



(Des)Empaquetamiento

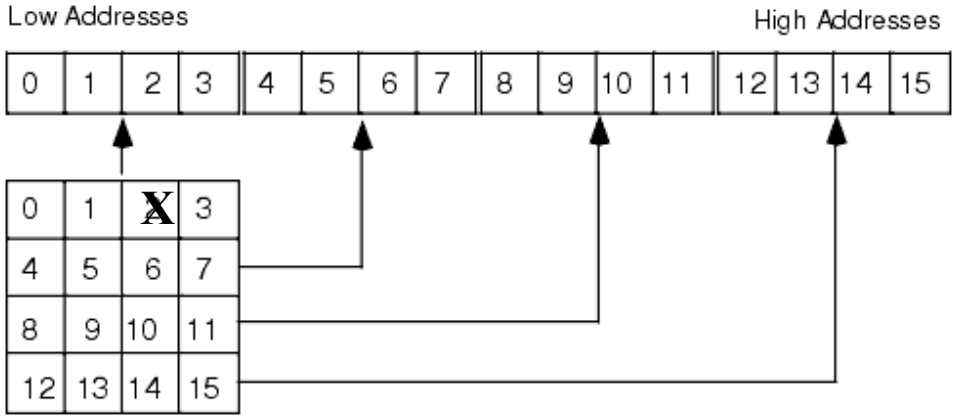
- "*Row-major*" (e.g., C/C++, Java)
  - Los elememtos de cada fila del array multidimensional se almacenan en posiciones de memoria consecutivas.



(Des)Empaquetamiento

- "*Row-major*" (e.g., C)

Element_Address = Base_Address + row_index * row_size + col_index



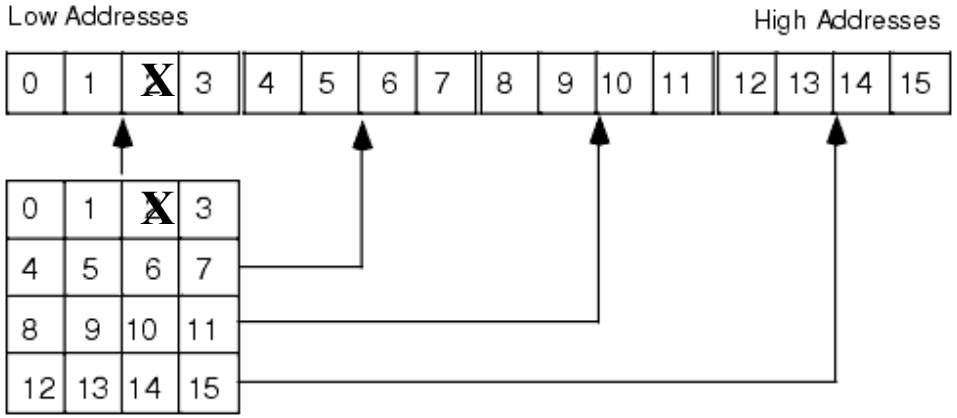
¿Posición de A[0][2]?

int A[4][4];

(Des)Empaquetamiento

- "*Row-major*" (e.g., C)

Element_Address = Base_Address + row_index * row_size + col_index



int A[4][4];

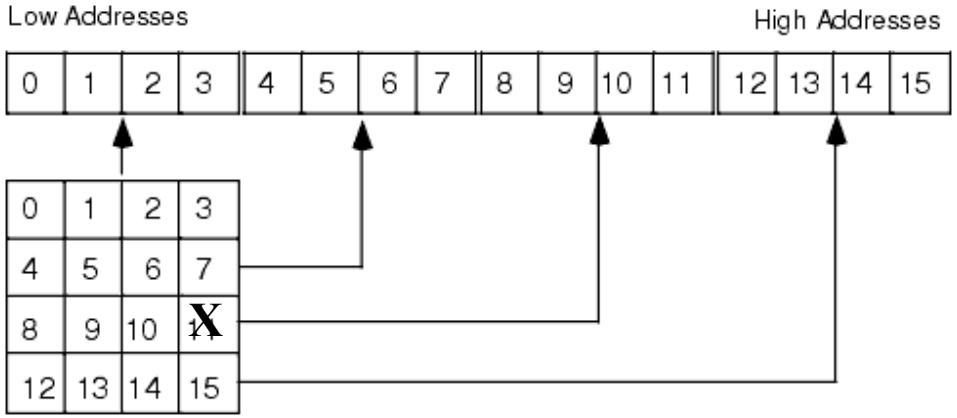
¿Posición de A[0][2]?

