**MareNostrum4 User Guide**  
http://www.bsc.es/user-support/mn4.php

*Remember to include the reservation name in your job scripts:*  
#SBATCH --reservation=MN4PATC19

---

**Hands-on II. MPI basic exercises**

Go to mpi dir, there you can find a folder for each exercise. You can use the mpich man pages for the MPI calls: http://www.mpich.org/static/docs/v3.2/

**Exercise 0**

Using MPI try to print out:  
Hello world, I am proc X of total Y

With a total number of tasks $Y = 4$ and $X$ ranging from 0 to 3. After having compiled the code, try to launch it as a batch job. You have to complete hello.c and job.cmd with proper values. Once completed, you can compile with  
'make'

And execute with  
'make submit'

HELP:  
int MPI_Init(int *argc, char ***argv)  
int MPI_Comm_size(MPI_Comm comm, int *size)  
int MPI_Comm_rank(MPI_Comm comm, int *rank)  
int MPI_Finalize(void)
Exercise 1

Write a code using point to point communication that makes two processes send each other an array of floats containing their rank. Each of the processes will declare two float arrays, A and B, of a fixed dimension (10000). All of the elements of the array A will be initialized with the rank of the process. Then, A and B will be used as the buffers for SEND and RECEIVE, respectively. The program terminates with each process printing out one element of the array B.

The program should follow this scheme:

Once you have compiled the code using ‘make’ you can try it doing: make submit

The Output out-exercise2 should look like:
   I am task 0 and I have received b(0) = 1.00
   I am task 1 and I have received b(0) = 0.00

HELP:
int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
Exercise 2

Let’s extend the previous exercise and let the program be executed with more than 2 processes. Each process sends the A array to its right process (the last one sends it to the process 0) and receive B from its left one (the first one receive from N). The send receive mechanism is represented in this figure:

![Diagram of send receive mechanism](image)

HINT: You can use module ‘%’ function in C.

The output should look like (the order may change):
- I am task 3 and I have received b(0) = 2.00
- I am task 1 and I have received b(0) = 0.00
- I am task 2 and I have received b(0) = 1.00
- I am task 0 and I have received b(0) = 3.00

HELP:
- int MPI_Isend(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
Exercise 3

Task 0 initializes a variable to a given value (in this case 47), then modifies the variable (for example, by calculating the square of its value) and finally broadcasts it to all the others tasks. The program will print the value of this number before and after the Broadcast.

HELP:

```c
int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm );
```
Hands-on II. MKL (Optional)

Go to mkl directory.

This hands-on explains the auto-parallelization report features of Intel compilers. It also introduces an optimized mathematical library, Intel MKL, for both Intel and GNU compilers and it's differences with the naive approach.

Actions

1. Read compile.sh and Makefile
2. Execute compile.sh
3. Run mm.icc and mm-mkl.icc and mm-mkl.gcc with 256 as input:
   
   ./mm.icc 256
   ./mm-mkl.icc 256
   ./mm-mkl.gcc 256
4. Try larger inputs (512, 1024, 2048) and compare time differences between gcc and icc and between mkl and non-mkl. You will need to write a jobscript and use a batch job for larger inputs.
5. [Optional]: Check the -vec-report output for 'make mm-rep.icc' and the differences between mm.icc, mm-mkl.icc and mm-vec.icc
6. [Optional]: Modify Makefile and sources to make mm.icc as fast as possible using OpenMP and the auto-vectorization features.