HPC Architectures and Parallelism

Jorge Rodríguez (jorge.rodriguez@bsc.es)

PATC Systems Workshop: Programming MareNostrum4
Index

- Chip in MareNostrum4
- Parallelism (MPI)
Chip in MareNostrum4
Intel® Xeon® Platinum

- 64-bit x86 multi-socket multi-core
- SkyLake microarchitecture:
  - Successor of BroadWell μarch
  - μarch redesign using the same 14nm manufacturing process technology
  - It is a “tock” in Intel’s “tick-tock” manufacturing and design model
  - Greater CPU and GPU performance and reduced power consumption
Intel® Xeon® Platinum

- Intel® Xeon® processor scalable family tech
Intel® Xeon® Platinum

What do we have in MareNostrum?

- 3456 compute nodes, each has:
  - 2 Intel® Xeon® Platinum 8160 24C @ 2.1GHz
  - 96GB of RAM
Intel® Xeon® Platinum

- What do we have in MareNostrum?
  - 3456 compute nodes, each has:
    - 2 Intel® Xeon® Platinum 8160 24C @ 2.1GHz
    - 96GB of RAM
Intel® Xeon® Platinum

● Some more technical details of 8160:
  ○ 24 physical cores (48 threads)
  ○ Intel® AVX-512 with 32 DP flops per core
  ○ Data center optimized cache hierarchy:
    ■ 1MB L2 per core
    ■ 33MB non-inclusive L3
  ○ Memory DDR4 up to 2666MHz
  ○ 3 UPI links
    ■ Intel® Ultra Path Interconnect (UPI)
    ■ High speed, point-to-point interconnect bus between the processors
    ■ Increased bandwidth and performance over Intel® QPI
  ○ Power consumption 70W - 205W
Intel® Xeon® Platinum

- Instruction sets:
Intel® Xeon® Platinum

- Performance and efficiency with Intel® AVX.
Intel® Xeon® Platinum

- How to manage them in MareNostrum?
  - Recompile your code!
  - Try to take benefit of AVX-512 instructions
  - Try to vectorize your code
  - Module for optimization flags:
    - module load opt/flags
Parallelism
Parallel programming models

- Distributed memory parallelism:
  - MPI (Message Passing Interface) is the most used programming model in HPC
    - http://www.mpi-forum.org/
    - http://www.mcs.anl.gov/mpi
  - Processes can have threads in the same node
    - Sharing the same address space
  - MPI is used for the communication
    - Separate address space
  - Interprocess communication consists of:
    - Synchronization
    - Moving data (P2P and Collective)
Distributed Memory Multiprocessors

- Each processor has local memory that is accessible through a fast path
- The different nodes are connected as I/O devices with (potentially) slower interconnect
- Local memory access is a lot faster than remote memory – non-uniform memory access (NUMA)
- Advantage: can be built with commodity processors and many applications will perform well thanks to locality
Parallel programming models

- Distributed memory parallelism:
  - MPI (Message Passing Interface)
Basic-Code-Fragment

Initialization and Exit:

1. ```
#include <mpi.h>
```  
2. ```
... 
```  
3. ```
int main(int argc, char *argv[]) 
```  
4. ```
{ 
```  
5. ```
   MPI_Init(&argc, &argv); 
```  
6. ```
... 
```  
7. ```
   MPI_Finalize(); 
```  
8. ```
} 
```
1.1 – Example

• Small MPI code example

```c
#include “mpi.h”
#include <stdio.h>

int main (int argc, char **argv)
{
    int rank, size;
    MPI_Init (&argc, &argv);
    MPI_Comm_rank ( MPI_COMM_WORLD, &rank ); /* Who am I? */
    MPI_Comm_size ( MPI_COMM_WORLD, &size ); /* How many are we? */
    printf ( “I am %d of %d\n”, rank, size );
    MPI_Finalize();
    return 0;
}
```

```
login3:~/examples> mpicc -o hello-world mpi-hello-world.c
login3:~/examples> mpirun -np 4 ./hello-world
I am 0 of 4
I am 3 of 4
I am 1 of 4
I am 2 of 4
```
Data Partitioning (SPMD) - Single Program, Multiple Data

