

Message Passing Interface MPI

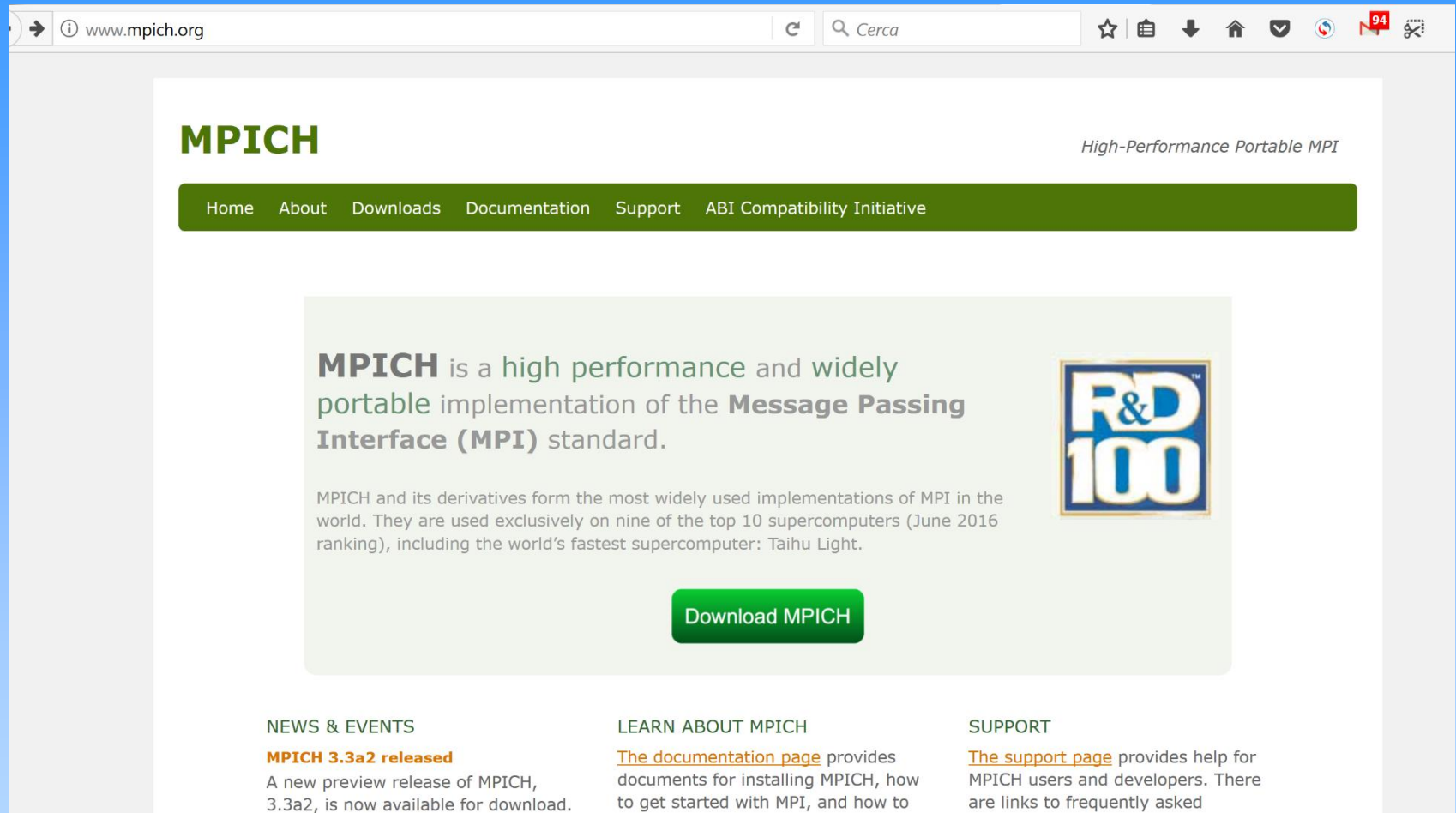
Message Passing Interface MPI

Implementazioni di MPI

- MPICH
 - Argonne National Laboratory (ANL) and
 - Mississippi State University (MSU)
- OpenMPI
- ...

Linguaggi ospiti: Fortran, C, C++, Matlab

MPI : dove trovarlo



The screenshot shows the MPICH website in a browser. The address bar displays 'www.mpich.org'. The page features a green navigation bar with links: Home, About, Downloads, Documentation, Support, and ABI Compatibility Initiative. The main content area has a large heading 'MPICH' and a subtitle 'High-Performance Portable MPI'. Below this, a green box contains the text: 'MPICH is a high performance and widely portable implementation of the Message Passing Interface (MPI) standard.' To the right of this text is an 'R&D 100' logo. Below the text, it states: 'MPICH and its derivatives form the most widely used implementations of MPI in the world. They are used exclusively on nine of the top 10 supercomputers (June 2016 ranking), including the world's fastest supercomputer: Taihu Light.' A green button labeled 'Download MPICH' is positioned below this text. At the bottom, there are three columns: 'NEWS & EVENTS' with a link to 'MPICH 3.3a2 released', 'LEARN ABOUT MPICH' with a link to 'The documentation page', and 'SUPPORT' with a link to 'The support page'.

MPICH *High-Performance Portable MPI*

Home About Downloads Documentation Support ABI Compatibility Initiative

MPICH is a high performance and widely portable implementation of the **Message Passing Interface (MPI)** standard.

MPICH and its derivatives form the most widely used implementations of MPI in the world. They are used exclusively on nine of the top 10 supercomputers (June 2016 ranking), including the world's fastest supercomputer: Taihu Light.

[Download MPICH](#)