A+0*4 +2 = A+2

(Des)Empaquetamiento

- "*Row-major*" (e.g., C)

Element_Address = Base_Address + row_index * row_size + col_index



int A[4][4];

¿Posición de A[0,2]?

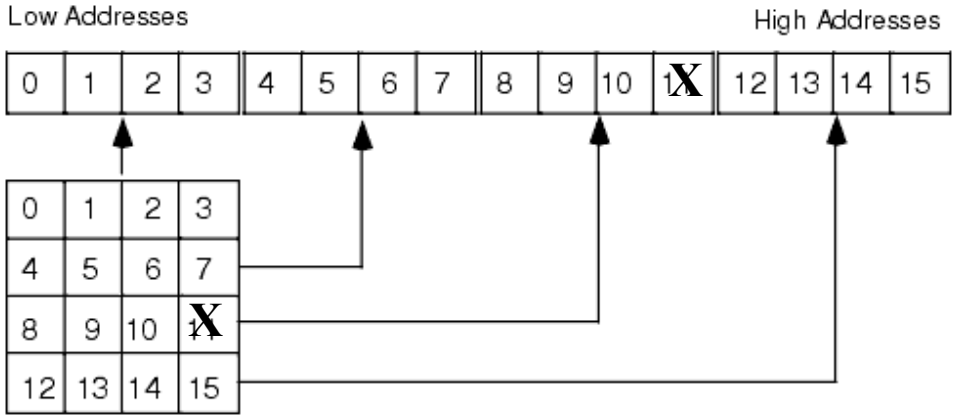
A+0*4 +2 = A+2

¿Posición de A[2][3]?

(Des)Empaquetamiento

- "*Row-major*" (e.g., C)

Element_Address = Base_Address + row_index * row_size + col_index



int A[4][4];

¿Posición de A[0,2]?

A+0*4 +2 = A+2

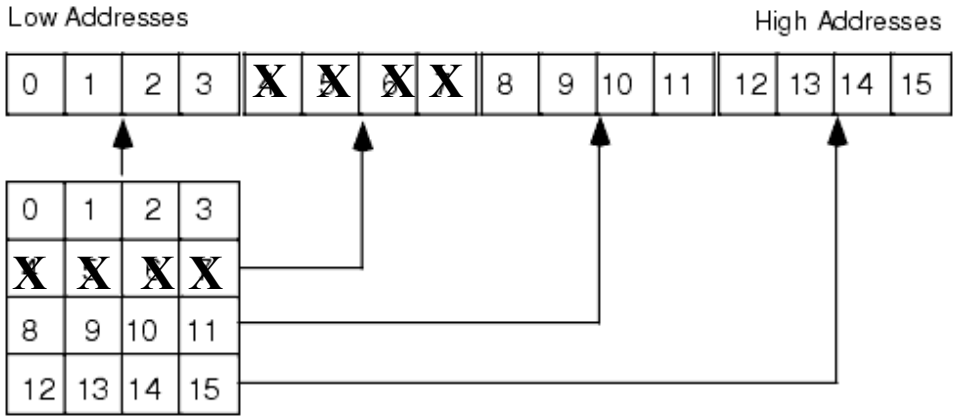
¿Posición de A[2][3]?

A+2*4+3 = A+11

(Des)Empaquetamiento

- Ejemplo: Empaquetamiento de una fila en C

int A[4][4];



Bucle de 4 iteraciones y
empaquetamiento con cnt=1 y stride=1

```
for(i=0;i<4;i++) {
   pvm_pkint(A[1][i],1,1);
}
```