**Implementation:**
1 Source code

```c
void main()
{
    int i,j;
    char a;
    for(i=0;
    ...
```

**Execution:**

n Executables

**Processor 1**

```c
void main()
{
    int i,j;
    char a;
    for(i=0;
    ...
```

**Processor 2**

```c
void main()
{
    int i,j;
    char a;
    for(i=0;
    ...
```

**Processor 3**

```c
void main()
{
    int i,j;
    char a;
    for(i=0;
    ...
```
Implementation: m Source codes

Execution: n Executables

void main()
{
    int i, j;
    char a;
    for (i = 0;
         ...
}

Processor 1

void main()
{
    int i, j;
    char a;
    for (i = 0;
         ...
}

Processor 2

void main()
{
    int k;
    char b;
    while (b =
         ...
}

Processor 3
Comparison: SPMD/MPMD

SPMD:

- One source code for all processes
- Distinction in the source code through control statements
  ```c
  if (pid() == MASTER) { ... } else { ... }
  ```
- Widely used

MPMD:

- One source for each process
- Higher flexibility and modularity
- Administration of source codes difficult
- Additional effort during process creation
- Dynamic process creation possible
Query Functions

Identification

- *Who am I?*
- *Which process number has the current process?*
  
  ```c
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank)
  ```

- *Who else is there?*
- *How many processes have been started?*
  
  ```c
  MPI_Comm_size(MPI_COMM_WORLD, &mysize)
  ```

- *Characteristics: 0 <= myrank < mysize*
Master-Slave Program

1. int main(int argc, char *argv[])
2. {
3.   MPI_Init(&argc, &argv);
4.   ...
5.   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
6.   if(myrank == 0)
7.     master();
8.   else
9.     slave();
10.  ...
11.  MPI_Finalize();
12. }
Global Variables

**Problem:**

1. `int main(int argc, char *argv[])`
2. `{ float big_array[10000];`

**Solution:**

1. `int main(int argc, char *argv[])`
2. `{`
3. `if(myrank == 0) {`
4. `float big_array[10000];`
Allocating Global Variables

- Global definition, local allocation where needed.

**Traditional:**
1. int main(int argc, char *argv[])
2. {
3. float big_array[10000];

**Rank based approach:**
1. int main(int argc, char *argv[])
2. {
3. float *big_array;
4. if(myrank == 0) {
5. big_array = (float *)malloc(...)}
Basic Functions:
• send(parameter_list)

Send Function:
• In origin process
• Creates message

recv(parameter_list)

Receive Function:
• In destination process
• Receives transmitted message
On the origin process:
\[ \text{send}(\&x, \text{destination}_\text{id}) \]

On the destination process:
\[ \text{recv}(\&y, \text{source}_\text{id}) \]

Simple Functions

Message passing

Process 1
\[ \text{Send}(\&x, 2) \]

Process 2
\[ \text{Recv}(\&y, 1) \]
int **MPI_Send** (void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

- **buf**   Address of message in memory
- **count** Number of elements in message
- **datatype** Data type of message
- **dest** Destination process of message
- **tag** Generic message tag
- **comm** Communication handler

**MPI_Datatype**  MPI_CHAR, MPI_INT, MPI_FLOAT, ...
**MPI_Comm**  MPI_COMM_WORLD
Standard Receive

```c
int MPI_Recv (void *buf, int count,
              MPI_Datatype datatype, int source,
              int tag,
              MPI_Comm comm, MPI_Status *status)
```

- **buf** Address of message in memory
- **count** *Expected* number of elements in message
- **datatype** Data type of message
- **source** Origin process of message
- **tag** Generic message tag
- **comm** Communication handler
- **status** Status-Information
Example: *Round Robin*

Process 0
- `MPI_Send(Str, ...)`
- `MPI_Recv(Str, ...)`
- `printf(Str)`

Process 1
- `MPI_Recv(Str, ...)`
- `printf(Str)`
- `MPI_Send(Str, ...)`

Process 2
- `MPI_Recv(Str, ...)`
- `printf(Str)`
- `MPI_Send(Str, ...)`

Process 3
- `MPI_Recv(Str, ...)`
- `printf(Str)`
- `MPI_Send(Str, ...)`
Remark - Maximum message length is fixed:
• If message is bigger ➔ overflow error
• If message is smaller ➔ unused memory

➔ Allocate sufficient space before calling MPI_Recv
Standard Receive

How many elements have been received?

```c
int MPI_Get_count (MPI_Status *status, MPI_Datatype datatype, int *count)
```

Status-Information:

