OpenMP* / MPI Hybrid Programming

Agenda

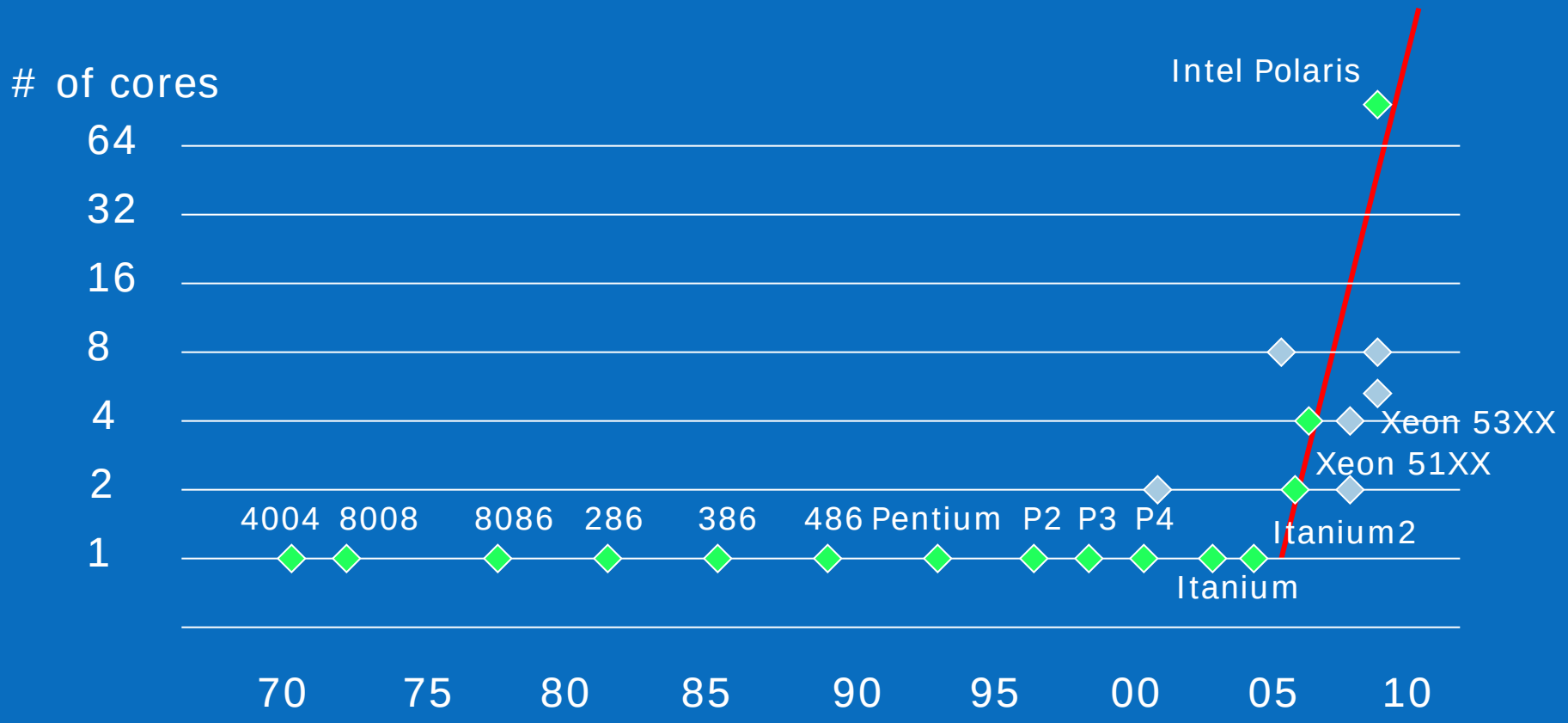
Why Hybrid Programming?

