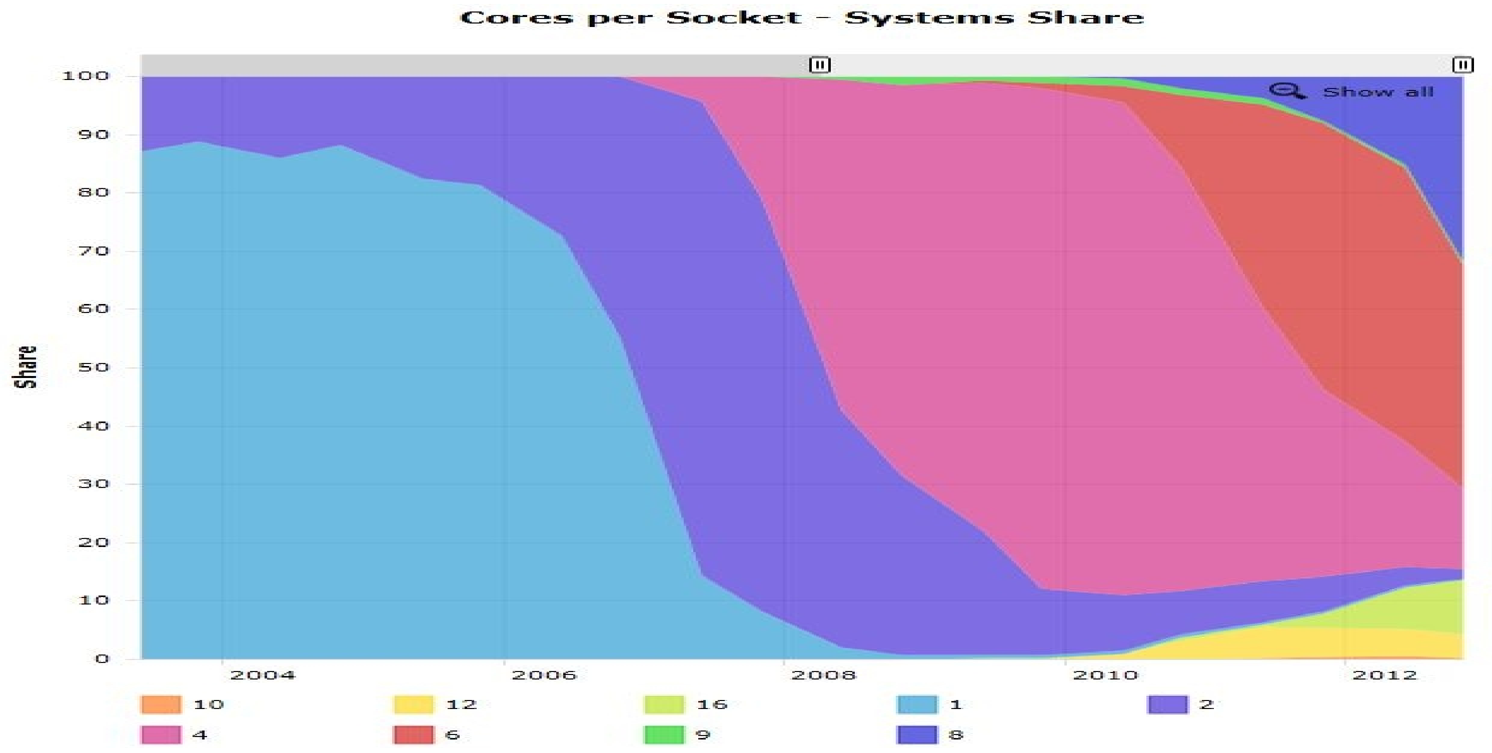
Introduction to MPI+OpenMP hybrid programming

SuperComputing Applications and Innovation Department

Basic concepts

Architectural trend



Architectural trend

- In a nutshell:

  - memory per core decreases
  - memory bandwidth per core decreases
  - number of cores per socket increases
  - single core clock frequency decreases

- Programming model should follow the new kind of architectures available on the market: what is the most suitable model for this kind of machines?

Programming models

- Distributed parallel computers rely on MPI
  - strong
  - consolidated
  - standard
  - enforce the scalability (depending on the algorithm) up to a very large number of tasks
- but... is it enough when memory is such small amount on each node?

Example: Bluegene/Q has 16GB per node and 16 cores. Can you imagine to put there more than 16MPI (tasks), i.e. less than 1GB per core?