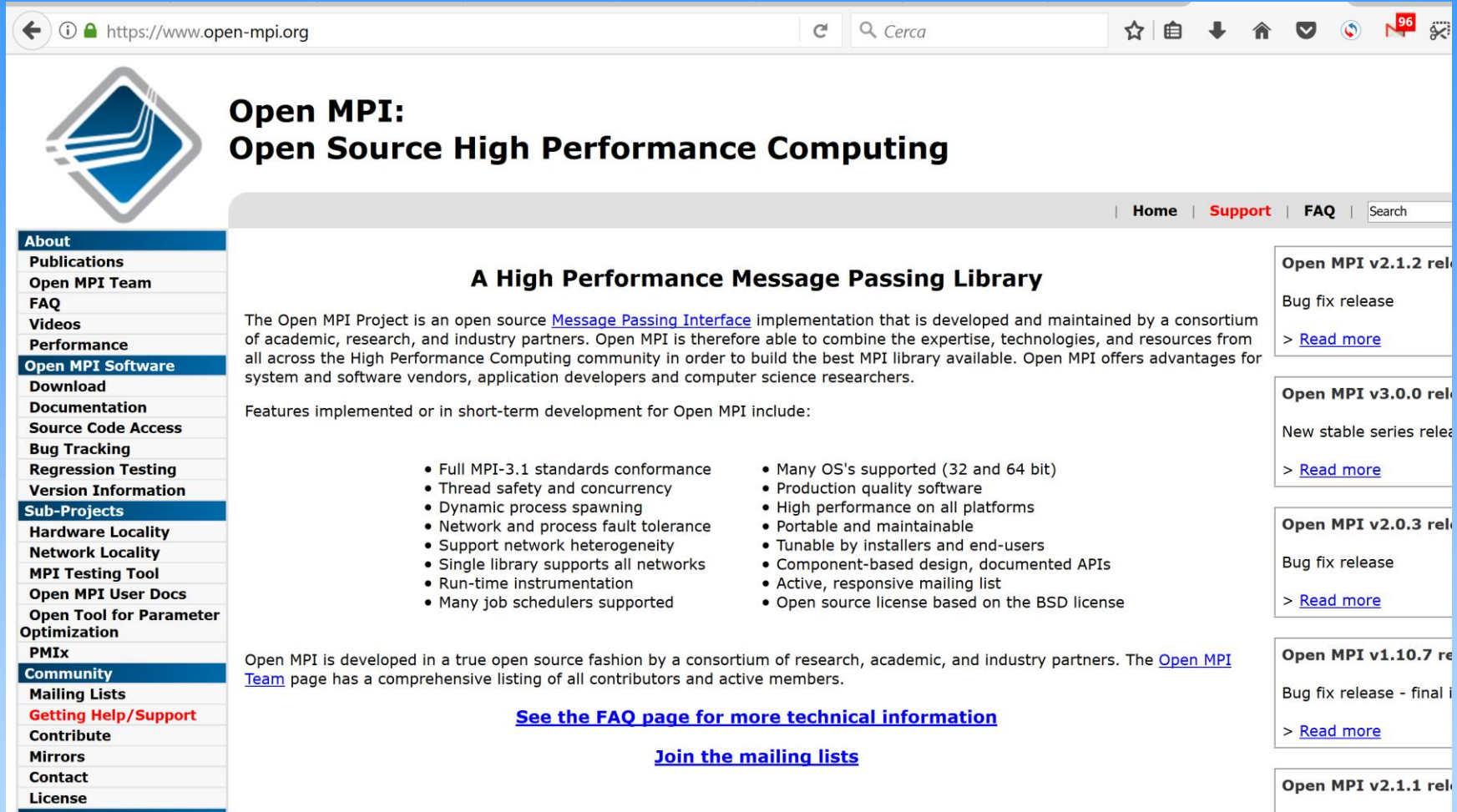
NEWS & EVENTS
[MPICH 3.3a2 released](#)
A new preview release of MPICH, 3.3a2, is now available for download.

LEARN ABOUT MPICH
[The documentation page](#) provides documents for installing MPICH, how to get started with MPI, and how to

SUPPORT
[The support page](#) provides help for MPICH users and developers. There are links to frequently asked

<http://www.mpich.org/>

MPI : dove trovarlo



The screenshot shows the Open MPI website. The browser address bar displays <https://www.open-mpi.org>. The website header features the Open MPI logo and the text "Open MPI: Open Source High Performance Computing". A navigation bar includes links for "Home", "Support", and "FAQ", along with a search bar. A left sidebar contains a menu with categories like "About", "Open MPI Software", "Sub-Projects", and "Community". The main content area is titled "A High Performance Message Passing Library" and describes the Open MPI Project as an open source Message Passing Interface implementation. It lists features implemented or in development, such as MPI-3.1 standards conformance, thread safety, dynamic process spawning, network fault tolerance, support for heterogeneous networks, run-time instrumentation, support for many job schedulers, support for many OSes, production quality software, high performance on all platforms, portability, tunability, component-based design, a documented API, an active mailing list, and an open source license based on the BSD license. It also mentions that the Open MPI is developed in a true open source fashion by a consortium of research, academic, and industry partners, and provides a link to the "Open MPI Team" page for a comprehensive listing of contributors and active members. A "See the FAQ page for more technical information" link and a "Join the mailing lists" link are also present. On the right side, there are several release announcements for Open MPI versions, including v2.1.2, v3.0.0, v2.0.3, v1.10.7, and v2.1.1, each with a "Read more" link.

Open MPI:
Open Source High Performance Computing

[Home](#) | [Support](#) | [FAQ](#) |

About
Publications
Open MPI Team
FAQ
Videos
Performance
Open MPI Software
Download
Documentation
Source Code Access
Bug Tracking
Regression Testing
Version Information
Sub-Projects
Hardware Locality
Network Locality
MPI Testing Tool
Open MPI User Docs
Open Tool for Parameter Optimization
PMIx
Community
Mailing Lists
[Getting Help/Support](#)
Contribute
Mirrors
Contact
License

A High Performance Message Passing Library

The Open MPI Project is an open source [Message Passing Interface](#) implementation that is developed and maintained by a consortium of academic, research, and industry partners. Open MPI is therefore able to combine the expertise, technologies, and resources from all across the High Performance Computing community in order to build the best MPI library available. Open MPI offers advantages for system and software vendors, application developers and computer science researchers.

Features implemented or in short-term development for Open MPI include:

- Full MPI-3.1 standards conformance
- Thread safety and concurrency
- Dynamic process spawning
- Network and process fault tolerance
- Support network heterogeneity
- Single library supports all networks
- Run-time instrumentation
- Many job schedulers supported
- Many OS's supported (32 and 64 bit)
- Production quality software
- High performance on all platforms
- Portable and maintainable
- Tunable by installers and end-users
- Component-based design, documented APIs
- Active, responsive mailing list
- Open source license based on the BSD license

Open MPI is developed in a true open source fashion by a consortium of research, academic, and industry partners. The [Open MPI Team](#) page has a comprehensive listing of all contributors and active members.