Testing Thread-safety of MPI Implementations

Master Communication

Multi-threaded Communication

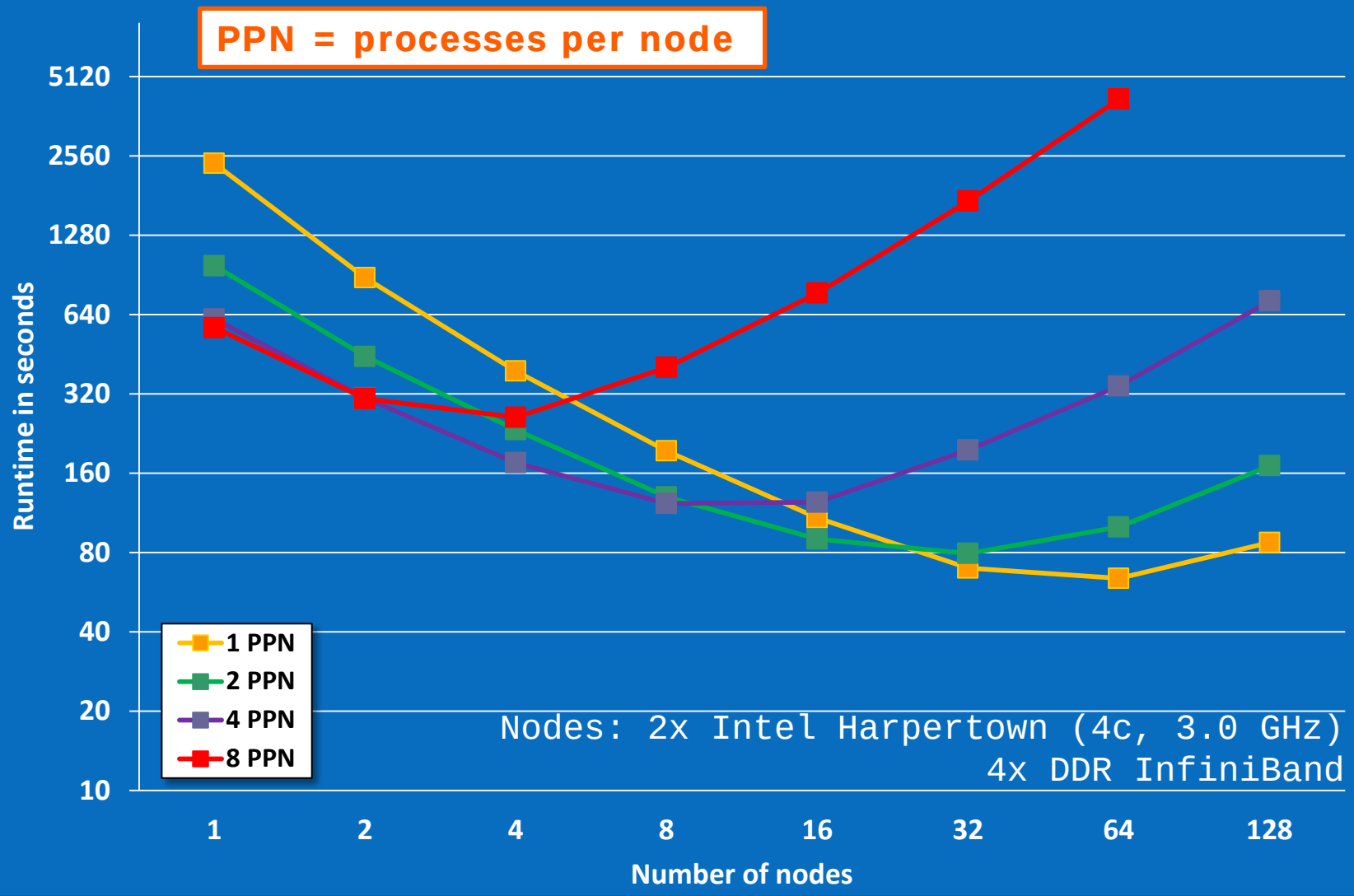
Why Hybrid Programming?



Core counts will steadily increase during the next years.

