Advanced Middleware: MPI & Interactivity

Grids & e-Science 2009

UIMP, Santander

Enol Fernández del Castillo

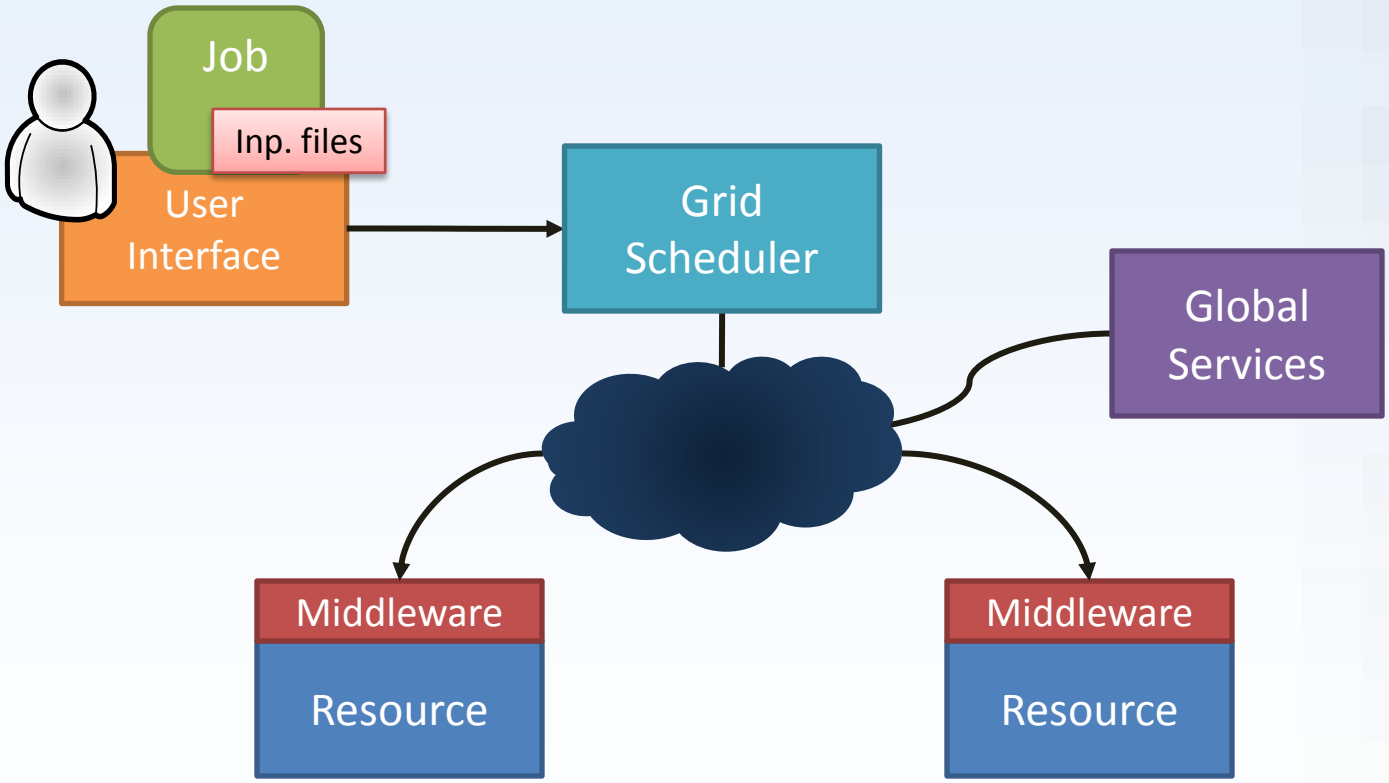
Instituto de Física de Cantabria

Instituto de Física de Cantabria

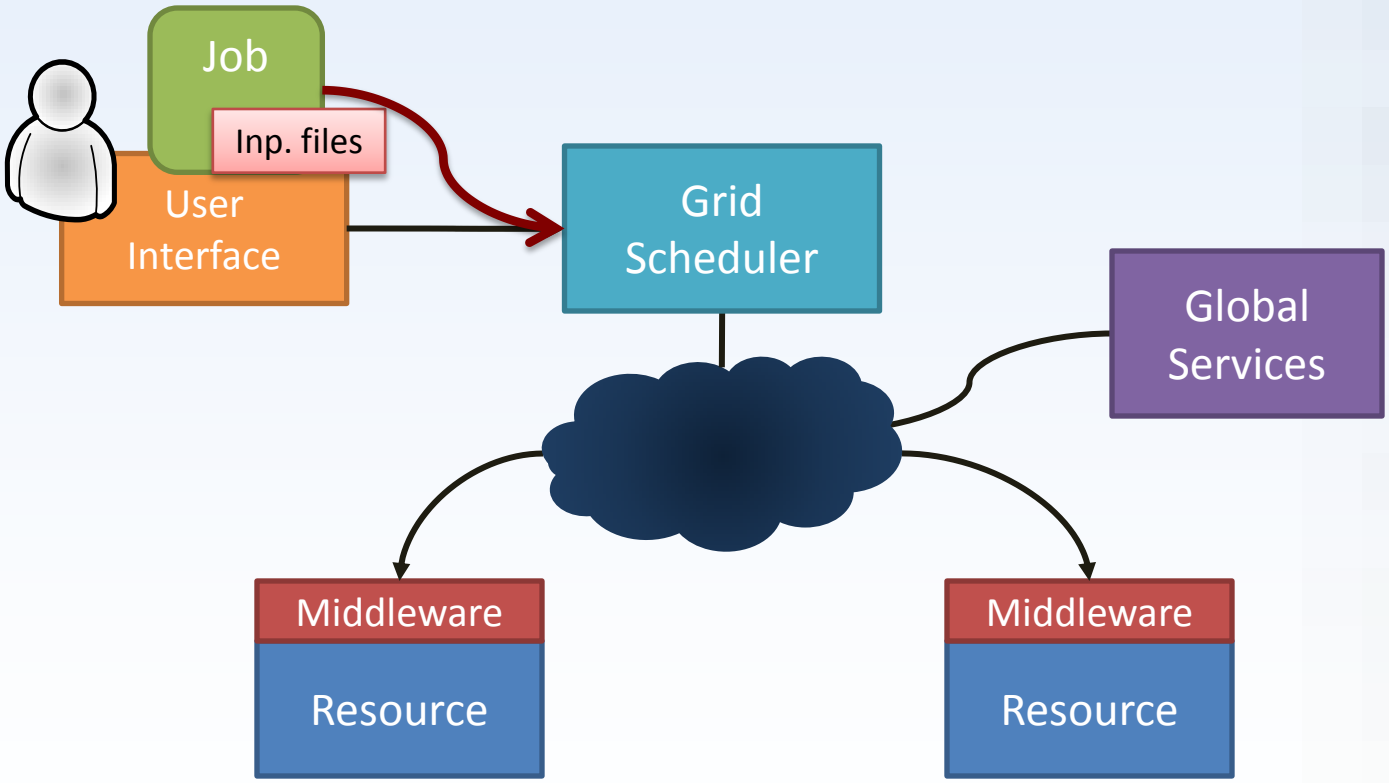
Outline

- Introduction
- MPI
- PACX-MPI
- MPI-START
- CrossBroker
  - Parallel jobs
  - Interactivity with i2glogin