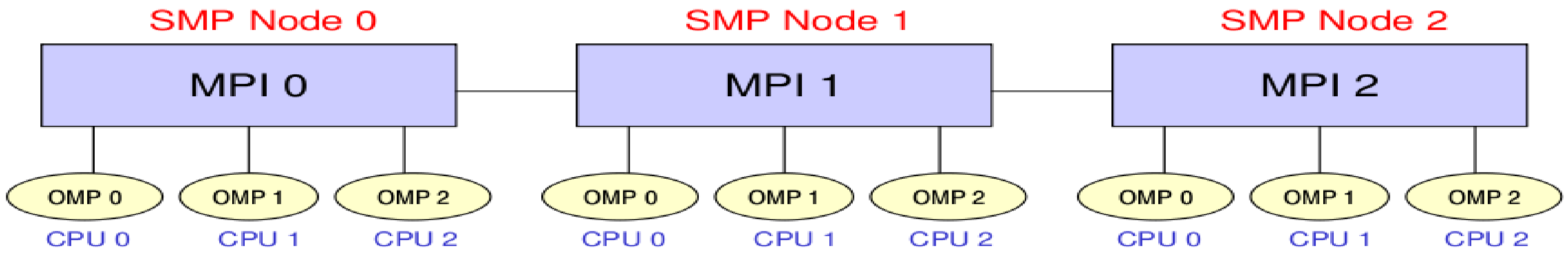
Programming models

- On the other side, OpenMP is a standard for shared memory systems
- Pthreads execution models is a lower-level alternative, but OpenMP is often a better choice for HPC programming
- OpenMP is robust, clear and sufficiently easy to implement but
  - depending on the implementation, typically the scaling on the number of threads is much less effective than the scaling on number of MPI tasks
- Putting together MPI with OpenMP could permit to exploit the features of the new architectures, mixing these paradigms

Hybrid model: MPI+OpenMP

- In a single node you can exploit a shared memory parallelism using OpenMP
- Across the nodes you can use MPI to scale up

Example: on a Bluegene/Q machine you can put 1 MPI task on each node and 16 OpenMP threads. If the scalability on threads is good enough, you can use all the node memory.



MPI vs OpenMP

- **Pure MPI Pro:**
  - High scalability
  - High portability
  - No false sharing
  - Scalability out-of-node

- **Pure MPI Con:**
  - Hard to develop and debug.
  - Explicit communications
  - Coarse granularity
  - Hard to ensure load balancing

MPI vs OpenMP

- **Pure MPI Pro:**
  - High scalability
  - High portability
  - No false sharing
  - Scalability out-of-node
- **Pure MPI Con:**
  - Hard to develop and debug.
  - Explicit communications
  - Coarse granularity
  - Hard to ensure load balancing

Pure OpenMP Pro:

Easy to deploy (often)
Low latency
Implicit communications
Coarse and fine granularity
Dynamic Load balancing

Pure OpenMP Con:

Only on shared memory machines
Intranode scalability
Possible data placement problem
Undefined thread ordering

MPI+OpenMP

- Conceptually simple and elegant
- Suitable for multicore/multinodes architectures
- Two-level hierarchical parallelism
- In principle, you can alleviate problems related to the scalability of MPI, reducing the number of tasks and network flooding

Increasing granularity

- OpenMP introduces fine granularity parallelism
- Loop-based parallelism
- Task construct (OpenMP 3.0): powerful and flexible
- Load balancing can be dynamic or scheduled
- All the work is in charge to the compiler
- No explicit data movement

Two level parallelism

- Using a hybrid approach means to balance the hierarchy between MPI tasks and thread.

- MPI in most cases (but not always) occupy the upper level respect to OpenMP
  - usually you assign *n* threads per MPI task, not m MPI tasks per thread

- The choice about the number of threads per MPI task strongly depends on the kind of application, algorithm or kernel. (this number can change inside the application)

- There's no golden rule. More often this decision is taken a-posteriori after benchmarks on a given machine/architecture

Saving MPI tasks

- Using a hybrid approach MPI+OpenMP can lower the number of MPI tasks used by the application.
- Memory footprint can be alleviated by a reduction of replicated data on MPI level
- Speed-up limited due algorithmic issues can be solved (because you're reducing the amount of communication)