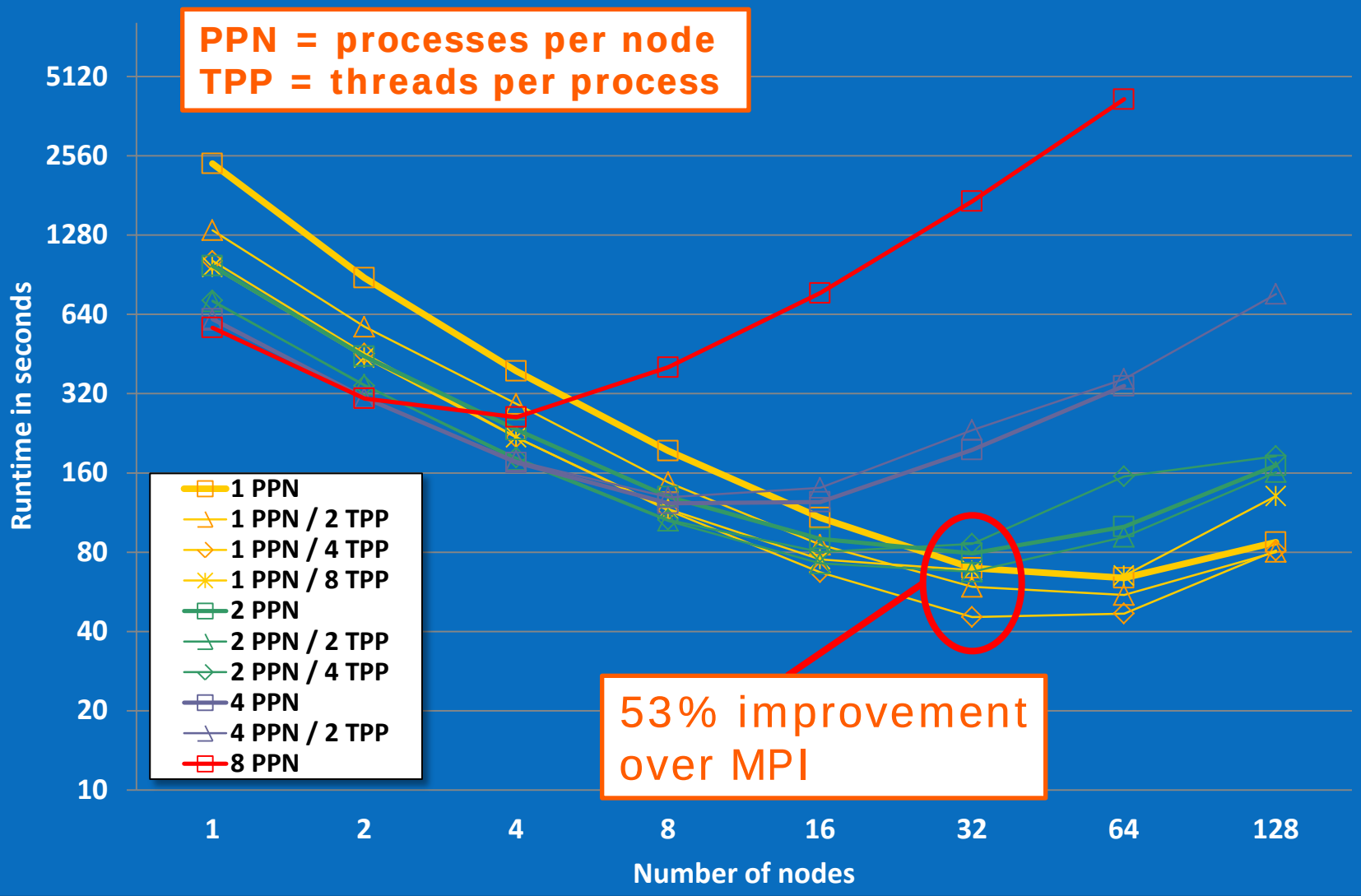
Cluster nodes will become highly parallel machines.