[See the FAQ page for more technical information](#)

[Join the mailing lists](#)

Open MPI v2.1.2 rel
Bug fix release
> [Read more](#)

Open MPI v3.0.0 rel
New stable series rele
> [Read more](#)

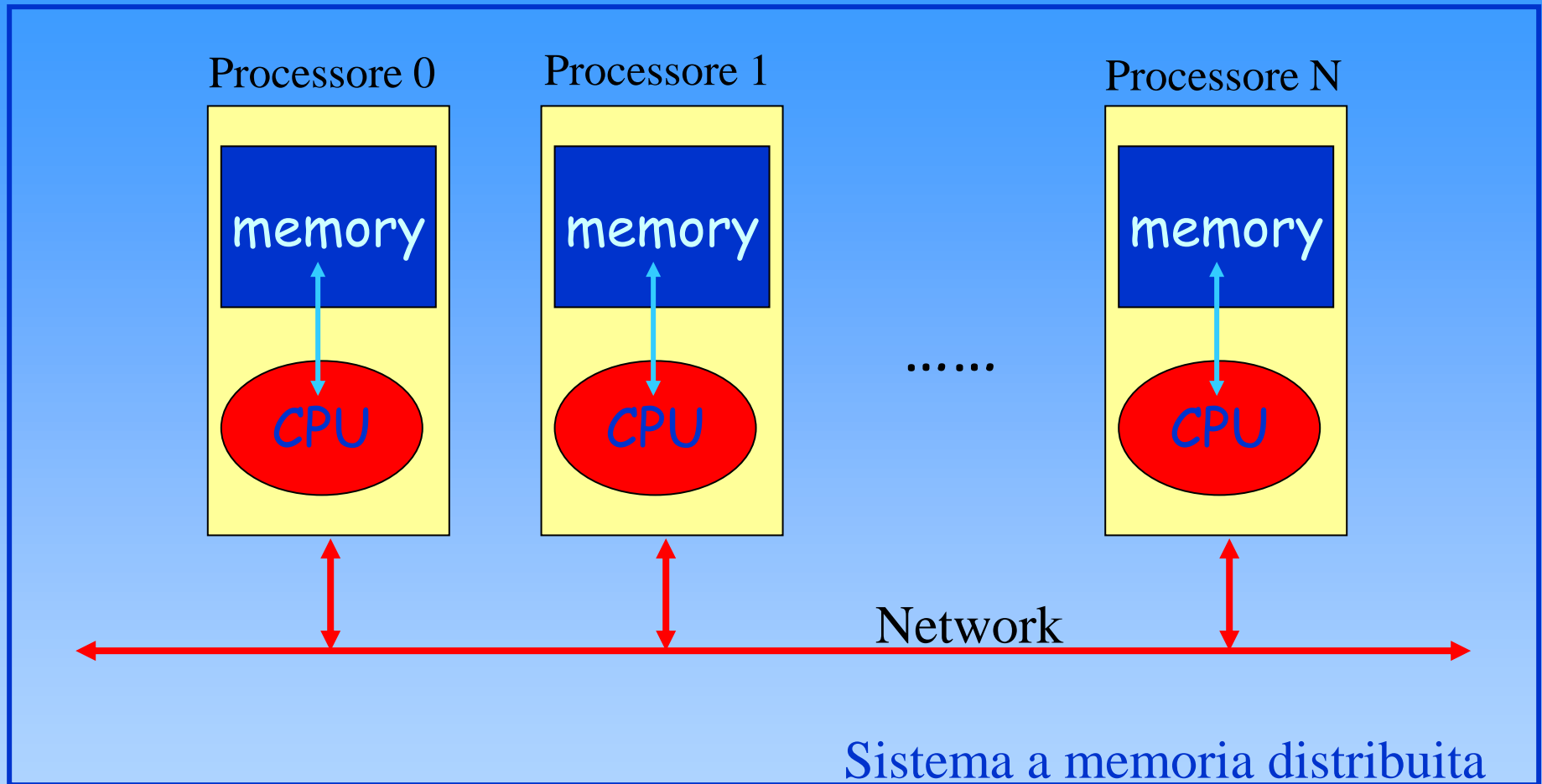
Open MPI v2.0.3 rel
Bug fix release
> [Read more](#)

Open MPI v1.10.7 re
Bug fix release - final i
> [Read more](#)

Open MPI v2.1.1 rel

<https://www.open-mpi.org/>

Paradigma del message passing

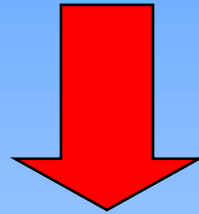


Ogni processore ha una propria memoria locale alla quale accede direttamente.

Ogni processore può conoscere i dati in memoria di un altro processore o far conoscere i propri, attraverso il trasferimento di dati.

Definizione: *message passing*

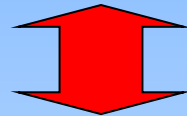
... ogni processore può conoscere i dati nella memoria di un altro processore o far conoscere i propri, attraverso il trasferimento di dati.



Message Passing :
modello per la progettazione
di software in
ambiente di Calcolo Parallelo.

Lo standard

La necessità di **rendere portabile**
il modello *Message Passing*
ha condotto alla definizione
e all'implementazione
di un **ambiente standard**.



Message Passing Interface
MPI

Per cominciare...

In ambiente MPI un programma
è visto come un insieme di
processi concorrenti

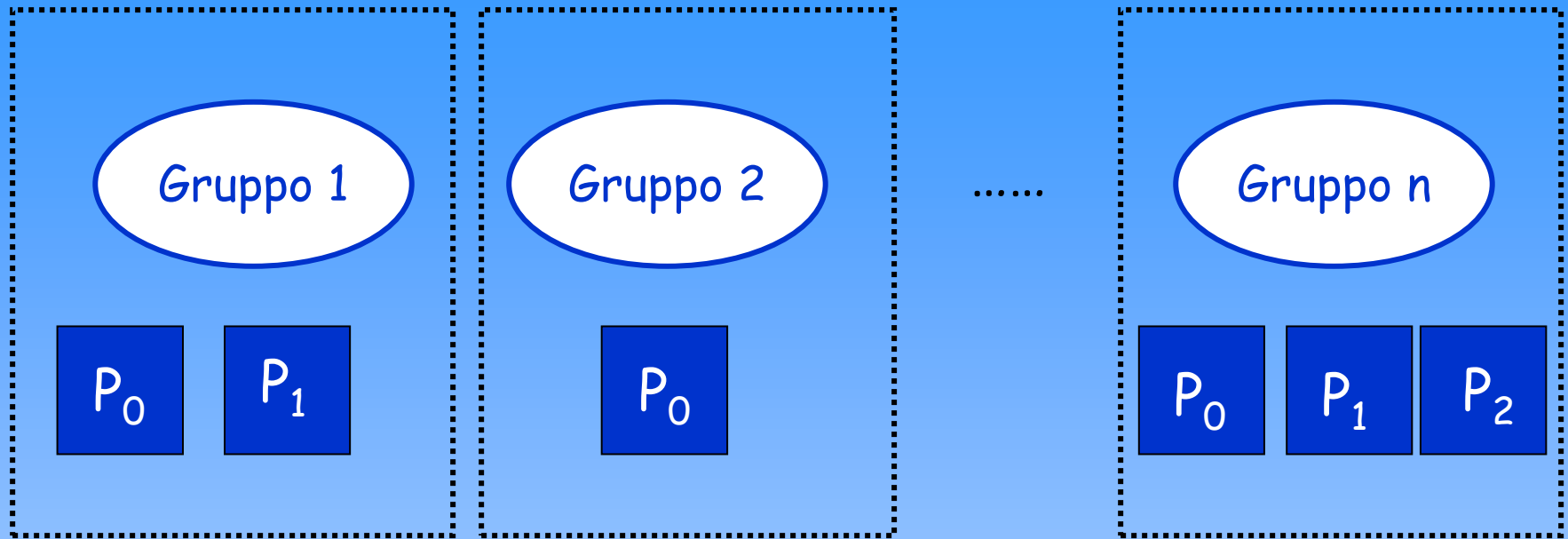
```
...  
int main()  
{  
  
...  
sum=0;  
for i= 0 to 14 do  
  sum=sum+a[i];  
endfor  
  
...  
return 0;  
}
```

```
...  
main()  
{  
  
...  
sum=0;  
for i=0 to 4 do  
  sum=sum+a[i];  
endfor  
  
...  
return 0;  
}
```

```
...  
main()  
{  
  
...  
sum=0;  
for i=5 to 9 do  
  sum=sum+a[i];  
endfor  
  
...  
return 0;  
}
```

