# Hybrid Programming with MPI and OpenMP

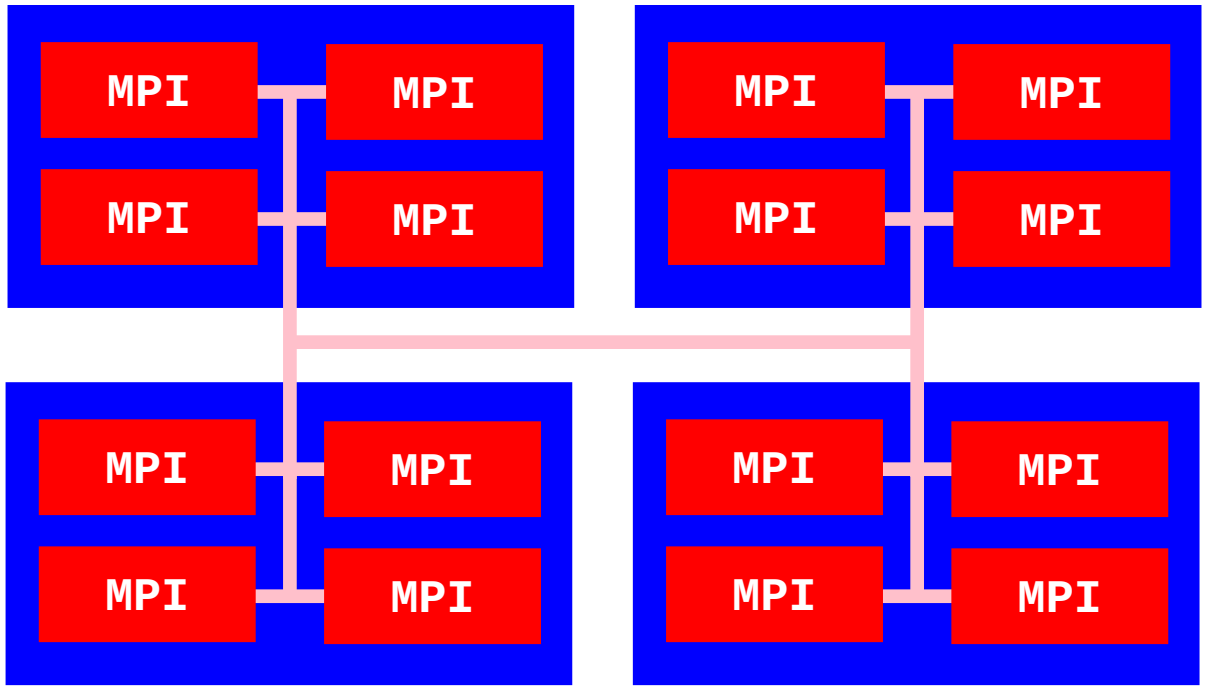
Ricardo Rocha and Fernando Silva

Computer Science Department
Faculty of Sciences
University of Porto

**Parallel Computing 2015/2016**

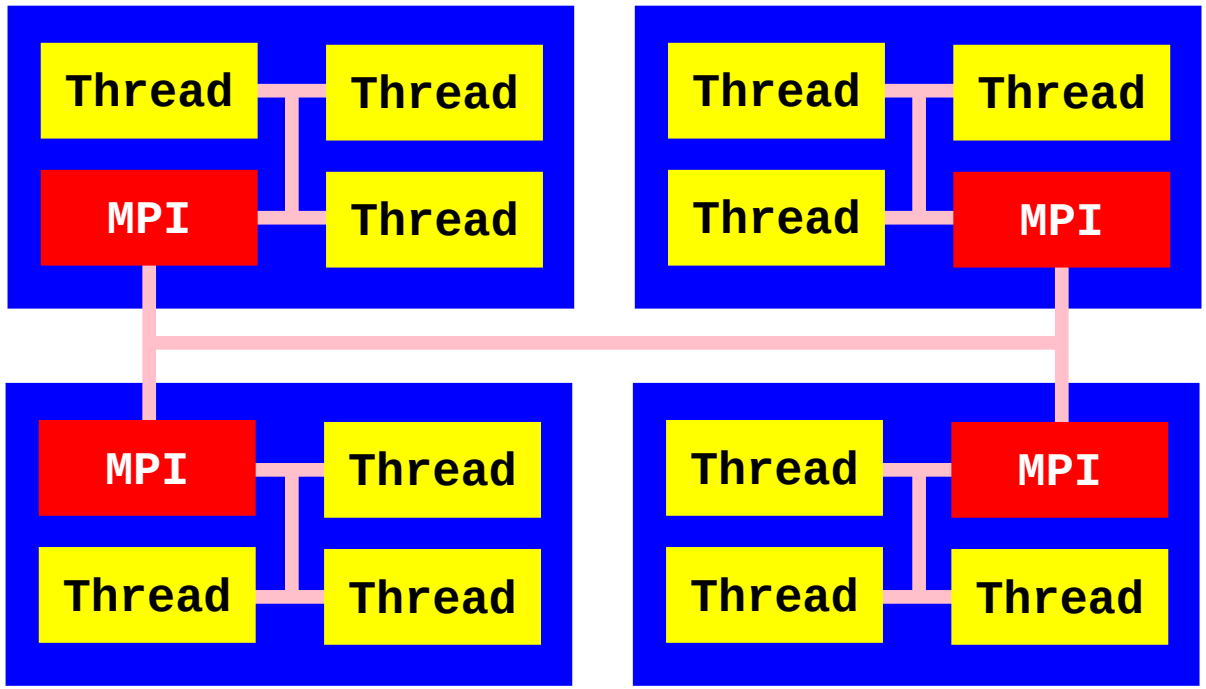
# MPI on Clusters

On a cluster of multiprocessors (or multicore processors), we can execute a parallel program by having a MPI process executing in each processor (or core). It may be the case that multiple MPI processes execute in