Why Hybrid Programming? MPI only



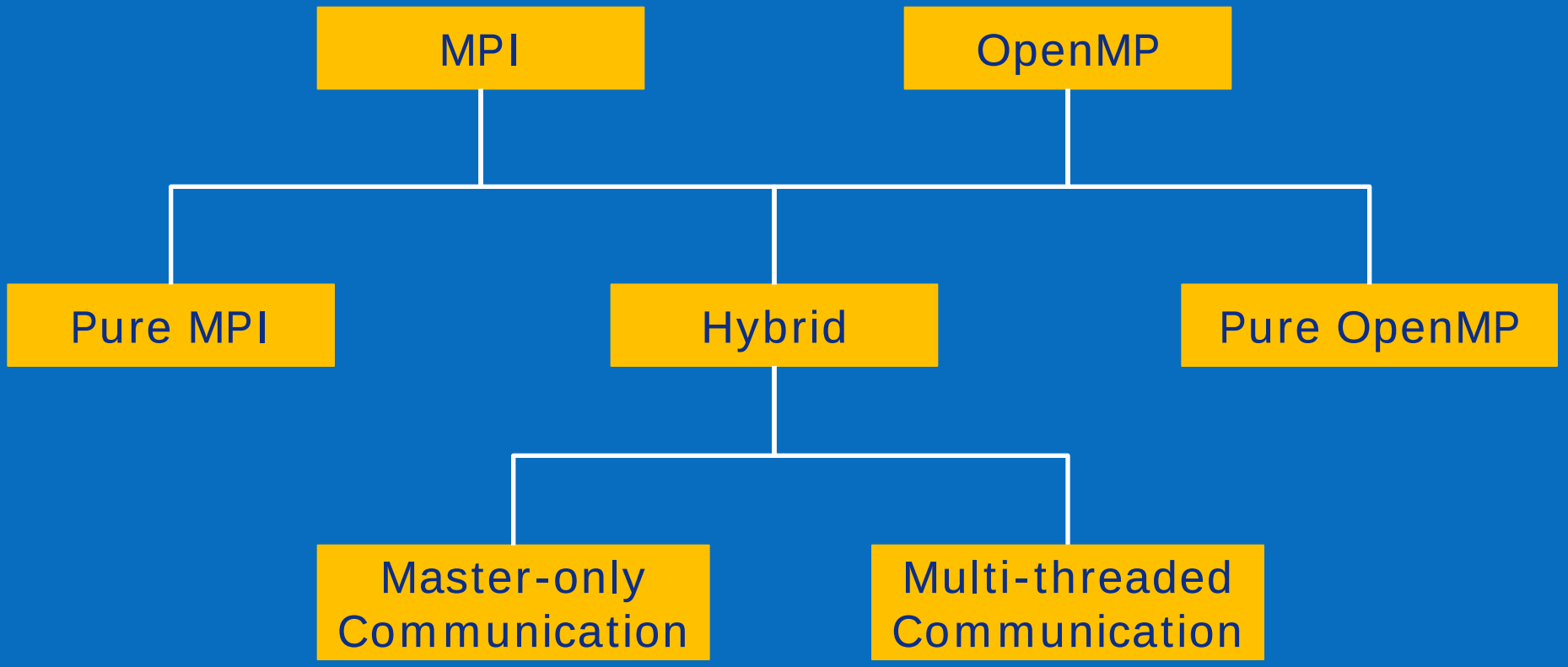
Simulation of Free-Surface Flows, Finite Element CFD solver written in Fortran and C
Figure kindly provided by HPC group of the Center of Computing and Communication, RWTH Aachen, Germany

Why Hybrid Programming? OpenMP / MPI



Simulation of Free-Surface Flows, Finite Element CFD solver written in Fortran and C
Figure kindly provided by HPC group of the Center of Computing and Communication, RWTH Aachen, Germany

Hybrid Programming with OpenMP and MPI



Using OpenMP Together with MPI

OpenMP and MPI blend well with each other

- OpenMP statements express multi-threading in processes
- MPI interconnects processes to form a distributed application

Notes:

- Not all MPI implementations are thread-safe.
- In older MPI implementations, only the OpenMP master thread should invoke MPI primitives.
- Implementations conforming to MPI 2.1 offer an API to query threading capabilities.

Agenda

Why Hybrid Programming?

Testing Thread-safety of MPI Implementations

Master Communication

Multi-threaded Communication

MPI_Init_thread

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

MPI_INIT_THREAD(REQUIRED, PROVIDED, IERROR)
        INTEGER REQUIRED, PROVIDED, IERROR
```

Initialize the MPI runtime and prepare for threaded execution

- Should be called instead of MPI_Init in multi-threaded applications
- Programmer specifies threading level *needed* for execution
- MPI library responds with threading level *supported*.

MPI_Init_thread

argc	C: `argc` argument of `main`
argv	C: `argv` argument of `main`
required	Threading level required (see below)
provided	Threading level supported (see below)

MPI_THREAD_SINGLE	Application only uses one thread
MPI_THREAD_FUNNELED	Application is multi-threaded, but only the master thread calls MPI routines
MPI_THREAD_SERIALIZED	Application is multi-threaded, but only one thread at a time calls MPI routines
MPI_THREAD_MULTIPLE	Multiple threads call MPI routines without restrictions

Note: MPI_THREAD_SINGLE < MPI_THREAD_FUNNELED < MPI_THREAD_SERIALIZED < MPI_THREAD_MULTIPLE

MPI_Is_thread_main & MPI_Query_thread

```
int MPI_Query_thread(int *provided)

MPI_QUERY_THREAD(PROVIDED, IERROR)
        INTEGER PROVIDED, IERROR
```

Queries the level of threading support of the MPI library.