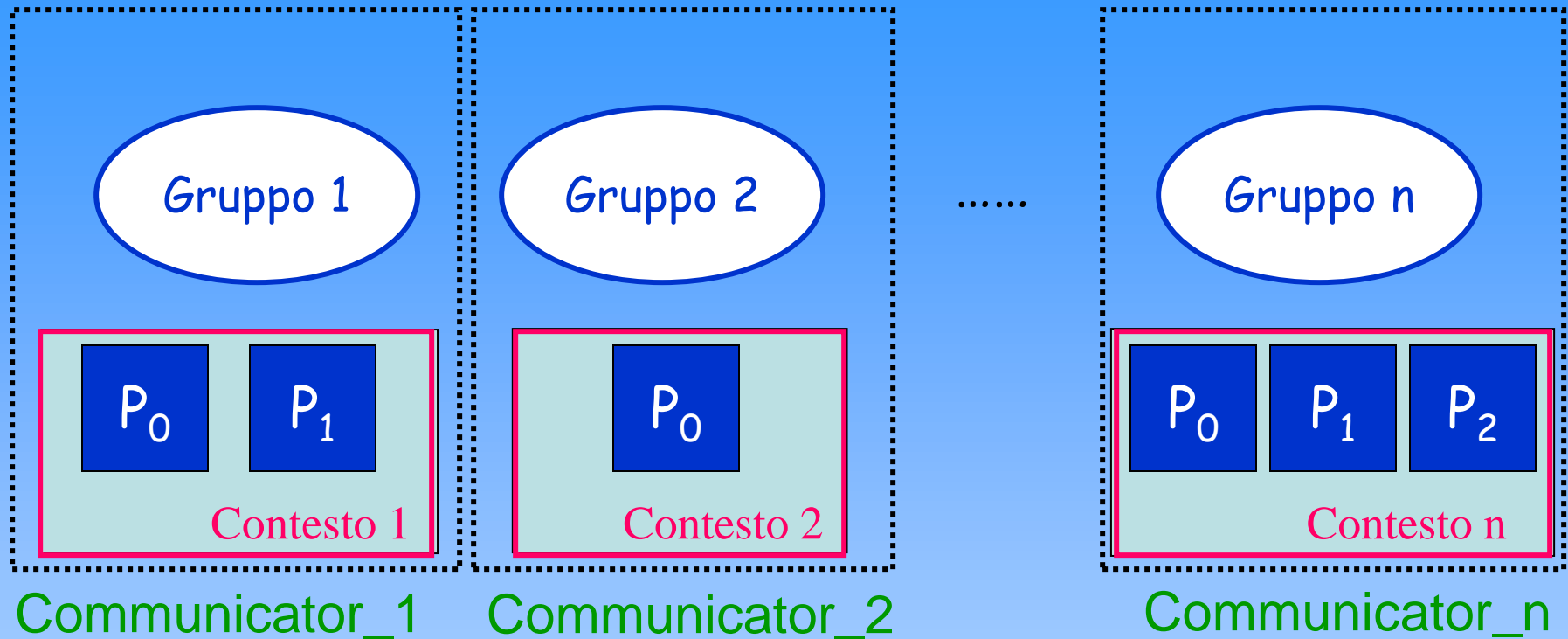
```
...  
main()  
{  
  
...  
sum=0;  
for i=10 to 14do  
  sum=sum+a[i];  
endfor  
  
...  
return 0;  
}
```


Alcuni concetti di base: GRUPPO...



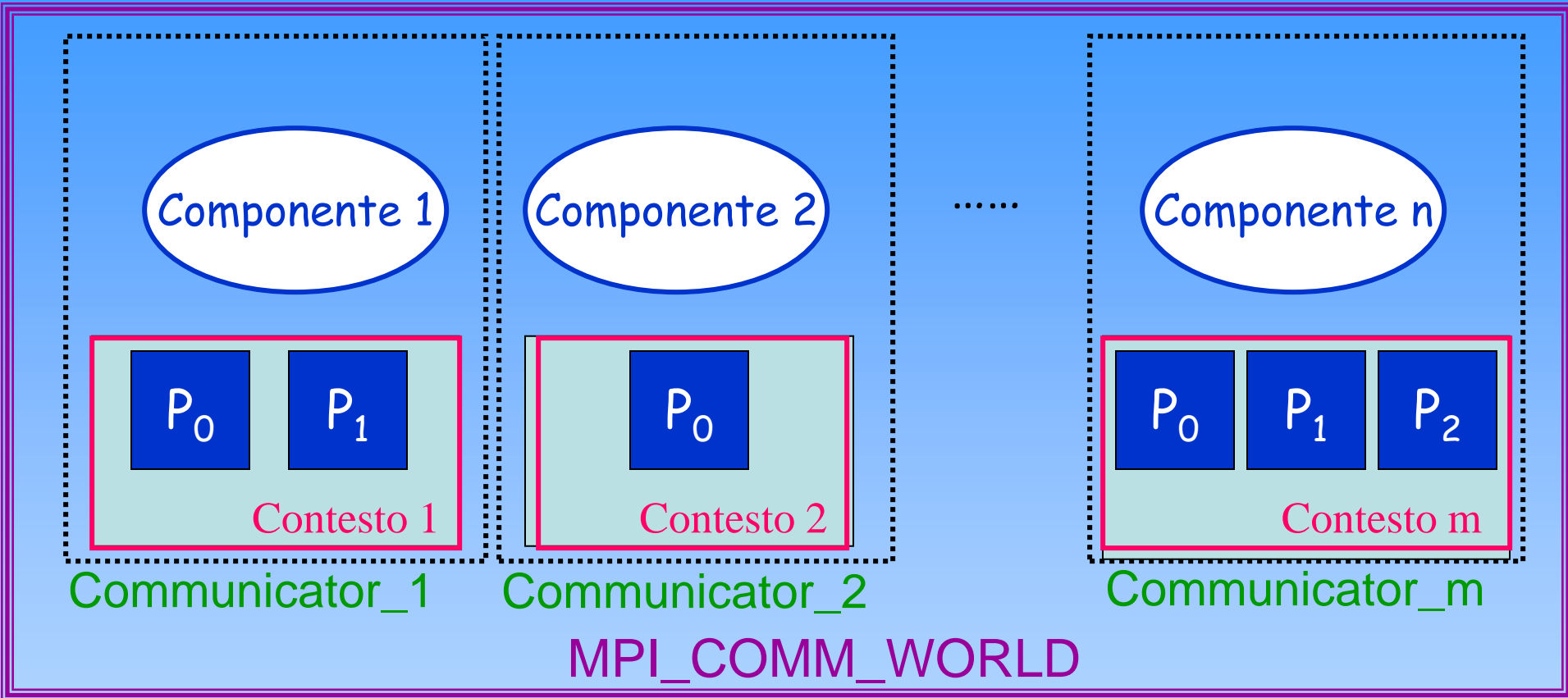
- Ogni attività del programma viene affidata ad un *gruppo di processi*.
- In ciascun gruppo ad ogni processo è associato un *identificativo*.
- L'*identificativo* è un intero, compreso tra 0 ed il numero totale di processi appartenenti ad un gruppo decrementato di una unità.
- Processi appartenenti a uno o più gruppi possono avere identificativi diversi, ciascuno relativo ad uno specifico gruppo

