PVM MPI 12

Sistemas Operativos y Distribuidos

Mg. Javier Echaiz
D.C.I.C. – U.N.S.
http://cs.uns.edu.ar/~jechaiz
je@cs.uns.edu.ar

Outline

- Motivation
- Modern scientific method
- Evolution of supercomputing
- Modern parallel computers
- Seeking concurrency
- Programming parallel computers
- **PVM**

What is Parallel and Distributed computing?

- Solving a single problem faster using multiple CPUs
- Parallel = Shared Memory among all CPUs
- Distributed = Local Memory/CPU
- Common Issues: Partition, Synchronization, Dependencies

Why Parallel and Distributed Computing?

- Grand Challenge Problems
  - Weather Forecasting; Global Warming
  - Materials Design – Superconducting material at room temperature; nano-devices; spaceships.
  - Organ Modeling; Drug Discovery
- Physical Limitations of Circuits
  - heat and light effect
  - Superconducting material to counter heat effect
  - Speed of light effect – no solution!

Why Parallel and Distributed Computing?

- VLSI – Effect of Integration
  - 1 M transistor enough for full functionality
  - Rest must go into multiple CPUs/chip
- Cost – Multitudes of average CPUs give better FLPOS/$ compared to traditional supercomputers
- Idling workstations should be utilized
- Everyday Reasons
  - Solve compute-intensive problems faster
    - Make infeasible problems feasible
    - Reduce design time
  - Solve larger problems in same amount of time
    - Improve answer's precision
    - Reduce design time
  - Gain competitive advantage

Definitions

- Parallel computer
  - Multiple-processor system supporting parallel programming
- Parallel programming
  - Programming in a language that supports concurrency explicitly

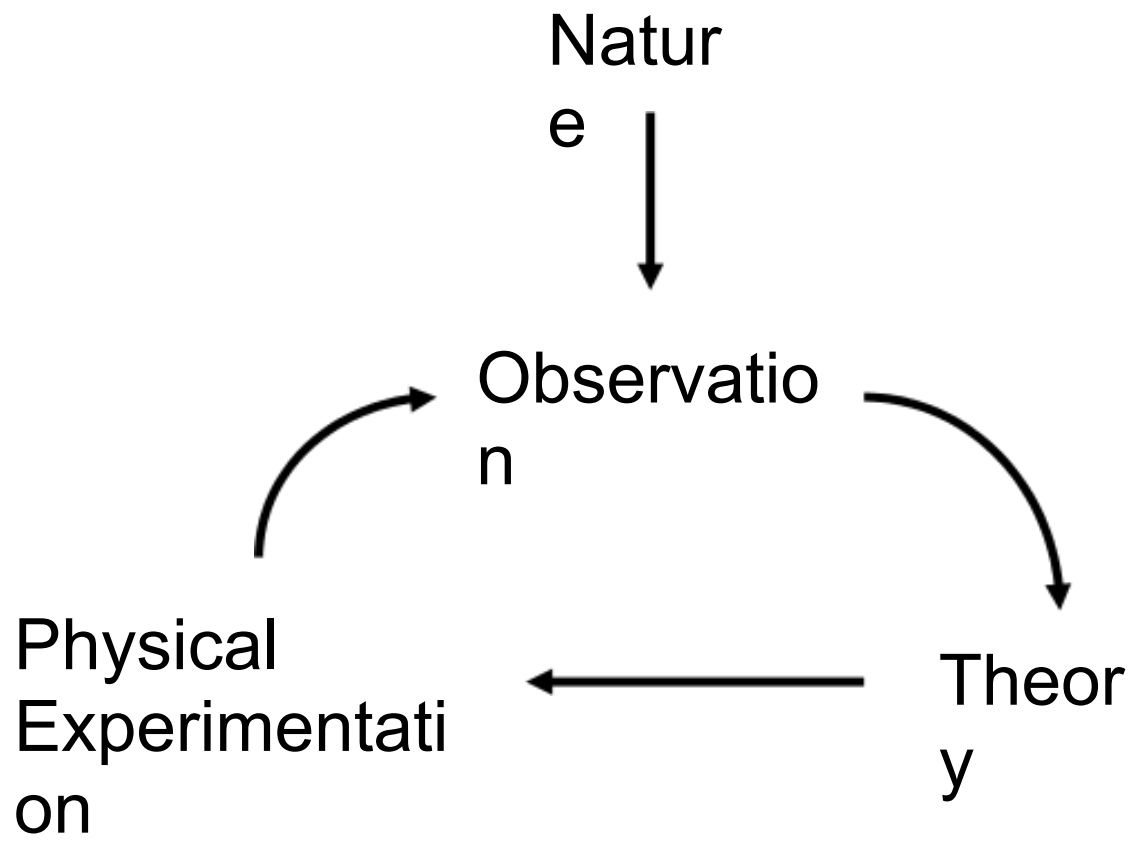
Why MPI and PVM?

- MPI = “Message Passing Interface”
- PVM = “Parallel Virtual Machine”
- Standard specification for message-passing libraries
- Libraries available on virtually all parallel computers
- Free libraries also available for networks of workstations or commodity clusters

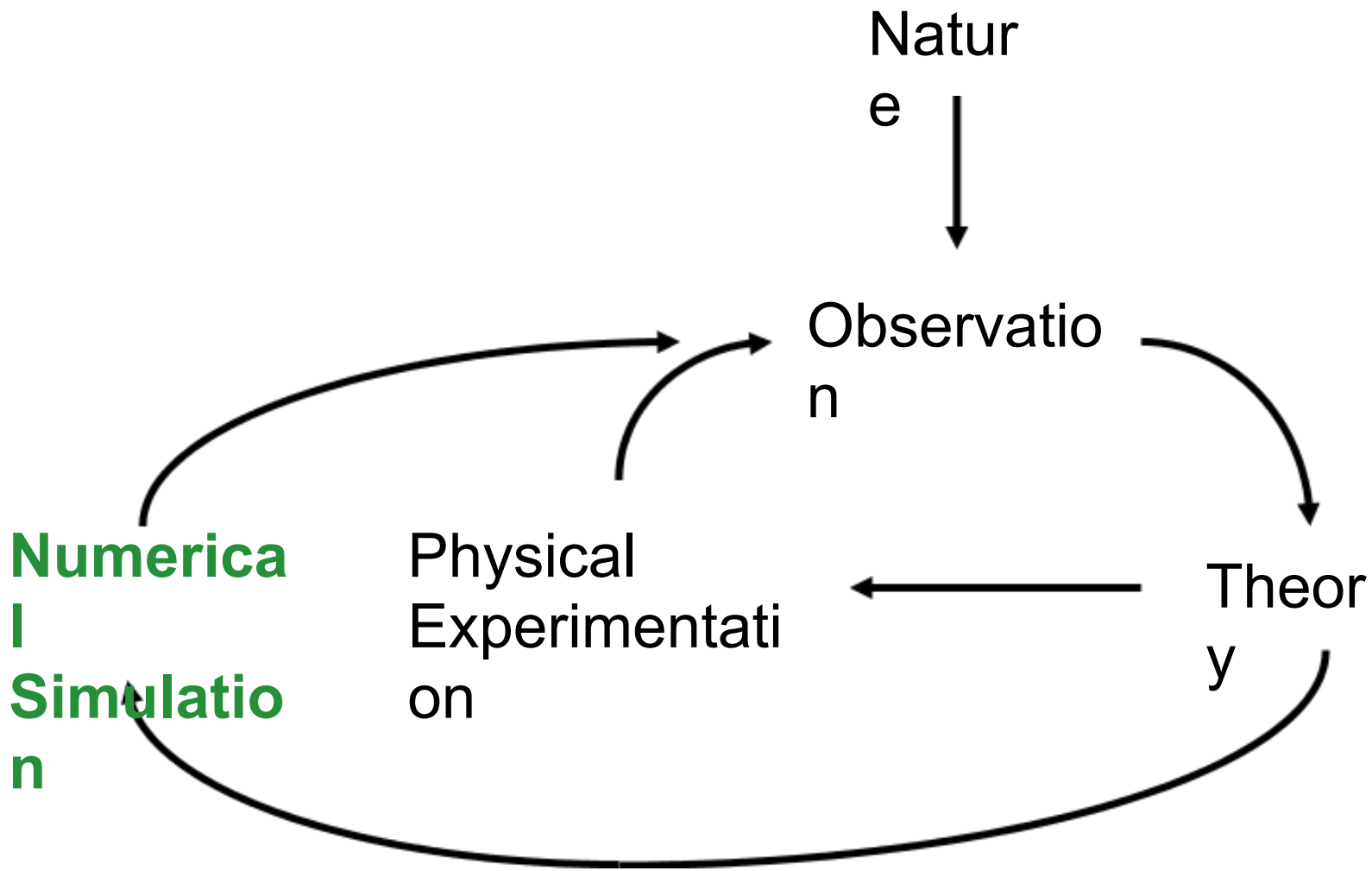
Why Shared Memory programming?

- Easier conceptual environment
- Programmers typically familiar with concurrent **threads** and **processes** sharing address space
- OpenMP an application programming interface (API) for shared-memory systems
  - Supports higher performance parallel programming of symmetrical multiprocessors

Classical Science



Modern Scientific Method



Evolution of Supercomputing

- World War II
  - Hand-computed artillery tables
  - Need to speed computations
  - ENIAC
- Cold War
  - Nuclear weapon design
  - Intelligence gathering
  - Code-breaking

Advanced Strategic Computing Initiative

- U.S. nuclear policy changes
  - Moratorium on testing
  - Production of new weapons halted
- Numerical simulations needed to maintain existing stockpile
- Five supercomputers costing up to $100 million each

ASCI White (10 teraops/sec)