```
int MPI_Is_thread_main(int *flag)

MPI_IS_THREAD_MAIN(FLAG, IERROR)
        LOGICAL FLAG
        INTEGER IERROR
```

Tests if the calling thread is the main thread.

Agenda

Why Hybrid Programming?

Testing Thread-safety of MPI Implementations

Master Communication

Multi-threaded Communication

Master-only Communication – Pattern 1 (MPI_THREAD_FUNNELED)

Only the sequential code invokes MPI routines.

```
void example() {
    /* main/master thread */
    MPI_Recv(/* receive data */);
#pragma omp parallel
    {
        /* multi-threaded execution */
    }

    /* main/master thread */
    MPI_Send(/* send back data */);
}
```

Master-only Communication – Pattern 2 (MPI_THREAD_FUNNELED)

Only the master thread invokes MPI routines.

```
void example() {
   #pragma omp parallel
        {
   #pragma omp master
            {
                MPI_Recv(/* receive data */);
            }
   #pragma omp barrier

            /* multi-threaded execution */

   #pragma omp master
            {
                MPI_Send(/* send back data */);
            }
   #pragma omp barrier
        }
   }
```

Discussion of MPI_THREAD_FUNNELED

Limited degree of parallelism when communication occurs:

- Only the master thread of the process communicates.
- Parallel OpenMP execution has to be stopped while communicating.

Performance suffers:

- The additional barrier needed introduces overheads and causes additional cache flushes.
- On NUMA architectures with first-touch policies, the master thread “touches” data when receiving a message.
- OpenMP threads might frequently access data stored remotely on the NUMA node of the master thread.

But: Easy to use and implement even with MPI libraries that are not fully multi-threaded.

Global Barriers

Global barriers involve a 2-stage barrier synchronization:

- intra-process OpenMP barrier
- inter-process MPI barrier

```
void barrier() {
   #pragma omp parallel
        {
             /* multi-threaded execution */

   #pragma omp barrier
   #pragma omp master
             {
                  MPI_Barrier(/* arguments */);
             }
   #pragma omp barrier

             /* multi-threaded execution */
        }
   }
```

Intra-process OpenMP barrier

Inter-process MPI barrier

Agenda

Why Hybrid Programming?

Testing Thread-safety of MPI Implementations

Master Communication

Multi-threaded Communication

Serialized Communication (MPI_THREAD_SERIALIZED)

Any thread invokes MPI calls, but communication is serialized by means of critical regions.

```
void example() {
   #pragma omp parallel
        {
   #pragma omp critical (MPI)
            {
                MPI_Recv(/* receive data */);
            }

        /* multi-threaded execution */

   #pragma omp critical (MPI)
            {
                MPI_Send(/* send back data */);
            }
        }
   }
```

Specify name for critical region to associate it with MPI communication.

Note: OpenMP lock routines could be used as well.

Discussion of MPI_THREAD_SERIALIZED

Limited degree of parallelism when communication occurs:

- OpenMP execution does not need to stop while communicating.
- However, only one thread of the process communicates at a time.

Performance does not suffer as with MPI_THREAD_FUNNELED

- Expensive barrier synchronization is avoided.
- Lock operations are not as expensive, but cache flushes still occur.
- On NUMA architectures with first-touch policies, the communicating threads “touch” the data received and cause allocation on the local NUMA node.

Multi-threaded Communication (MPI_THREAD_MULTIPLE)

Any thread invokes MPI calls and communicates with other threads of other processes.

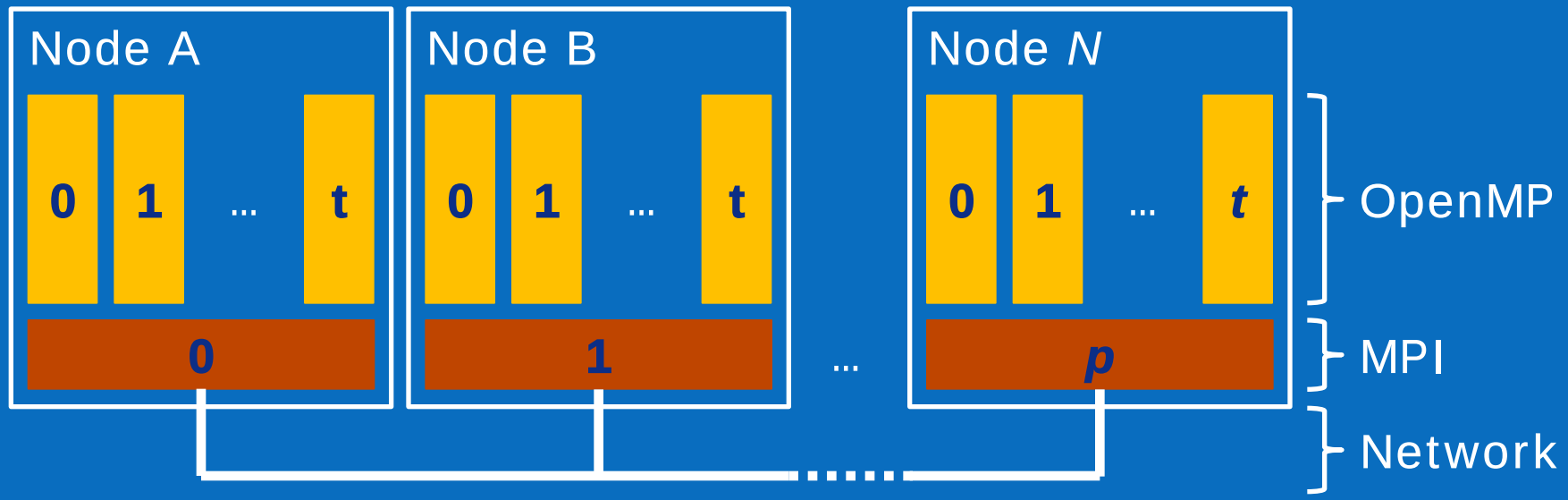
```
void example() {
   #pragma omp parallel
       {
           MPI_Recv(/* receive data */);