1. `struct MPI_Status {
   2.   int MPI_SOURCE;
   3.   int MPI_TAG;
   4.   int MPI_ERROR;
   5.   int count;
   6.   ...
   7.   };
```
MPI Specifics

- Communicators: Scope of communication operations
- Structure of messages: complex data types
- Data transfer:
  - Synchronous/asynchronous
  - Blocking/non-blocking
- Message tags/identifiers
- Communication partners:
  - Point-to-point
  - Wild card process and message tags
MPI:

- Data types for message contents:
  - Standard types:
    - MPI_INT
    - MPI_FLOAT
    - MPI_CHAR
    - MPI_DOUBLE
    - ...
  - User defined types: derived from standard types
Data Transfer

Blocking:
• Function does not return, before message can be accessed again
• Process is „blocked“

Non-blocking:
• Function returns, whether data transfer is finished or not
• Requires function to query the status of the data transfer
• Message buffers are needed
  – Length of message is limited
• Overlapping of communication and computation is possible
  – → Reduction of execution time
Non-Blocking Functions

Example: Overlapping of Computation and Communication
Synchronous Data Transfer

Case 1: Send is called before receive

Process 1
- Send(&x, 2)
- Execution blocked

Process 2
- Request
- Ack
- Message
- Recv(&y, 1)

Time
Synchronous Data Transfer

Case 2: Recv is called before send

Process 1

Send(&x, 2)

Request

Message

Process 2

Recv(&y, 1)

Time

Execution blocked
Collective Operations

Possibilities:

- **MPI_Barrier**: has to be passed by all processes
- **MPI_Bcast**: one process to all others
- **MPI_Gather**: collect data of other processes
- **MPI_Scatter**: distribute data onto other processes
- **MPI_Reduce**: combine data of other processes
- **MPI_Reduce_scatter**: combine and distribute
- ...

[Image of the slide with the text]
Broadcast/Multicast

Process 0
- data
- msg
- BCast()

Process 1
- data
- BCast()

Process 2
- data
- BCast()
Scatter

- Distribute the array `msg_arr` of process `root` to all other processes
  - Contents at index i is sent to process i
  - Different implementations possible:
    - Data may be returned to `root`, ...
  - Widely used in SPMD Model
• Collect data of all processes on process *root* in array *msg_arr*
  – Data of process i is stored at index i
  – Opposite of Scatter-Operation
  – Usually at the end of a distributed computation
  – Different implementations possible
• Global operation on process root during data collection
  – Combination of Gather + global operation
  – logical or arithmetic operation possible
  – Different implementations possible: operation on root, partial, distributed operations, ...
For an arbitrary MPI program, only 6 Functions are needed

- MPI_Init(...)
- MPI_Finalize(...)
- MPI_Comm_rank(...)
- MPI_Comm_size(...)
- MPI_Send(...)
- MPI_Recv(...)
1.2 – MPI Implementations @ BSC

• How to compile MPI programs

<table>
<thead>
<tr>
<th>LANGUAGE</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>mpicc</td>
</tr>
<tr>
<td>C++</td>
<td>mpiCC/mpiCxx/mpiC++</td>
</tr>
<tr>
<td>Fortran 77</td>
<td>mpif77/mpiFort</td>
</tr>
<tr>
<td>Fortran 90</td>
<td>mpif90/mpiFort</td>
</tr>
</tbody>
</table>

• All the backend compilers are managed by environment variables (modules intel and gcc)

• Several MPI implementations available
  – Intel MPI
  – OpenMPI
  – MVAPICH2
  – IBM Parallel Environment (PE)
### 1.2 – INTEL MPI

<table>
<thead>
<tr>
<th>Version</th>
<th>2017.4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module</td>
<td>module load impi</td>
</tr>
<tr>
<td>Description</td>
<td>Intel® MPI Library focuses on making applications perform better on Intel® architecture-based clusters—implementing the high performance Message Passing Interface Version 2.2 specification on multiple fabrics.</td>
</tr>
<tr>
<td>Location</td>
<td>/apps/INTEL/impi/2017.4</td>
</tr>
<tr>
<td>Features</td>
<td>Fully compatible with Slurm, and LSF/POE</td>
</tr>
<tr>
<td>Benchmarks</td>
<td>/apps/INTEL/mkl/benchmarks/mp_linpack/bin_intel/intel64/</td>
</tr>
<tr>
<td>Launcher</td>
<td>mpirun / srun</td>
</tr>
</tbody>
</table>
## 1.2 – OpenMPI

<table>
<thead>
<tr>
<th>Versions</th>
<th>1.10.7 (default) …</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module</td>
<td>module load openmpi[/version]</td>
</tr>
<tr>
<td>Description</td>
<td>The Open MPI Project is an open source MPI-2 implementation that is developed and maintained by a consortium of academic, research, and industry partners.</td>
</tr>
<tr>
<td>Web</td>
<td><a href="http://www.open-mpi.org/">http://www.open-mpi.org/</a></td>
</tr>
<tr>
<td>Location</td>
<td>/apps/OPENMPI/&lt;version&gt;/bin</td>
</tr>
<tr>
<td>Features</td>
<td>Fully compatible with Slurm and LSF</td>
</tr>
<tr>
<td>Launcher</td>
<td>mpirun / srun</td>
</tr>
</tbody>
</table>
1.3 – How to switch MPI implementations

- InelMPI 2017.4 is loaded by default
- You can change your environment easily using ‘module’ commands
- IntelMPI -> OpenMPI