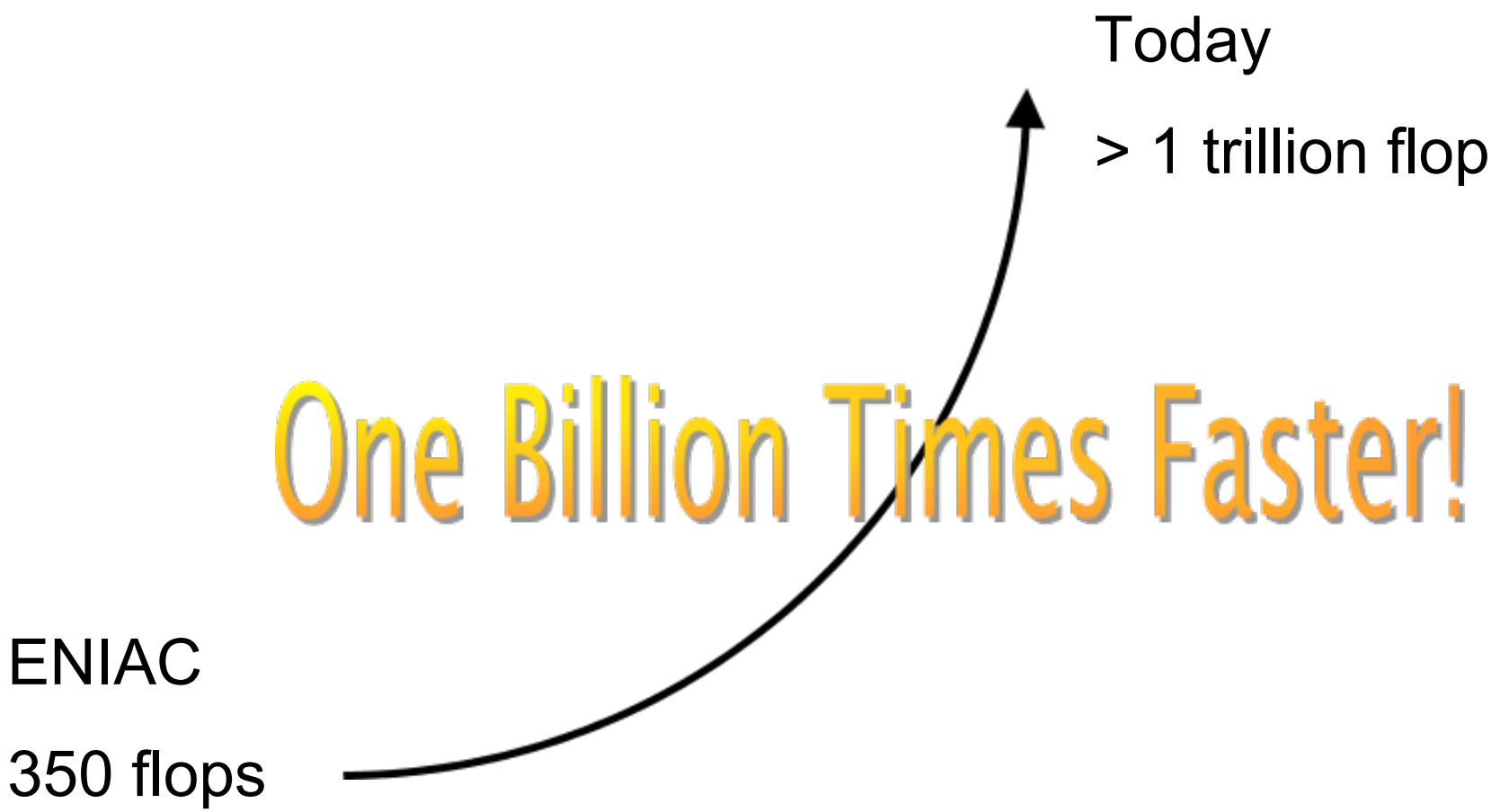
Supercomputer

- Fastest General-purpose computer
- Solves individual problems at high speeds, compared with contemporary systems
- Typically costs $10 million or more
- Traditionally found in government labs

Commercial Supercomputing

- Started in capital-intensive industries
  - Petroleum exploration
  - Automobile manufacturing
- Other companies followed suit
  - Pharmaceutical design
  - Consumer products
  - Financial Transactions

50 Years of Speed Increases



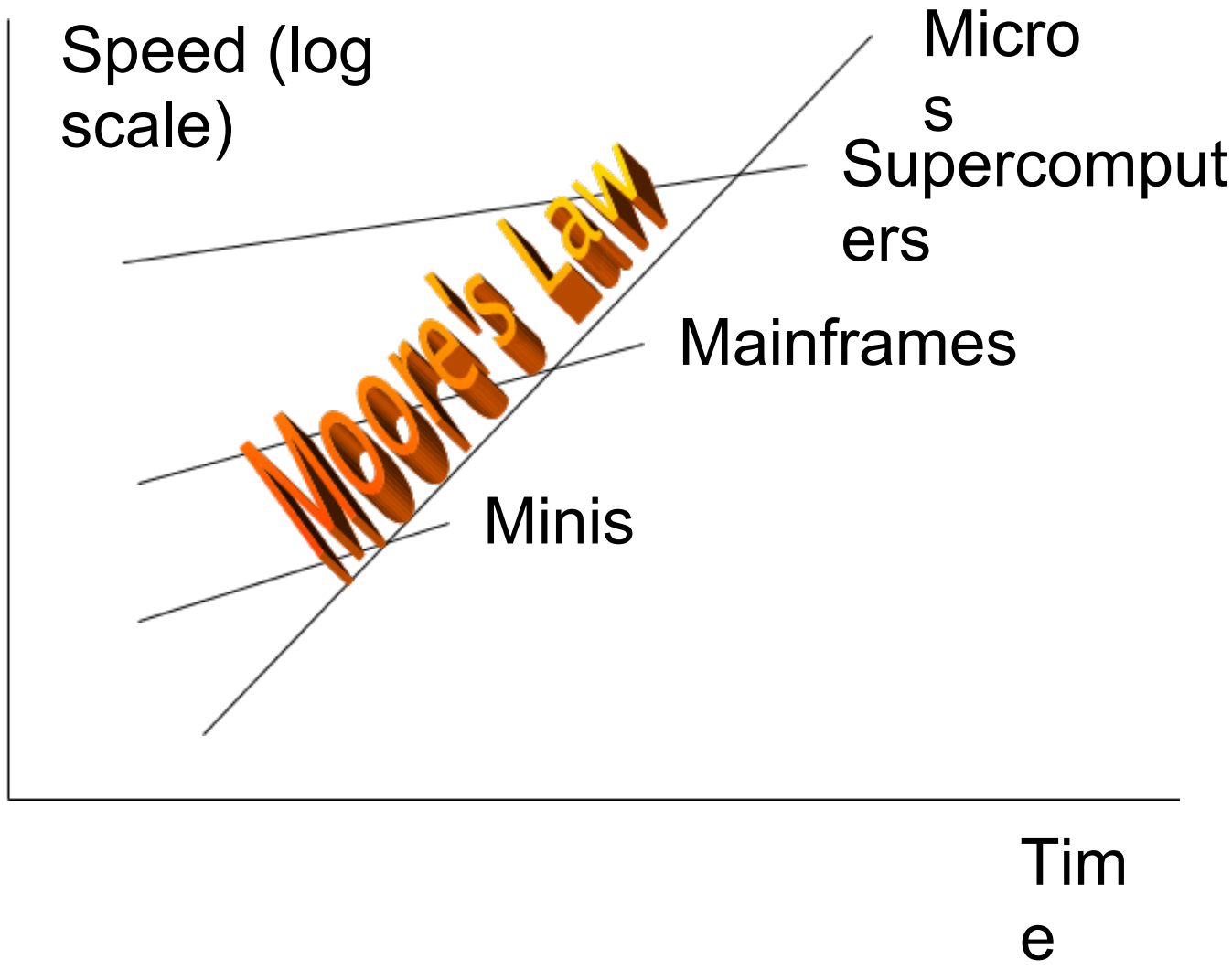
CPUs 1 Million Times Faster

- Faster clock speeds
- Greater system concurrency
  - Multiple functional units
  - Concurrent instruction execution
  - Speculative instruction execution

Systems 1 Billion Times Faster

- Processors are 1 million times faster
- Combine thousands of processors
- Parallel computer
  - Multiple processors
  - Supports parallel programming
- Parallel computing = Using a parallel computer to execute a program faster

Microprocessor Revolution



Modern Parallel Computers

- Caltech's Cosmic Cube (Seitz and Fox)
- Commercial copy-cats
  - nCUBE Corporation (512 CPUs)
  - Intel's Supercomputer Systems
    - iPSC1, iPSC2, Intel Paragon (512 CPUs)
  - Lots more
- Thinking Machines Corporation
  - CM2 (65K 4-bit CPUs) – 12-dimensional hypercube - SIMD
  - CM5 – fat-tree interconnect - MIMD

Copy-cat Strategy

- Microprocessor
  - 1% speed of supercomputer
  - 0.1% cost of supercomputer
- Parallel computer = 1000 microprocessors
  - 10 x speed of traditional supercomputer
  - Same cost as supercomputer

Why Didn't Everybody Buy One?

- Supercomputer □ □ CPUs
  - Computation rate □ throughput (#jobs/time)
  - Slow Interconnect
  - Inadequate I/O
  - Customized Compute and Communication processors meant inertia in adopting the fastest commodity chip with least cost and effort
  - Focus on high end computation meant no sales volume to reduce cost
- Software
  - Inadequate operating systems
  - Inadequate programming environments

After the “Shake Out”

- IBM – SP1 and SP2
- Hewlett-Packard – Tandem
- Silicon Graphics – Origin
- Sun Microsystems - Starfire

Commercial Parallel Systems

- Relatively costly per processor
- Primitive programming environments
- Focus on commercial sales
- Scientists looked for alternative

Beowulf Concept

SD2

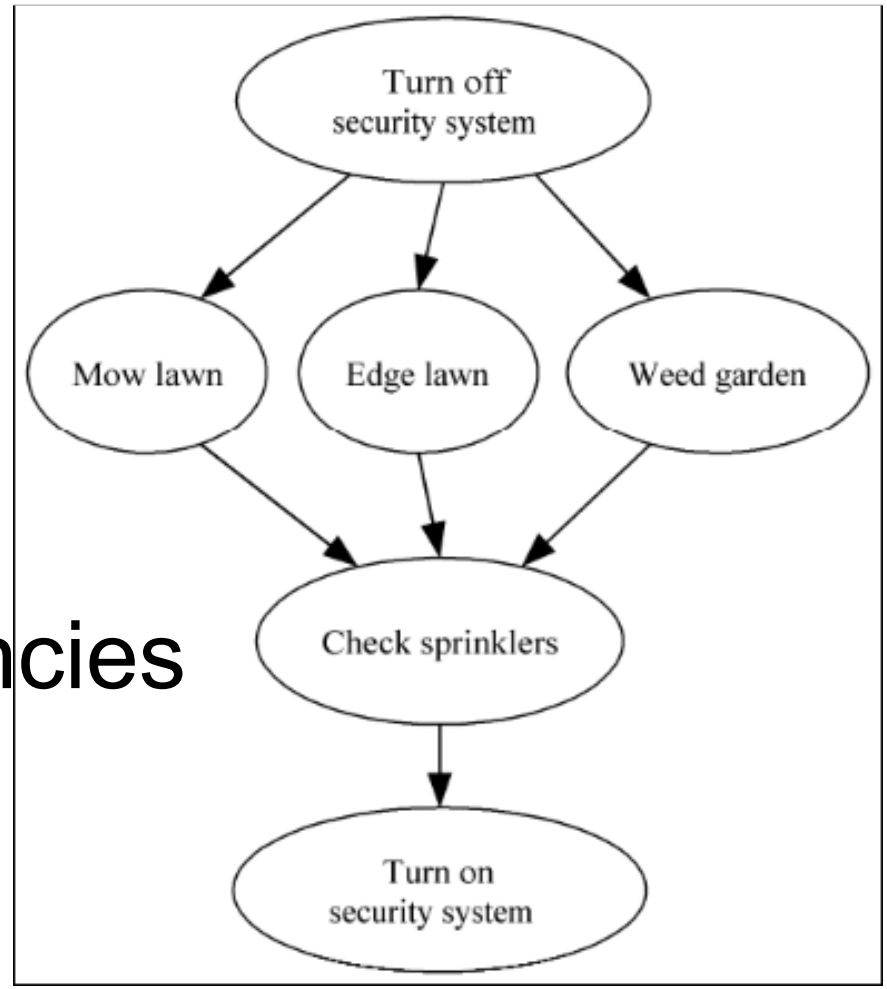
- NASA (Sterling and Becker)
- Commodity processors
- Commodity interconnect
- Linux operating system
- MPI/PVM library
- High performance/$ for *certain* applications

Seeking Concurrency

- Data dependence graphs
- Data parallelism
- Functional parallelism
- Pipelining

Data Dependence Graph

- Directed graph
- Vertices = tasks
- Edges = dependencies



Data Parallelism

- Independent tasks apply same operation to different elements of a data set

```
for i □ 0 to 99 do
a[i] □ b[i] + c[i]
endfor
```

- Okay to perform operations concurrently

Functional Parallelism

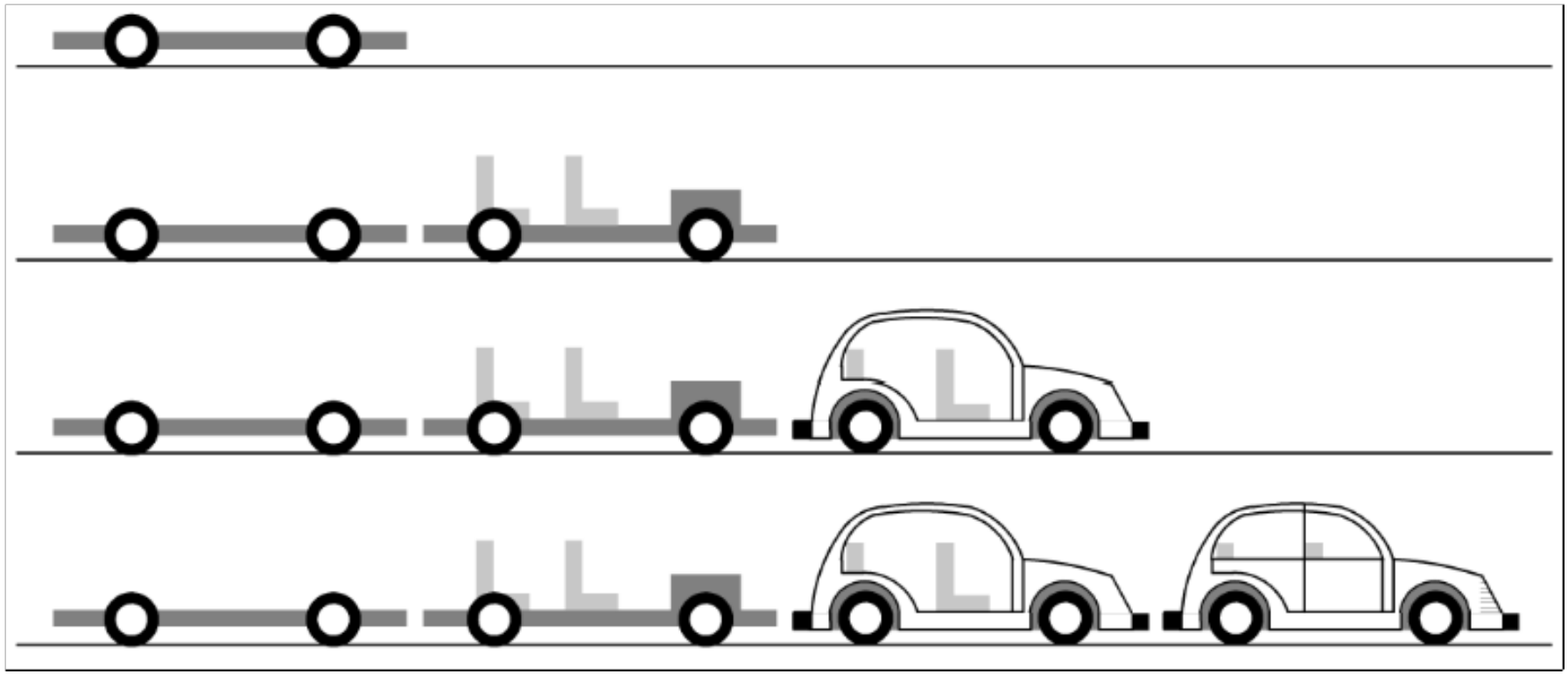
- Independent tasks apply different operations to different data elements

  a □ 2
  b □ 3
  m □ (a + b) / 2
  s □ $(a^2 + b^2)$ / 2
  v □ s - m^2

- First and second statements
- Third and fourth statements

Pipelining

- Divide a process into stages
- Produce several items simultaneously



Programming Parallel Computers

- Extend compilers: translate sequential programs into parallel programs
- Extend languages: add parallel operations
- Add parallel language layer on top of sequential language
- Define totally new parallel language and compiler system

Strategy 1: Extend Compilers

- Parallelizing compiler
  - Detect parallelism in sequential program
  - Produce parallel executable program
- Focus on making Fortran programs parallel

Extend Compilers (cont.)

- Advantages
  - Can leverage millions of lines of existing serial programs
  - Saves time and labor
  - Requires no retraining of programmers
  - Sequential programming easier than parallel programming

Extend Compilers (cont.)

- Disadvantages
  - Parallelism may be irretrievably lost when programs written in sequential languages
  - Parallelizing technology works mostly for easy codes with loops, etc.
  - Performance of parallelizing compilers on broad range of applications still up in air

Extend Language

- Add functions to a sequential language
  - Create and terminate processes
  - Synchronize processes
  - Allow processes to communicate
  - E.g., MPI, PVM, Process/thread, OpenMP

Extend Language (cont.)

- Advantages
  - Easiest, quickest, and least expensive
  - Allows existing compiler technology to be leveraged
  - New libraries can be ready soon after new parallel computers are available

Extend Language (cont.)

- Disadvantages
  - Lack of compiler support to catch errors
  - Easy to write programs that are difficult to debug

Add a Parallel Programming Layer

- Lower layer
  - Core of computation
  - Process manipulates its portion of data to produce its portion of result (persistent object like)
- Upper layer
  - Creation and synchronization of processes
  - Partitioning of data among processes
- A few research prototypes have been built based on these principles – Linda, SyD Middleware – System on Devices (Prasad, et al., MW-04)

Create a Parallel Language

- Develop a parallel language “from scratch”
  - occam is an example
- Add parallel constructs to an existing language
  - Fortran 90
  - High Performance Fortran
  - C*

New Parallel Languages (cont.)

- Advantages
  - Allows programmer to communicate parallelism to compiler
  - Improves probability that executable will achieve high performance
- Disadvantages
  - Requires development of new compilers
  - New languages may not become standards
  - Programmer resistance

Current Status

- Low-level approach is most popular
  - Augment existing language with low-level parallel constructs
  - MPI, PVM, threads/process-based concurrency and OpenMP are examples
- Advantages of low-level approach
  - Efficiency
  - Portability
- Disadvantage: More difficult to program and debug

Summary

- High performance computing
  - U.S. government
  - Capital-intensive industries
  - Many companies and research labs
- Parallel computers
  - Commercial systems
  - Commodity-based systems

Summary – contd.

- Power of CPUs keeps growing exponentially
- Parallel programming environments changing very slowly
- Two standards have emerged
  - MPI/PVM library, for processes that do not share memory
  - OpenMP directives and thread based concurrency, for processes that do share memory

LENGUAJES PARALELOS

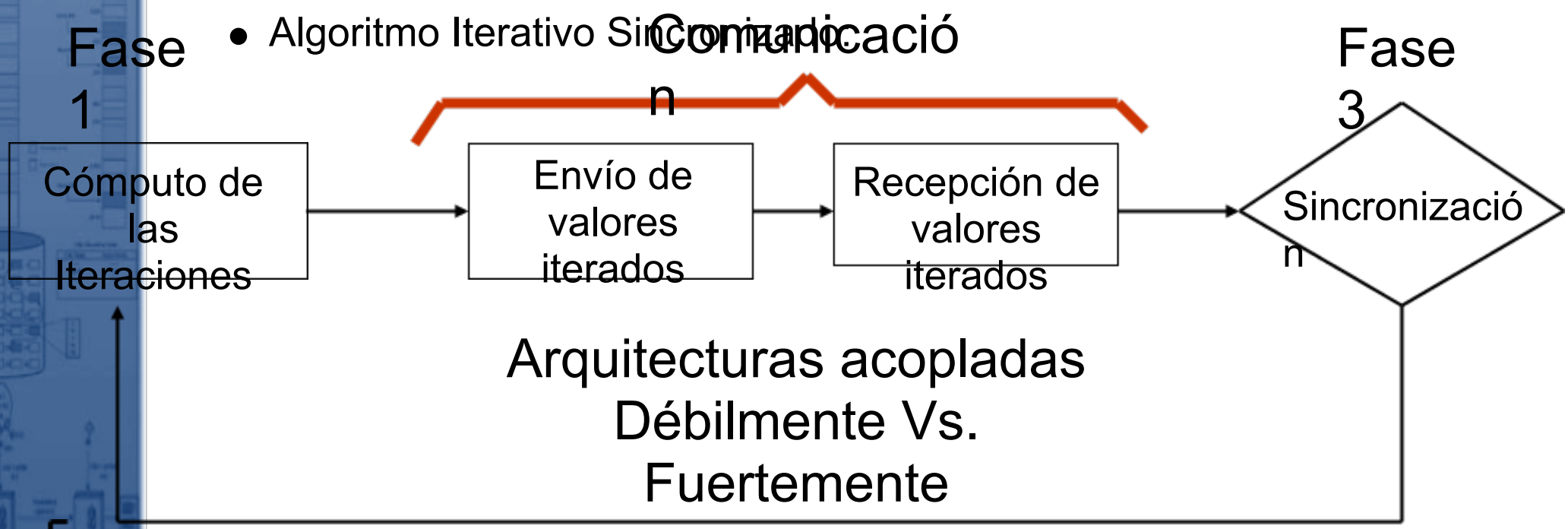
- Chang y Smith (1990) clasificación:
- 1) Lenguajes con características de Optimización:
  - FX Fortran, DINO, MIMDizer.
- 2) Características de disponibilidad (Escalabilidad).
  - Compatibilidad. JAVA.
  - Portable. PVM, MPI y JAVA.
- 3) Características de comunicación/sincronización:
  - PVM, MPI y CORBA.
- 4) Control del paralelismo (Nivel de paralelismo).
  - Grano Burdo (Unix) PVM, MPI.
  - Grano Fino. OCCAM, Threads, Pthreads Y Cthreads.
- 5) Características del paralelismo de datos (SIMD o MIMD):
  - Propietarios. Lenguajes con extensiones (C*, ADA, Pascal C., C++, etc)
- 6) Caracteríticas para el Control de procesos. Creación eficiente de procesos (Multitarea o multithread).
  - JAVA, Pthreads, C*, CORBA, Sistemas Propietarios.

IMPLEMENTACION DE ALGORITMOS

- Para la programación de algoritmos en máquinas SIMD y MIMD, es necesario la utilización de estrategias de planificación de multiprocesadores.
- Al diseñar algoritmos se deben de tomar en cuenta:
  - Decisión de Ubicación. Donde situar el código.
  - Decisión de Asignación (Administración del procesador). En que procesador ejecutar el código.
- El objetivo primordial es desarrollar la asignación y las técnicas de planificación que utilizarán el número mínimo de procesadores para ejecutarlos en el menos tiempo posible.
- Algoritmos deterministas y no deterministas.
- Clasificación de algoritmos paralelos (SIMD y MIMD):
  - Algoritmos Sincronizados. Puntos de Interación.
  - Algoritmos Asíncronos (Mem. Compartida). No hay punto de interacción.
  - Macro segmentación encauzada. División del problema en segmentos denominados etapas, la salida de uno es la entrada de otro segmento.

IMPLEMENTACION DE ALGORITMOS (contd.)

- La eficiencia de los algoritmos paralelos esta regida por medidas de rendimiento:
  - 1) Fase de Procesamiento. Costo de procesamiento puro.
  - 2) Fase de comunicación. Costo de la comunicación (Envío y recepción).
  - 3) Fase de sincronización. Costo de esperar los resultados de una etapa.
- Algoritmo Iterativo Sincronizado.



Arquitecturas acopladas Débilmente Vs. Fuertemente

IMPLEMENTACION DE ALG. PARALELOS (PVM Y MPI)

- En actualidad se disponen de muchos ambientes de desarrollo paralelo: CORBA, Cthreads, JAVA, OCCAM, ADA, C*, PVM, MPI etc. Cada uno propone características diferentes.
- La mayoría se basan en estándares de internacionales como:
  - POSIX (IEEE 1003.1). Llamadas al sistema compatibles, con tiempo real.
  - Threads (IEEE 1003.1c). Multitarea estándares.
  - BSD Unix. IPCS.
  - AT&T System V versión 4 (SVR4). IPCS.
- Los IPCs más utilizados son:
  - Signals. Notificación de recepción de datos asíncronos.
  - Shared Memory. Segmentos de memoria compartida entre procesos.
  - Semaphores. Funciones P y V, coordinan el acceso exclusivo a recursos.
  - Locks, Mutexes y Variables de Condición. Exclusión mutua recursos simples.
  - Barries (IRIX). Para sincronizar un grupo de procesos.
  - Fiels Locks. Para asegurar el uso exclusivo de una parte de un archivo.

Model	Programming Paradigms	Flint Taxonomy
Domain decomposition	Message Passing **MPI**, **PVM**	Single Program Multiple Data (**SPMD**)
	Data Parallel **HPF**	
Functional decomposition	Data Parallel **OpenMP**	Multiple Program Single Data (**MPSD**)
	Message Passing **MPI**, **PVM**	Multiple Program Multiple Data (**MPMD**)

LOS MODELOS DE COMPUTACION DISTRIBUIDA

- En la actualidad existen dos herramientas muy populares para la programación paralela y distribuida. Que funcionan en sistemas heterogéneos basados en el protocolo TCP/IP.
- **Message-Passing Interface (MPI).**
  - Es un estándar de programación para construcción aplicaciones portables en sistemas heterogéneos, utilizando el lenguaje Fortran 77 o C.
  - Las aplicaciones se pueden desarrollar suponiendo un número fijo de procesadores en una topología fija de procesadores (Pipeline, Mesh, Hipercube, Pyramid, HyperTree, etc.).
  - Provee una serie de funciones con las cuales es posible lograr la ejecución, comunicación y sincronización de procesos paralelos.
- **Parallel Virtal Machine (PVM).**
  - Es un conjunto de herramientas y bibliotecas que emulan aplicaciones de propósito general, en un ambiente de computación concurrente con computadoras interconectadas en una arquitectura variada.
  - Permite la creación de aplicaciones que ejecutan una serie de procesos concurrentes en un conjunto de computadoras que pueden incluir uniprocesadores, multiprocesadores y nodos de un arreglo de procesadores.

MODELOS PVM Y MPI

- PVM y MPI son modelos abstractos y formales que permiten el diseño y distribución de aplicaciones en los nodos de sistemas heterogéneos con memoria PRAM, cada uno provee diferentes formas de desarrollar aplicaciones:
  - Procesos y Threads puede ser ejecutados en paralelo en un sistema heterogéneo.
  - Cuando los procesos son distribuidos en máquinas independientes se puede construir programas paralelos utilizando el paso de mensajes.
- En una ambiente de pasos de mensajes las aplicaciones consisten de :
  - Múltiples procesos independientes.
  - Cada uno tiene su propio espacio de direcciones.
  - Se ejecutan utilizando el modelo PRAM.
  - Cada proceso se coordina con otros utilizando el paso de mensajes.
- En sistema UNIX actualmente es posible integrar estas dos librerías.

Próximamente: PVM vs MPI.

PVM

PVM Resources

- Web site www.epm.ornl.gov/pvm/pvm_home.html
- Book

PVM: Parallel Virtual Machine
A Users' Guide and Tutorial for Networked Parallel Computing

Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, Vaidy Sunderam

www.netlib.org/pvm3/book/pvm-book.html

PVM = Parallel Virtual Machine

- Software package
  - Standard daemon: pvmd
  - Application program interface
- Network of heterogeneous machines
  - Workstations
  - Supercomputers
  - UNIX, M$ Win, VMS, OS/2

Configuración de PVM

- Es necesario tener el software siguiente antes de iniciar PVM:
  - Archivo de instalación de PVM; pvm34.exe o pvm34.z
  - En el caso de los sistemas Windows es necesario agregar un servicio de shell remoto mediante el cual PVM ejecutará procesos. Wrshd95 o WrshNT.
- La variables de entorno necesarias para un correcto funcionamiento son:
  - PVM_ROOT. Directorio Raíz del PVM. Set PVM_ROOT=c:\pvm3.4
  - PVM_ARCH. Arquitectura de la maquina.

Configuración de PVM

- Antes de iniciar la ejecución de una aplicación paralela se requiere activar la máquina virtual, incluyendo sus host.
- Activar pvmd3.exe, ejecutando % PVM_ROOT%\lib\pvmd. Servidor de PVM.
- Activar la consola Pvm.exe; % PVM_ROOT%\console\%PVM_ARCH% \Pvm.exe.
- Agregar con comandos en la consola las máquinas que participaran en la máquina virtual.

Configuración de PVM

- Antes de agregar una nueva máquina a PVM, es necesario asegurarse que el archivo /hosts existan las ligas entre el IP y el Hostname que se agregará:
  - 200.49.226.22 stargazer
  - 200.49.226.38 grid
- O bien asegurarse que un DNS puede resolver el nombre por una dirección IP. Ver archivo named. hosts del DNS.
- En la consola PVM se agrega con el comando add un nuevo host especificando su directorio donde se encuentra el pvmd3.exe

Consola PVM

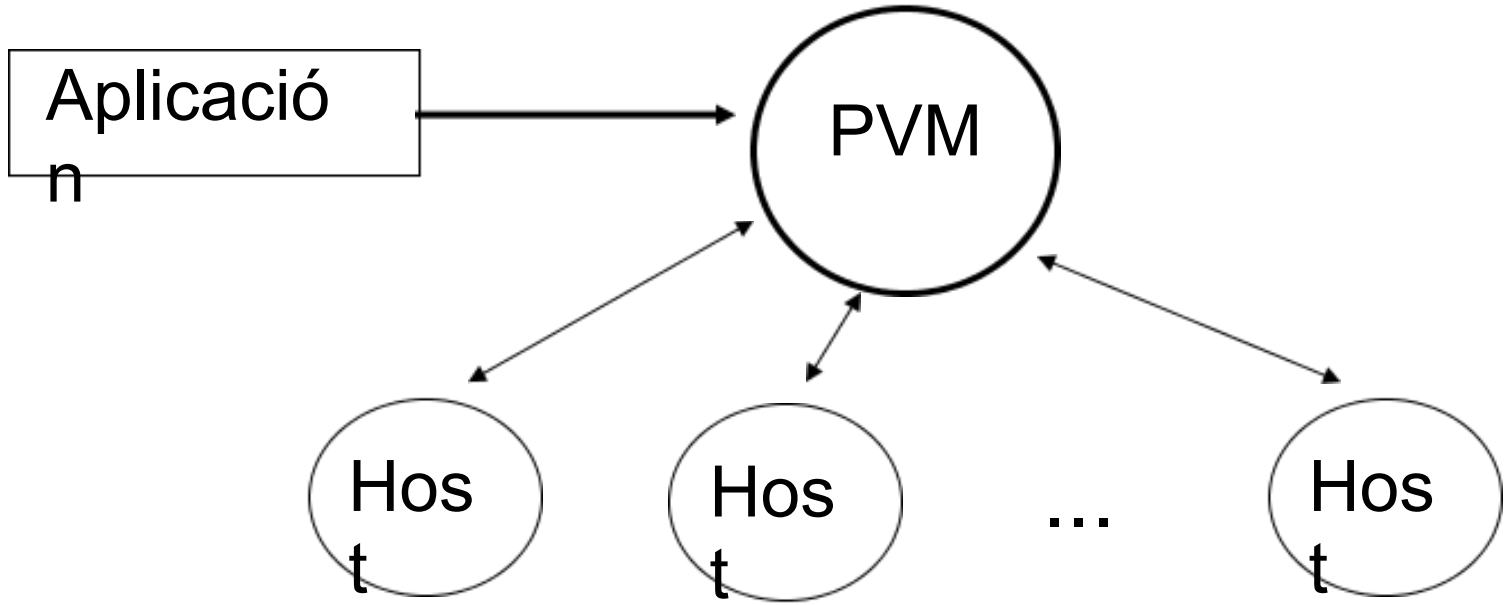
- Los comandos más utilizados en la consola son:
  - add hostname(s) Add hosts to virtual machine
  - alias Define/list command aliases
  - conf List virtual machine configuration
  - delete hostname(s) Delete hosts from virtual machine
  - halt Stop pvmds
  - help [command] Print helpful information about a command
  - id Print console task id

Consola PVM

- Comandos de la consola PVM:
  - jobs List running jobs.
  - kill task-tid Terminate tasks.
  - mstat host, tid Show status of hosts.
  - ps -a List all PVM tasks.
  - pstat task-tid Show status of tasks.
  - quit Exit console.
  - reset Kill all tasks.
  - setenv Display/set environment variables.
  - sig signum task Send signal to task.
  - spawn [opt] a.out Spawn task.
    - (count) number of tasks, default is 1.
    - (host) spawn on host, default is any.
    - (ARCH) spawn on hosts of ARCH.
  - trace Set/display trace event mask.

Librería PVM

- Es necesario incluir las bibliotecas de PVM en un programa que se ejecutará en la máquina virtual.



Librería PVM

- La aplicación debe diseñarse de tal forma que esta considera el número de copias que hará de ella misma en que Host, también deberá incluir la etapa de sincronización y la etapa de comunicación.
- Las funciones de PVM ayudan a controlar las tres fases de un algoritmo paralelo.

Programa de Ejemplo

- El siguiente código muestra un programa que indica a PVM la creación de una tarea en otro host (“Hello Other”), retornado desde el otro host el mismo mensaje enviado con algunos parámetros extras.

Programa de Ejemplo

```
#include <stdio.h>
#include "pvm3.h" //librería PVM
main()
{
int cc, tid;
char buf[100];
printf("i'm t%x\n", pvm_mytid()); //INDICA EL TID DEL PROGRAMA
cc = pvm_spawn("hello_other", (char**)0, 0, "", 1, &tid); //Crea Proceso en otro Host
if (cc == 1) { //Fue Posible
cc = pvm_recv(-1, -1); //Esperar Respuesta del proceso remoto.
pvm_bufinfo(cc, (int*)0, (int*)0, &tid); //Obtener información del canal.
pvm_upkstr(buf); //Traer información a buf (xdr)
printf("from t%x: %s\n", tid, buf); //Imprimir mensaje
} else
printf("can't start hello_other\n"); //No se incio proceso Hello Other
pvm_exit();
exit(0);
}
```

Programa de Ejemplo

- **Archivo Hello Other**
  - **Programa que responde a la petición de otro programa Hello.c, se debe colocar el archivo ejecutable en %PVM_ROOT% de la máquina remota.**

Archivo Hello Other

```
#include <stdio.h>
#include "pvm3.h”
main()
{
int ptid;
char buf[100];

ptid = pvm_parent(); //SOLICITA EL PID DEL PROCESO PADRE
strcpy(buf, "hello, world from "); //CONSTRUIR EL MENSAJE DE RETORNO
gethostname(buf + strlen(buf), 64); //INSERTA EL NOMBRE DEL HOST LOCAL

pvm_initsend(PvmDataDefault); //PREPARA ENVIO DE MSGs ESTANDARES
pvm_pkstr(buf); //CONSTRUYE EL PAQUETE PARA ENVIAR (xdr)
pvm_send(ptid, 1); //ENVIA EL PAQUETE

pvm_exit();
exit(0);
}
```

Funciones de la Librería

- LAS FUNCIONES DE PVM: Se dividen en 5 clases:
- **Message Passing:**

pvm_bufinfo, pvm_freebuf, pvm_getrbuf, pvm_getsbuf, pvm_initsend, pvm_mcast, pvm_mkbuf, pvm_nrecv, pvm_pack, pvm_precv, pvm_probe, pvm_psend, pvm_recv, pvm_recvf, pvm_send, pvm_sendsig, pvm_setmwid, pvm_setrbuf, pvm_setsbuf, pvm_trecv, pvm_unpack

- **Task Control:**

pvm_exit, pvm_kill, pvm_mytid, pvm_parent, pvm_pstat, pvm_spawn, pvm_tasks

- **Group Library Functions:**

pvm_barrier, pvm_bcast, pvm_gather, pvm_getinst, pvm_gettid, pvm_gsize, pvm_joingroup, pvm_lvgroup, pvm_reduce, pvm_scatter

- **Virtual Machine Control:**

pvm_addhosts, pvm_config, pvm_delhosts, pvm_halt, pvm_mstat, pvm_reg_hoster, pvm_reg_rm, pvm_reg_tasker, pvm_start_pvmd

- **Miscellaneous:**

Some more details about

PVM

The *pvmd3* process

- Manages each host of vm
  - Message router
  - Create, destroy, … local processes
  - Fault detection
  - Authentication
  - Collects output
- Inter-host point of contact
  - One pvmd3 on each host
- Can be started during boot

The program named *pvm*

- Interactive control console
  - Configuration of PVM
  - Status checking
- Can start pvmd

The library *libpvm*

- Linked with application programs
- Functions
  - Compose a message
  - Send
  - Receive
- System calls to pvmd
- libpvm3.a, libfpvm3.a, libgpvm3.a

libpvm3.a

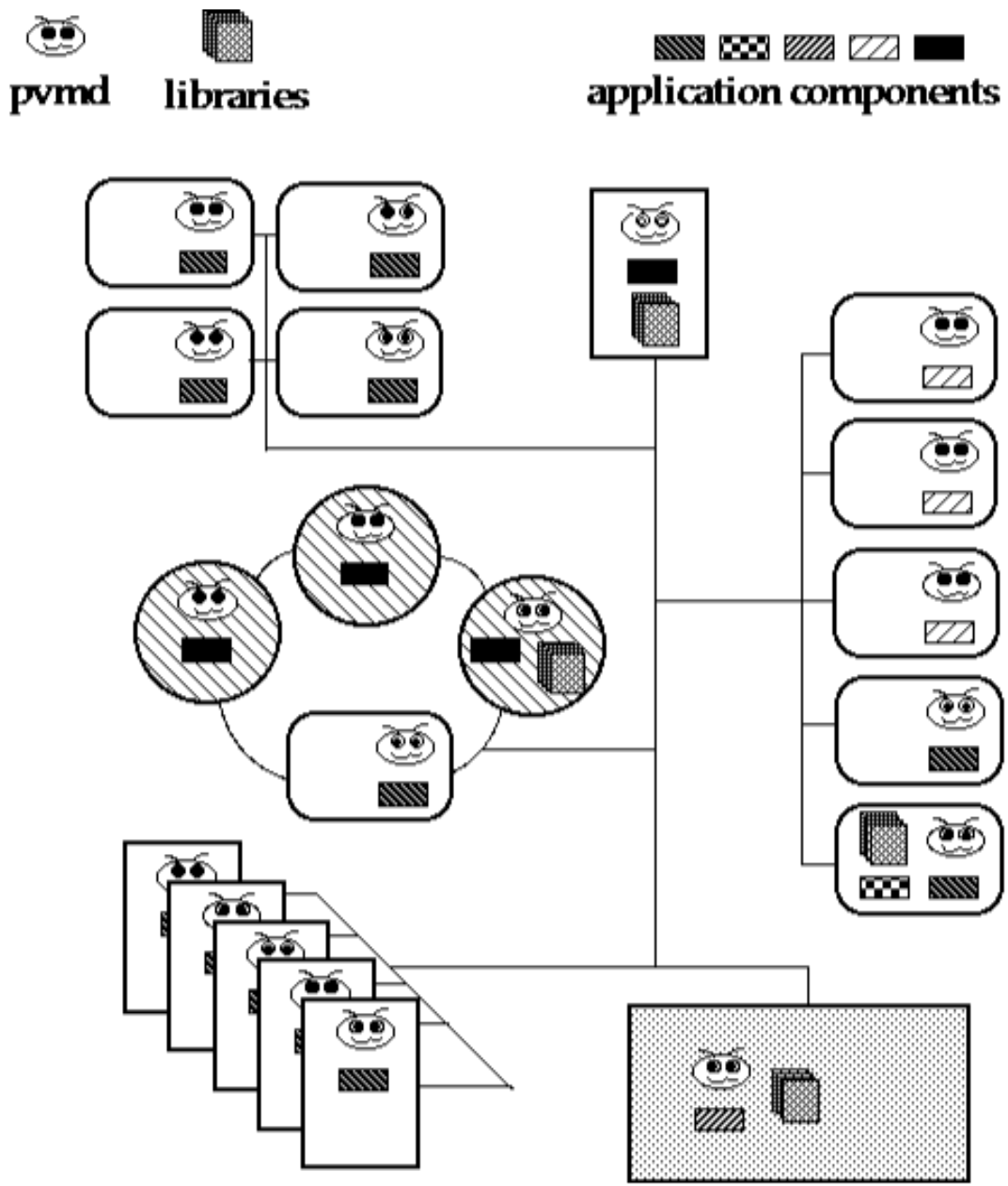
- Initiate and terminate processes
- Pack, send, and receive messages
- Synchronize via barriers
- Query and change configuration of the pvm
- Talk to local pvmd3
- Data format conversion (XDR)

libfpvm3.a

- additionally required for Fortran codes

libgpvm3.a

- Dynamic groups of processes


pvmd
libraries
application components

PVM: Hello World!

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

main()
{
me = pvm_mytid());

cc = pvm_spawn("hello_other”, 0, 0, "",
1, &tid);

…

pvm_exit();
exit(0);
}
```

```
#include "pvm3.h"

main()
{
ptid = pvm_parent();

strcpy(buf, "hello, world from ");
gethostname(buf + strlen(buf), 64);

pvm_initsend(PvmDataDefault);
pvm_pkstr(buf);
pvm_send(ptid, 1);

pvm_exit();
exit(0);
}
```

PVM: Hello World!

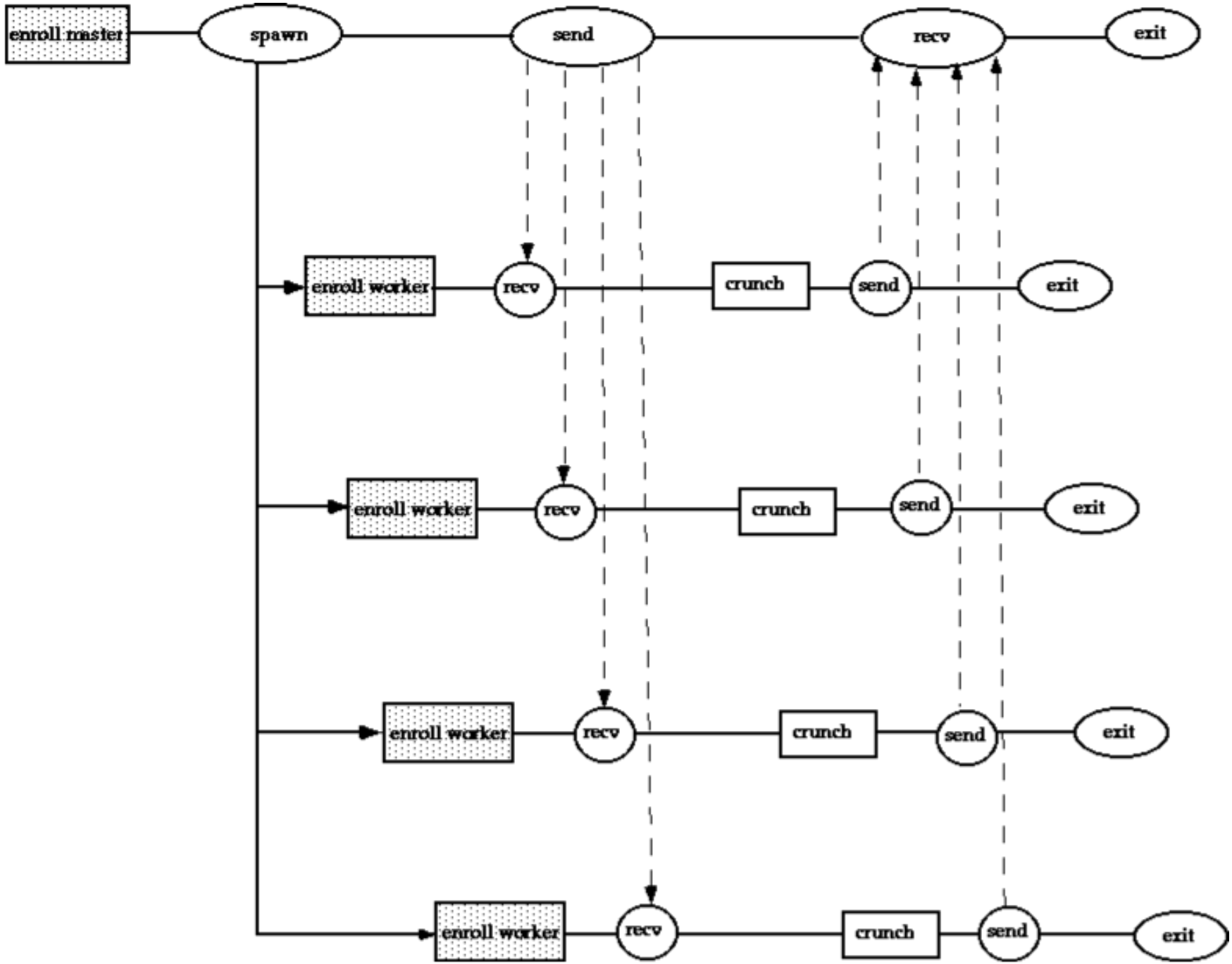
```
#include <stdio.h>
#include "pvm3.h"

main()
{
cc = pvm_spawn("hello_other", 0, 0, "", 1, &tid);

if (cc == 1) {
cc = pvm_recv(-1, -1);
pvm_bufinfo(cc, 0, 0, &tid);
pvm_upkstr(buf);
printf("from t%x: %s\n", tid, buf);
} else …
}
```

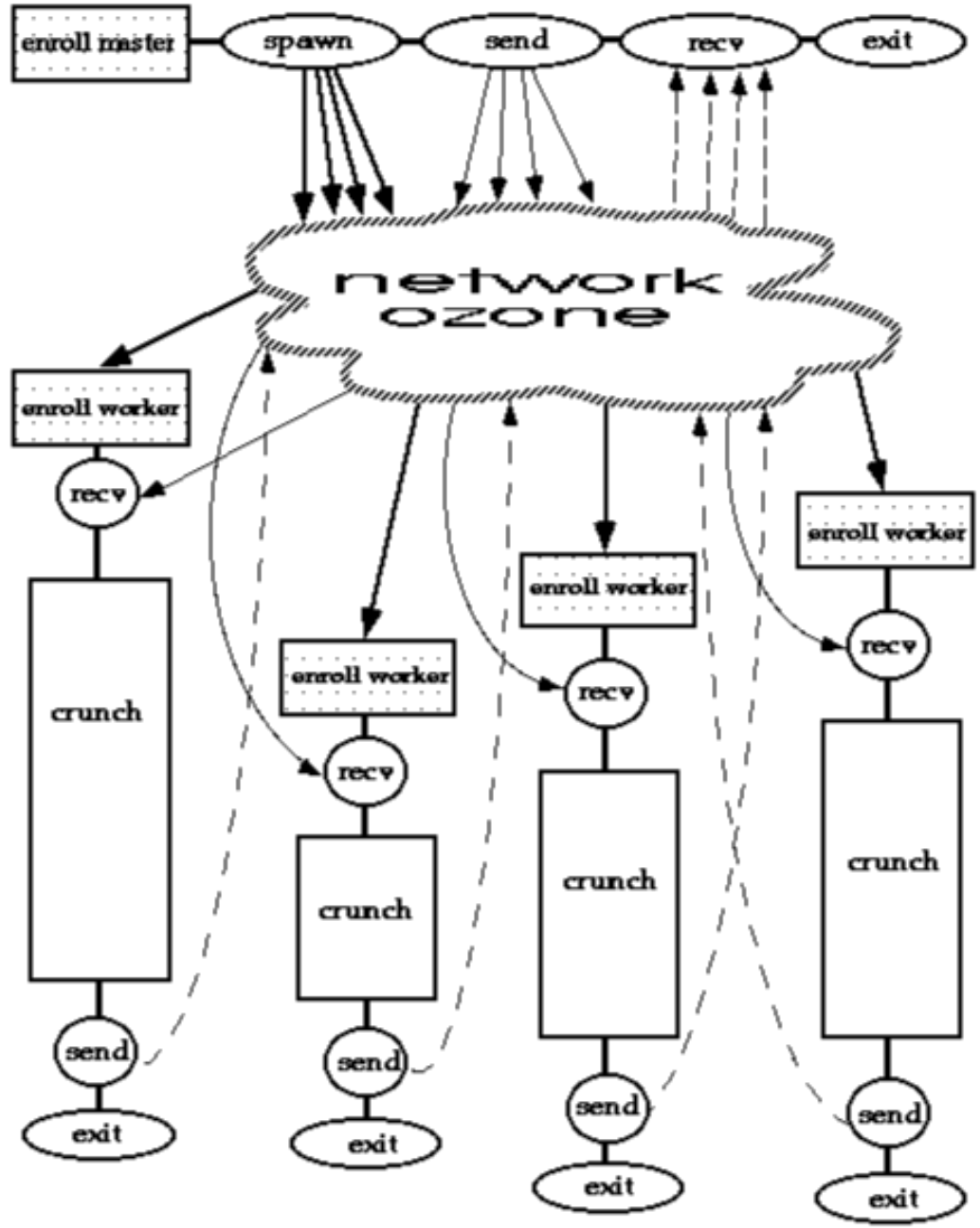
```
#include "pvm3.h"

main()
{
int ptid;
char buf[100];

ptid = pvm_parent();

strcpy(buf, "hello, world from ");
gethostname(buf + strlen(buf), 64);

pvm_initsend(PvmDataDefault);
pvm_pkstr(buf);
pvm_send(ptid, 1);

pvm_exit();
exit(0);
}
```

PVM: Hello World!

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

main()
{
int cc, tid;
char buf[100];

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

cc = pvm_spawn("hello_other", 0, 0, "", 1, &tid);

if (cc == 1) {
cc = pvm_recv(-1, -1);
pvm_bufinfo(cc, 0, 0, &tid);
pvm_upkstr(buf);
printf("from t%x: %s\n", tid, buf);
} else
printf("can't start hello_other\n");

pvm_exit();
exit(0);
}
```

```
#include "pvm3.h"

main()
{
int ptid;
char buf[100];

ptid = pvm_parent();

strcpy(buf, "hello, world from ");
gethostname(buf + strlen(buf), 64);

pvm_initsend(PvmDataDefault);
pvm_pkstr(buf);
pvm_send(ptid, 1);

pvm_exit();
exit(0);
}
```


enroll master
spawn
send
recv
exit
enroll worker
recv
crunch
send
exit
enroll worker
recv
crunch
send
exit
enroll worker
recv
crunch
send
exit
enroll worker
recv
crunch
send
exit


enroll master
spawn
send
recv
exit
network
ozone
enroll worker
recv
crunch
send
exit
enroll worker
recv
crunch
send
exit
enroll worker
recv
crunch
send
exit
enroll worker
recv
crunch
send
exit

pvm_mytid

- Enrolls the calling process into PVM and generates a unique task identifier if this process is not already enrolled in PVM. If the calling process is already enrolled in PVM, this routine simply returns the process's tid.
- tid = pvm_mytid ();

pvm_spawn

- Starts new PVM processes. The programmer can specify the machine architecture and machine name where processes are to be spawned.
- numt = pvm_spawn ("worker",0, PvmTaskDefault,"",1,&tids[i]);
- numt = pvm_spawn ("worker",0,PvmTaskArch," RS6K",1,&tid[i]);

pvm_exit

- Tells the local pvmd that this process is leaving PVM. This routine should be called by all PVM processes before they exit.

pvm_addhosts

- Add hosts to the virtual machine. The names should have the same syntax as lines of a pvmd hostfile.
- pvm_addhosts (hostarray,4, infoarray);

pvm_delhost

- Deletes hosts from the virtual machine.
- pvm_delhosts (hostarray,4);

pvm_pkdatatype

- Pack the specified data type into the active send buffer. Should match a corresponding unpack routine in the receive process. Structure data types must be packed by their individual data elements.

pvm_upkdatatype

- Unpack the specified data type into the active receive buffer. Should match a corresponding pack routine in the sending process. Structure data types must be unpacked by their individual data elements.

pvm_send

- Immediately sends the data in the message buffer to the specified destination task. This is a blocking, send operation. Returns 0 if successful, < 0 otherwise.
- pvm_send (tids[1],MSGTAG);

pvm_psend

- Both packs and sends message with a single call. Syntax requires specification of a valid data type.

pvm_mcast

- Multicasts a message stored in the active send buffer to tasks specified in the tids[]. The message is not sent to the caller even if listed in the array of tids.
- pvm_mcast (tids,ntask,msgtag);

pvm_recv

- Blocks the receiving process until a message with the specified tag has arrived from the specified tid. The message is then placed in a new active receive buffer, which also clears the current receive buffer.
- pvm_recv (tid,msgtag);

pvm_nrecv

- Same as pvm_recv, except a non-blocking receive operation is performed. If the specified message has arrived, this routine returns the buffer id of the new receive buffer. If the message has not arrived, it returns 0. If an error occurs, then an integer < 0 is returned.
- pvm_nrecv (tid,msgtag);

PVM Collective Communication

pvm_barrier

- Blocks the calling process until all processes in a group have called pvm_barrier().

- pvm_barrier ("worker",5);

pvm_bcast

- Asynchronously broadcasts the data in the active send buffer to a group of processes. The broadcast message is not sent back to the sender.
- pvm_bcast ("worker",msgtag);

pvm_gather

- A specified member receives messages from each member of the group and gathers these messages into a single array. All group members must call pvm_gather().
- pvm_gather (&getmatrix,&myrow,10, PVM_INT,msgtag,"workers",root);

pvm_scatter

- Performs a scatter of data from the specified root to each of the members of the group, including itself. All group members must call pvm_scatter(). Each receives a portion of the data array from the root in their local result array.
- pvm_scatter (&getmyrow,&matrix,10, PVM_INT, msgtag,"workers",root);

pvm_reduce

- Performs a reduce operation over members of the group. All group members call it with their local data, and the result of the reduction operation appears on the root. Users can define their own reduction functions or the predefined PVM reductions
- pvm_reduce (PvmMax,&myvals,10, PVM_INT,msgtag,"workers",root);

Prepare to Execute Your PVM session

PVM expects executables to be located in ~/pvm3/bin/$PVM_ARCH

% ln -s $PVM_ROOT/lib ~/pvm3/lib

% cc -o myprog myprog.c -I$PVM_ROOT/include -L$PVM_ROOT/lib/$PVM_ARCH -lpvm3

Create your PVM hostfile

- PVM hostfile defines your parallel virtual machine. It contains the names of all desired machines, one per line.

Create Your $HOME/.rhosts file

- Example .rhosts file

```
grid.cs.uns.edu.ar user02
fr2s02.mhpcc.edu user02
beech.tc.cornell.edu jdoe
machine.mit.edu user02
```

Start pvmd3

- % pvmd3 hostfile &

- This starts up daemons on all other machines (remote) specified in your hostfile.
- PVM console can be started after pvmd3 by typing "pvm".

Execute your application

```
% myprog
```

Quitting PVM

- Application components must include call of pvm_exit().
- Halting the master pvmd3 will automatically kill all other pvmd3s and all processes enrolled in this PVM.
- In pvm console: "halt"
- Running in the background: enter console mode by typing "pvm" and halt.

MPI 1.2

Message Passing Interface

Sistemas Operativos y Distribuidos

Mg. Javier Echaiz

D.C.I.C. – U.N.S.

http://cs.uns.edu.ar/~jechaiz

je@cs.uns.edu.ar

Introducción

- El pasaje de mensajes es una tarea ampliamente usada en ciertas clases de maquinas paralelas, especialmente aquellas con memoria distribuida. Recientemente diferentes sistemas han demostrado que un sistema de pasaje de mensajes puede ser implementado eficientemente y con un alto grado de portabilidad.

Introducción

- Al diseñarse MPI, se tomaron en cuenta las características más atractivas de los sistemas existentes para el pasaje de mensajes, en vez de seleccionar uno solo de ellos y adoptarlo como el estándar. Resultando así, en una fuerte influencia para MPI los trabajos hechos por IBM, INTEL NX/2, Express, nCUBE's Vernex, p4 y PARMACS. Otras contribuciones importantes provienen de Zipcode,Chimp, PVM, Chameleon y PICL.

Introducción

- La meta de MPI o Message Passing Interface (Interfase de pasaje de mensajes), es el desarrollar un estándar (que sea ampliamente usado) para escribir programas que implementen el pasaje de mensajes. Por lo cual la interfase intenta establecer para esto un estándar práctico, portable, eficiente y flexible.

- El esfuerzo para estandarizar MPI involucró cerca de 60 personas de 40 organizaciones diferentes principalmente de USA y Europa. La mayoría de los vendedores de computadoras concurrentes estaban involucrados con MPI, así como con investigadores de diferentes

MPI

- Es una interfaz estándar (1993) para la programación siguiendo el paradigma de pasaje de mensajes.
- Sirve tanto para grandes computadores de memoria compartida, como para clusters o redes de ordenadores heterogéneos (incluido el grid computing).
- Definida para C, C++ y FORTRAN.
- Comprende una gran cantidad de funciones (129), macros, etc.
- Pero con 6 funciones básicas se puede empezar a programar usando MPI.

MPI

- Para que se puede usar MPI:
  - Programas paralelos “portables”.
  - Librerías paralelas.

- Cuando no usar MPI:
  - Si se pueden usar librerías de más alto nivel (que pueden haber sido escritas usando MPI).
  - Si no se necesita realmente el paralelismo.

MPI

- En el modelo de programación MPI, un cómputo comprende uno o más procesos comunicados a través de llamadas a rutinas de librería para mandar (send) y recibir (receive) mensajes a otros procesos. En la mayoría de las implementaciones de MPI, se crea un conjunto fijo de procesos al inicializar el programa, y un proceso es creado por cada tarea. Sin embargo, estos procesos pueden ejecutar diferentes programas. De ahi que, el modelo de programación MPI es algunas veces referido como MPMD (multiple program multiple data).

MPI

- MPI permite definir grupos de procesos. Un grupo es una colección de procesos, y define un espacio de direcciones (desde 0 hasta el tamaño del grupo menos 1). Los miembros del grupo tienen asignada una dirección dentro de él. Un proceso puede pertenecer simultáneamente a varios grupos, y tener una dirección distinta en cada uno de ellos.
- Un comunicador es un universo de comunicación. Básicamente consiste en un grupo de procesos, y un contexto de comunicación. Las comunicaciones producidas en dos comunicadores diferentes nunca interfieren entre sí.

MPI

- Funciones básicas:
  - MPI_Init => Inicialización de MPI.
  - MPI_Finalize => Termina MPI.
  - MPI_Comm_size => Para averiguar el número de procesos.
  - MPI_Comm_rank => Identifica el proceso.
  - MPI_Send => Envía un mensaje.
  - MPI_Recv => Recibe un mensaje.

- Existen tanto implementaciones gratuitas (por ejemplo MPICH, LAM y Open MPI) como pagas. Típicamente los proveedores de computadores de alto rendimiento tienen implementaciones propias optimizadas para su hardware.

Comunicación Punto a Punto

Modelos de Comunicación

El modelo de comunicación tiene que ver con el tiempo que un proceso pasa bloqueado tras llamar a una función de comunicación.

Dos tipos:

- bloqueante (blocking)
- No bloqueante (nonblocking)

Modelos de Comunicación

Una función bloqueante mantiene a un proceso bloqueado hasta que la operación solicitada finalice.

Una no bloqueante supone simplemente “encargar” al sistema la realización de una operación, recuperando el control inmediatamente.

- Recepción: cuando tengamos un mensaje nuevo, completo, en el buffer asignado al efecto.
- Envío como finalizado cuando el emisor puede reutilizar, sin problemas de causar interferencias, el buffer de emisión que tenía el mensaje.

Bloqueante Vs. No bloqueante

- SEND bloqueante podemos reutilizar el buffer asociado sin problemas.
- SEND no bloqueante tenemos riesgo de alterar inadvertidamente la información que se está enviando.

Modos de Comunicación

- Definición: la forma en la que se realiza y completa un envío.
- MPI define 4 modos de envío:
  - Básico (basic).
  - Con buffer (buffered).
  - Síncrono (synchronous).
  - Listo (ready).

Con Buffer

Cuando se hace un envío **con buffer** se guarda inmediatamente, en un buffer al efecto en el emisor, una copia del mensaje. La operación se da por completa en cuanto se ha efectuado esta copia. Si no hay espacio en el buffer, el envío fracasa.

Síncrono

Si se hace un envío **síncrono**, la operación se da por terminada sólo cuando el mensaje ha sido recibido en destino. Este es el modo de comunicación habitual en los sistemas basados en *Transputers* . En función de la implementación elegida, puede exigir menos copias de la información conforme ésta "viaja" desde el buffer del emisor al buffer del receptor.

Básico

El modo de envío **básico** no especifica la forma en la que se completa la operación: es algo dependiente de la implementación. Normalmente equivale a un envío con buffer para mensajes cortos y a un envío síncrono para mensajes largos. Se intenta así agilizar el envío de mensajes cortos a la vez que se procura no perder demasiado tiempo realizando copias de la información.

Listo

En cuanto al envío en modo **listo**, sólo se puede hacer si antes el otro extremo está preparado para una recepción inmediata. No hay copias adicionales del mensaje (como en el caso del modo con buffer), y tampoco podemos confiar en bloquearnos hasta que el receptor esté preparado.

Semántica de comunicación P2P MPI

- Los mensajes son “no-alcanzables”.
- Progreso.
- No se garantiza “Justicia” en el manejo de mensajes.
- Cualquier operación de comunicación pendiente consume recursos del sistema (que son limitados).

Modos de comunicación y Comunicadores

Modos de comunicación

- MPI define cuatro modos de comunicación:
  - Modo síncrono: seguro.
  - Modo listo (ready): menor overhead en el sistema.
  - Modo Buffer (buffered mode): separa el remitente (sender) del receptor (receiver).
  - Modo estándar: comprometido.

- El modo de comunicación es escogido mediante la rutina del send.

Modos de comunicación

Modo de comunicación	Rutinas bloqueantes	Rutinas no bloqueantes
Síncrono	MPI_SSEND	MPI_ISSEND
Ready	MPI_RSEND	MPI_IRSEND
Buffered	MPI_BSEND	MPI_IBSEND
Estándar	MPI_SEND MPI_RECV	MPI_ISEND MPI_IRECV

Modos de comunicación

- Modo síncrono: Bloquea el proceso hasta que el receive correspondiente fue realizado en el proceso destino.
- Modo ready: Puede utilizarse si se tiene la garantía de que la solicitud de recepción ya fue realizada. Puede implementarse mediante un protocolo de comunicación optimizado, dependiendo de la versión.
- Modo buffered: El programador reserva un buffer para los datos hasta que puedan ser enviados. Ofrece seguridad sobre el espacio de buffer de sistema necesario.
- Modo estándar: el MPI decide cuando se colocan en el buffer los mensajes salientes (asíncrono). El

Modos de comunicación

Modo de comunicación	Ventajas	Desventajas
Síncrono	Más seguro, y por lo tanto más portable SEND/RECV	Puede incurrir en overhead de sincronización
Ready	Menor cantidad de overhead en el SEND/RECV	El RECV debe preceder el SEND
Buffered	Separa el SEND del RECV. No hay overhead síncrono en el SEND. El programador puede controlar el tamaño del espacio en el buffer.	Se incurre en un sistema adicional de overhead al copiar en el buffer
Estándar	Es bueno en muchos casos	El programa puede que no sea conveniente

Primitivas

- **MPI_Send(buffer, count, datatype, dest, tag, comm)**

(MPI_Ssend, MPI_Bsend, MPI_Rsend)

- buffer: Dirección de inicio del buffer de envío o de recepción.
- count: Número de elementos a enviar o a recibir.
- datatype: Tipo de dato de cada elemento.
- dest: Identificador del destinatario.
- source: Identificador del remitente .
- tag: Etiqueta del mensaje.
- comm: Representa el dominio de comunicación.

Primitivas

- **MPI_Recv(buffer, count, datatype, source, tag, comm, status)**

- buffer: Dirección de inicio del buffer de envío o de recepción.
- count: Número de elementos a enviar o a recibir.
- datatype: Tipo de dato de cada elemento.
- dest: Identificador del destinatario.
- source: Identificador del remitente .
- tag: Etiqueta del mensaje.
- comm: Representa el dominio de comunicación.
- status: Permite obtener fuente, tag y contador del

Comunicaciones Asíncronas

Tipo de comunicación	Rutinas	Descripción
Envío	MPI_Isend	Envío no bloqueado. Mismos parámetros del MPI_Send más parámetro *request.*
Recepción	MPI_Irecv	Recepción sin bloqueo. Mismos parámetros de MPI_Recv, sustituyendo *status* por *request.*
Espera	MPI_Wait	Es usada para completar comunicaciones sin bloqueo.
Verificación	MPI_Test	Prueba si una comunicación sin bloqueo ha finalizado.

Rutinas de comunicadores

- **MPI_Comm_size:** devuelve el número de procesos en el grupo de comunicadores
- **MPI_Comm_rank:** devuelve el rango de procesos llamadores (calling process) en el grupo de comunicadores
- **MPI_Comm_compare:** compara dos comunicadores
- **MPI_Comm_dup:** duplica un comunicador
- **MPI_Comm_create:** crea un nuevo comunicador para un grupo
- **MPI_Comm_split:** separa un comunicador en múltiples, comunicadores sin superposición
- **MPI_Comm_free:** marca un comunicador para desasignación

Comunicadores

- MPI utiliza objetos llamados *comunicadores,* los cuales definen cuál colección de procesos se va a comunicar con otra.
- MPI_INIT define un comunicador llamado MPI_COMM_WORLD para cada proceso que lo llama.
- La palabra clave MPI_COMM_WORLD hace referencia al comunicador universal, un comunicador predefinido por MPI que incluye a todos los procesos de la aplicación.
- Todos los comunicadores MPI requieren un argumento comunicador.
- Los procesos MPI sólo se pueden comunicar si comparten un comunicador.

Comunicadores

- Cada proceso tiene su id en el comunicador
- Un proceso puede tener varios comunicadores
- Un proceso puede solicitar información de un comunicador
  - **MPI_Comm_rank(MPI_comm comm, int *rank)**
    - es utilizado para que cada proceso averigue su dirección (identificador) dentro de la colección de procesos que componen la aplicación
  - **MPI_Comm_size(MPI_Comm comm, int *size)**
    - devuelve el número de procesos que participan en la aplicación

Comunicaciones Colectivas y Operaciones Globales

Comunicaciones Colectivas

- Permiten la transferencia de datos entre todos los procesos que pertenecen a un grupo específico.
- No se usan etiquetas para los mensajes, estas se sustituyen por identificadores de los grupos (comunicadores).
- Comprenderán a todos los procesos en el alcance del comunicador.
- Por defecto, todos los procesos se incluyen en el comunicador genérico MPI_COMM_WORLD.

Operaciones Globales

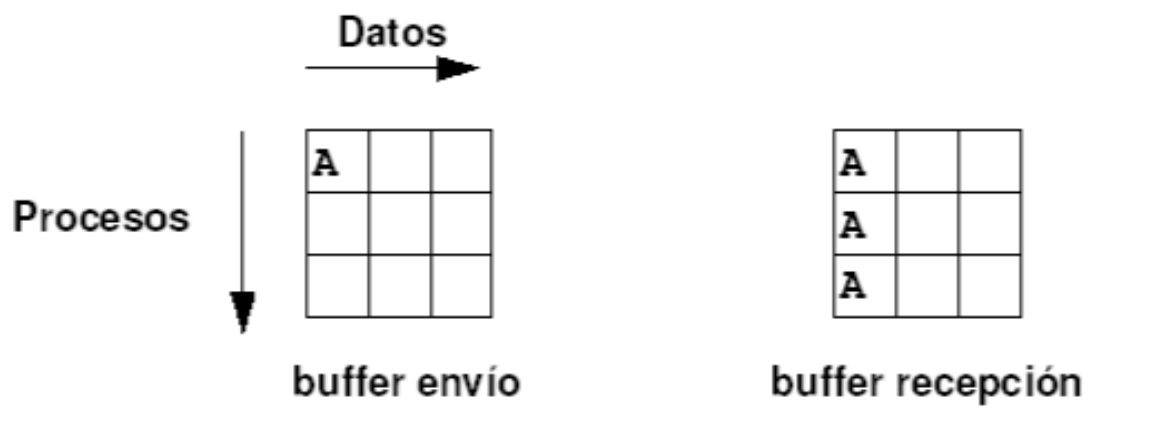
- Permiten coordinar la comunicación en múltiples procesos.
- Se ejecutada por todos los procesos que intervienen en un cálculo o comunicación.
- Ejemplo: todos los procesos pueden necesitar cooperar para invertir una matriz distribuida o para sumar un conjunto de números distribuidos en cada proceso.

Operaciones Globales

- Tipos de Operaciones:
  - Operaciones de movimiento de datos: se utilizan para intercambiar datos entre un conjunto de procesos.
  - Operaciones de cálculo colectivo: permiten hacer cálculos colectivos como mínimo, máximo, suma, OR lógico, así como operaciones definidas por el usuario.
  - Operaciones de barrera: procesos que esperan a que otros miembros del grupo

Operaciones Globales

- **MPI_Barrier():** es una operación puramente de sincronización, que bloquea a los procesos de un comunicador hasta que todos ellos han pasado por la barrera.
- **MPI_Broadcast():** sirve para que un proceso, el raíz, envíe un mensaje a todos los miembros del comunicador.



Operadores Globales

- **MPI_Gather:** realiza una recolección de datos en el proceso raíz. Este proceso recopila un vector de datos, al que contribuyen todos los procesos del comunicador con la misma cantidad de datos.

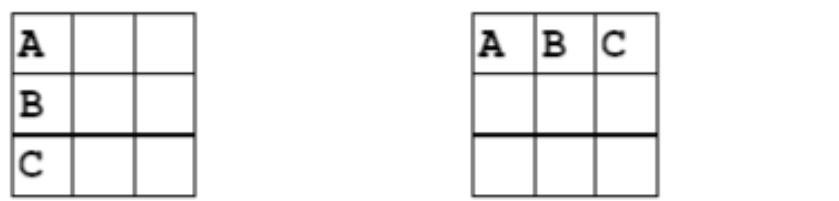


Figura 3. Esquema de la operación colectiva MPI_Gather().

Operaciones Globales

- **MPI_Allgather():** para distribuir a todos los procesos el resultado de una recolección previa.

Figura 4. Esquema de la operación colectiva MPI_Allgather().

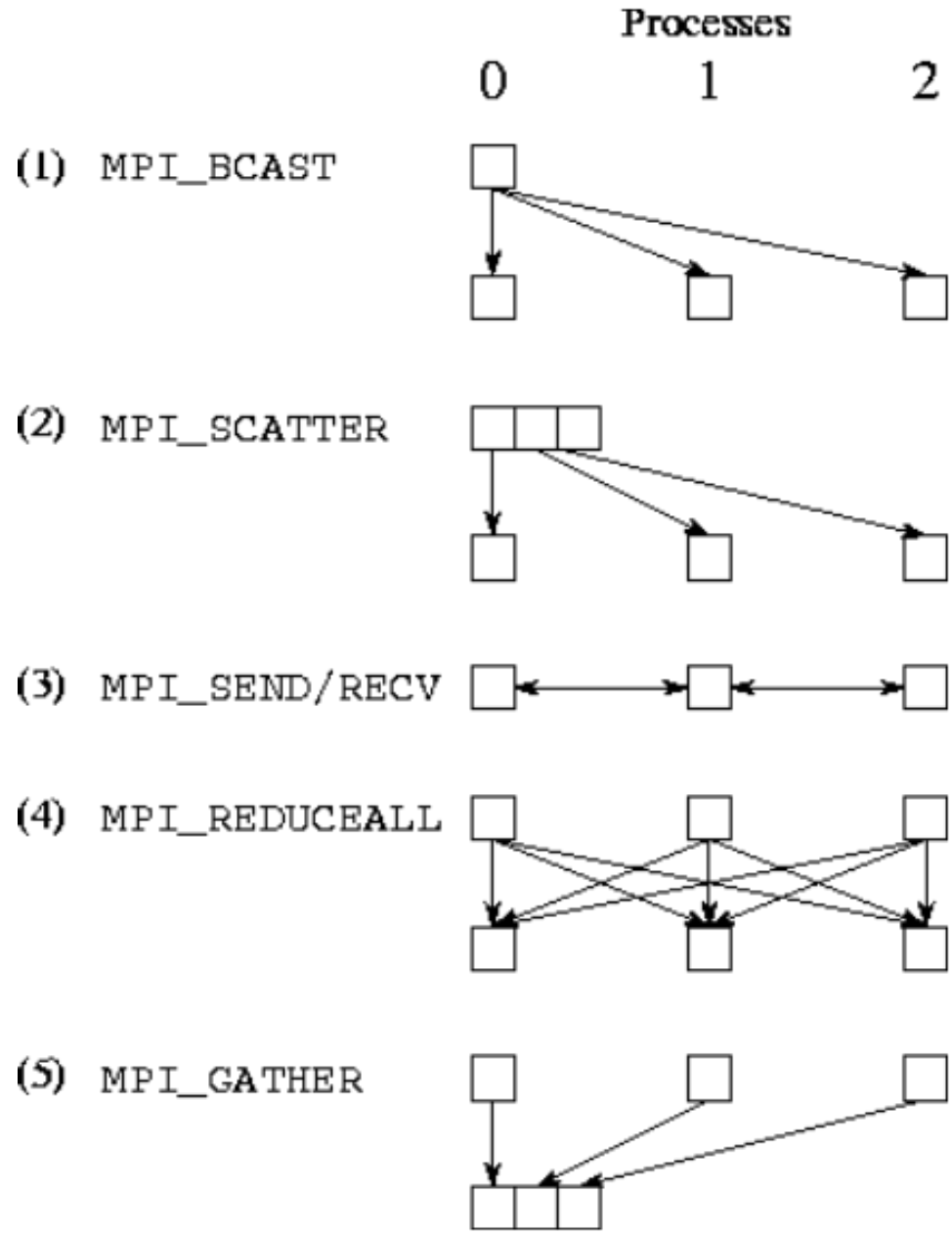
Operaciones Globales

- **MPI_Scatter()** realiza la operación simétrica a MPI_Gather(). El proceso raíz posee un vector de elementos, uno por cada proceso del comunicador. Tras realizar la distribución, cada proceso tiene una copia del vector inicial.

Figura 5. Esquema de la operación colectiva MPI_Scatter().

Operaciones Globales

1. Hace una transmisión del proceso 0 a los n procesos existentes.
   - 2. Distribuye la información del proceso 0 a otros procesos. Distribuye distintos mensajes desde una tarea a cada tarea en el grupo.
   - 3. Intercambian datos entre vecinos.
   - 4. Determina los valores computados en diferentes procesos y los distribuye a cada proceso. (reduce la carga en los procesos).
   - 5. Acumula los valores de otros procesos en un solo proceso.



Operaciones Globales

- Características:
  - Las operaciones colectivas son bloqueantes.
  - Las operaciones colectivas que involucran un subconjunto de procesos deben antes hacer un particionamiento de los conjuntos y relacionar los nuevos grupos con nuevos comunicadores.

Tipos de Datos

Tipos de Datos

- Ofrece un amplio conjunto de tipos de datos predefinidos: caracteres, enteros, números, flotante, entre otros
- Ofrece la posibilidad de definir tipos de datos derivados.

Tabla 1. Tipos de datos MPI

Tipos MPI	Tipos C equivalentes
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	Sin equivalente

Tipos de Datos

- Es posible que existan diferencias en la representación de los datos en las distintas máquinas.
- Para eliminar los problemas que puedan surgir, MPI realiza, si son necesarias, transformaciones de sintaxis que posibilitan la comunicación en entornos heterogéneos.
- La excepción la constituyen los datos de tipo MPI_BYTE, quese copian sin más de una máquina a otra.

LAM & MPICH

LAM (Local Area Multicomputer)

- Es una implementación open source del estándar MPI, en particular MPI-1 y varios elementos de MPI-2.
- Ofrece varias herramientas de monitoreo y depuración.
- Se ejecuta sobre una amplia variedad de plataformas Unix, estaciones de trabajo, grandes supercomputadores.
- Para el pasaje de mensajes puede usar TCP/IP, memoria compartida, Myrinet o Infiniband.

MPICH

- Es una implementación open source de la librería de pasaje de mensajes MPI.
- La principal característica es su adaptabilidad en gran número de plataformas, incluyendo clusters, GNU/Linux y M$/Windows, de workstations y procesadores masivamente paralelos (MPP).
- Soporta paralelismo del tipo SPMD (Single Program Multiple Data) y MPMD (Multiple Program Multiple Data).

MPICH

- La transferencia de mensajes se realiza de forma cooperativa.
- Permite establecer comunicación punto-a-punto o comunicación colectiva.
- Proporciona cuatro modos de comunicación: standard, synchronous, buffered y ready.

Ejemplo de programa con MPI

- Programa trivial:

```
#include "mpi.h"
#include <stdio.h>
int main( int argc, char **argv )
{
MPI_Init( &argc, &argv );
printf( "Hola Mundo\n" );
MPI_Finalize();
return 0;
}
```

- Compilación: mpicc –o hello hello.c

Ejecución de un programa MPI con MPICH

- El comando mpirun se usa para ejecutar los programas MPI en MPICH.
- mpirun admite diversas opciones. Destacaremos las siguientes:
  - -np N: N indica el número de procesos que se quiere en la ejecución del programa.
  - -p4pg pgfile: indica explicitamente las máquinas en las que se quiere correr y el path del programa y usuario que se usará en cada una de ellas.

Ejecución de un programa MPI con MPICH

- mpirun admite diversas opciones. Destacaremos las siguientes:
  - -machinesfile mfile: fichero de máquinas para correr en lugar del estándar (se usa en combinación con –np).
- Digamos que queremos correr nuestro programa en dos máquinas de la lista estándar:

```
mpirun -np 2 hello
```

Coming Next