Alcuni concetti di base: COMMUNICATOR...



- Ad un gruppo di processi appartenenti ad uno stesso contesto viene assegnato un ulteriore identificativo: il *communicator*.
- Il *communicator* racchiude tutte le caratteristiche dell'ambiente di comunicazione: topologia, quali contesti coinvolge, ecc...

...Alcuni concetti di base: COMMUNICATOR



- Tutti i processi fanno parte per default di un unico communicator detto **MPI_COMM_WORLD**.

MPI è una libreria
che comprende:

- Funzioni per definire l'ambiente
- Funzioni per comunicazioni uno a uno
- Funzioni per comunicazioni collettive
- Funzioni per operazioni collettive

Le funzioni dell'ambiente.

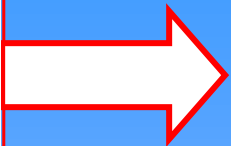
Un semplice programma :

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[])
{
    int menum, nproc;
    MPI_Init(&argc,&argv);

    MPI_Comm_rank(MPI_COMM_WORLD,&menum);
    MPI_Comm_size(MPI_COMM_WORLD,&nproc);

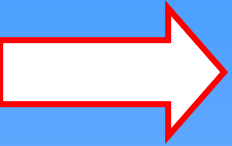
    printf("Sono %d di %d\n",menum,nproc);

    MPI_Finalize();
    return 0;
}
```



Tutti i processi di MPI_COMM_WORLD
stampano a video il proprio
identificativo menum ed il numero di processi nproc.

Nel programma ... :



```
#include "mpi.h"
```

mpi.h : Header File

Il file contiene alcune direttive necessarie al preprocessore per l'utilizzo dell'MPI

Nel programma ... :

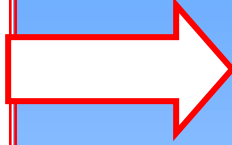
```
#include <stdio.h>
#include "mpi.h"

int main(int argc, char *argv[])
{
    int menum, nproc;
    MPI_Init(&argc,&argv);

    MPI_Comm_rank(MPI_COMM_WORLD,&menum);
    MPI_Comm_size(MPI_COMM_WORLD,&nproc);

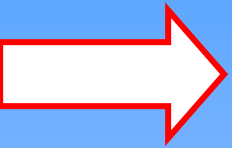
    printf("Sono %d di %d\n",menum,nproc);

    MPI_Finalize();
    return 0;
}
```



**Tutti i processi di MPI_COMM_WORLD
stampano a video il proprio
identificativo menum ed il numero di processi nproc.**

Nel programma ... :



`MPI_Init(&argc,&argv);`

- Inizializza l'ambiente di esecuzione MPI
- Inizializza il communicator `MPI_COMM_WORLD`
- I due dati di input: `argc` ed `argv` sono gli argomenti del main

In generale ... :

```
MPI_Init(int *argc, char ***argv);
```

Input: argc, **argv;

- Questa routine inizializza l'ambiente di esecuzione di MPI. Deve essere chiamata una sola volta, prima di ogni altra routine MPI.
- Definisce l'insieme dei processi attivati (communicator).

Nel programma ... :

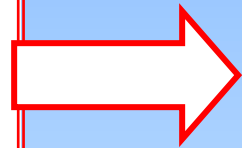
```
#include <stdio.h>
#include "mpi.h"

int main(int argc, char *argv[])
{
    int menum, nproc;
    MPI_Init(&argc,&argv);

    MPI_Comm_rank(MPI_COMM_WORLD,&menum);
    MPI_Comm_size(MPI_COMM_WORLD,&nproc);

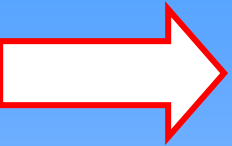
    printf("Sono %d di %d\n",menum,nproc);

    MPI_Finalize();
    return 0;
}
```



**Tutti i processi di MPI_COMM_WORLD
stamperanno sul video il proprio
identificativo menum ed il numero di processi nproc.**

Nel programma ... :



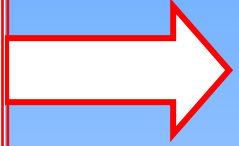
`MPI_Finalize();`

- Questa routine determina la fine del programma MPI.
- Dopo questa routine **non** è possibile richiamare nessun'altra routine di MPI.

Nel programma ... :

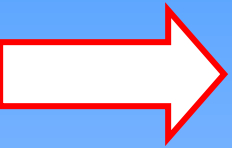
```
#include <stdio.h>
#include "mpi.h"

int main(int argc, char *argv[])
{
    int menum, nproc;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&menum);
    MPI_Comm_size(MPI_COMM_WORLD,&nproc);
    printf("Sono %d di %d\n",menum,nproc);
    MPI_Finalize();
    return 0;
}
```



**Tutti i processi di MPI_COMM_WORLD
stamperanno sul video il proprio
identificativo menum ed il numero di processi nproc.**

Nel programma ... :



```
MPI_Comm_rank(MPI_COMM_WORLD,&menum);
```

- Questa routine permette al processo chiamante, appartenente al communicator `MPI_COMM_WORLD`, di memorizzare il proprio identificativo nella variabile `menum`.

In generale ... :