the same multiprocessor (or multicore processor), but still the interactions among those processes are based on message passing.



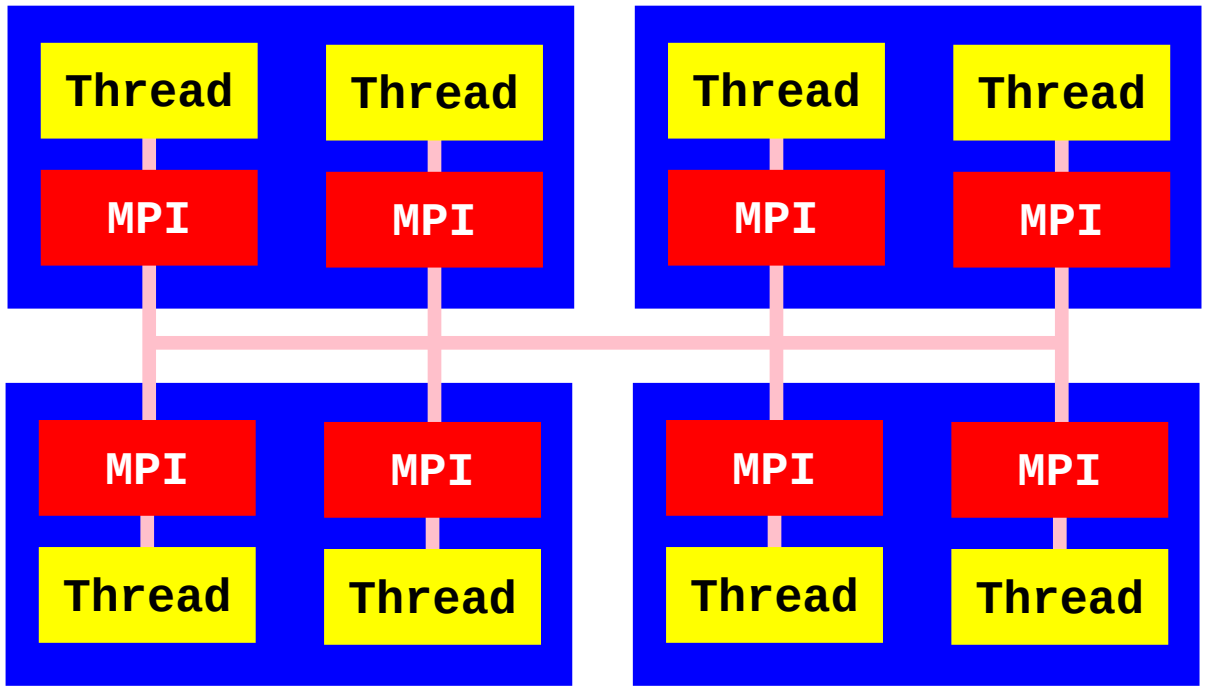
# MPI and OpenMP on Clusters

A different approach is to design an **hybrid program** in which only one MPI process executes in each multiprocessor (or multicore processor), and then launch a set of threads equal to the number of processors (or cores) in each machine to execute the parallel regions of the program.



# MPI and OpenMP on Clusters

As an alternative, we could combine both strategies and **adapt the division between MPI processes and threads** that optimizes the use of the available resources without violating possible constraints or requirements of the problem being solved.



# Hybrid Programming with MPI and OpenMP

Hybrid programming **adapts perfectly to current architectures** based on clusters of multiprocessors and/or multicore processors, as it induces less communication among different nodes and increases performance of each node without having to increase memory requirements.

Applications with **two levels of parallelism** may use MPI processes to exploit large grain parallelism, occasionally exchanging messages to synchronize information and/or share work, and use threads to exploit medium/small grain parallelism by resorting to a shared address space.

Applications with constraints or requirements that may limit the number of MPI processes that can be used (e.g., Fox's algorithm), may take advantage of OpenMP to exploit the remaining computational resources.

Applications for which **load balancing is hard** to achieve with only MPI processes, may benefit from OpenMP to balance work, by assigning a different number of threads to each MPI process as a function of its load.

# Hybrid Programming with MPI and OpenMP

The simplest and safe way to combine MPI with OpenMP is to **never use the MPI calls inside the OpenMP parallel regions**. When that happens, there is no problem with the MPI calls, given that only the master thread is active during all MPI communications.

```
main(int argc, char **argv) {
  ...
  MPI_Init(&argc, &argv);
  ... // master thread only --> MPI calls here
  #pragma omp parallel
  {
    ... // team of threads --> no MPI calls here
  }
  ... // master thread only --> MPI calls here
  MPI_Finalize();
  ...
}
```

# Matrix-Vector Product (mpi-omp-matrixvector.c)

The product of a matrix `mat[ROWS,COLS]` and a column vector `vec[COLS]` is a row vector `prod[ROWS]` such that each `prod[i]` is the scalar product of row `i` of the matrix by the column vector.

If we have `P` MPI processes and `T` threads per MPI process, then each MPI process can compute `ROWS/P` (`P_ROWS`) elements of the result and each thread can compute `P_ROWS/T` (`T_ROWS`) elements of the result.

```
// distribute the matrix
MPI_Scatter(mat, P_ROWS * COLS, MPI_INT, submat, P_ROWS * COLS,
            MPI_INT, ROOT, MPI_COMM_WORLD);
// calculate the matrix-vector product
#pragma omp parallel for num_threads(NTHREADS)
for (i = 0; i < P_ROWS; i++)
  subprod[i] = scalar_product(&submat[i * COLS], vec, COLS);
// gather the submatrices
MPI_Gather(subprod, P_ROWS, MPI_INT, prod, P_ROWS, MPI_INT, ROOT,
           MPI_COMM_WORLD);
```

# Matrix Multiplication with Fox's Algorithm

Given two matrices `A[N][N]` and `B[N][N]` and `P` processes (`P=Q*Q`), Fox's algorithm divides both matrices in `P` submatrices of size `(N/Q)*(N/Q)` and assigns each submatrix to one of the `P` available processes. To compute the multiplication of its submatrix, each process only communicates with the processes in the same row and in the same column of its own submatrix.

```
for (stage = 0; stage < Q; stage++) {
  bcast = ... // pick a submatrix A in each row of processes and ...
  MPI_Bcast(...); // ... send it to all processes in the same row
  // multiply the received submatrix A with current submatrix B
  #pragma omp parallel for private(i,j,k)
  for (i = 0; i < N/Q; i++)
    for (j = 0; j < N/Q; j++)
      for (k = 0; k < N/Q; k++)
        C[i][j] += A[i][k] * B[k][j];
  MPI_Send(...) // send submatrix B to process above and ...
  MPI_Recv(...) // ... receive the submatrix B from process below
}
```

# MPI Thread Safe

If a program is parallelized so that it has MPI calls inside OpenMP parallel regions, then multiple threads can call the same MPI communications and at the same time. For this to be possible, it is necessary that the **MPI implementation be thread safe**.

```
#pragma omp parallel private(tid)
{
  tid = omp_get_thread_num();
  if (tid == id1)
    mpi_calls_f1(); // thread id1 makes MPI calls here
  else if (tid == id2)
    mpi_calls_f2(); // thread id2 makes MPI calls here
  else
    do_something();
}
```

MPI-1 does not support multithreading. Such support was only considered with MPI-2 with the introduction of the `MPI_Init_thread()` call.

# Initializing MPI with Support for Multithreading

```
int MPI_Init_thread(int *argc, char ***argv, int required,
  int *provided)
```

`MPI_Init_thread()` initializes the MPI execution environment (similarly to `MPI_Init()`) and defines the support level for multithreading:

- `required` is the aimed support level
- `provided` is the support level provided by the MPI implementation

The support level for multithreading can be:

- `MPI_THREAD_SINGLE` – only one thread will execute (the same as initializing the environment with `MPI_Init()`)
- `MPI_THREAD_FUNNELED` – only the master thread can make MPI calls
- `MPI_THREAD_SERIALIZED` – all threads can make MPI calls, but only one thread at a time can be in such state
- `MPI_THREAD_MULTIPLE` – all threads can make simultaneous MPI calls without any constraints

# MPI_THREAD_FUNNELED

With support level `MPI_THREAD_FUNNELED` only the master thread can make MPI calls. One way to ensure this is to protect the MPI calls with the `omp master` directive.

However, the `omp master` directive does not define any implicit synchronization barrier among all threads in the parallel region (at entrance or exit of the `omp master` directive) in order to protect the MPI call.

```
#pragma omp parallel
{
  ...
  #pragma omp barrier // explicit barrier at entrance
  #pragma omp master // only the master thread makes the MPI call
    mpi_call();
  #pragma omp barrier // explicit barrier at exit
  ...
}
```

# MPI_THREAD_SERIALIZED

With support level `MPI_THREAD_SERIALIZED` all threads can make MPI calls, but only one thread at a time can be in such state. One way to ensure this is to protect the MPI calls with the `omp single` directive that allows to define code blocks that should be executed only by one thread.

However, the `omp single` directive does not define an implicit synchronization barrier at the entrance of the directive. In order to protect the MPI call, it is necessary to set an explicit `omp barrier` directive at entrance of the `omp single` directive.

```
#pragma omp parallel
{
  ...
  #pragma omp barrier // explicit barrier at entrance
  #pragma omp single // only one thread makes the MPI call
    mpi_call(); // explicit barrier at exit
  ...
}
```

# MPI_THREAD_MULTIPLE

With support level **MPI_THREAD_MULTIPLE** all threads can make simultaneous MPI calls without any constraints. Given that the implementation is **thread safe**, there is no need for any additional synchronization mechanism among the threads in the parallel region in order to protect the MPI calls.

```
#pragma omp parallel private(tid)
{
  tid = omp_get_thread_num();
  ...
  if (tid == id1)
    mpi_calls_f1(); // thread id1 makes MPI calls here
  else if (tid == id2)
    mpi_calls_f2(); // thread id2 makes MPI calls here
  else
    do_something();
  ...
}
```

# MPI_THREAD_MULTIPLE

The communication among threads of different MPI processes raises the **problem of identifying the thread that is involved in the communication** (as MPI communications only include arguments to identify the ranks of the MPI processes).

A simple way to solve this problem is to **use the tag argument** to identify the thread involved in the communication.

```
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
...
#pragma omp parallel num_threads(NTHREADS) private(tid)
{
  tid = omp_get_thread_num();
  ...
  if (my_rank == 0) // MPI process 0 sends NTHREADS messages
    MPI_Send(a, 1, MPI_INT, 1, tid, MPI_COMM_WORLD);
  else if (my_rank == 1) // MPI process 1 receives NTHREADS messages
    MPI_Recv(b, 1, MPI_INT, 0, tid, MPI_COMM_WORLD, &status);
  ...
}
```