X

Empaquetamiento con
cnt=4 y stride=1

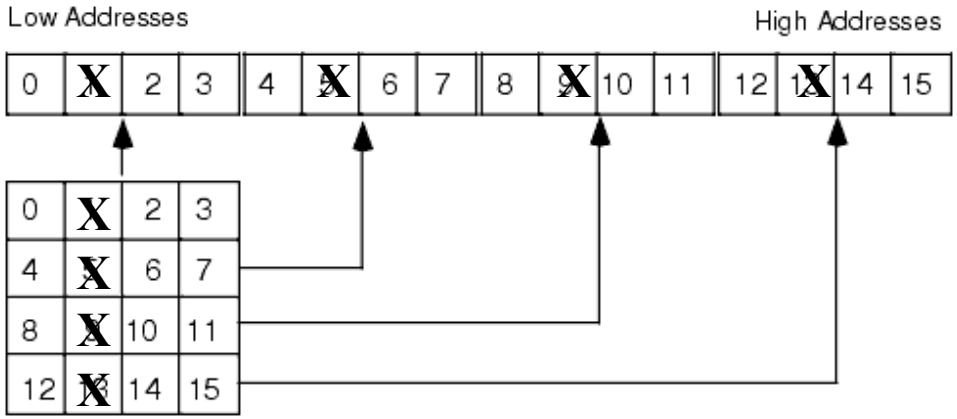
```
pvm_pkint(&A[1][0],4,1);
```

√

(Des)Empaquetamiento

- Ejemplo: Empaquetamiento de una columna en C

int A[4][4];



Bucle de 4 iteraciones y
empaquetamiento con cnt=1 y stride=1

```
for(i=0;i<4;i++) {
   pvm_pkint(A[i][1],1,1);
}
```

X

Empaquetamiento con
cnt=4 y stride=4

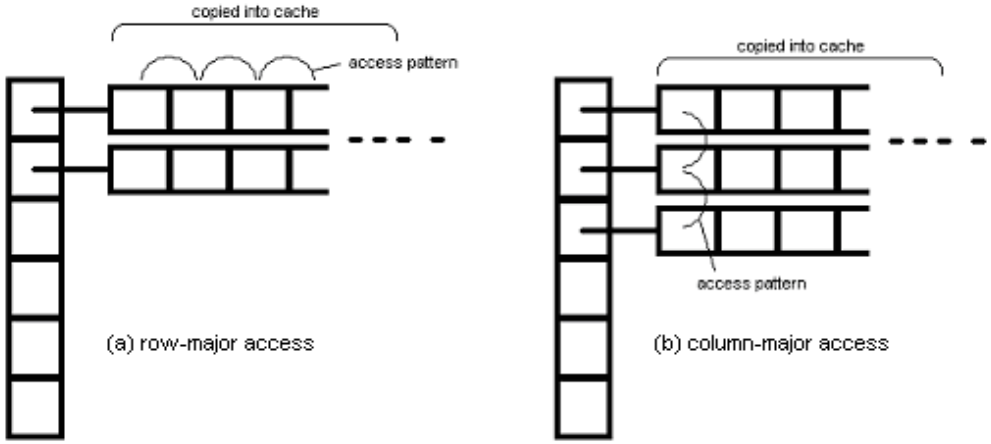
```
pvm_pkint(&A[0][1],4,4);
```

√

(Des)Empaquetamiento

- Ejemplo: Empaquetamiento de una fila/columna en C

int **A;



Empaquetamiento de una fila

```
for(i=0;i<4;i++) {
   pvm_pkint(A[0][i],1,1);
}
```

X

```
pvm_pkint(&A[0][0],4,1);
```

√

Empaquetamiento de una columna

```
for(i=0;i<4;i++) {
   pvm_pkint(A[i][0],1,1);
}
```

√

Funciones del API de PVM

- Control de procesos
- Información
- Paso de mensajes
  - Manipulación de buffers
  - Empaquetamiento de datos
  - Envío
  - Recepción
  - Desempaquetamiento de datos
- **Grupos de procesos dinámicos**

Grupos de procesos dinámicos

int inum = pvm_joingroup(char *group)

- Crea un grupo de nombre *group* y añade al proceso llamante.
- Devuelve un identificador del proceso *inum* dentro del grupo, de valor entre 0 y el nº de miembros del grupo menos 1.

int info = pvm_lvgroup(char *group)

- Elimina el proceso llamante del grupo *group*.

int info = pvm_barrier(char *group, int count)

- El proceso llamante se bloque hasta que *count* procesos miembros del grupo invoquen la rutina *pvm_barrier()*.
- Es necesario indicar el número de procesos *count* debido a la naturaleza totalmente dinámica de los grupos de procesos de PVM.

