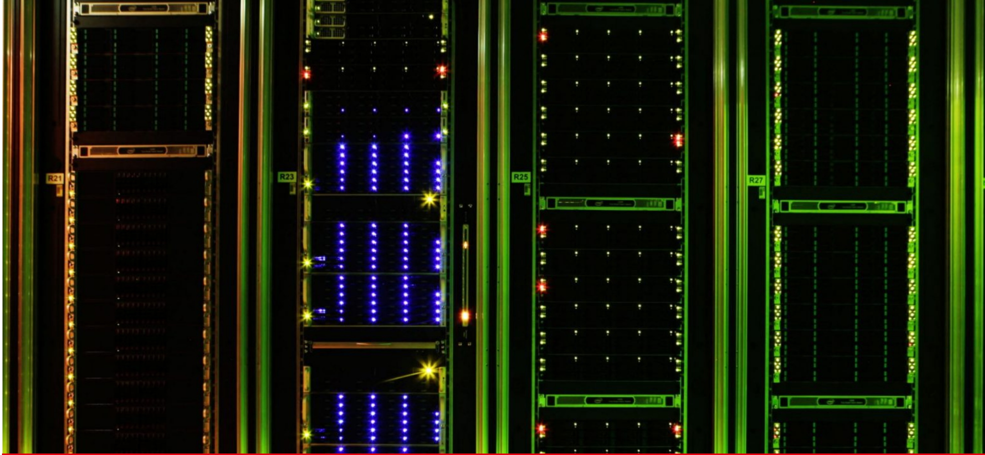
EPFL



Parallel Programming

Single-core optimization, MPI, OpenMP, and hybrid programming

Nicolas Richart
Emmanuel Lanti

Course based on V. Keller's lecture notes

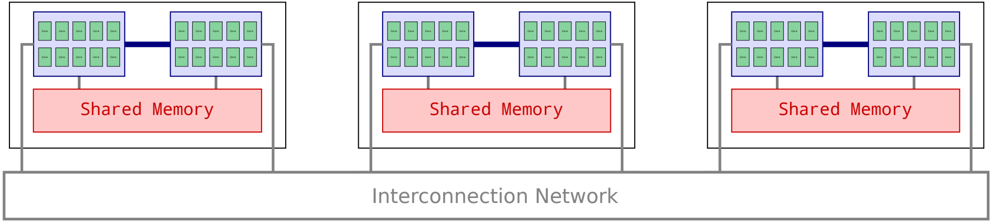
15th - 19th of November 2021

SCITAS

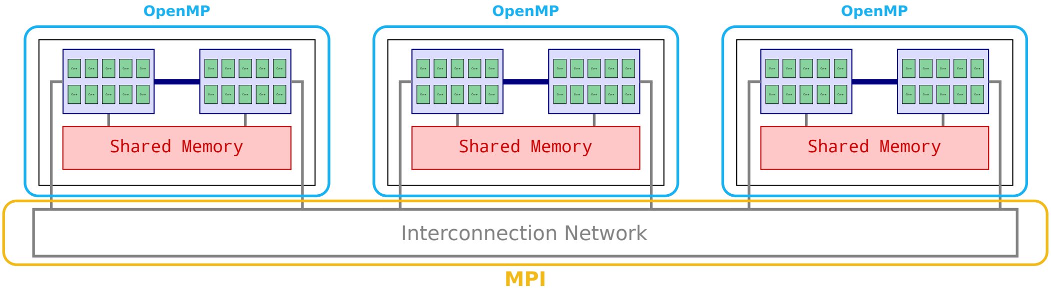
What will we learn today?

- Hybrid MPI + OpenMP programming
- Partitioned point-to-point communications
- Matching probe/receive

Situation


Shared Memory
Shared Memory
Shared Memory
Interconnection Network

Situation


OpenMP
OpenMP
OpenMP
Shared Memory
Shared Memory
Shared Memory
Interconnection Network
MPI

- Thread safety?
- Which thread/process can/will call the MPI library?
- MPI process placement in the case of multi-CPU processors?
- Data visibility? OpenMP private?
- Does my problem fits with the targeted machine?
- Levels of parallelism within my problem?