Reality is bitter

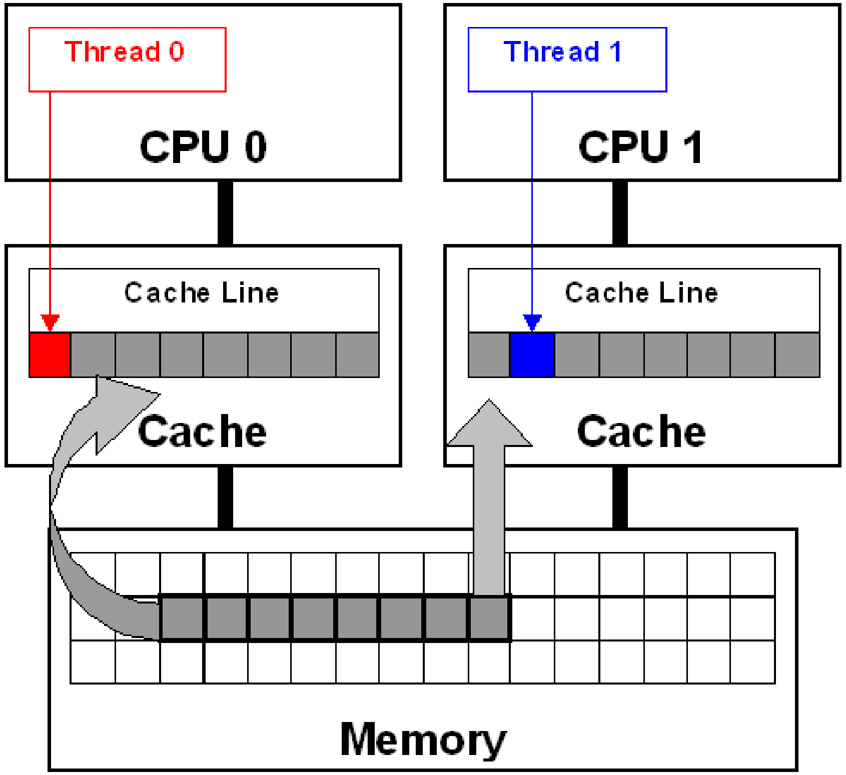
- In real scenarios, mixing MPI and OpenMP, sometimes, can make your code slower
  - If you exceed with the number of OpenMP threads you can encounter problems with locking of resources
  - Sometimes threads can stay in a idle state (spin) for a long time
  - Problems with cache coherency and false sharing
  - Difficulties in the management of variables scope

Cache coherency and false sharing

- It is a side effects of the cache-line granularity of cache coherence implemented in shared memory systems.
- The cache coherency implementation keep track of the status of cache lines by appending *state bits to* indicate whether data on cache line is still valid or outdated.
- Once the cache line is modified, cache coherence notifies other caches holding a copy of the same line that its line is invalid.
- If data from that line is needed, a new updated copy must to be fetched.

False sharing

```
#pragma omp parallel for
shared(a) schedule(static,1)
for (int i=0; i<n; i++)
       a[i] = i;
```



Let's start

- The most simple recipe is:
  - start from a serial code and make it a MPI-parallel code
  - implement for each of the MPI task a OpenMP-based parallelization
- Nothing prevents to implement a MPI parallelization inside a OpenMP parallel region
  - in this case, you should take care of the thread-safety
- To start, we will assume that only the master thread is allowed to communicate with others MPI tasks

A simple hybrid code

```
call MPI_INIT (ierr)
call MPI_COMM_RANK (…)
call MPI_COMM_SIZE (…)
 …  some computation and MPI communication
 call OMP_SET_NUM_THREADS(4)
!$OMP PARALLEL
!$OMP DO
   do i=1,n
       … computation
   enddo
 !$OMP END  DO
 !$OMP END  PARALLEL
 …  some computation and MPI communication
call MPI_FINALIZE (ierr)
```

Master-only approach

Advantages:

- Simplest hybrid parallelization (easy to understand and to manage)
- No message passing inside a SMP node

Disadvantages:

- All other threads are sleeping during MPI communications
- Thread-safe MPI is required

MPI_Init_thread support

- **MPI_INIT_THREAD** (required, provided, ierr)
  - IN: required, desired level of thread support (integer).
  - OUT: provided, provided level (integer).
    provided may be less than required.