           /* multi-threaded execution */

           MPI_Send(/* send back data */);
       }
   }
```

How to address OpenMP threads?

Addressing OpenMP Threads



OpenMP and MPI use different numbering schemes.

- OpenMP threads IDs are not recognized as valid MPI ranks.
- Programmers need to work around this restriction.
- Solution: Use message tags to address threads.

Addressing OpenMP Threads

Sending messages:

```
MPI_Send(buf, count, dtype, dest, threadID, comm)

call MPI_SEND(BUF, COUNT, DTYPE, DEST, THREADID, COMM, ERR)
```

Receiving messages:

```
MPI_Recv(buf, count, dtype, source, omp_get_thread_num(),
         comm, status)

call MPI_Recv(BUF, COUNT, DTYPE, SOURCE, OMP_GET_THREAD_NUM(),
              COMM, STATUS, ERR)
```

Addressing OpenMP Threads

If tags are needed to distinguish message types, thread IDs and message tags need to be fused to a single value:

```
#define THREADID(id, tag) (((id) << 16) | (tag))

#define THISTHREAD(tag) (omp_get_thread_num() << 16 | (tag))
```

(These macros leave 16 bits for the message tags.)

Usage examples:

```
MPI_Send(buf, count, dtype, dest, THREADID(1, 42), comm)

MPI_Recv(buf, count, dtype, source, THISTHREAD(42),
         comm, status)
```

Global Barriers

Global barriers still involve a 2-stage barrier synchronization:

- intra-process OpenMP barrier
- inter-process MPI barrier

```
void barrier() {
   #pragma omp parallel
        {
            /* multi-threaded execution */

            MPI_Barrier(/* arguments */);

            /* multi-threaded execution */
        }
   }
```

This does not establish a barrier for OpenMP threads!

Collective operations need to be performed by one thread only.

The Good, the Bad, and the Ugly

The Good

- OpenMP and MPI blend well with each other if certain rules are respected by programmers.
- Hybrid programs can greatly improve performance of parallel codes.

The Bad

- Programmers need to be aware of the issues of hybrid programming.
- Not all MPI implementations are thread-safe (but situation improves)

The Ugly

- Work-arounds needed to make things work.
- Hybrid programming does not automatically lead to efficient programs. What's the best setting for PPN and TPP for a given machine?

References

Message Passing Interface (MPI) Forum, *MPI: A Message-Passing Interface Standard*, Version 2.1, 2008, available at www.mpi-forum.org.

The OpenMP Architecture Review Board, *OpenMP Application Program Interface*, Version 3.0, 2008, available at www.openmp.org.

Michael J. Quinn, *Parallel Programming in C with MPI and OpenMP*, McGraw-Hill, 2004.


intel®
Leap ahead™