Hybrid MPI/OpenMP hello world

hybrid/hello_world.cc

```cpp
#include <iostream>
#include <mpi.h>
#include <omp.h>

int main(int argc, char *argv[]) {
  int provided, size, rank, nthreads, tid;
  MPI_Init_thread(&argc, &argv, MPI_THREAD_SINGLE, &provided);

  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  #pragma omp parallel default(shared) private(tid, nthreads)
  {
    nthreads = omp_get_num_threads();
    tid = omp_get_thread_num();
    std::printf("Hello from thread %i out of %i from process %i out of %i\n", tid, nthreads, rank,
    ↪ size);
  }
  MPI_Finalize();
  return 0;
}
```

Hybrid MPI/OpenMP hello world

Compilation using the GNU g++ compiler:

```
$> mpicxx -fopenmp hello_world.cc -o hello_world
```

Compilation using the Intel C++ compiler:

```
$> mpiicpc -qopenmp hello_world.cc -o hello_world
```

Submission script the clusters

```bash
#!/bin/bash
#SBATCH --nodes 1
#SBATCH --ntasks 2
#SBATCH --cpus-per-task 3

export OMP_NUM_THREADS=3
srun -n 2 ./hello_world
```

It will start 2 MPI processes each will spawn 3 threads

```
Hello from thread 0 out of 3 from process 0 out of 2
Hello from thread 1 out of 3 from process 0 out of 2
Hello from thread 0 out of 3 from process 1 out of 2
Hello from thread 1 out of 3 from process 1 out of 2
Hello from thread 2 out of 3 from process 0 out of 2
Hello from thread 2 out of 3 from process 1 out of 2
```

Changes to your code

- Change your MPI initialization routine
  - **MPI_Init** is replaced by **MPI_Init_thread**
  - **MPI_Init_thread** has two additional parameters for the level of thread support required, and for the level of thread support provided by the library implementation

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

- Make sure that the *provided* support matches the *required* one

```c
if (provided < required)
MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
```

- Add OpenMP directives as long as you stick to the level of thread safety you specified in the call to **MPI_Init_thread**

The 4 options for thread support

- **MPI_THREAD_SINGLE**
  - Only one thread will execute (no multi-threading)
  - Standard MPI-only application
- **MPI_THREAD_FUNNELED**
  - Only the Master Thread will make calls to the MPI library
  - A thread can determine whether it is the master thread by a call to `MPI_Is_thread_main`
- **MPI_THREAD_SERIALIZED**
  - Only one thread at a time will make calls to the MPI library, but all threads are eligible to make such calls
- **MPI_THREAD_MULTIPLE**
  - Any thread may call the MPI library at any time

The 4 options for thread support

- **MPI_THREAD_SINGLE**
  - Only one thread will execute (no multi-threading)
  - Standard MPI-only application
- **MPI_THREAD_FUNNELED**
  - Only the Master Thread will make calls to the MPI library
  - A thread can determine whether it is the master thread by a call to `MPI_Is_thread_main`
- **MPI_THREAD_SERIALIZED**
  - Only one thread at a time will make calls to the MPI library, but all threads are eligible to make such calls
- **MPI_THREAD_MULTIPLE**
  - Any thread may call the MPI library at any time

In most cases **MPI_THREAD_FUNNELED** provides the best choice for hybrid programs

The 4 options for thread support

- **MPI_THREAD_SINGLE**
  - Only one thread will execute (no multi-threading)
  - Standard MPI-only application
- **MPI_THREAD_FUNNELED**
  - Only the Master Thread will make calls to the MPI library
  - A thread can determine whether it is the master thread by a call to `MPI_Is_thread_main`
- **MPI_THREAD_SERIALIZED**
  - Only one thread at a time will make calls to the MPI library, but all threads are eligible to make such calls
- **MPI_THREAD_MULTIPLE**
  - Any thread may call the MPI library at any time

In most cases **MPI_THREAD_FUNNELED** provides the best choice for hybrid programs

```c
int MPI_Query_thread(int * thread_level_provided);
```

Returns the level of thread support provided by the MPI library

The 4 options for thread support

- Thread support values are monotonic, i.e.
  MPI_THREAD_SINGLE < MPI_THREAD_FUNNELED < MPI_THREAD_SERIALIZED < MPI_THREAD_MULTIPLE
- Different processes in **MPI_COMM_WORLD** can have different thread safety
- The level(s) of provided thread support depends on the implementation
- The rules for thread support attribution are done in the following order:
  - return provided = required
  - return the least supported level such that provided > required
  - return the highest supported level

MPI partitioned communications

- New feature from MPI 4.0 standard (June 2021!)
- We have already talked about persistent point-to-point communications
- Partitioned comms are just persistent comms where the message is constructed in partitions
- Typical case: multi-threading with each thread building a portion of the message

MPI partitioned communications

- Remember the typical cycle for persistent point-to-point communications

  Init (Start Test/Wait)* Free

  where * means zero or more
- Partitioned are very similar

  PInit (PStart PReady)* Free

```c
MPI_Psend_init(msg, parts, count, MPI_INT, dest, tag, info, MPI_COMM_WORLD, &request);
MPI_Start(&request);
#pragma omp parallel for shared(request)
for (int i = 0; i < parts; ++i) {
  /* compute and fill partition #i, then mark ready: */
  MPI_Pready(i, request);
}
while(!flag) {
  /* Do useful work */
  MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
  /* Do useful work */
}
MPI_Request_free(&request);
```

MPI matching probe

- We have already talked about **MPI_Probe** to obtain information about a message waiting to be received
- This is typically used when the size of the message is unknown (probe, allocate, receive)

MPI matching probe

- We have already talked about **MPI_Probe** to obtain information about a message waiting to be received
- This is typically used when the size of the message is unknown (probe, allocate, receive)
- Care must be taken because it is a stateful method:
  *A subsequent receive [...] will receive the message that was matched by the probe, **if no other intervening receive occurs after the probe** [...]*

MPI matching probe

- We have already talked about **MPI_Probe** to obtain information about a message waiting to be received
- This is typically used when the size of the message is unknown (probe, allocate, receive)

- Care must be taken because it is a stateful method:
  *A subsequent receive [...] will receive the message that was matched by the probe, **if no other intervening receive occurs after the probe** [...]*

- Problem with multi-threading!
- Imagine two threads A and B that must do a Probe, Allocation, and Receive

$$A_P \longrightarrow A_A \longrightarrow A_R \longrightarrow B_P \longrightarrow B_A \longrightarrow B_R$$

  but may also be

$$A_P \longrightarrow B_P \longrightarrow B_A \longrightarrow B_R \longrightarrow A_A \longrightarrow A_R$$

  Thread B stole thread A's message!

MPI matching probe

- The solution of this problem is the matching probe
- MPI provides two versions, **MPI_Improbe** and **MPI_Mprobe**
- It allows to receive only a message matching a specific probe

MPI matching probe

- The solution of this problem is the matching probe
- MPI provides two versions, **MPI_Improbe** and **MPI_Mprobe**
- It allows to receive only a message matching a specific probe

- Counter part operation are the matching receive **MPI_Imrecv** and **MPI_Mrecv**
- They are used to receive messages that have been previously matched by a matching probe

Concluding remarks

- Always keep in mind that you are mixing threads and processes
- You will need to test your code performance on every machine
- There are no magic rule on the best configuration to use
- Often 1 MPI task per NUMA region seems to give the best performance