- Four levels are supported:
  - **MPI_THREAD_SINGLE**: Only one thread will run. Equals to MPI_INIT.
  - **MPI_THREAD_FUNNELED**: processes may be multithreaded, but only the main thread can make MPI calls (MPI calls are delegated to main thread)
  - **MPI_THREAD_SERIALIZED**: processes could be multi-threaded More than one thread can make MPI calls, but only one at a time.
  - **MPI_THREAD_MULTIPLE**: multiple threads can make MPI calls, with no restrictions.

MPI_Init_thread

- The various implementations differ in levels of thread-safety
- If your application allows multiple threads to make MPI calls simultaneously, without MPI_THREAD_MULTIPLE, is not thread-safe
- Using OpenMPI, you have to use –enable-mpi-threads at configure time to activate all levels
  - see more later
- Higher level corresponds to higher thread-safety. Use the required safety needs.

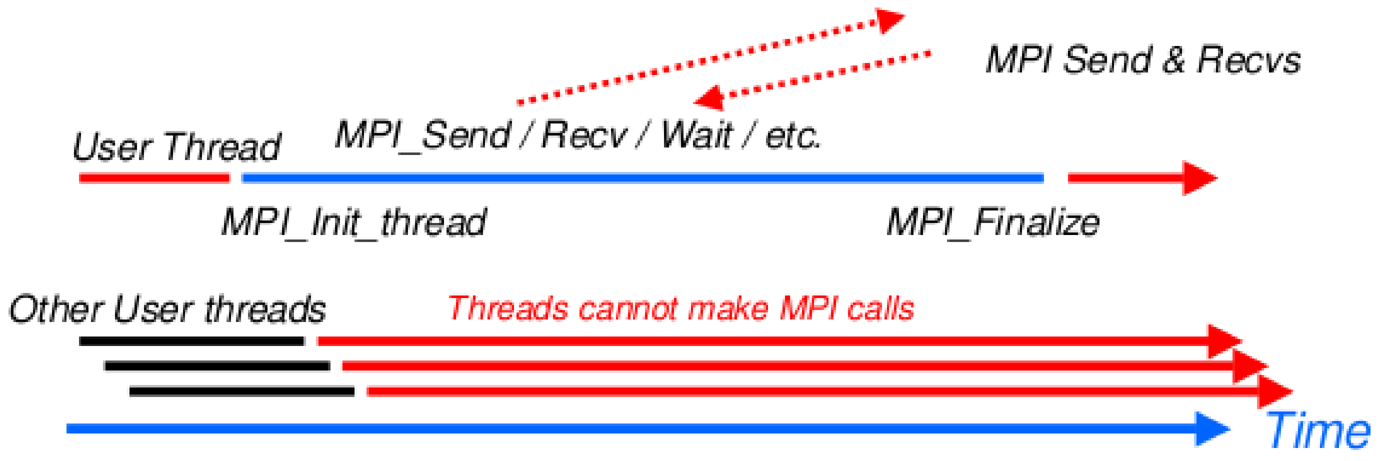
MPI_THREAD_SINGLE

- There are no additional user thread in the system
  - E.g., there are no OpenMP parallel regions
  - MPI_Init_thread with MPI_THREAD_SINGLE is fully equivalent to MPI_Init

```
int main(int argc, char ** argv)
{
    int buf[100];
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    for (i = 0; i < 100; i++)
       compute(buf[i]);
    /* Do MPI stuff */
    MPI_Finalize();
    return 0;
}
```

MPI_THREAD_FUNNELED

- It adds the possibility to make MPI calls inside a parallel region, but only the master thread is allowed to do so
  - All MPI calls are made by the master thread
  - **The programmer must guarantee that!**



MPI_THREAD_FUNNELED

```
int main(int argc, char ** argv)
{
int buf[100], provided;
MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &provided);
if (provided < MPI_THREAD_FUNNELED) MPI_Abort(MPI_COMM_WORLD,1);
#pragma omp parallel for
for (i = 0; i < 100; i++)
    compute(buf[i]);
/* Do MPI stuff */
MPI_Finalize();
return 0;
}
```