```
MPI_Comm_rank(MPI_Comm comm, int *menum);
```

Input: comm ;

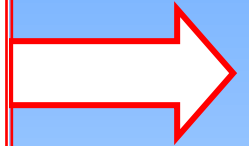
Output: menum.

- Fornisce ad ogni processo del communicator comm l'identificativo menum.

Nel programma ... :

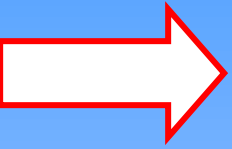
```
#include <stdio.h>
#include "mpi.h"

int main(int argc, char *argv[])
{
    int menum, nproc;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&menum);
    MPI_Comm_size(MPI_COMM_WORLD,&nproc);
    printf("Sono %d di %d\n",menum,nproc);
    MPI_Finalize();
    return 0;
}
```



**Tutti i processi di MPI_COMM_WORLD
stamperanno sul video il proprio
identificativo menum ed il numero di processi nproc.**

Nel programma ... :



```
MPI_Comm_size(MPI_COMM_WORLD,&nproc);
```

- Questa routine permette al processo chiamante di memorizzare nella variabile `nproc` il numero totale dei processi concorrenti appartenenti al communicator `MPI_COMM_WORLD`.

In generale ... :

```
MPI_Comm_size(MPI_Comm comm, int *nproc);
```

Input: comm ;

Output: nproc.

- Ad ogni processo del communicator `comm` , restituisce in `nproc`, il numero totale di processi che costituiscono `comm`.
- Permette di conoscere quanti processi concorrenti possono essere utilizzati per una determinata operazione.

Eseguire un programma: esempio di job-script

```
#!/bin/bash
```

```
#####  
#  
# The PBS directives #  
#  
#####  
#PBS -q studenti  
#PBS -l nodes=8  
#PBS -N hello  
#PBS -o hello.out  
#PBS -e hello.err  
#####  
# -q coda su cui va eseguito il job #  
  
# -l numero di nodi richiesti #  
# -N nome job(stesso del file pbs) #  
# -o, -e nome files contenente l'output #  
#####  
#  
# qualche informazione sul job #  
#  
#####  
  
NCPU=`wc -l < $PBS_NODEFILE`  
echo -----  
echo ' This job is allocated on '${NCPU}' cpu(s)'  
echo 'Job is running on node(s): '  
cat $PBS_NODEFILE
```

Job-script che compila
ed esegue Hello.c

Eseguire un programma: esempio di job-script

```
#!/bin/bash
```

```
#####
```

```
#
```

```
# The PBS directives #
```

```
#
```

```
#####
```

```
#PBS -q studenti
```

```
#PBS -l nodes=8
```

```
#PBS -N Hello
```

```
#PBS -o Hello.out
```

```
#PBS -e Hello.err
```

```
#####
```

```
# -q coda su cui va eseguito il job #
```

```
# -l numero di nodi richiesti #
```

```
# -N nome job(stesso del file pbs) #
```

```
# -o, -e nome files contenente l'output #
```

```
#####
```

```
#
```

```
# qualche informazione sul job #
```

```
#
```

```
#####
```

```
NCPU=`wc -l < $PBS_NODEFILE`
```

```
echo -----
```

```
echo ' This job is allocated on '${NCPU}' cpu(s)'
```

```
echo 'Job is running on node(s): '
```

```
cat $PBS_NODEFILE
```

La prima riga deve essere sempre questa

Le righe che iniziano con # sono commenti.
Le righe che iniziano con **#PBS** sono direttive.

In rosso le informazioni tipiche di questa particolare sottomissione

Eseguire un programma: esempio di job-script

```
PBS_O_WORKDIR=$PBS_O_HOME/ProgettoHello
echo -----
echo PBS: qsub is running on $PBS_O_HOST
echo PBS: originating queue is $PBS_O_QUEUE
echo PBS: executing queue is $PBS_QUEUE
echo PBS: working directory is $PBS_O_WORKDIR
echo PBS: execution mode is $PBS_ENVIRONMENT
echo PBS: job identifier is $PBS_JOBID
echo PBS: job name is $PBS_JOBNAME
echo PBS: node file is $PBS_NODEFILE
echo PBS: current home directory is $PBS_O_HOME
echo PBS: PATH = $PBS_O_PATH
echo -----
```

Stampa informazioni

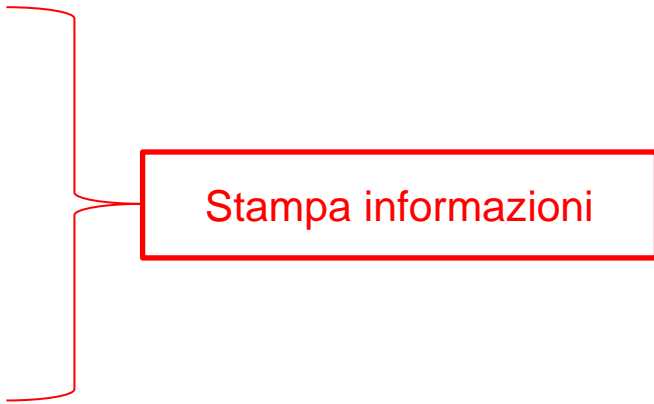
```
echo "Eseguo/usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS_O_WORKDIR/Hello $PBS_O_WORKDIR/Hello.c"
/usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS_O_WORKDIR/Hello $PBS_O_WORKDIR/Hello.c
```

```
echo "Eseguo:/usr/lib64/openmpi/1.4-gcc/bin/-machinefile $PBS_NODEFILE E -np $NCPU $PBS_O_WORKDIR/Hello"
/usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile $PBS_NODEFILE -np $NCPU $PBS_O_WORKDIR/Hello
```

Eseguire un programma: esempio di job-script

```
PBS_O_WORKDIR=$PBS_O_HOME/ProgettoHello
echo -----
echo PBS: qsub is running on $PBS_O_HOST
echo PBS: originating queue is $PBS_O_QUEUE
echo PBS: executing queue is $PBS_QUEUE
echo PBS: working directory is $PBS_O_WORKDIR
echo PBS: execution mode is $PBS_ENVIRONMENT
echo PBS: job identifier is $PBS_JOBID
echo PBS: job name is $PBS_JOBNAME
echo PBS: node file is $PBS_NODEFILE
echo PBS: current home directory is $PBS_O_HOME
echo PBS: PATH = $PBS_O_PATH
echo -----
echo "Eseguo/usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS_O_WORKDIR/Hello $PBS_O_WORKDIR/Hello.c"
/usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS_O_WORKDIR/Hello $PBS_O_WORKDIR/Hello.c

echo "Eseguo/usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile $PBS_NODEFILE -np $NCPU
$PBS_O_WORKDIR/Hello"
/usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile $PBS_NODEFILE -np $NCPU $PBS_O_WORKDIR/Hello
```



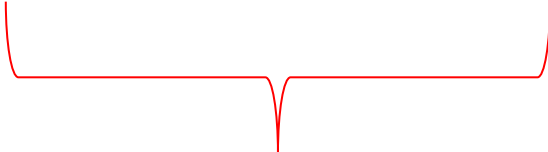
Stampa informazioni

**CAMBIARE IL NUMERO
\$NCPU IN UNA CIFRA
QUALUNQUE E FARE TUTTE
LE PROVE POSSIBILI**

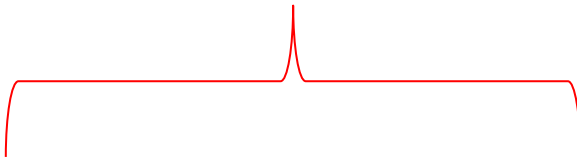
Eseguire un programma: esempio di job-script

Vediamo nel dettaglio la compilazione e l'esecuzione...