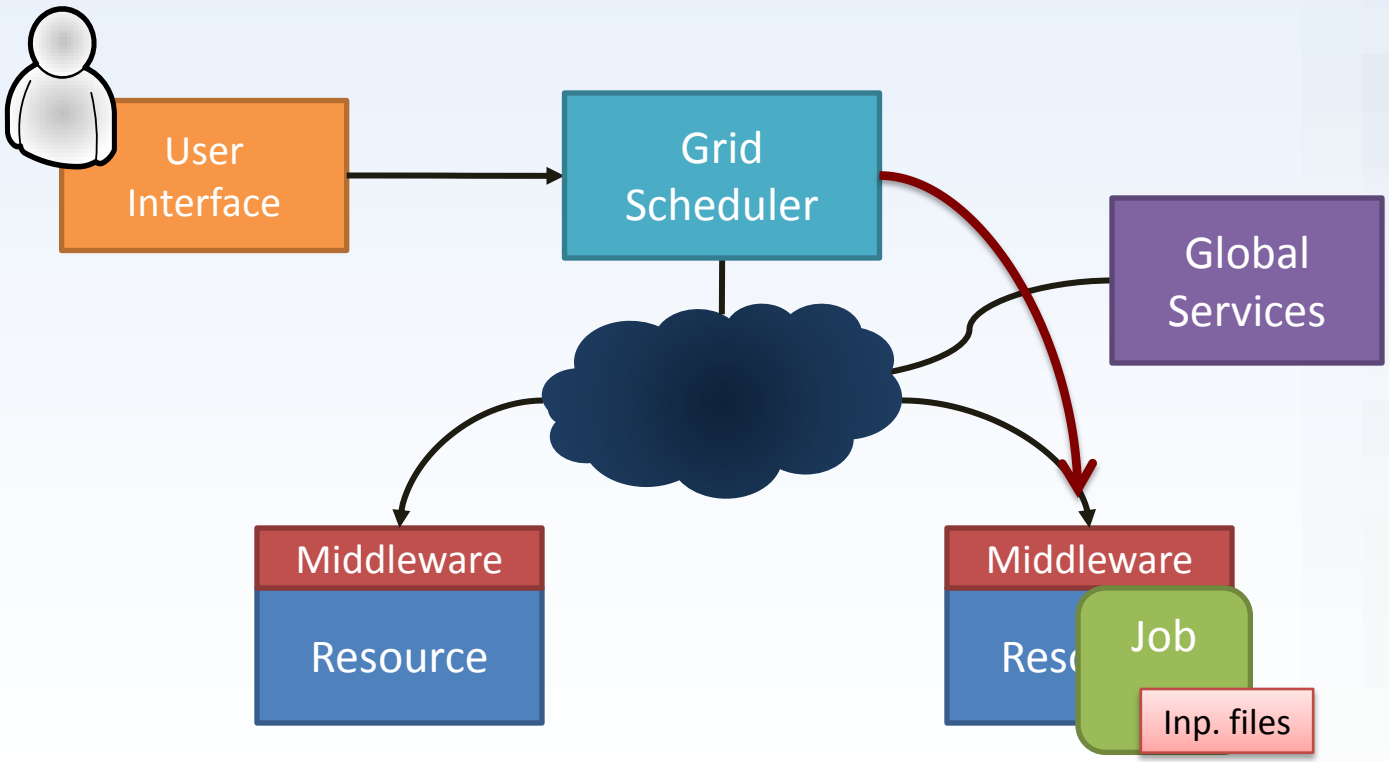
Batch Execution on Grids



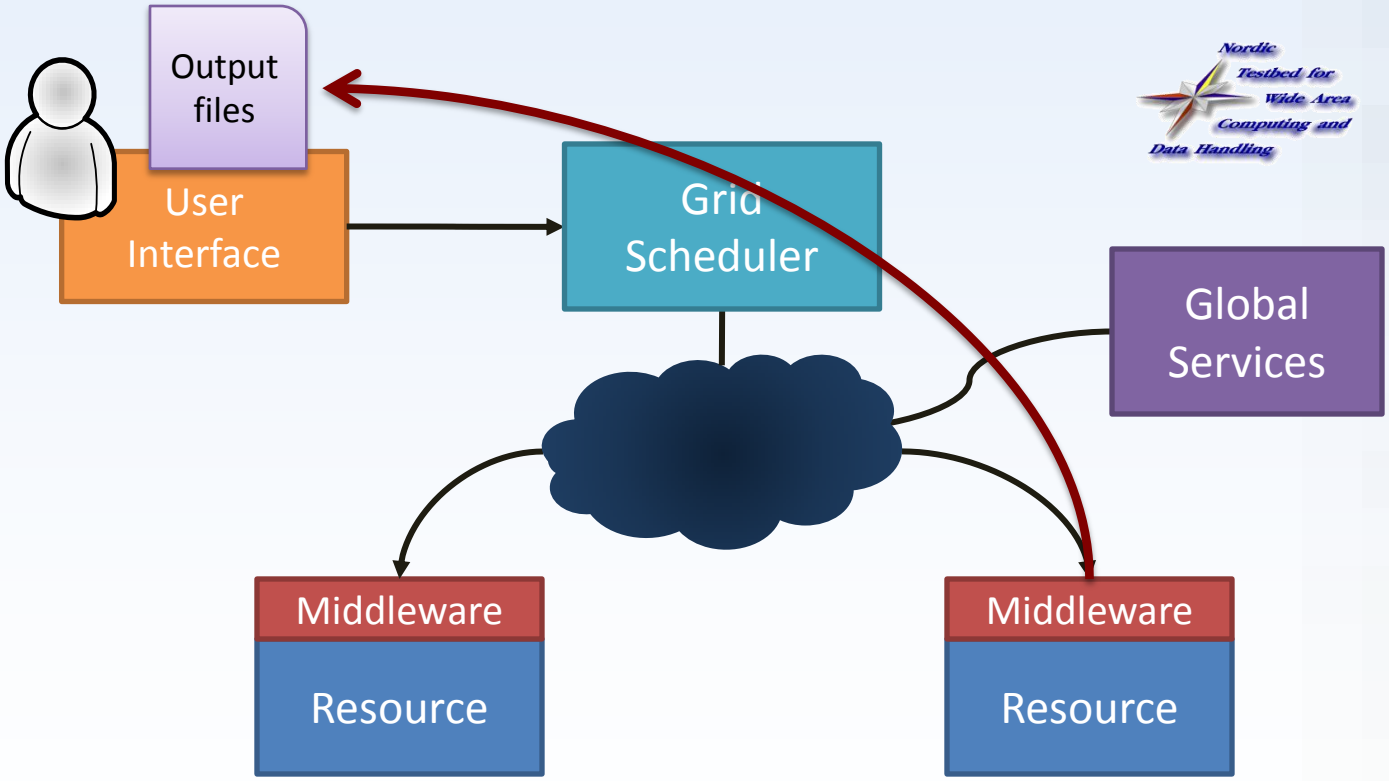
Batch Execution on Grids



Batch Execution on Grids



Batch Execution on Grids



But…

- Users need more computing power:
  - Using more than one core for execution
  - Or even more than one site for execution
- Users need to interact with the application:
  - Monitoring the output of the app
  - Or even changing the behavior of the app while it is running

Parallel Jobs

- Parallel jobs use more than one core… how to use all the cores efficiently?
  - **Shared memory**: all cores access a common data area
  - **Message Passing**: cores interchange messages with the data
- MPI (Message Passing Interface) provides a standard interface for programming parallel jobs

MPI

- Defines uniform and standard API (vendor neutral) for message passing
- Allows efficient implementation
- Provides C, C++ and Fortran bindings
- Several MPI implementations available:
  - From hardware providers (IBM, HP, SGI….) optimized for their systems
  - Academic implementations
- Most extended implementations:
  - MPICH (from ANL/MSU), includes support for a wide range of devices (even using globus from communication)
  - Open MPI (join effort from FT-MPI, LA-MPI, LAN/MPI and PACX-MPI developers): modular implementation that allows the use of advanced hardware during runtime

MPI-1 & MPI-2

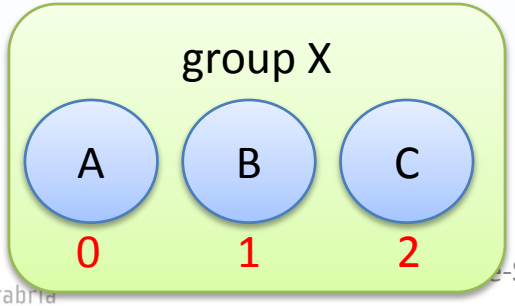
- MPI-1 standard includes:
  - Point to point communication
  - Collective operations
  - Process groups and topologies
  - Communication contexts
  - Datatype Management
- MPI-2 adds:
  - Dynamic Process Management
  - File I/O
  - One Sided Communicactions
  - Extension of Collective Operations

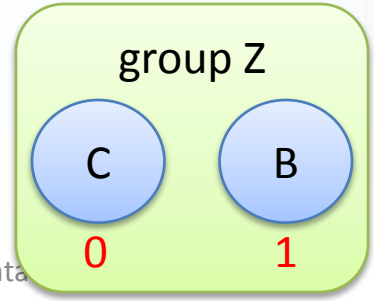
Processes

- Every MPI job consist of N processes
  - 1 <= N , N == 1 is valid
- Each process in a Job could execute a different binary
  - In general it's always the same binary which executes different code path based on the process number
- Processes are included in **groups**

Groups

- A group is an ordered set of processes
- Every process in a group has an unique rank inside the group
  - From 0 to #PROCS -1
  - Processes can have different ranks in different groups
- Groups are defined with MPI Communicators





Hello World

```
#include <mpi.h>    /* for MPI functions */
#include <stdio.h>  /* for printf */

int main(int argc, char *argv[] {
   int rank = 0, size = 0;
   MPI_Init(&argc, &argv); MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   printf("Hello my rank is %i of %i\n", rank, size);

   MPI_Finalize();
   return 0;
}
```

Compiling

- Basics
  - C programs have to include mpi.h
  - Fortran programs include mpif.h
  - All MPI functions/symbols prefixed with “MPI_”
- Compiling/Linking is implementations specific
  - Most of them provide wrapper compilers
  - mpicc, mpicxx, mpif90… add required flags and libraries
  - Just use mpicc instead of gcc:

```
mpicc -o hello hello.c
```

Hello World

- MPI_Init
  - initialize the MPI system
  - must be called before any other MPI function can be called
- MPI_Comm_size
  - return the number of processes in the processes group
  - MPI_COMM_WORLD is a default group with all processes
- MPI_Comm_rank
  - return the rank of the current process in the group
- MPI_Finalize
  - Shutdown the MPI system
  - After this call MPI must not be called

Hello World

```
#include <mpi.h>    /* for MPI functions */
#include <stdio.h>  /* for printf */

int main(int argc, char *argv[] {
   int rank = 0, size = 0;
   MPI_Init(&argc, &argv); MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   printf("Hello my rank is %i of %i\n", rank, size);

   MPI_Finalize();
   return 0;
}
```

```
Hello my rank is 2 of 5
Hello my rank is 1 of 5
Hello my rank is 0 of 5
Hello my rank is 4 of 5
Hello my rank is 3 of 5
```

Communication between processes

- Two types of processes communication:
  - Point-to-point: the source process knows the rank of the destination process and sends message directed to it.
  - Collective: all the processes in the group are involved in the communication.

- Most functions are *blocking.* That is: the process waits until it has received completely the message.

Point-to-point

- MPI_Send:

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
   int dest, int tag, MPI_Comm comm)
```

  - When MPI_Send return it means that the user can safely re-use the buffer, but not that the send operation already completed of even that the remote process received the data

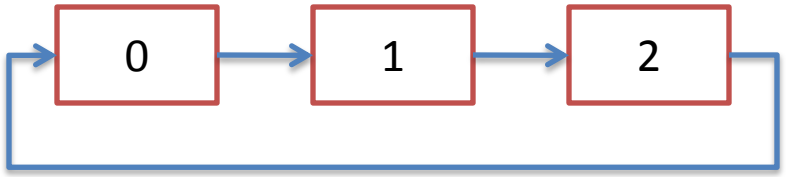
- MPI_Recv:

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,  int source, i
   nt tag, MPI_Comm comm, MPI_Status *status)
```

  - When MPI_Recv return the data has been received into the specified buffer (check status for possible errors)

Point-to-point: ring example

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

int main(int argc, char *argv[]) {
    MPI_Status status;
    MPI_Request request;
    int rank = 0, size = 0;
    int next = 0, prev = 0;
    int data = 0;

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

    prev = (rank + (size-1)) % size;
    next = (rank + 1) % size;

    MPI_Send(&rank, 1, MPI_INT, next, 0, MPI_COMM_WORLD);
    MPI_Recv(&data, 1, MPI_INT, prev, 0, MPI_COMM_WORLD, &status);

    printf("%i received %i\n", rank, data);
    MPI_Finalize();
    return 0;
}
```



Point-to-point

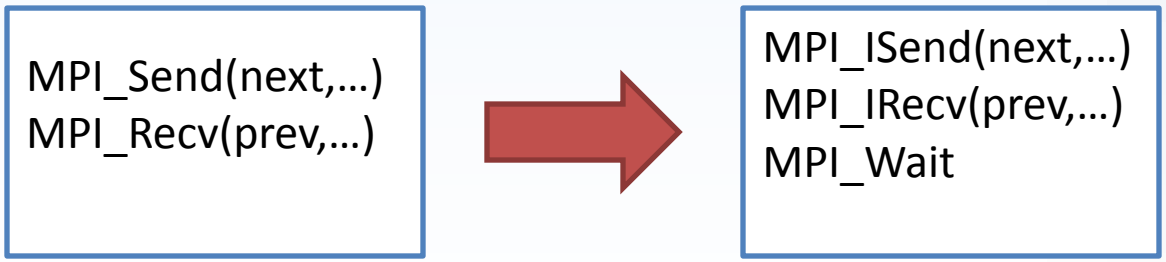
- Non blocking:
  - MPI_Isend/MPI_Irecv
    - return as fast as possible,
      - Operation already finished
      - Operation is on going
      - Operation not even started
    - return a request to be used for testing of the completion of the operation

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int
    tag, MPI_Comm comm, MPI_Request *request);
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
   int source, int tag, MPI_Comm comm, MPI_Request *request);
```

Point-to-point

- Checking status:
  - MPI_Wait: return when the operation is complete
  - MPI_Test: non-blocking version of MPI_Wait

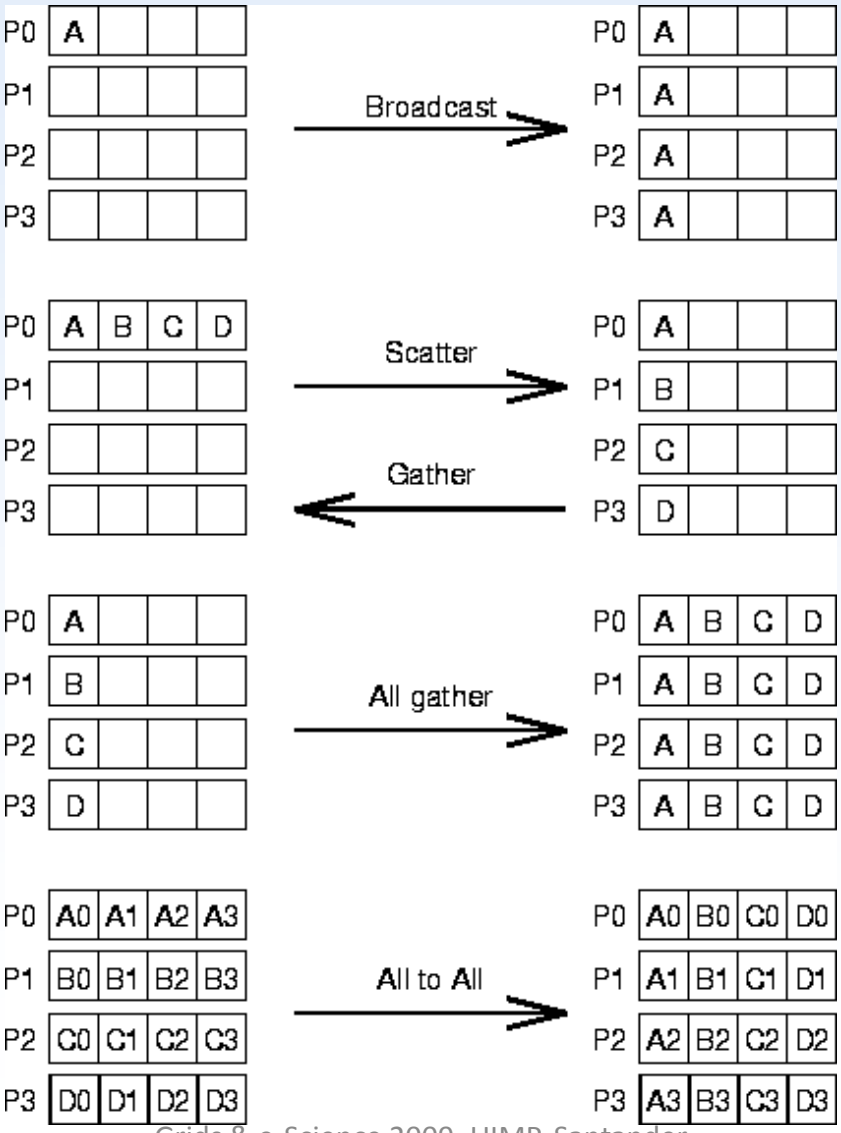
```
int MPI_Wait(MPI_Request *req, MPI_Status *st)
int MPI_Test(MPI_Request *req, int *flag,  MPI_Status *st)
```



Collective Operations

- MPI provides several collective operations
  - All processes of a communicators have to call it
  - Rapid software development
    - User does not have to implement the algorithms again
  - Better performance
    - Allows the implementation to exploit system specific features
    - The implementation can select the best possible algorithm for the operation

Collective Operations



Collective Operations: Reduce

- MPI_Reduce:
  - Applies one OP on the input buffer of all processes and stores result in the output buffer on the root processes
  - Predefined operations
    - MPI_MIN/MPI_MINLOC/MPI_MAX/MPI_MAXLOC
    - MPI_SUM/MPI_PROD
    - MPI_LAND
    - …
  - User defined operations

Collective Operations Example

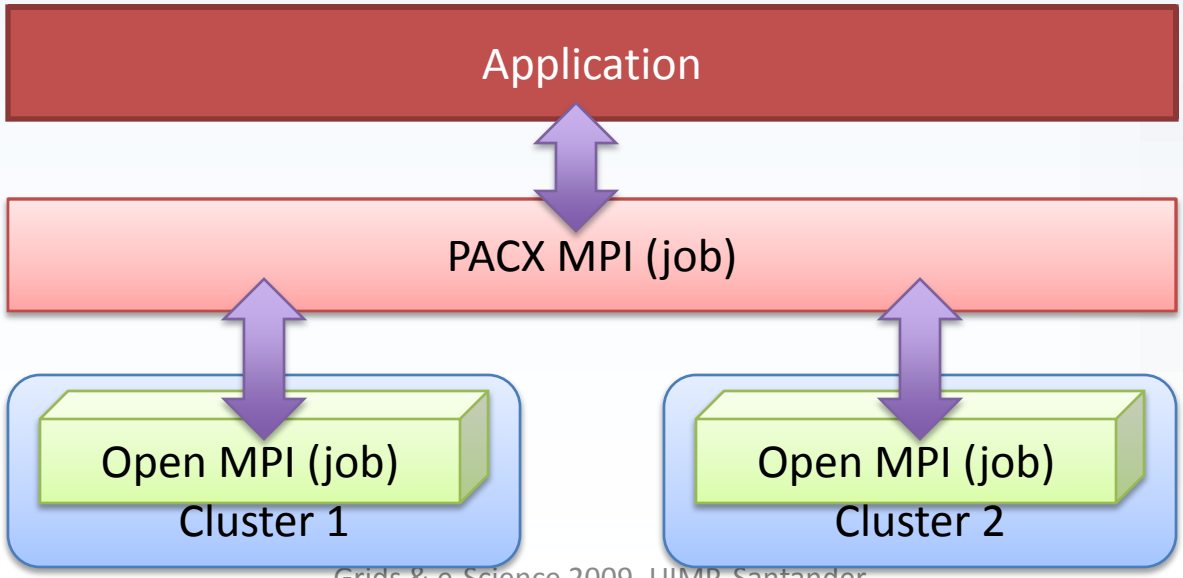
```
int main(int argc,char *argv[])
{
    int    n, myid, numprocs, i;
    double mypi, pi, h, sum, x;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);

    n = 10000;              /* default # of rectangles */

    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

    h   = 1.0 / (double) n;
    sum = 0.0;
    for (i = myid + 1; i <= n; i += numprocs) {
         x = h * ((double)i - 0.5);
         sum += 4/(1+x*x);
    }
    mypi = h * sum;

    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    if (myid == 0) {
           printf("pi is approximately %.16f\n", pi);
     }

    MPI_Finalize();
    return 0;
}
```

MPI-2

- Parallel I/O
  - Allows several process to access data (read or write) from a common file
  - High performance
- DPM (Dynamic Process Management)
  - Supports the creation of new processes during runtime and communicate with them
- One sided operations
  - Remote Memory Access
  - Can provide good performance on hardware paltforms with this kind of communication (Infiniband, Myrinet)

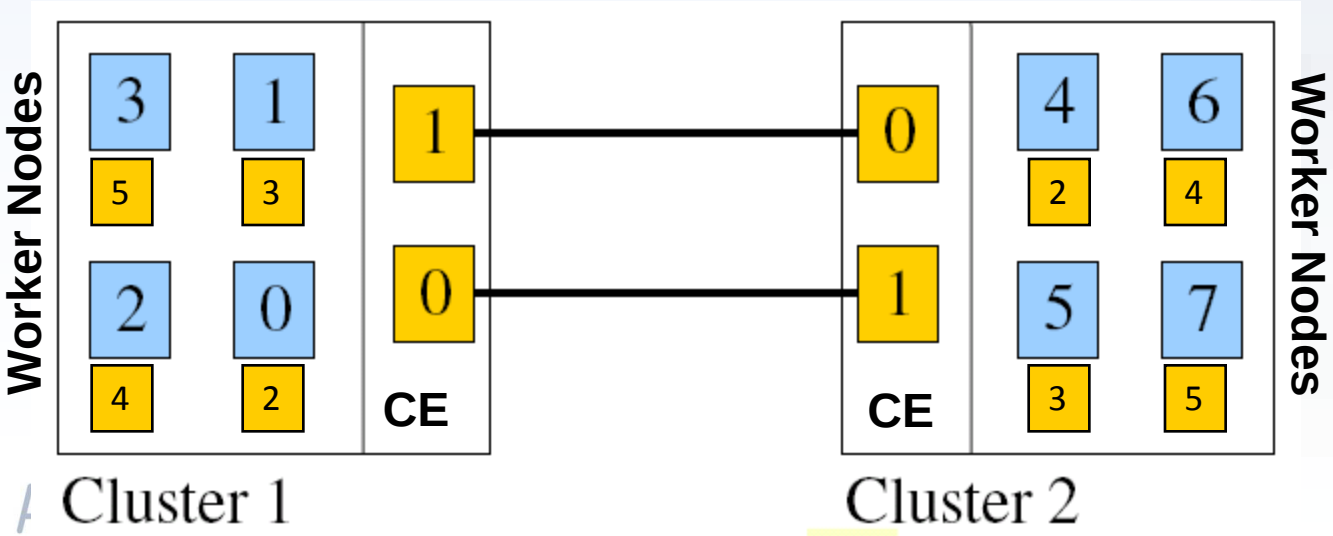
Pacx MPI

- Middleware to run MPI applications on a network of parallel computers
  - Starts MPI jobs in each cluster
  - PACX merges them into a bigger/unique MPI job
- PACX conforms to the MPI standard
  - Applications just need to be recompiled!



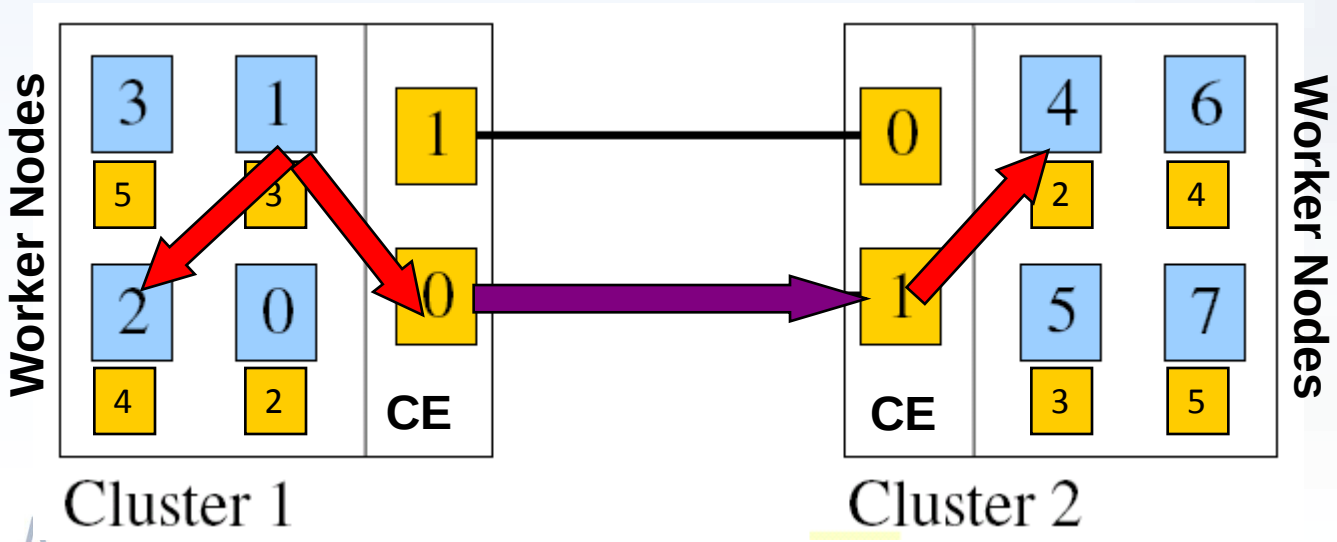
Pacx MPI communication

- Pacx-MPI maps the MPI ranks of the big job to the MPI processes running on each cluster
- Pacx-MPI maps 2 additional "**hidden**" processes on the local MPI jobs for external communication
  - Rank 0 of the local MPI jobs is always the "out" daemon
  - Rank 1 of the local MPI jobs is always the "in" daemon



Pacx MPI communication

- Internal Communication
  - Communication between processes running inside the same local cluster is performed via the local, optimized MPI implementation
- External Communication
  - Send message to “out” daemon using local MPI
  - “out” daemon send message to destination host over the network using a protocol (TCP)
  - The “in” daemon send message to destination using local MPI

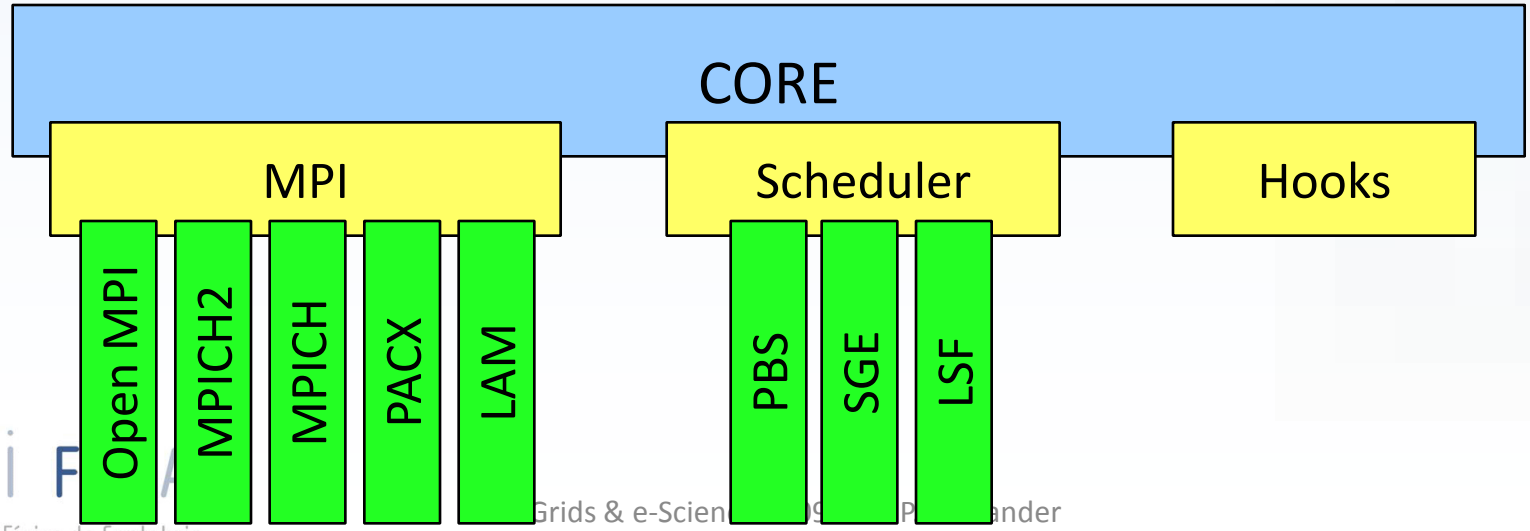


Executing MPI (on the Grid)

- There is no standard way of starting an MPI application
  - No common syntax for mpirun, mpiexec support optional
- The cluster where the MPI job is supposed to run doesn't have a shared file system
  - How to distribute the binary and input files?
  - How to gather the output?
- Different clusters over the Grid are managed by different Local Resource Management Systems (PBS, LSF, SGE,…)
  - Where is the list of machines that the job can use?
  - What is the correct format for this list?
- How to compile MPI program?
  - How can a physicist working on Windows workstation compile his code for/with an Itanium MPI implementation?

MPI-START

- Specifies a unique interface to the upper layer in the middleware to describe MPI jobs
- Support basic file distributions
- Implemented as portable shell scripts
- Extensible via user hooks and plugins at the site level
- User specifies the job characteristics using env variables



MPI-START variables

- Interface Intra Cluster MPI:
  - I2G_MPI_APPLICATION:
    - The executable
  - I2G_MPI_APPLICATION_ARGS:
    - The parameters to be passed to the executable
  - I2G_MPI_TYPE:
    - The MPI implementation to use (e.g openmpi, ...)
  - I2G_MPI_VERSION:
    - Specifies which version of the the MPI implementation to use. If not defined the default version will be used

MPI-START variables

- Interface Intra Cluster MPI
  - I2G_MPI_PRECOMMAND
    - Specifies a command that is prepended to the mpirun (e.g. time).
  - I2G_MPI_PRE_RUN_HOOK
    - Points to a shell script that must contain a “pre_run_hook” function.
    - This function will be called before the parallel application is started (usage: compilation of the executable)
  - I2G_MPI_POST_RUN_HOOK
    - Like I2G_MPI_PRE_RUN_HOOK, but the script must define a “post_run_hook” that is called after the parallel application finished (usage: upload of results).

MPI-START invocation

- **The script:**

```
[imain179@i2g-ce01 ~]$ cat test2mpistart.sh
#!/bin/sh
# This is a script to show how mpi-start is called

# Set environment variables needed by mpi-start
export I2G_MPI_APPLICATION=/bin/hostname
export I2G_MPI_APPLICATION_ARGS=
export I2G_MPI_NP=2
export I2G_MPI_TYPE=openmpi
export I2G_MPI_FLAVOUR=openmpi
export I2G_MPI_JOB_NUMBER=0
export I2G_MPI_STARTUP_INFO=/home/imain179
export I2G_MPI_PRECOMMAND=time
export I2G_MPI_RELAY=
export I2G_MPI_START=/opt/i2g/bin/mpi-start

# Execute mpi-start
$I2G_MPI_START
```

- **The submission (in SGE):**

```
[imain179@i2g-ce01 ~]$ qsub -S /bin/bash -pe openmpi 2 -l
allow_slots_egee=0 ./test2mpistart.sh
```

- **The StdOut:**

```
[imain179@i2g-ce01 ~]$ cat test2mpistart.sh.o114486
Scientific Linux CERN SLC release 4.5 (Beryllium)
Scientific Linux CERN SLC release 4.5 (Beryllium)
lflip30.lip.pt
lflip31.lip.pt
```

- **The StdErr:**

```
[lflip31] /home/imain179 > cat test2mpistart.sh.e114486
Scientific Linux CERN SLC release 4.5 (Beryllium)
Scientific Linux CERN SLC release 4.5 (Beryllium)
real    0m0.731s
user    0m0.021s
sys     0m0.013s
```

- MPI commands are transparent to the user
  - No explicit mpiexec/mpirun instruction
  - Start the script via normal LRMS submission

Submitting MPI app to the Grid

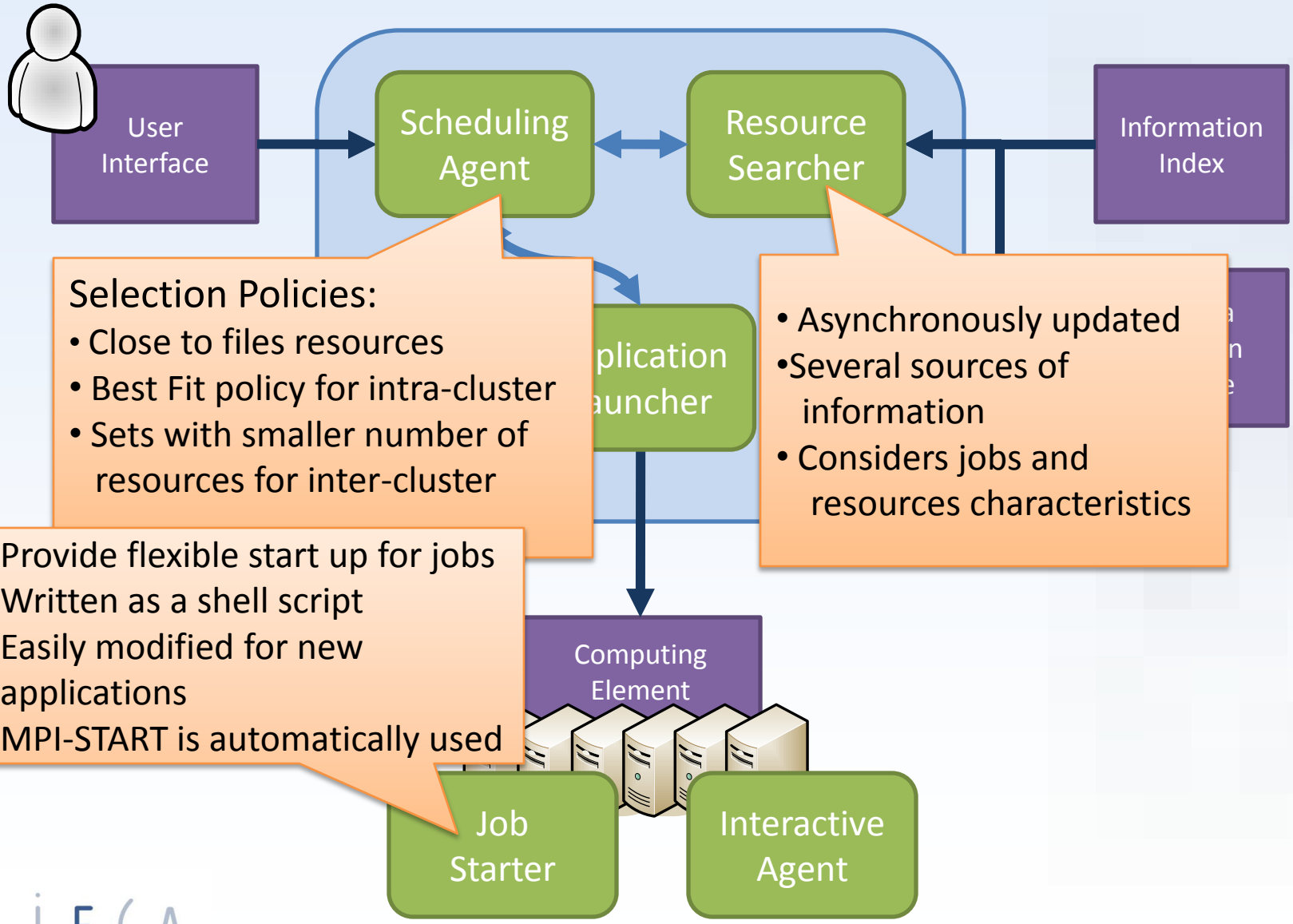
- Definition of the job requirements
  - Need for JDL extensions that allow the user to express their job characteristics
- Check if the site supports the MPI implementation
  - Usually published in the site-BDII (lcg-info magic)
  - Need a service that checks this transparently for the user
- Select and use resources from different administrative domains
  - No reservation or control over the resources
  - Need a service that handles automatically co-allocation and selects the best resources
- Start the job on the sites
  - Find MPI-START or download to the site, set properly the variables
  - Need a service that is able to do this for the user (and allows advanced users to enhance it)

CrossBroker

- CrossBroker is a metascheduler that provides automatic execution of interactive and parallel applications on grid (gLite based) environments
  - Transparent
  - Flexible modular components

- Leverages existing developments:
  - Fork from EDG-RB + gLite WMS
  - Condor
  - C++, perl, python and shell script

CrossBroker architecture



JDL: Normal jobs

```
Type          = "Job";
JobType       = "Normal";
Executable    = "my_app";
Arguments     = "-n 356 -p 4";
StdOutput     = "std.out";
StdError      = "std.err";
InputSandBox  = {"my_app"};
OutputSandBox = {"std.out", "std.err"};
Requirements  =
  other.GlueHostBenchmarkSI00 >= 1000;
Rank          = other.GlueHostFreeCPUs;
```

JDL: Parallel jobs

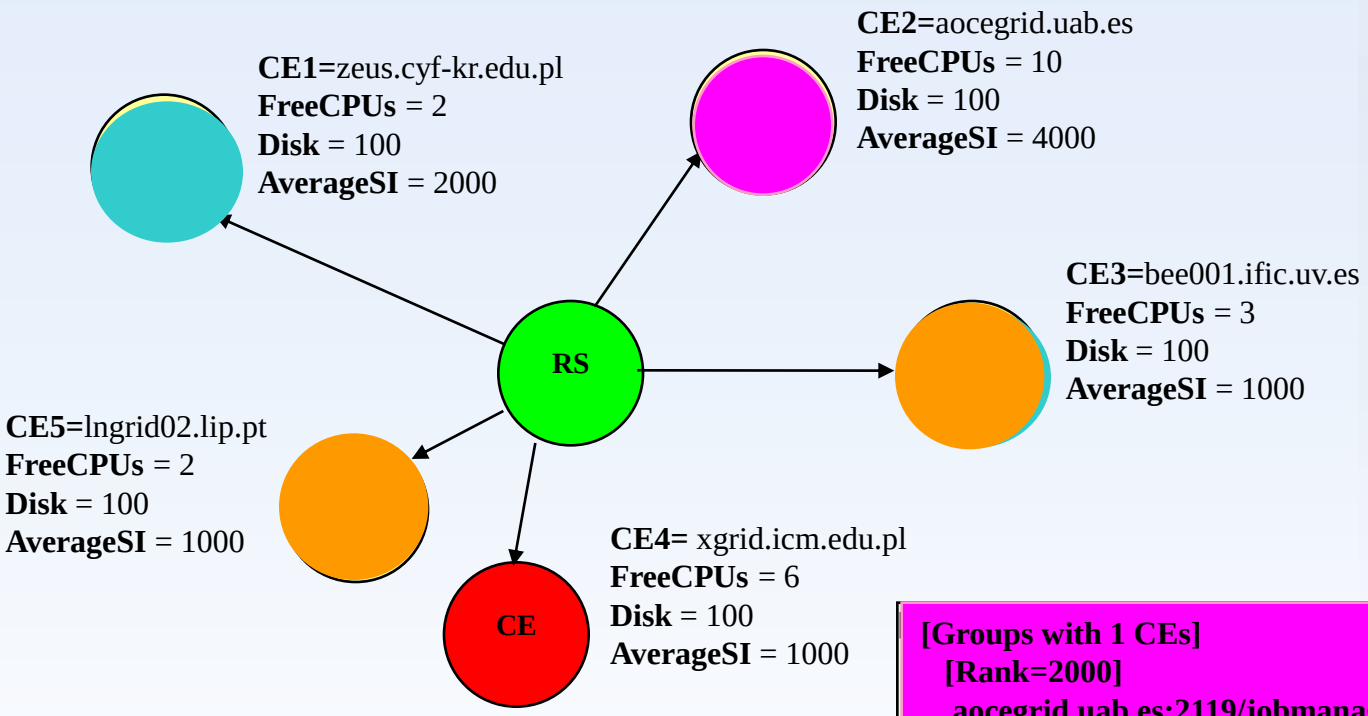
```
Type          = "Job";
JobType       = "Parallel";
NodeNumber    = 23;
SubJobType    = "openmpi";
Executable    = "my_app";
Arguments     = "-n 356 -p 4";
StdOutput     = "std.out";
StdError      = "std.err";
InputSandBox  = {"my_app"};
OutputSandBox = {"std.out", "std.err"};
Requirements  = other.GlueHostBenchmarkSI00 >= 1000;
Rank          = other.GlueHostFreeCPUs;
```

```
openmpi
mpich
mpich-g2
pacx-mpi
```

Set-matching

```
Executable    = "testMpi";
JobType       = "Parallel";
SubJobType    = "pacx-mpi";
NodeNumber    = 5;
StdOutput     = "testMpi.out";
StdError      = "testMpi.err";
Requirements  =
  other.GlueCEInfoLRMSType == "pbs";
Rank          = other.GlueHostBenchmarkSI00;
InputSandbox  = {"testMpi"};
OutputSandbox = {"testMpi.out", testMpi.err"};
```

Set-matching

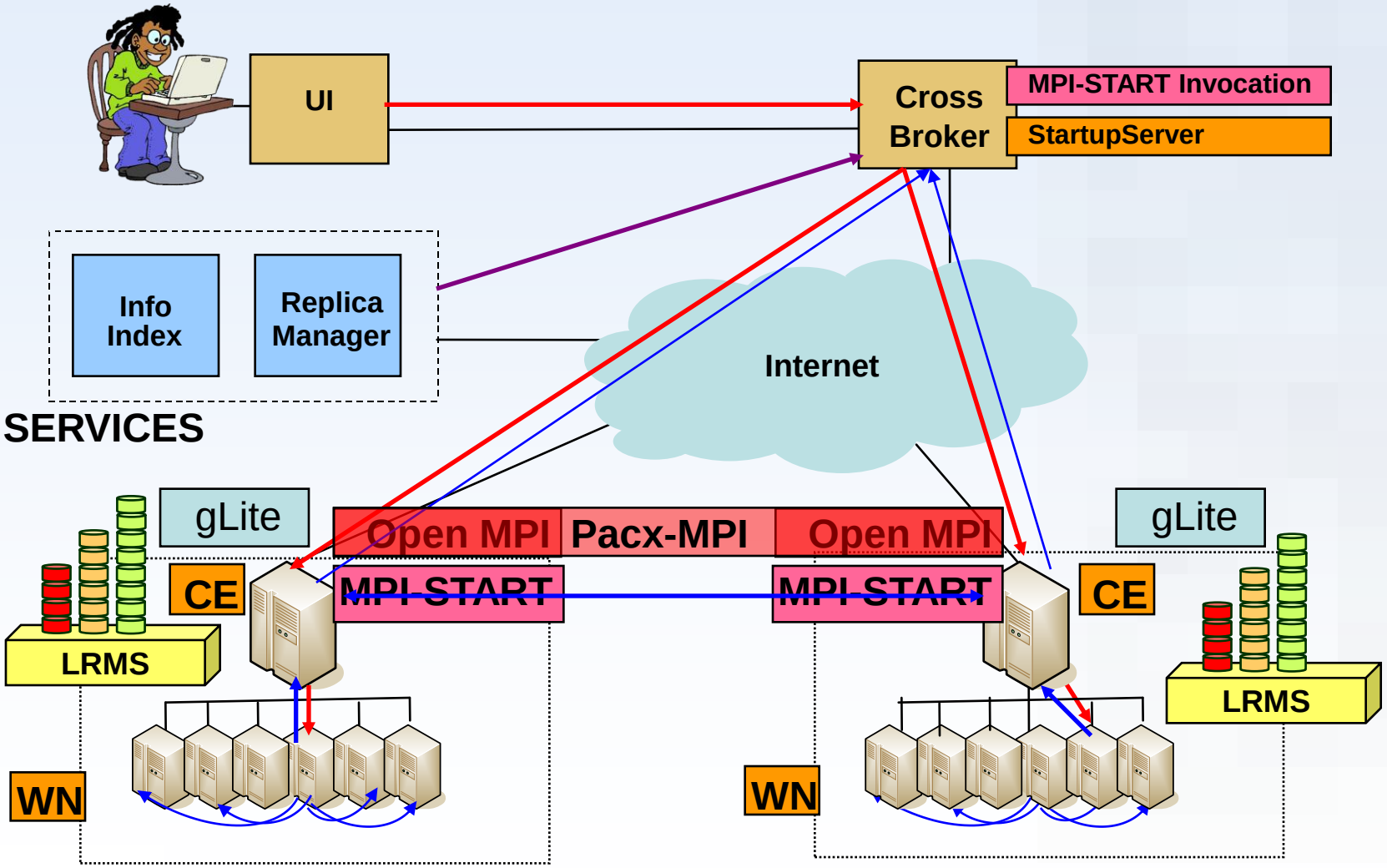


[Groups with 1 CEs]
[Rank=2000]
aocegrid.uab.es:2119/jobmanager-pbs-workq
freeCPUs = 10

[Groups with 2 CEs]
[Rank=1500]
zeus.cyf-kr.edu.pl:2119/jobmanager-pbs-workq
freeCPUs = 2
bee001.ific.uv.es:2119/jobmanager-pbs-workq
freeCPUs = 3

Rank=1000]
lngrid02.lip.pt/jobmanager-pbs-workq
freeCPUs = 2
bee001.ific.uv.es:2119/jobmanager-pbs-workq
freeCPUs = 3

Executing PACX job



JDL: Parallel jobs

```
Type            = "Job";
JobType         = "Parallel";
NodeNumber      = 23;
SubJobType      = "plain";
JobStarter      = "my-starter.sh"
JobStarterArguments = "-foo –bar"
Executable      = "my_app";
Arguments       = "-n 356 -p 4";
StdOutput       = "std.out";
StdError        = "std.err";
InputSandBox    = {"my_app"};
OutputSandBox   = {"std.out", "std.err"};
Requirements    = other.GlueHostBenchmarkSI00 >= 1000;
Rank            = other.GlueHostFreeCPUs;
```

Advanced users can tune the environment to fit their needs

Interactivity Support

- Interactivity allows researchers to visualize results and obtain them faster
- Requirements:
  - Fast startup: the possibility of starting the application immediately, even in high occupancy scenarios
  - Online Input-Output streaming: the ability to have application input and output online.

Interactive Agents

- "Traditional" remote interactivity tools: VNC, ssh are not directly applicable to grids
- glogin/i2glogin is able to create communication channels from the WN to the user
  - Interactive GSS secured Grid-shells
  - TCP tunnels
  - Posix pipes
  - VPN support
  - Integrated with other interactivity/visualization tools

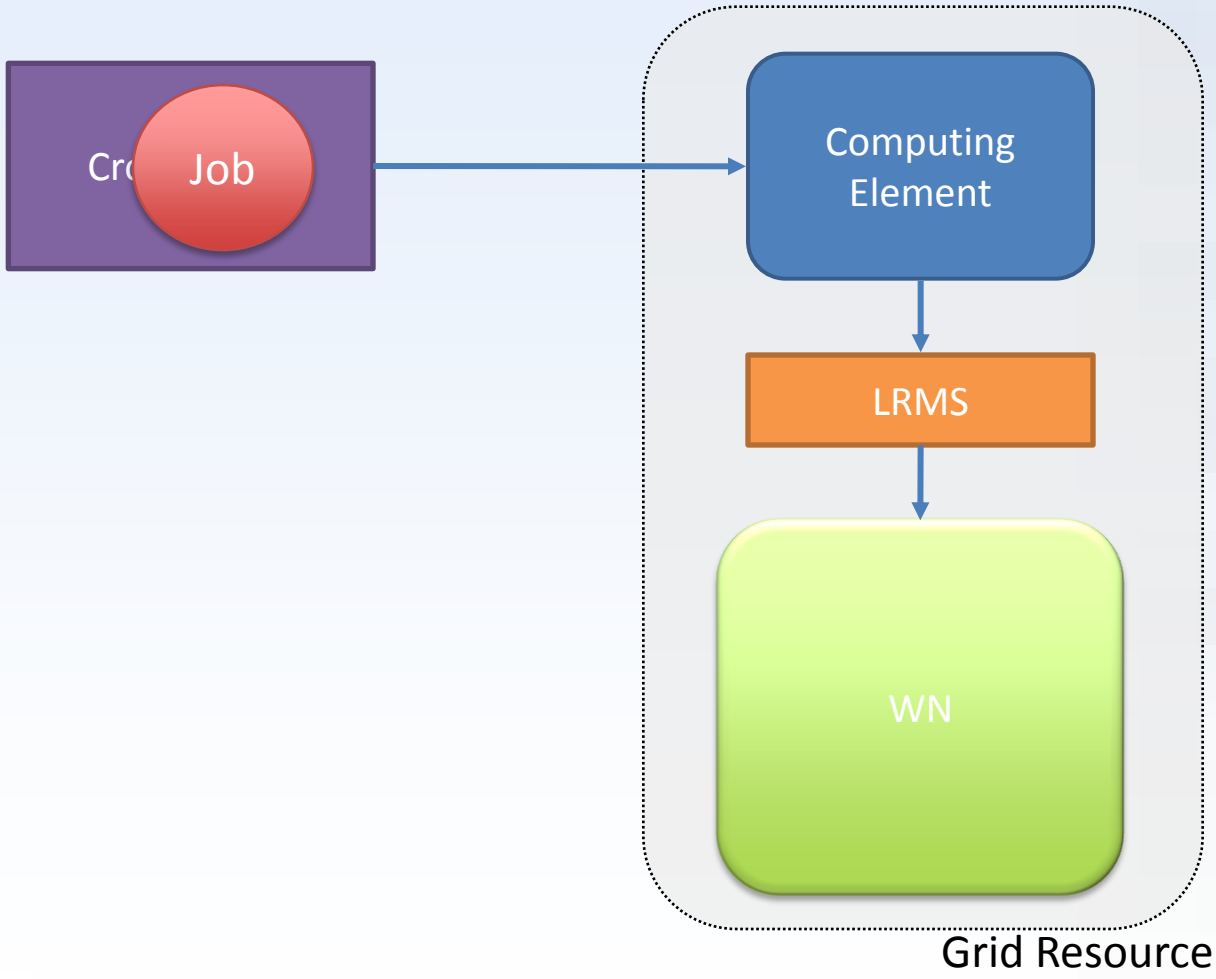
JDL: Interactive jobs

```
Type = "Job";
VirtualOrganisation = "imain";
JobType = "Parallel";
SubJobType = “openmpi";
NodeNumber =  11;
Interactive = TRUE;
InteractiveAgent = “i2glogin“;
InteractiveAgentArguments = “-r -p 195.168.105.65:23433“;
Executable = "test-app";
InputSandbox = {"test-app", "inputfile"};
OutputSanbox = {"std.out", "std.err"};
StdErr = "std.err“;
StdOutput = "std.out";
Rank = other.GlueHostBenchmarkSI00 ;
Requirements = other.GlueCEStateStatus == "Production";
```

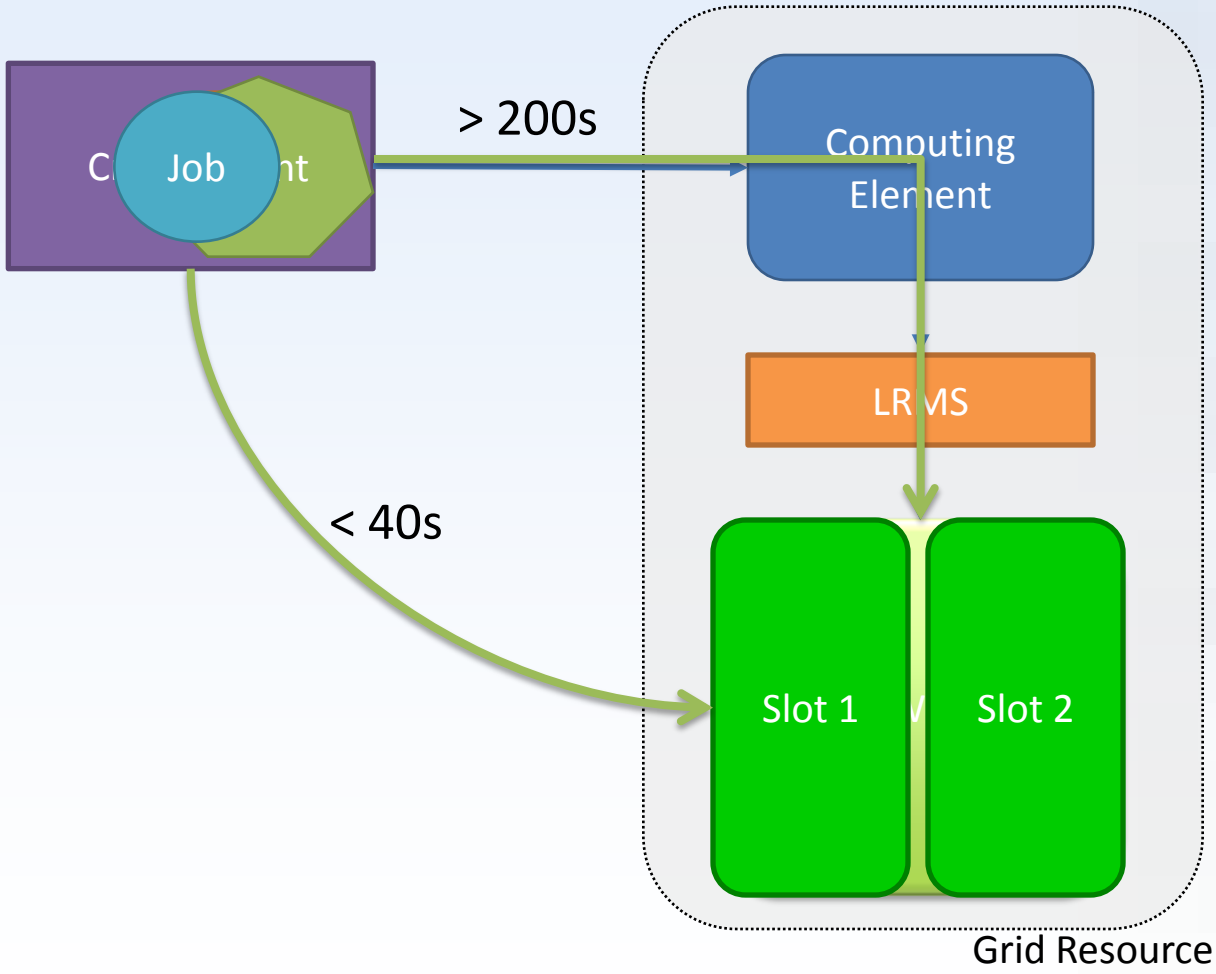
Multiprogramming

- The idea
  - Each job is encapsulated in an agent that takes control over the WN independently of its LRMS
- Lightweight "Virtual Machines"
  - Each Worker Node is divided in 2 execution slots
  - Each VM can execute jobs independently (e.g. batch and interactive)
  - NOT a full virtual machine (Xen, VMWare,…)
  - NO need for special priviledges in the WN

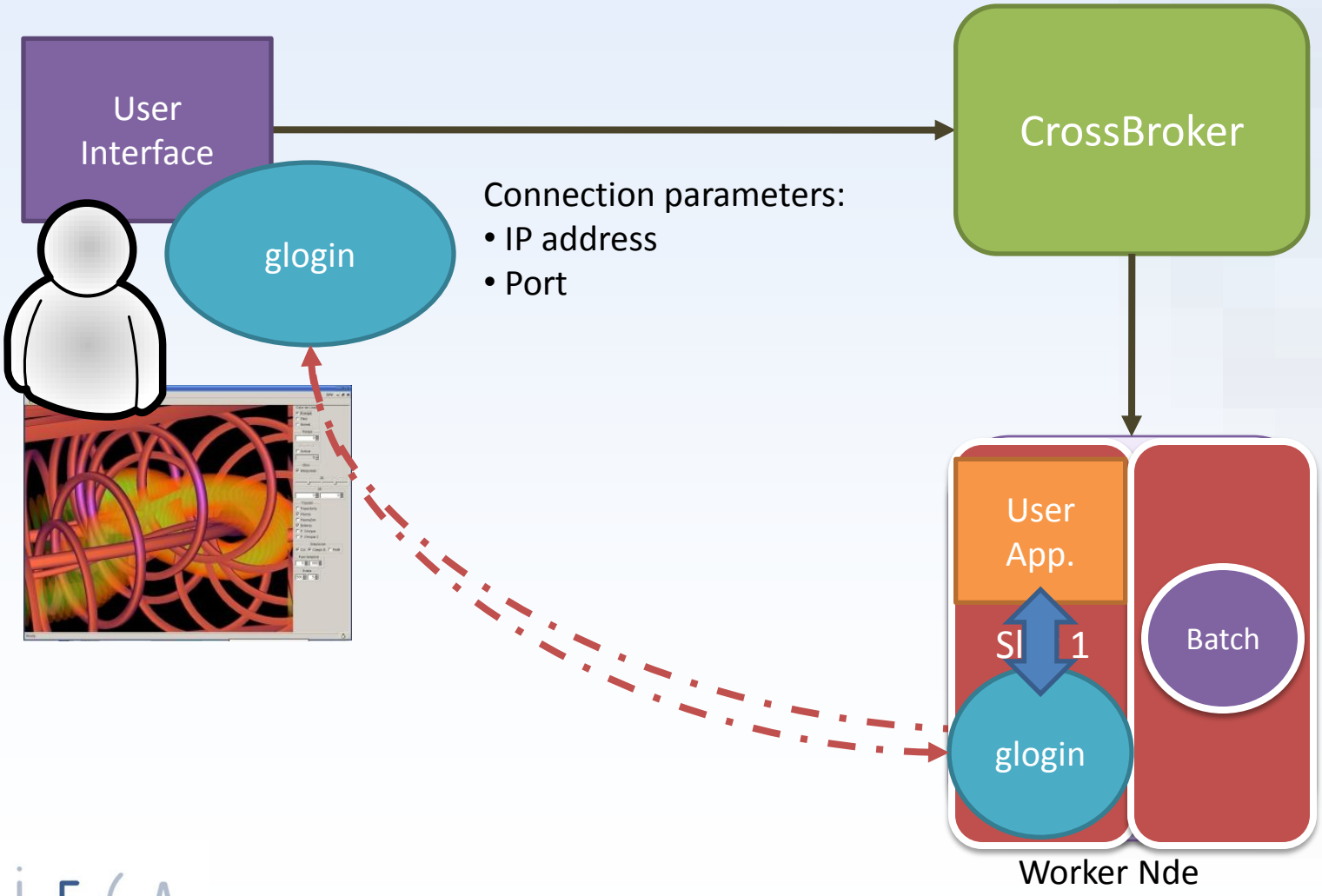
Multiprogramming



Multiprogramming



Interactive Agents



What about gLite/EGEE?

- Broken support for MPI until latest production version of WMS
  - MPICH job type with hardcoded parameters
  - Support only for specific LRMS
- Now you can submit parallel jobs specifying Jobtype="Normal" + NodeNumber, but…
  - MPI-START has to be configured by the user
  - No requirement checking done at matchmaking time
- A big amount of the sites do not provide support for MPI
- If you want to use interactivity:
  - No automatic setup of the agent
  - No prioritization of the jobs

Questions?