MPI_THREAD_FUNNELED

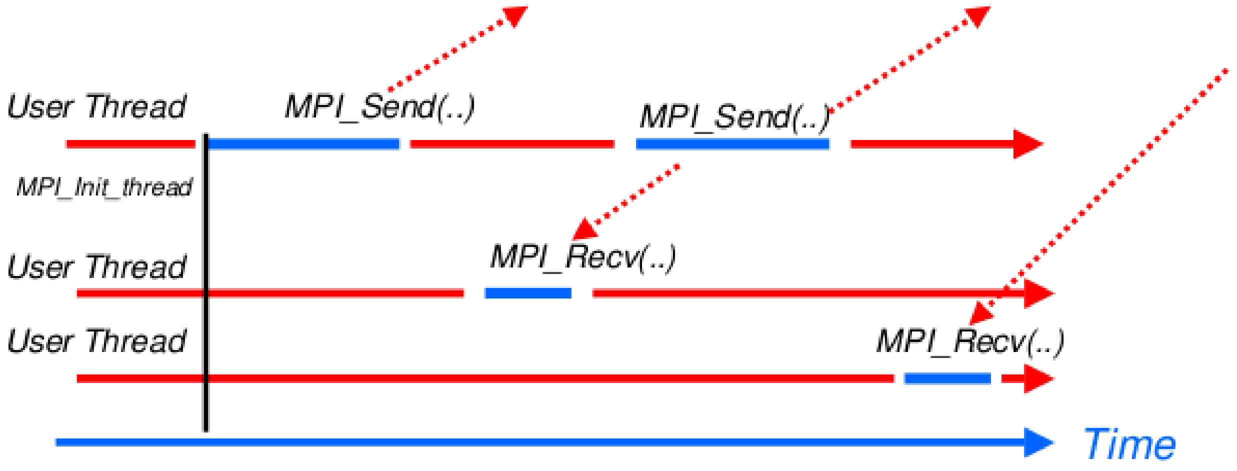
- MPI function calls can be: outside a parallel region or in a parallel region, enclosed in “omp master” clause
- There is no synchronization at the end of a “omp master” region, so a barrier is needed before and after to ensure that data buffers are available before/after the MPI communication

```
!$OMP BARRIER
!$OMP MASTER
      call MPI_Xxx(...)
!$OMP END MASTER
!$OMP BARRIER
```

```
#pragma omp barrier
#pragma omp master
     MPI_Xxx(...);
#pragma omp barrier
```

MPI_THREAD_SERIALIZED

- Multiple threads may make MPI calls, but only one at a time:
  - MPI calls are **not** made concurrently from two distinct threads. MPI calls are "serialized"
  - **The programmer must guarantee that!**



MPI_THREAD_SERIALIZED

```
int main(int argc, char ** argv)
{
int buf[100], provided;
MPI_Init_thread(&argc, &argv, MPI_THREAD_SERIALIZED, &provided);
if (provided < MPI_THREAD_SERIALIZED) MPI_Abort(MPI_COMM_WORLD,1);
#pragma omp parallel for
for (i = 0; i < 100; i++) {
    compute(buf[i]);
#pragma omp critical
    /* Do MPI stuff */
}
MPI_Finalize(); Return 0;
}
```

MPI_THREAD_SERIALIZED

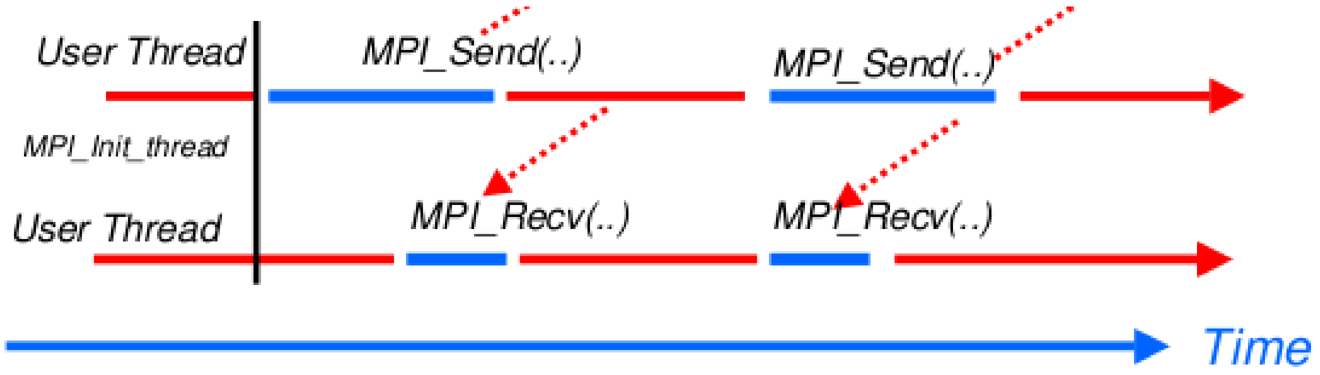
- MPI calls can be outside parallel regions, or inside, but enclosed in a “omp single” region (it enforces the serialization) or “omp critical” or ...
- Again, a starting barrier may be needed to ensure data consistency
  - But at the end of ***omp single*** there is an automatic barrier
  - Unless ***nowait*** is specified

```
!$OMP BARRIER
!$OMP SINGLE
      call MPI_Xxx(...)
!$OMP END SINGLE
```

```
#pragma omp barrier
#pragma omp single
     MPI_Xxx(...);
```