Grupos de procesos dinámicos

int info = pvm_bcast(char *group, int mstag)

- Pone la etiqueta *mstag* a un mensaje y lo envía a todos los procesos del grupo (excepto a si mismo).

int info = pvm_reduce(void (*func)(), void *data,
int nitem, int datatype,
int mstag, char *group,
int root)

- Realiza una operación aritmética entre todos los miembros de un grupo, devolviendo el resultado en el proceso con identificador *root*.
- Funciones predefinidas: *PvmMax*, *PvmMin*, *PvmSum*, *PvmProduct*.
- Nota: pvm_reduce() es no bloqueante, por lo que si un proceso llama a pvm_reduce() y luego a pvm_lvgroup(), se puede producir un error.

Otras funciones del API de PVM

- Control de procesos
- Información
- Paso de mensajes
  - Manipulación de buffers
  - Empaquetamiento de datos
  - Envío
  - Recepción
  - Desempaquetamiento de datos
- Grupos de procesos dinámicos

Control de procesos

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

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

- Añade/Elimina los *nhost* computadores cuyos nombres se indican en el array *hosts*.

  ⇒ Configuración dinámica de la máquina virtual (e.g., aumentar flexibilidad y/o tolerancia a fallos)
- Devuelve el número de computadores añadidos/eliminados (*info*), y el código de estado (e.g., en funcionamiento, perteneciente a la máquina virtual) de cada computador añadido/eliminado (*infos*).

Información

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

- Devuelve información de configuración de la máquina virtual: número de computadores (nhost) y formatos de datos (narch), etc.

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

- Devuelve información sobre una tarea PVM (*which*): TID de la tarea, TID del padre, fichero ejecutable asociado, etc.

pvm_pstat(int tid)

- Estado del poceso de identificador *tid* (e.g., en ejecución, finalizada)

pvm_mstat(char *host)

- Estado del computador de nombre *host*

pvm_perror(char *msg)

- Imprime el estado de error de la última llamada a una función PVM

Manipulación de buffers

int bufid = pvm_mkbuf(int encoding)

- Creación de un nuevo buffer vacío, indicando el método de codificación.
  ⇒ Implementación de aplicaciones paralelas con múltiples buffers

int info = pvm_freebuf(int bufid)

- Elimina el buffer de identifcador *bufid*.
- La función pvm_initsend() realiza la función de pvm_mkbuf() y pvm_freebuf(), haciendo innecesario el uso de dichas funciones.

int bufid = pvm_getsbuf(void)
int bufid = pvm_getrbuf(void)

- Obtener el identificador del buffer de envío/recepción activo.

int oldbufid = pvm_setsbuf(int bufid)
int oldbufid = pvm_setrbuf(int bufid)

- Establecer el buffer *bufid* como buffer de envío/recepción activo.

Manipulación de mensajes

int bufid = pvm_nrecv(int tid, int mstag)

- Recepción asíncrona no bloqueante (i.e., no se asegura la recepción del mensaje).

int bufid = pvm_probe(int tid, int mstag)

- Comprueba si ha llegado un mensaje, pero no lo recibe. La recepción se realiza mediante una llamada posterior a pvm_recv() o pvm_nrecv().

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

- Devuelve inforamación sobre el mensaje contenido en el buffer *bufid*: número de bytes (*bytes*), etiqueta (*mstag*) y proceso de procedencia (*tid*).

Manipulación de mensajes

**int info = pvm_psend(int tid, int mstag, void *vp,
int cnt, int type)**

**int info = pvm_precv(int tid, int mstag, void *vp,
int cnt, int type, int *rtid,
int *rtag, int *rcnt)**

- Procedimientos para realizar en un único paso el empaquetamiento-envío y la recepción-desempaquetamiento.
- Sólo se pueden usar con datos homogéneos y sin *stride*.

Grupos de procesos dinámicos

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

- Obtener el TID de un miembro de un grupo dinámico.

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

- Obtener el identificador como miembro de un grupo dinámico del proceso cuyo TID es *tid*.

int size = pvm_gsize(char *group)

- Devuelve el número de miembros de un gruo dinámico.