```
/usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS_O_WORKDIR/Hello $PBS_O_WORKDIR/Hello.c
```



Path assoluto del comando di compilazione (mpicc) e di esecuzione (mpiexec)



```
/usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile $PBS_NODEFILE -np $NCPU $PBS_O_WORKDIR/Hello
```

Eseguire un programma: esempio di job-script

Vediamo nel dettaglio la compilazione e l'esecuzione...

```
/usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS_O_WORKDIR/Hello $PBS_O_WORKDIR/Hello.c
```




Opzione **-o**, ben nota!

```
/usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile $PBS_NODEFILE -np $NCPU $PBS_O_WORKDIR/Hello
```


Eseguire un programma: esempio di job-script

Vediamo nel dettaglio la compilazione e l'esecuzione...

```
/usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS_O_WORKDIR/Hello $PBS_O_WORKDIR/Hello.c
```



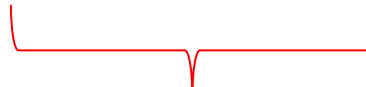
Nome dell'eseguibile che si vuole creare (path assoluto)

```
/usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile $PBS_NODEFILE -np $NCPU $PBS_O_WORKDIR/Hello
```

Eeguire un programma: esempio di job-script

Vediamo nel dettaglio la compilazione e l'esecuzione...

```
/usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS_O_WORKDIR/Hello $PBS_O_WORKDIR/Hello.c
```



Nome del sorgente che si vuole compilare (path assoluto)

```
/usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile $PBS_NODEFILE -np $NCPU $PBS_O_WORKDIR/Hello
```

Eseguire un programma: esempio di job-script

Vediamo nel dettaglio la compilazione e l'esecuzione...

```
/usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS_O_WORKDIR/Hello $PBS_O_WORKDIR/Hello.c
```

Elenco delle macchine utilizzabili

```
/usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile $PBS_NODEFILE -np $NCPU $PBS_O_WORKDIR/Hello
```

Eseguire un programma: esempio di job-script

Vediamo nel dettaglio la compilazione e l'esecuzione...

```
/usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS_O_WORKDIR/Hello $PBS_O_WORKDIR/Hello.c
```

Numero di processi che si devono lanciare

```
/usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile $PBS_NODEFILE -np  $NCPU $PBS_O_WORKDIR/Hello
```

Eeguire un programma: esempio di job-script

Vediamo nel dettaglio la compilazione e l'esecuzione...

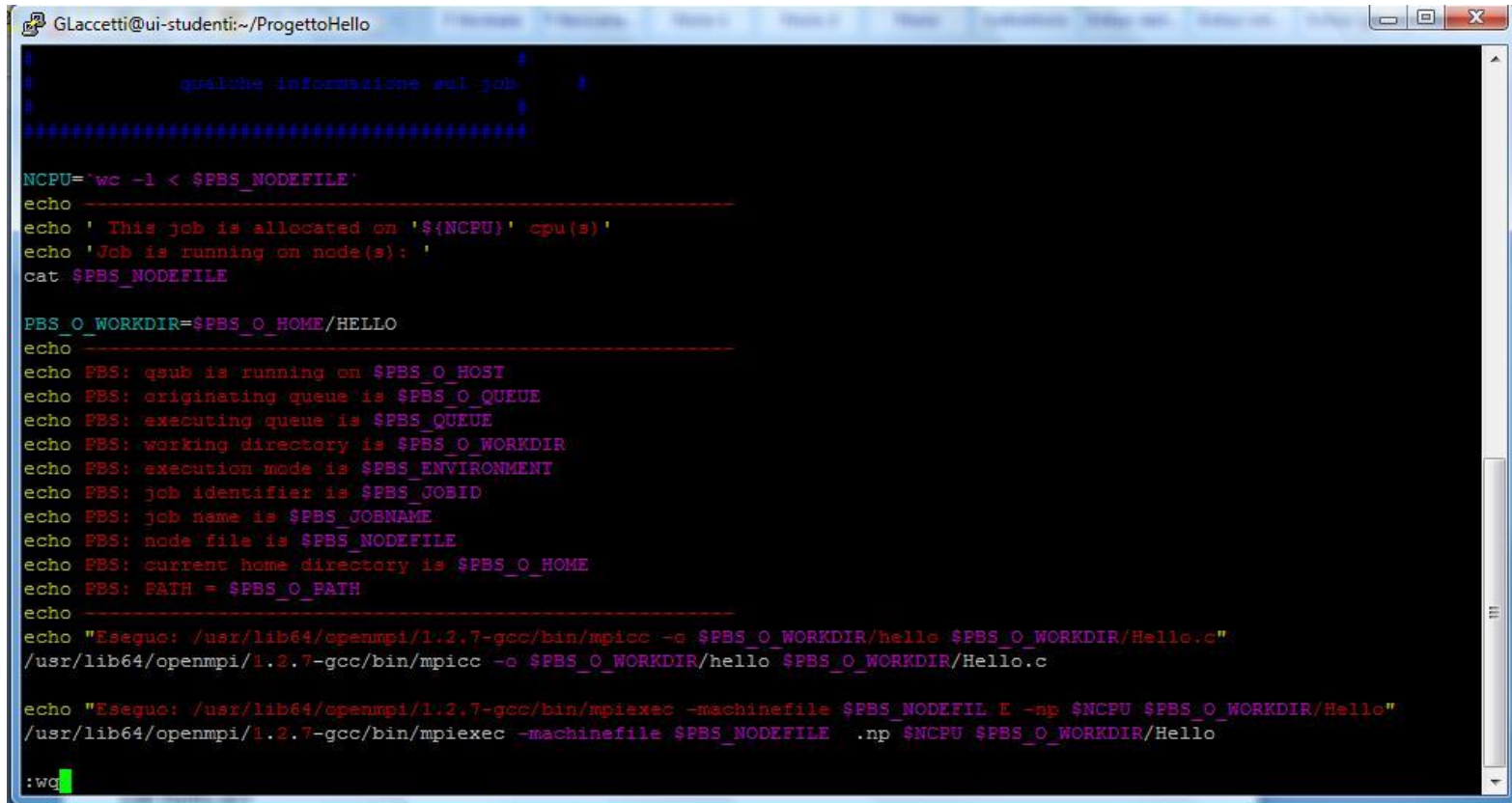
```
/usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS_O_WORKDIR/Hello $PBS_O_WORKDIR/Hello.c
```

Eseguibile da lanciare