MPI_THREAD_MULTIPLE

- It is the most flexible mode, but also the most complicate one
- Any thread is allowed to perform MPI communications, without any restrictions.



MPI_THREAD_MULTIPLE

```
int main(int argc, char ** argv)
{
int buf[100], provided;
MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
if (provided < MPI_THREAD_MULTIPLE) MPI_Abort(MPI_COMM_WORLD,1);
#pragma omp parallel for
for (i = 0; i < 100; i++) {
    compute(buf[i]);
    /* Do MPI stuff */
}
MPI_Finalize();
return 0;
}
```

Specs of MPI_THREAD_MULTIPLE

- **Ordering**: when multiple threads make MPI calls concurrently the outcome will be as if the calls executed sequentially in some (any) order
  - Ordering is maintained within each thread
  - User must ensure that collective operations on the same communicator windows or file handle are correctly ordered among threads
    - E.g. cannot call a broadcast on one thread and a reduce on another thread on the same communicator
  - It is the user's responsibility to prevent races when threads in the same application post conflicting MPI calls
    - E.g. accessing an info object from one thread and freeing it from another thread
- **Blocking**: Blocking MPI calls will block only the calling thread and will not prevent other threads from running or executing MPI functions

Ordering in MPI_THREAD_MULTIPLE

- **Incorrect** example with collectives
  - P0 and P1 can have different ordering of Bcase on Barrier
  - Here the user must use some kind of synchronization to ensure that either thread 1 or thread 2 gets scheduled first on both processes
  - Otherwise a broadcast may get matched with a barrier on the same communicator, which is not allowed in MPI

	Process 0	Process 1
Thread 0	MPI_Bcast(comm)	MPI_Bcast(comm)
Thread 1	MPI_Barrier(comm)	MPI_Barrier(comm)

Ordering in MPI_THREAD_MULTIPLE

- **Incorrect** example with object Management
  - The user has to make sure that one thread is not using an object while another thread is freeing it
  - This is essentially an ordering issue; the object might get freed before it is used

	Process 0	Process 1
Thread 0	MPI_Bcast(comm)	MPI_Bcast(comm)
Thread 1	MPI_Comm_free(comm)	MPI_Comm_free(comm)

Blocking in MPI_THREAD_MULTIPLE

- **Correct** example with point-to-point
  - An implementation must ensure that the example below never deadlocks for any ordering of thread execution
  - That means the implementation cannot simply acquire a thread lock and block within an MPI function. It must release the lock to allow other threads to make progress

	Process 0	Process 1
Thread 0	MPI_Recv(src=1)	MPI_Recv(src=0)
Thread 1	MPI_Send(dst=1)	MPI_Send(dst=0)

Comparison to pure MPI

Funneled

- All threads but the master are sleeping during MPI communications
- Only one thread may not be able to lead up to max inter-node bandwidth

Pure MPI

- Each CPU can lead up max inter-node bandwidth

Hints: overlap as much as possible communications and computations

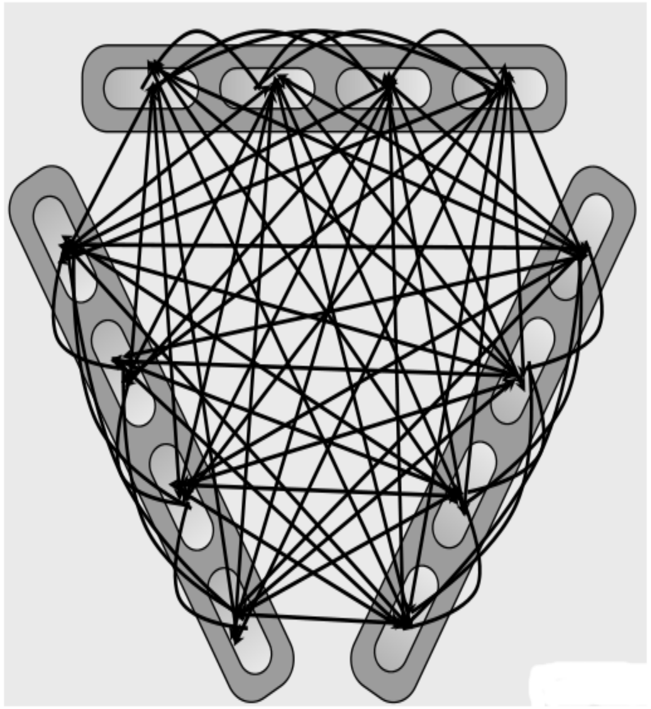
Overlap communications and computations

- In order to overlap communications with computations, the first step is using MPI_THREAD_FUNNELED mode
- While the master threads (a master thread for each MPI rank) are exchanging data, the other threads performs computation
- The tricky part is separating code that can run before or after the data exchanged are available

```
!$OMP PARALLEL
    if (thread_id==0) then
        call MPI_xxx(…)
    else
        do some computation
    endif
!$OMP END PARALLEL
```

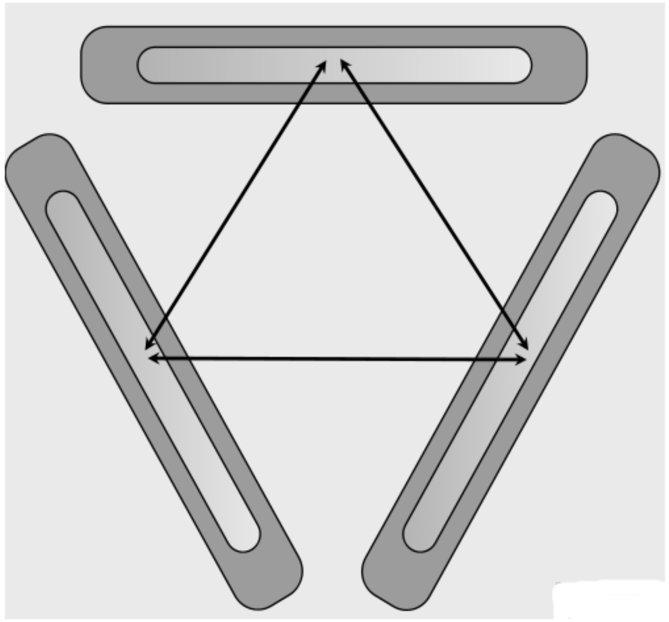
MPI collective hybridization

- MPI collectives are highly optimized
- Several point-to-point communication in one operations
- They can hide from the programmer a huge volume of transfer (MPI_Alltoall generates almost 1 million point-to-point messages using 1024 cores)
- There is no non-blocking (no longer the case in MPI 3.0)



MPI collective hybridization

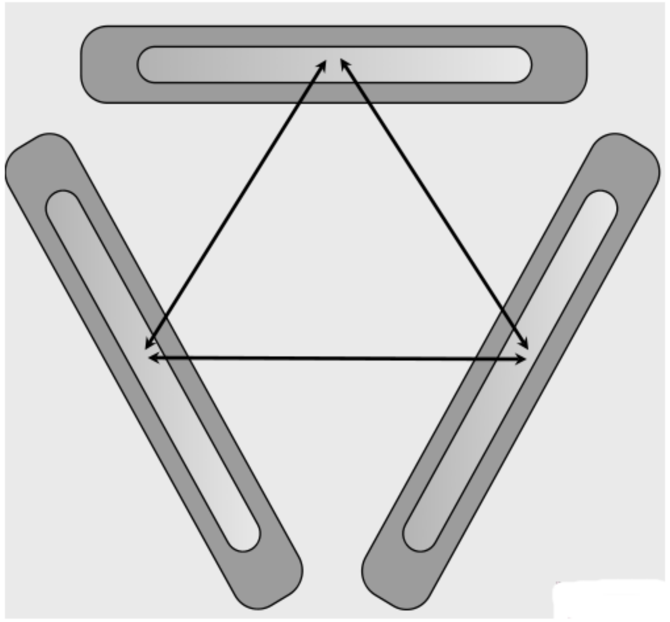
- Better scalability by a reduction of both the number of MPI messages and the number of process. