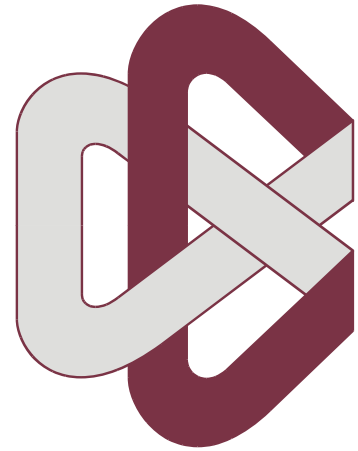


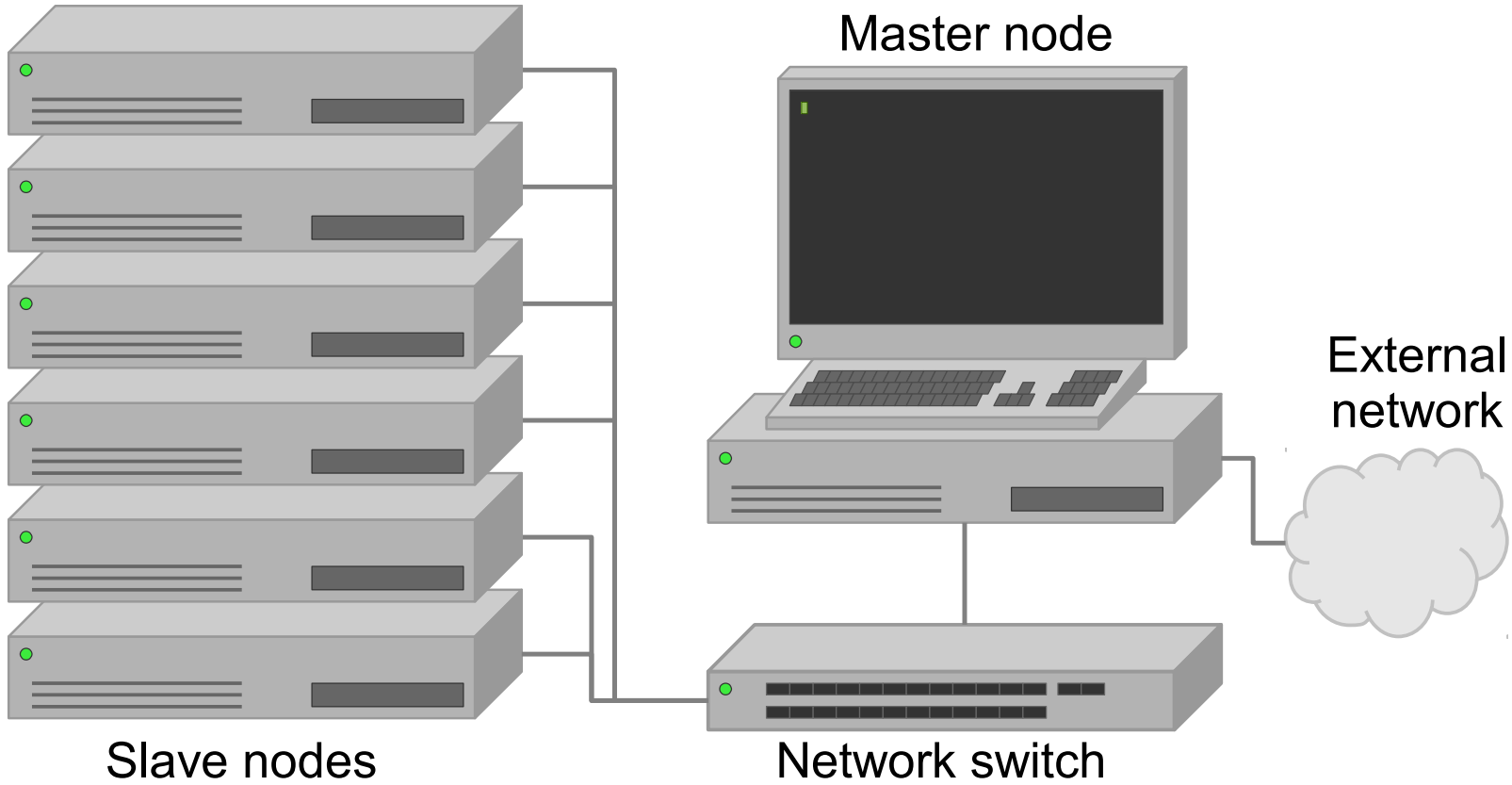
CIMAT

Cómputo en paralelo con MPI

Miguel Vargas-Félix

miguelvargas@cimat.mx
http://www.cimat.mx/~miguelvargas

Clusters Beowulf



Características:

- Tecnología estandar
- Fáciles de instalar
- Costo reducido

Existen sistemas operativos open source preparados para este tipo de arquitectura:

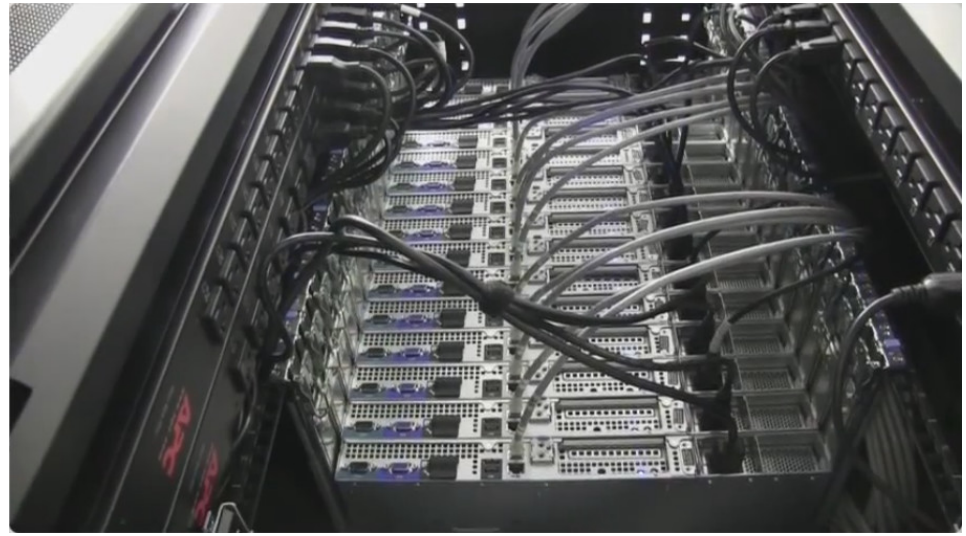
- Rocks Clusters: http://www.rocksclusters.org
- PelicanHPC: http://pareto.uab.es/mcreel/PelicanHPC
- Scientific Linux: http://www.scientificlinux.org

También es posible utilizar Windows

- Windows HPC Server 2008: http://www.microsoft.com/hpc

Cluster del CIMAT “El Insurgente”

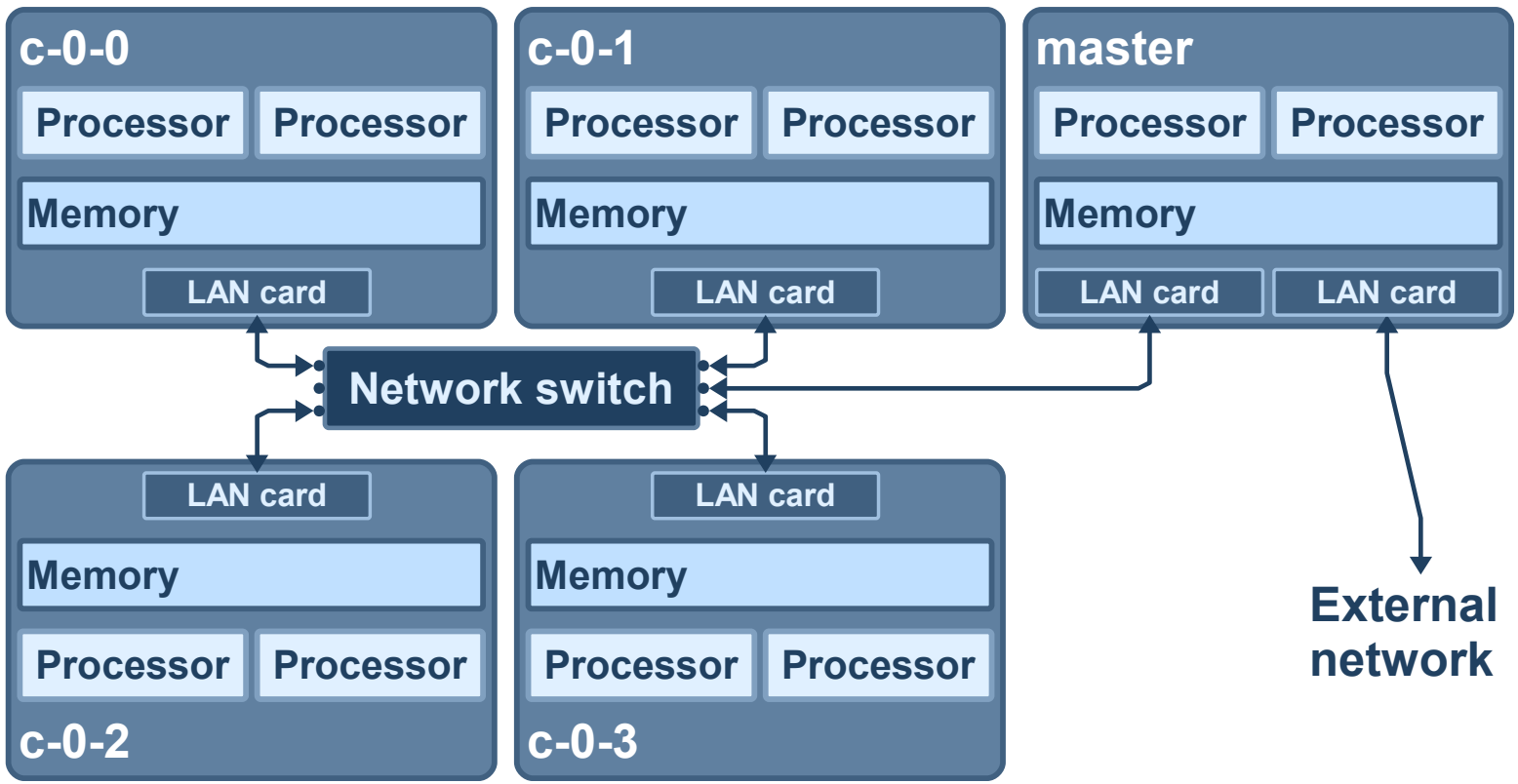
Clúster “El Insurgente”



MPI (Message Passing Interface)

Es una colección de funciones y programas para:

- Facilitar la comunicación entre procesos
- Ejecutar multiples veces un programa en un cluster



Usualmente se ejecuta un proceso por *core*.

Tiene soporte para threads y además es posible combinarlo con OpenMP

La comunicación entre procesos se realiza por medio de:

- Mensajes de red entre nodos (via Ethernet, Myrinet, InfiniBand)
- Memoria compartida (polling) entre procesos en una misma computadora
- Mensajes por sockets locales entre procesos en una misma computadora

Funciona con:

- C
- C++
- Fortran 77/90/95

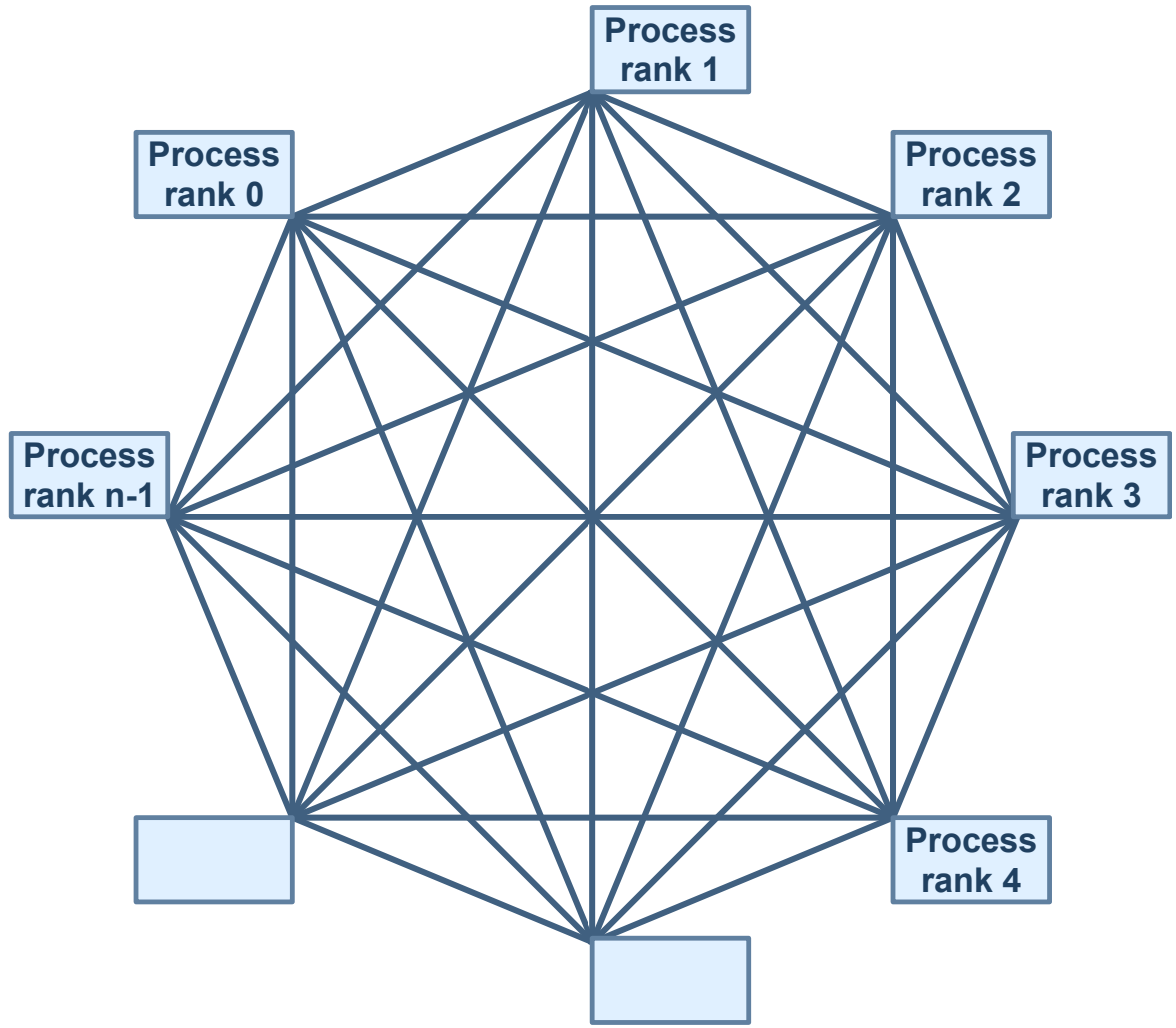
Es el estandard “de facto” para comunicaciones en clusters con aplicaciones al cómputo matemático.

Implementaciones open source:

- MPICH2: http://www.mcs.anl.gov/research/projects/mpich2
- Open-MPI: http://www.open-mpi.org

Comunicación entre procesos

Cada proceso se identifica por un número llamado rango (rank). Desde el punto de vista del software un proceso puede enviar y recibir mensajes de/y hacia cualquier otro proceso:



El programador no se tiene que preocupar por el medio de comunicación.

Cada nodo reserva memoria independientemente, MPI proporciona las funciones para “copiar” esa información a otros nodos y para recibir esa información.

MPI tiene funciones para:

- Enviar datos nodo a nodo
- Recibir datos nodo a nodo
- Enviar datos a muchos nodos (broadcasting)
- Recibir datos de muchos nodos (reduction)
- Sincronización de procesos
- Lectura/escritura de datos en paralelo
- Manejo de threads

Vamos a revisar brevemente 10 funciones básicas 4 de control y 6 de comunicación.

Un programa simple con MPI

El ejemplo siguiente muestra el programa mínimo con MPI.

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char** argv)
{
    int size, rank;

    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    printf("Hola mundo (size = %i, rank = %i)\n", size, rank);

    MPI_Finalize();

    return 0;
}
```

Compilación

En Windows con GCC y MPICH2

```
gcc -o hola hola.cpp -l mpi
```

En Linux con GCC y OpenMPI o MPICH2

```
mpicc -o hola hola.cpp
```

Para programas en C++

```
mpicxx -o hola hola.cpp
```

Ejecución

El comando para ejecutar es **mpiexec**, hay que especificar el número de procesos y el path al programa:

```
mpiexec -n 3 ./hola
```

./hola

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char** argv)
{
  int size, rank;

  MPI_Init(&argc, &argv);

  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  printf("Hola mundo (size=%i, rank=%i)\n",size,rank);

  MPI_Finalize();

 return 0;
}
```

```
Hola mundo (size=3, rank=0)
```

./hola

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char** argv)
{
  int size, rank;

  MPI_Init(&argc, &argv);

  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  printf("Hola mundo (size=%i, rank=%i)\n",size,rank);

  MPI_Finalize();

 return 0;
}
```

```
Hola mundo (size=3, rank=1)
```

./hola

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char** argv)
{
  int size, rank;

  MPI_Init(&argc, &argv);

  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  printf("Hola mundo (size=%i, rank=%i)\n",size,rank);

  MPI_Finalize();

 return 0;
}
```

```
Hola mundo (size=3, rank=2)
```

Funciones básicas

MPI_Init(<pointer_to_int>, <pointer_to_char*>)

Inicializa el proceso en MPI, bloquea la ejecución hasta que todos los procesos hayan sido inicializados.

MPI_Comm_size(*MPI_COMM_WORLD*, <pointer_to_int>)

Obtiene el número de procesos ejecutados en el grupo MPI_COMM_WORLD, éste es el grupo por default. Se pueden crear subgrupos de procesos.

MPI_Comm_rank(*MPI_COMM_WORLD*, <pointer_to_int>)

Obtiene el número de rango del proceso actual, los rangos van de 0 a $size-1$.

MPI_Finalize()

Termina la conección MPI, bloquea la ejecución hasta que todos los procesos terminen.

Comunicación con bloqueo

La comunicación con bloqueo significa que la ejecución del proceso se para hasta que se envíe o reciba el mensaje.

MPI_Send(<pointer_to_data>, <length>, <type>, <target>, <tag>, *MPI_COMM_WORLD*)

Envia los datos en <pointer_to_data> de tamaño <length> y tipo <type> al proceso <target>. El identificador del mensaje es <tag> (un entero).

MPI_Recv(<pointer_to_data>, <length>, <type>, <source>, <tag>, *MPI_COMM_WORLD*, *MPI_STATUS_IGNORE*)

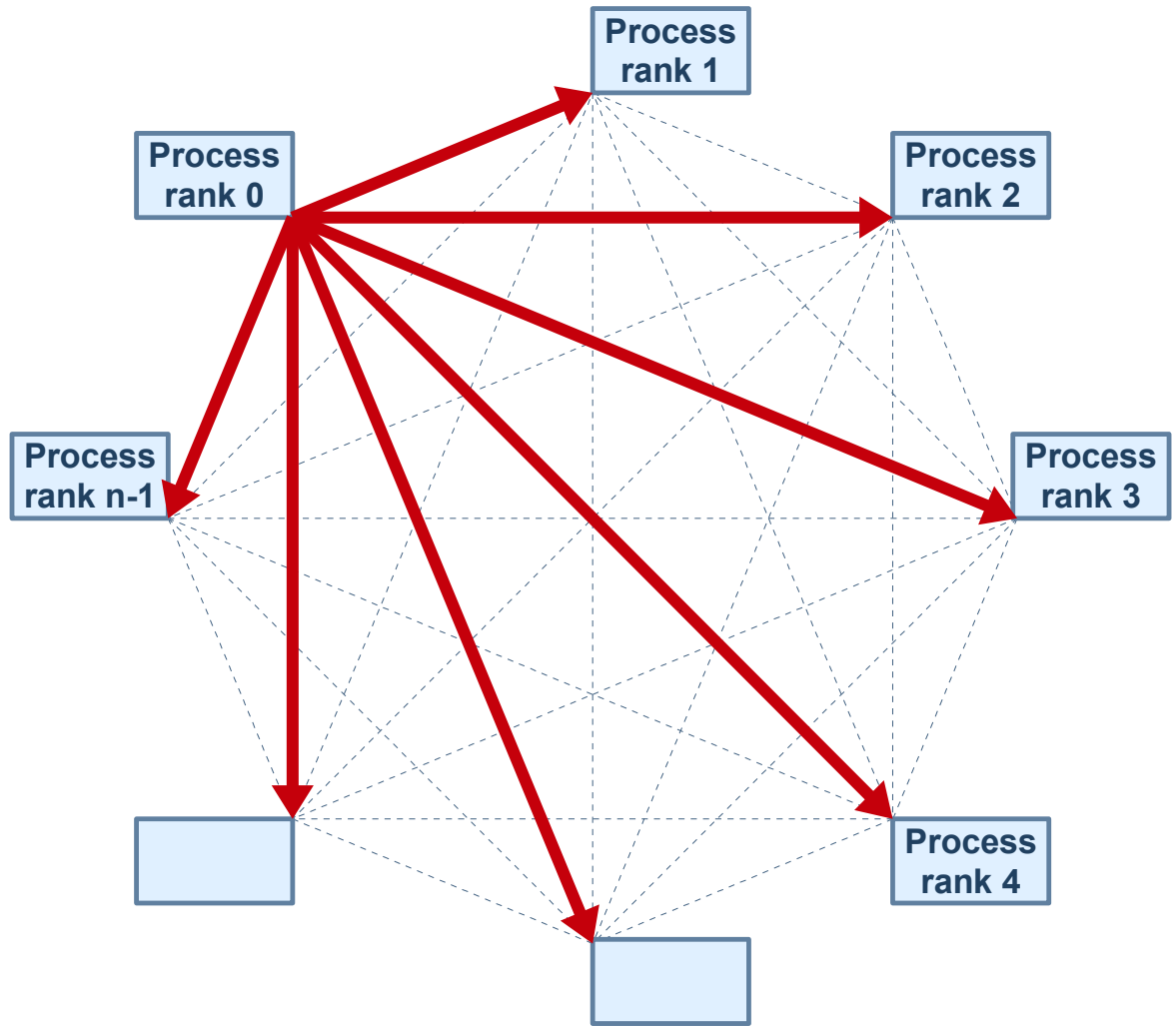
Recibe <length> datos de tipo <type> del proceso <source> y lo almacena en <pointer_to_data>. El identificador del mensaje es <tag> (un entero).

Los nombres predefinidos para los tipos de datos más comunes son:

MPI_CHAR	**char**
MPI_INT	**int**
MPI_UNSIGNED	**unsigned int**
MPI_LONG	**long**
MPI_UNSIGNED_LONG	**unsigned long**
MPI_FLOAT	**float**
MPI_DOUBLE	**double**

El siguiente es un ejemplo de comunicación con bloqueo en el que el nodo con rango 0 actuará como “maestro” y los nodos con rangos 1 a n-1 actuarán como “esclavos”.

El nodo maestro enviará un valor a cada nodo esclavo.



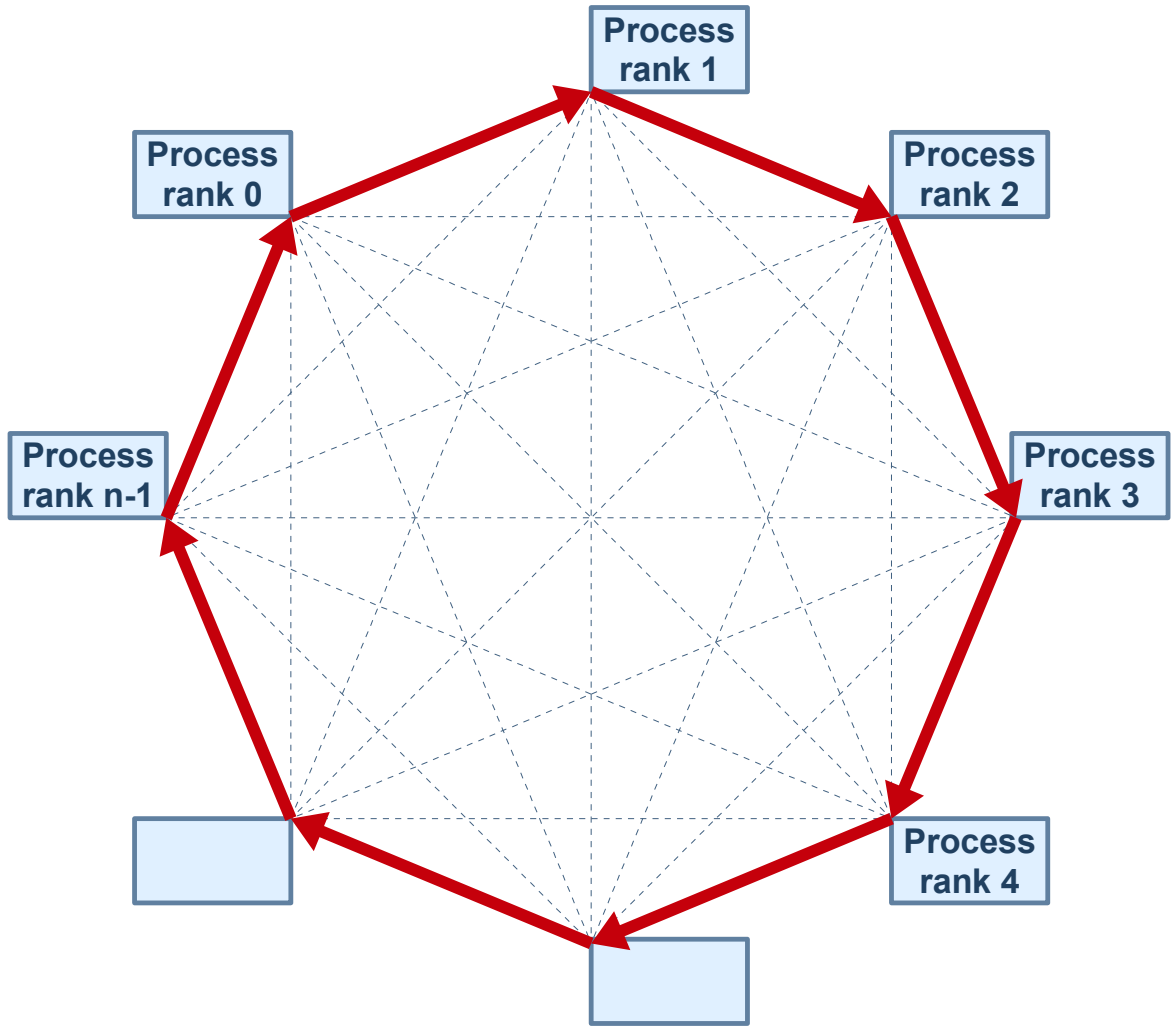
```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

#define TAG_DATA 73

int main(int argc, char** argv)
{
  int size;
  int rank;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank == 0) // Master
  {
    printf("Started %i processes\n", size);
    for (int s = 1; s < size; ++s)
    {
      double data = (double)rand()/RAND_MAX;
      MPI_Send(&data, 1, MPI_DOUBLE, s, TAG_DATA, MPI_COMM_WORLD);
      printf("Master sent %f to slave %i\n", data, s);
    }
  }
  else // Slaves
  {
    double data;
    MPI_Recv(&data, 1, MPI_DOUBLE, 0, TAG_DATA, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("Slave %i received %f\n", rank, data);
  }

  MPI_Finalize();
  return 0;
}
```

En el siguiente ejemplo, los esclavos reciben un vector. Primero reciben el tamaño del vector y luego el contenido del vector.

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

#define TAG_LENGTH 1
#define TAG_DATA   2

int main(int argc, char** argv)
{
  int size;
  int rank;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank == 0) // Master
  {
    srand(time(NULL));

    printf("MASTER (size = %i)\n", size);

    int length = rand()/(RAND_MAX/50);
    printf("Vector length = %i\n", length);

    double* data = new double[length];
    for (int s = 1; s < size; ++s)
    {
      MPI_Send(&length, 1, MPI_INT, s, TAG_LENGTH,
→       MPI_COMM_WORLD);
      for (int i = 1; i < length; ++i)
      {
        data[i] = (double)rand()/RAND_MAX;
      }
      MPI_Send(data, length, MPI_DOUBLE, s, TAG_DATA,
→       MPI_COMM_WORLD);
    }

    delete [] data;
  }
  else // Slaves
  {
    printf("SLAVE (rank = %i)\n", rank);

    int length;
    MPI_Recv(&length, 1, MPI_INT, 0, TAG_LENGTH,
→     MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    double* data = new double[length];
    MPI_Recv(data, length, MPI_DOUBLE, 0, TAG_DATA,
→     MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    delete [] data;
  }

  MPI_Finalize();
  return 0;
}
```

Comunicación sin bloqueo

En el ejemplo siguiente es un esquema “peer-to-peer”. Cada nodo envía un mensaje al nodo con el siguiente rango. La comunicación no puede hacerse con bloqueo. ¿Por qué?



En el siguiente código muestra la implementación de este ejemplo:

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

#define TAG_MESSAGE 1

int main(int argc, char** argv)
{
  int size;
  int rank;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  int send_data = rank + 100;
  MPI_Request send_request;
  MPI_Isend(&send_data, 1, MPI_INT, (rank + 1) % size, TAG_MESSAGE, MPI_COMM_WORLD, &send_request);

  int recv_data;
  MPI_Recv(&recv_data, 1, MPI_INT, (rank + size - 1) % size, TAG_MESSAGE, MPI_COMM_WORLD,
→ MPI_STATUS_IGNORE);

  printf("Peer %i received %i\n", rank, recv_data);

  MPI_Finalize();
  return 0;
}
```

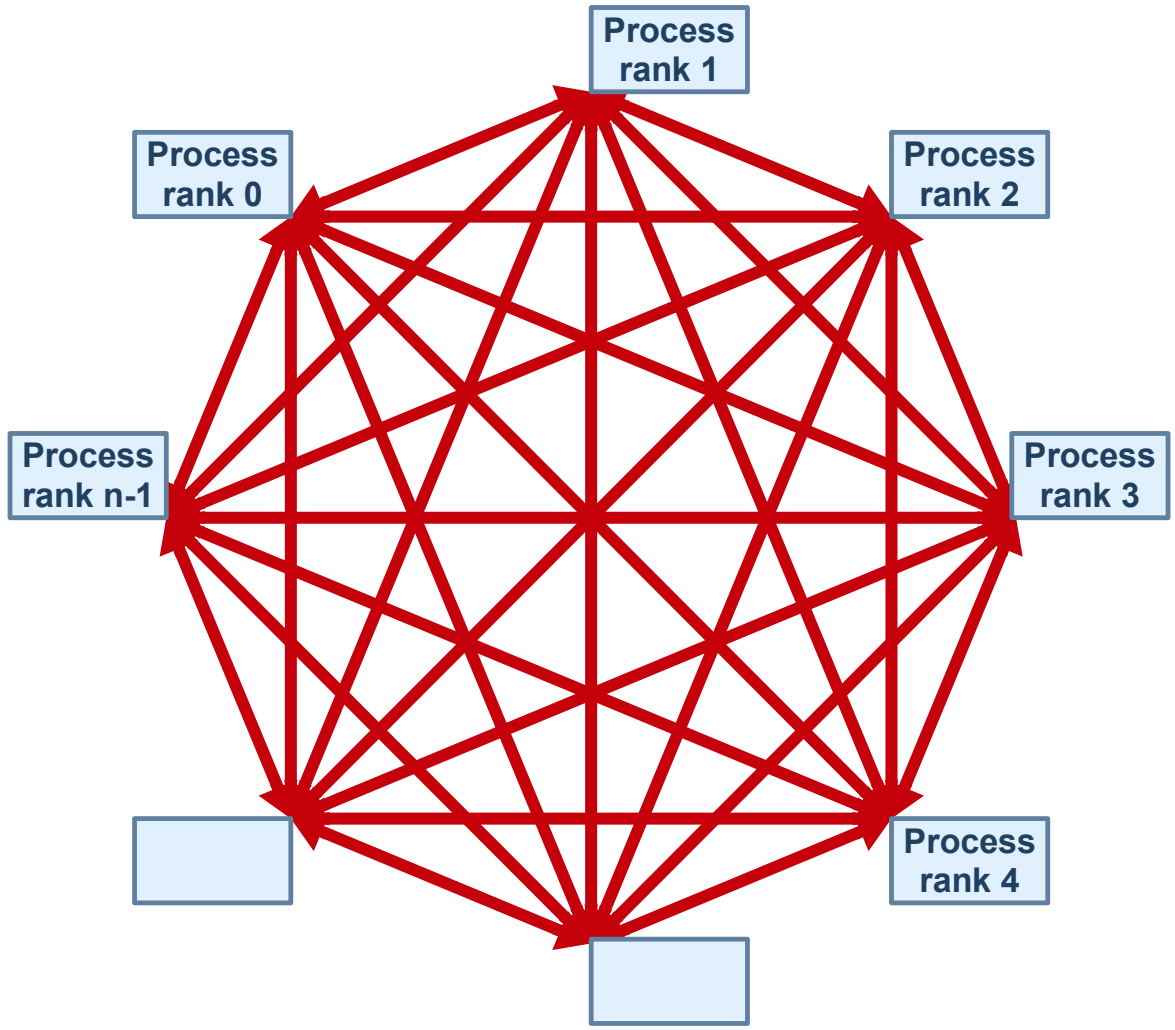
Introducimos el tipo de dato **MPI_Request** que es un identificador de solicitud. Sirve tanto para identificar solicitudes de envio o de recepción.

La nueva función utilizada es:

MPI_Isend(<pointer_to_data>, <length>, <type>, <target>, <tag>, *MPI_COMM_WORLD*, <pointer_to_request>)

Solicita enviar <pointer_to_data> a <target>, con mensaje <tag>. La solicitud se almacena en <pointer_to_request>

El siguiente es un ejemplo de una comunicación donde todos los nodos envian mensajes a todos los demás nodos.



Ahora necesitaremos arreglos de solicitudes (MPI_Request) para manejar las comunicaciones sin bloqueo.

A continuación el código de este ejemplo:

```
#include <stdio.h>
#include <mpi.h>

#define TAG_MESSAGE 1

int main(int argc, char** argv)
{
  int size;
  int rank;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  int* send_data = new int[size];
  int* recv_data = new int[size];

  MPI_Request* send_request = new MPI_Request[size];
  MPI_Request* recv_request =new MPI_Request[size];

  for (int r = 0; r < size; ++r)
  {
    if (r != rank)
    {
      send_data[r] = rank*1000 + r;
      MPI_Isend(&send_data[r], 1, MPI_INT, r, TAG_MESSAGE,
→       MPI_COMM_WORLD, &send_request[r]);
      MPI_Irecv(&recv_data[r], 1, MPI_INT, r, TAG_MESSAGE,
→       MPI_COMM_WORLD, &recv_request[r]);
    }
    else
    {
      send_request[r] = MPI_REQUEST_NULL;
      recv_request[r] = MPI_REQUEST_NULL;
    }
  }

  int received = 0;
  do
  {
    int r;
    MPI_Waitany(size, recv_request, &r, MPI_STATUS_IGNORE);
    printf("Peer %i received %i from %i\n", rank, recv_data[r], r);
    ++received;
  } while (received < size - 1);

  delete [] recv_request;
  delete [] send_request;
  delete [] recv_data;
  delete [] send_data;

  MPI_Finalize();
  return 0;
}
```

Las nuevas funciones utilizadas son:

MPI_Irecv(<pointer_to_data>, <length>, <type>, <source>, <tag>, *MPI_COMM_WORLD*, <pointer_to_request>)

Solicita recibir de <source> el mensaje <tag>, los datos se almacenarán en <pointer_to_data>. La solicitud se almacena en <pointer_to_request>

MPI_Waitany(<request_count>, <pointer_to_requests>, <pointer_to_int>, *MPI_STATUSES_IGNORE*)

Espera a que se complete una solicitud de un arreglo de solicitudes. El número de solicitud cumplida se almacena en <pointer_to_int>.

Alternamente a **MPI_Waitany** podemos utilizar **MPI_Waitall**.

```
#include <stdio.h>
#include <mpi.h>

#define TAG_MESSAGE 1

int main(int argc, char** argv)
{
  int size;
  int rank;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  int* send_data = new int[size];
  int* recv_data = new int[size];

  MPI_Request* send_request = new MPI_Request[size];
  MPI_Request* recv_request =new MPI_Request[size];

  for (int r = 0; r < size; ++r)
  {
    if (r != rank)
    {
      send_data[r] = rank*1000 + r;
      MPI_Isend(&send_data[r], 1, MPI_INT, r, TAG_MESSAGE,
→       MPI_COMM_WORLD, &send_request[r]);
      MPI_Irecv(&recv_data[r], 1, MPI_INT, r, TAG_MESSAGE,
→       MPI_COMM_WORLD, &recv_request[r]);
    }
    else
    {
      send_request[r] = MPI_REQUEST_NULL;
      recv_request[r] = MPI_REQUEST_NULL;
    }
  }

  MPI_Waitall(size, recv_request, MPI_STATUS_IGNORE);
  for (int r = 0; r < size; ++r)
  {
    if (r != rank)
    {
      printf("Peer %i received %i from %i\n", rank, recv_data[r], r);
    }
  }

  delete [] recv_request;
  delete [] send_request;
  delete [] recv_data;
  delete [] send_data;

  MPI_Finalize();
  return 0;
}
```

La nueva función es:

MPI_Waitall(<request_count>, <pointer_to_requests>, *MPI_STATUSES_IGNORE*)

Espera a que se completen todo un arreglo de solicitudes.

Depuración de programas MPI

Depurar programas con MPI puede ser difícil, procesos ejecutándose en varias computadoras al mismo tiempo, muchos gigabytes de información, problemas de sincronización, etc.

La forma más simple de hacer esto es hacer que los programas impriman mensajes.

Hay debugers comerciales:

- TotalView (GNU-Linux/Unix)
  http://www.totalviewtech.com/products/totalview.html
- Microsoft Visual Studio Professional Edition (Windows)
  http://www.microsoft.com/visualstudio/en-us/products/professional

También puede hacerse con herramientas gratuítas/libres como el GNU Debugger (GDB) y el Microsoft Visual Studio Express Edition.

A continuación vamos a mostrar varios trucos para depurar programas con MPI. Estos trucos solo funcionarán con unos pocos procesos (probablemente menos de cinco) ejecutándose en le misma computadora. Esto puede ser suficiente para pruebas conceptuales del código.

Tenemos que ejecutar el GDB de tal manera que sea éste el que llame nuestro programa. Para hacer esto visualmente más cómodo, vamos a ejecutar cada instacia del GDB en una ventana diferente.

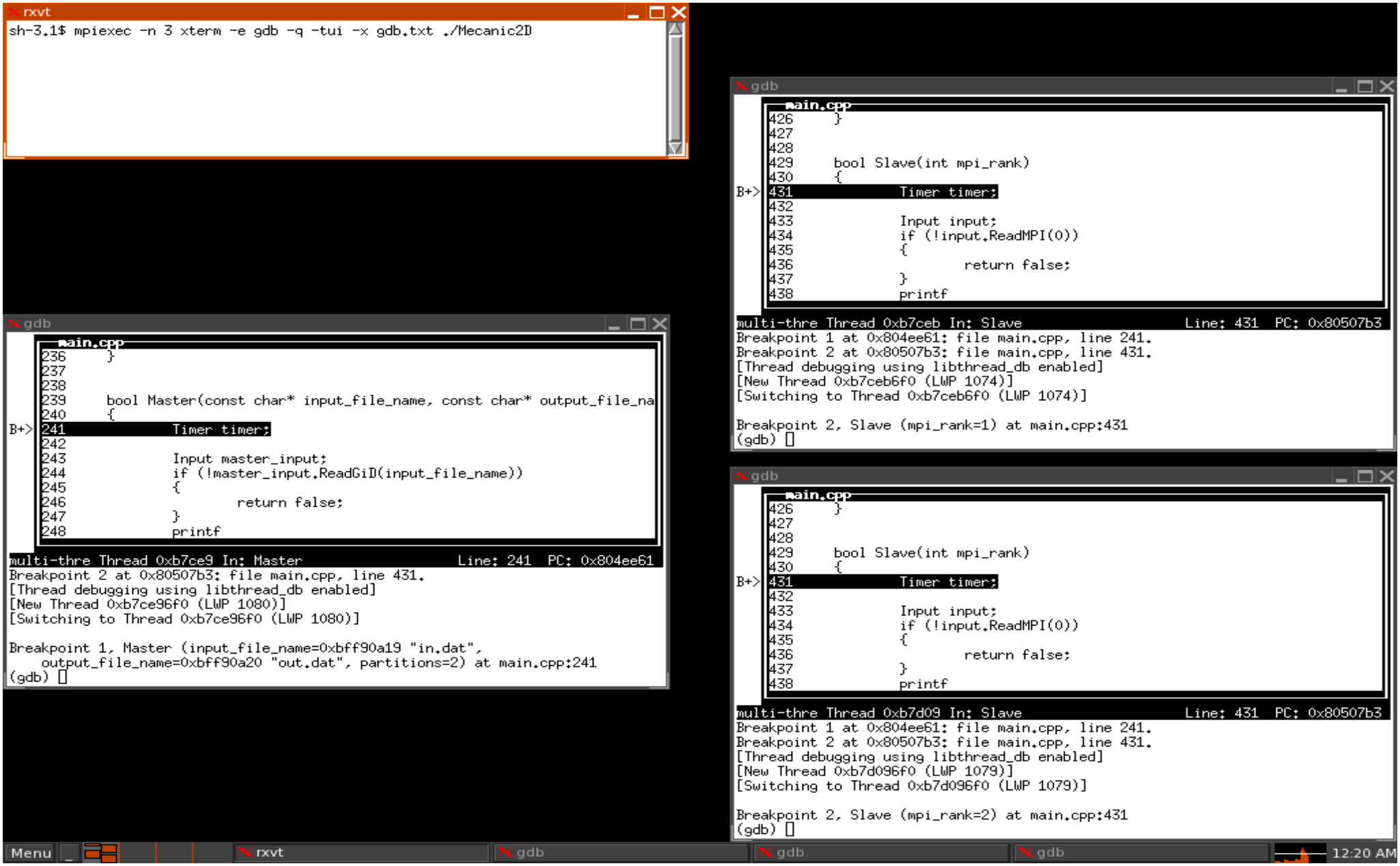
Primero hay que crear un archivo con los comandos para inicializar el GDB. Por ejemplo, el archivo “gdb.txt” pone “breakpoints” en las funciones “main” “Master” y “Slave”, además ejecutará el programa utilizando “in.dat” y “out.dat” como argumentos.

```
b main
b Master
b Slave
r in.dat out.dat
```

Para correr tres instancias de un programa llamando al GDB tres veces, cada una en una terminal diferente:

```
mpiexec -n 3 xterm -e gdb -q -tui -x gdb.txt ./Programa
```

La siguiente imagen muestra la ejecución de este ejemplo:



Windows

Tenemos que ejecutar el Visual Studio varias veces, una vez por cada instancia del programa.

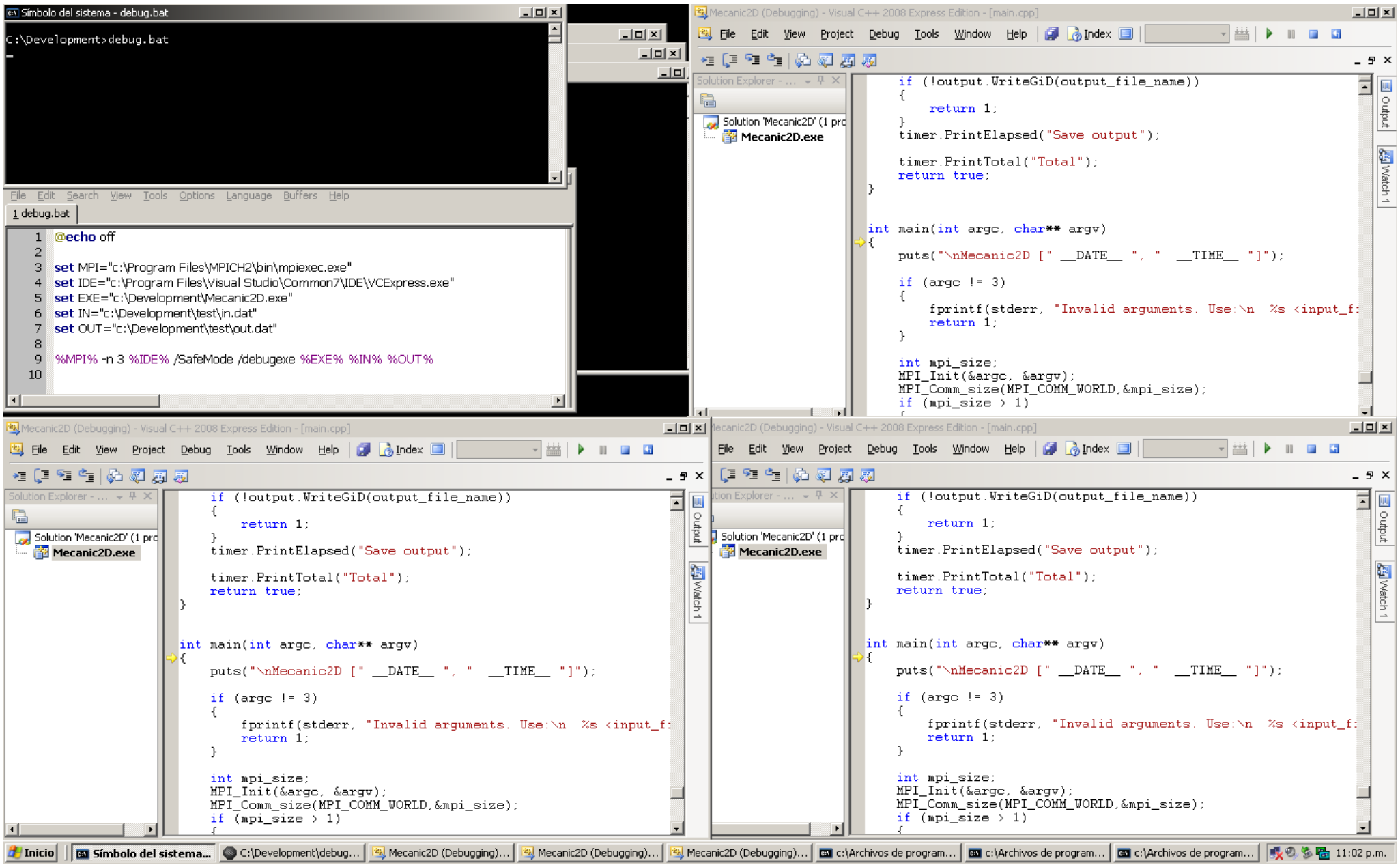
Para depurar es necesario que el código fuente del programa esté ubicado en el directorio donde se compilo el programa.

Hay que crear un archivo “batch” que contenga los parametros, la ruta del ejecutable, etc.

```
@echo off
set MPI = "c:\Program Files\MPICH2\bin\mpiexec.exe"
set IDE = "c:\Program Files\Visual Studio\Common7\IDE\VCExpress.exe"
set EXE = "c:\Development\Program.exe"
set IN = "c:\Development\test\in.dat"
set OUT = "c:\Development\test\out.dat"

%MPI% -n 3 %IDE% /SafeMode /debugexe %EXE% %IN% %OUT%
```

La base de datos de símbolos del programa (archivo .pdb) debe estar en el mismo directorio que el ejecutable.

La siguiente imagen muestra la ejecución de este ejemplo:



Ejecución en un cluster

Necesitamos un archivo “hosts”

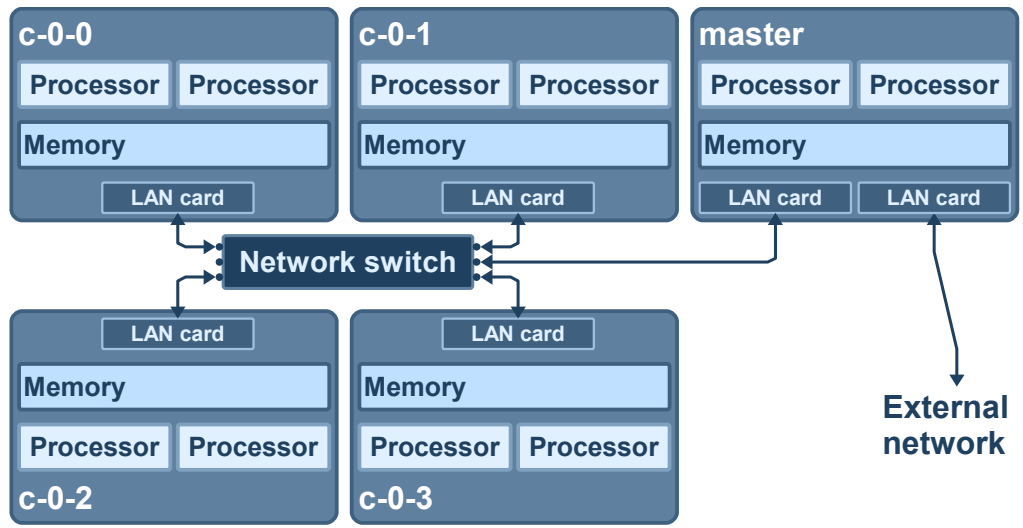
```
compute-0-0.local
compute-0-1.local
compute-0-2.local
compute-0-3.local
```

Ejemplo de archivo de hosts

La ejecución sería en la computadora “master.local” con:

```
mpiexe -n 4 -hostfile hosts <ejecutable>
```

De esta forma se ejecutaría una instancia del programa (proceso) en cada computadora esclavo.



Multiples procesos por nodo

Es posible ejecutar varios procesos por nodo.

Usando MPICH2 necesitamos un archivo “machines”

```
master.local:1
compute-0-0.local:2
compute-0-1.local:2
compute-0-2.local:2
compute-0-3.local:2
```

La ejecución sería:

```
mpiexe -n 9 -machinefile machines <ejecutable>
```

Usando OpenMPI se usa el archivo “hosts” con el formato:

```
master.local slots=1
compute-0-0.local slots=2
compute-0-1.local slots=2
compute-0-2.local slots=2
compute-0-3.local slots=2
```

La ejecución sería:

```
mpiexe -n 9 -hostfile hosts <ejecutable>
```

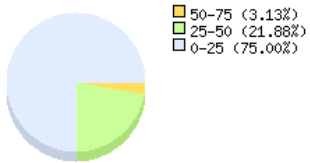
El estado del cluster

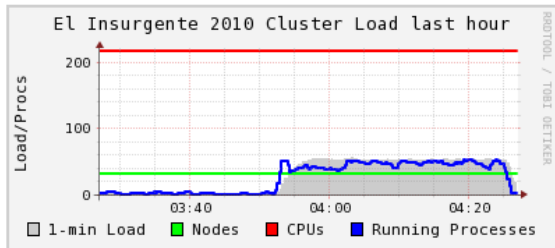
CPUs Total: **216**
Hosts up: **32**
Hosts down: **0**

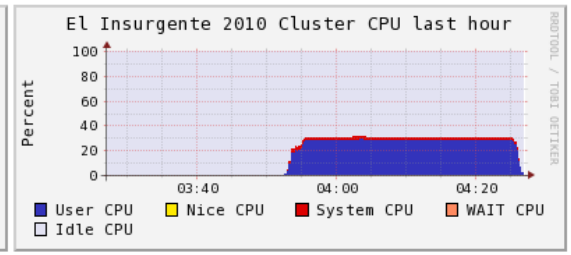
Avg Load (15, 5, 1m):
19%, 19%, 8%
Localtime:
2010-10-20 04:26

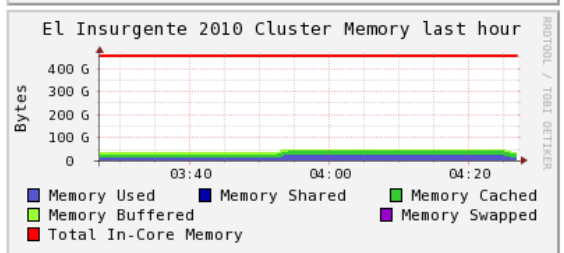
Rocks Tools:
Job Queue | **Cluster Top**

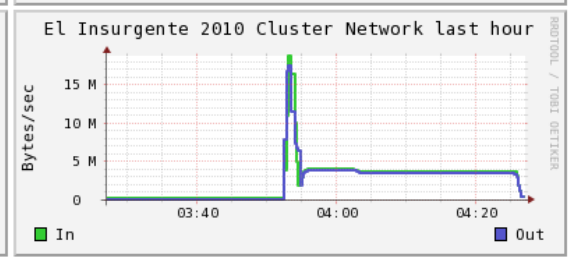
Cluster Load Percentages

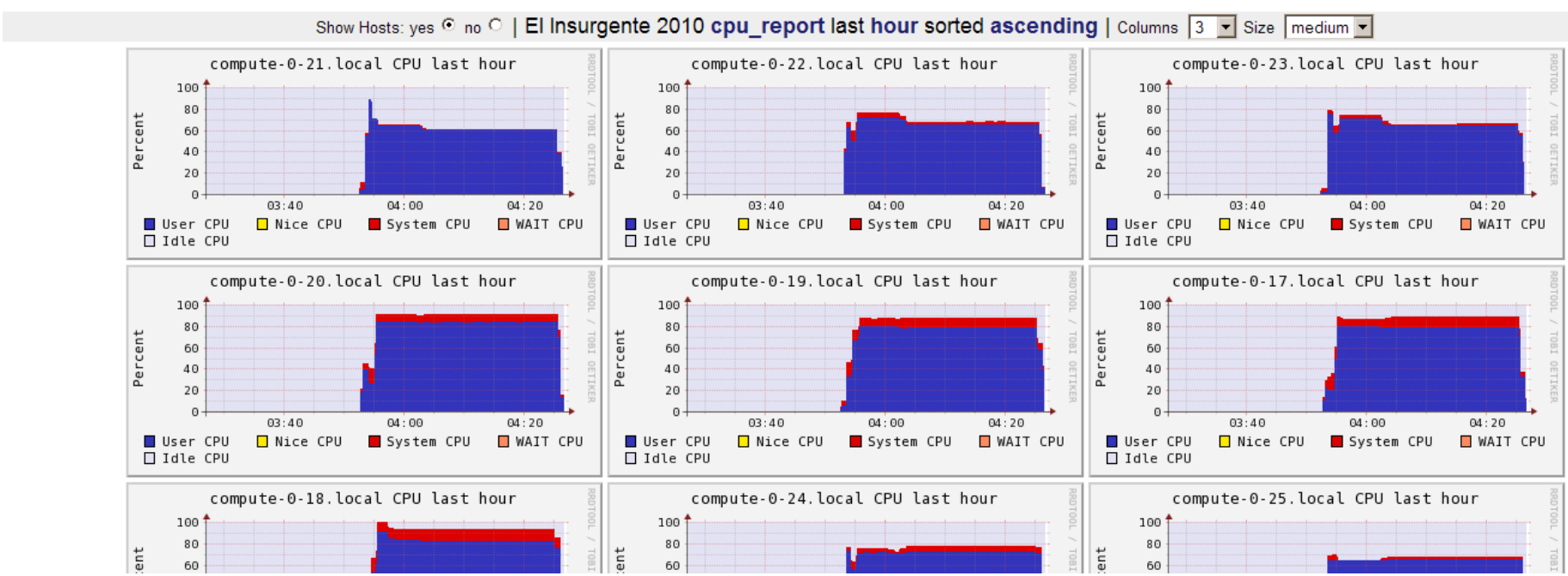












Como instalar Open-MPI desde el código fuente

Bajar el código fuente de OpenMPI de:

http://www.open-mpi.org/software/ompi/v1.6/downloads/openmpi-1.6.2.tar.bz2

Ir al directorio donde se bajó OpenMPI y ejecutar:

```
tar xjf openmpi-1.6.2.tar.bz2
cd openmpi-1.6.2
./configure -q --prefix=$HOME/local --disable-mpi-cxx --disable-mpi-cxx-
seek --disable-mpi-f77 --disable-mpi-f90 --disable-mpi-profile --disable-
shared --enable-static
make
make install
```

Esto compila e instala OpenMPI en su directorio "*home*/local".

Para poder utilizar esta instalación de OpenMPI hay que actualizar el PATH:

```
export PATH=$HOME/local/bin:$PATH
```

Para compilar el hay que usar:

```
mpicxx -o programa programa.cpp
```

Para ejecutar:

```
mpiexec -n 9 programa
```

¿Preguntas?

miguelvargas@cimat.mx

Referencias

[Ster95] T. Sterling, D. J. Becker, D. Savarese, J. E. Dorband, U. A. Ranawake, C. V. Packer. BEOWULF: A Parallel Workstation For Scientific Computation. Proceedings of the 24th International Conference on Parallel Processing, 1995.

[MPIF08] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, Version 2.1. University of Tennessee, 2008.