```
/usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile $PBS_NODEFILE -np $NCPU $PBS_O_WORKDIR/Hello
```

Eseguire un programma

Una volta che si è scritto il file PBS



```
Glaccetti@ui-studenti:~/ProgettoHello
#
#               qualche informazione sul job               #
#
#####

NCPU=`wc -l < $PBS_NODEFILE`
echo
echo ' This job is allocated on '${NCPU}' cpu(s)'
echo 'Job is running on node(s): '
cat $PBS_NODEFILE

PBS_O_WORKDIR=$PBS_O_HOME/HELLO
echo
echo -----
echo PBS: qsub is running on $PBS_O_HOST
echo PBS: originating queue is $PBS_O_QUEUE
echo PBS: executing queue is $PBS_QUEUE
echo PBS: working directory is $PBS_O_WORKDIR
echo PBS: execution mode is $PBS_ENVIRONMENT
echo PBS: job identifier is $PBS_JOBID
echo PBS: job name is $PBS_JOBNAME
echo PBS: node file is $PBS_NODEFILE
echo PBS: current home directory is $PBS_O_HOME
echo PBS: PATH = $PBS_O_PATH
echo
echo -----
echo "Eseguo: /usr/lib64/openmpi/1.2.7-gcc/bin/mpicc -o $PBS_O_WORKDIR/hello $PBS_O_WORKDIR/Hello.c"
/usr/lib64/openmpi/1.2.7-gcc/bin/mpicc -o $PBS_O_WORKDIR/hello $PBS_O_WORKDIR/Hello.c

echo "Eseguo: /usr/lib64/openmpi/1.2.7-gcc/bin/mpiexec -machinefile $PBS_NODEFILE -np $NCPU $PBS_O_WORKDIR/Hello"
/usr/lib64/openmpi/1.2.7-gcc/bin/mpiexec -machinefile $PBS_NODEFILE -np $NCPU $PBS_O_WORKDIR/Hello

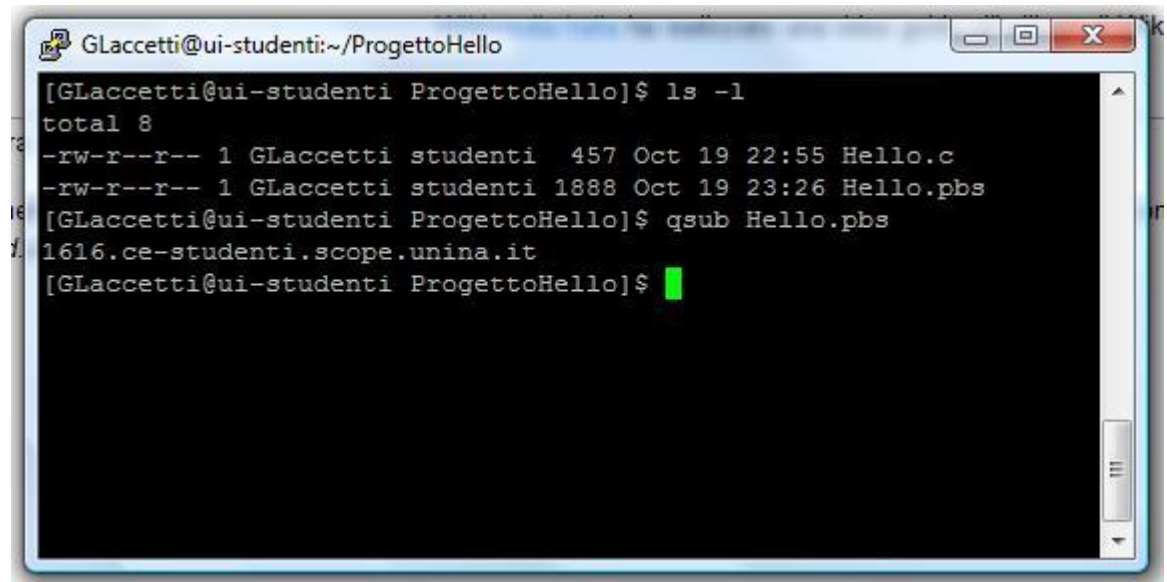
:wc
```

Eseguire un programma

Una volta che si è scritto il file PBS, e si è quindi preparata l'esecuzione parallela, si lancia con il comando

qsub Hello.pbs

qsub è in grado di interpretare le direttive contenute nel PBS

A terminal window titled 'GLaccetti@ui-studenti:~/ProgettoHello' showing the execution of the 'qsub' command. The user first runs 'ls -l' to list files, showing 'Hello.c' and 'Hello.pbs'. Then, they run 'qsub Hello.pbs', which returns the job ID '1616.ce-studenti.scope.unina.it'.

```
GLaccetti@ui-studenti:~/ProgettoHello
[GLaccetti@ui-studenti ProgettoHello]$ ls -l
total 8
-rw-r--r-- 1 GLaccetti studenti 457 Oct 19 22:55 Hello.c
-rw-r--r-- 1 GLaccetti studenti 1888 Oct 19 23:26 Hello.pbs
[GLaccetti@ui-studenti ProgettoHello]$ qsub Hello.pbs
1616.ce-studenti.scope.unina.it
[GLaccetti@ui-studenti ProgettoHello]$
```

Eseguire un programma

Una volta che si è scritto il file PBS, e si è quindi preparata l'esecuzione parallela, si lancia con il comando

```
qsub Hello.pbs
```

qsub è in grado di interpretare le direttive contenute nel PBS

bisognerà aspettare che il job venga gestito dal sistema (è in coda con altri job) e che termini la sua esecuzione.

A questo punto sarà possibile visualizzare l'output e l'error con i comandi:

```
cat Hello.err
```

```
cat Hello.out
```


Eseguire un programma

Per visualizzare i job:

qstat

Per visualizzare la coda ed il loro stato

(E=eseguito, R=in esecuzione, Q= è stato accodato):

qstat -q

Per eliminare un job dalla coda:

qdel jobid

Esercizi

Scrivere, compilare ed eseguire sul cluster attraverso il PBS i seguenti esercizi.
Inviare il codice, il PBS e uno o più esempi di output a valeria.mele@unina.it

1. N processi. Ognuno stampa il proprio identificativo nel communicator globale
2. N processi. In input: intero $M > 200$.
Ogni processo con identificativo > 0 divide M per il proprio identificativo. Il processo 0 divide M per il numero di processi N.
Ogni processo stampa il risultato ottenuto.

Entro una settimana

FINE LEZIONE