```
login3-mn3:~> module unload openmpi
remove openmpi/1.8.1 (PATH, MANPATH, LD_LIBRARY_PATH)
login3-mn3:~> module load impi
load impi/4.1.3.049 (PATH, MANPATH, LD_LIBRARY_PATH)
login3-mn3:~> module list
Currently Loaded Modulefiles:
  1) intel/13.0.1     2) transfer/1.0     3) impi/4.1.3.049
login3-mn3:~> mpicc -show
icc -m64 -D__64BIT__ -Wl,--allow-shlib-undefined -Wl,--enable-new-dtags
-Wl,-rpath,/opt/ibmhpc/pecurrent/mpich2/intel/lib64  -I/opt/ibmhpc/pecurrent/mpich2/intel/include64
-I/opt/ibmhpc/pecurrent/base/include -L/opt/ibmhpc/pecurrent/mpich2/intel/lib64 -lmpi -ldl -L -lirc
-lpthread -lrt
```

- You can not have 2 MPI implementations loaded at the same time (command will not permit loading conflicting modules)
1.3 – How to switch MPI implementations

• Switch between OpenMPI versions

```
login3-mn3:~> module switch openmpi/1.8.1 openmpi/1.6.4
switch1 openmpi/1.8.1 (PATH, MANPATH, LD_LIBRARY_PATH)
switch2 openmpi/1.6.4 (PATH, MANPATH, LD_LIBRARY_PATH)
switch3 openmpi/1.8.1 (PATH, MANPATH, LD_LIBRARY_PATH)
ModuleCmd_Switch.c(243):VERB:4: done
login3-mn3:~> module list
Currently Loaded Modulefiles:
   1) intel/13.0.1  2) transfer/1.0  3) openmpi/1.6.4
login3-mn3:~> mpicc -show
icc -I/apps/OPENMPI/1.6.4/include -pthread -L/apps/OPENMPI/1.6.4/lib -lmpi -ldl -lm -lnuma
-Wl,--export-dynamic -lrt -lnsl -lutil -lm -ldl
```

• Remember to use the same module to compile and execute
  – Otherwise the job will fail or not behave properly

```
./hello_world: error while loading shared libraries: libmpi.so.1: cannot open shared object file:
No such file or directory
login3-mn3:~> mpirun -np 4 ./hello_world
Hello world! I'm process 0 of 1 on login3
Hello world! I'm process 0 of 1 on login3
Hello world! I'm process 0 of 1 on login3
```


1.3 – Application advice

- There is no ‘best’ MPI implementation in MareNostrum. (However, we recommend IntelMPI)
1.4 - Possible issues

• Memory problems
  – Be careful with the memory consumption of your job
  – 28GB per node
• If your program tries to use more memory, then Slurm will send SIGTERM/SIGKILL to your processes
  – You will see: mpirun: killing job…
• You can avoid this problem:
  – Set system limit on the memory
    ```
    ulimit -m 1700000
    sbatch job.cmd
    ```
• For executions up to 216 nodes, there are fat-memory nodes:
  – #SBATCH --constraint=highmem
1.4 - Running jobs

- How to decide which number of cores to use?
  - Depends on the scalability of your code
  - Avoid wasting resources!! (>70% efficiency)
Thank you!

jorge.rodriguez@bsc.es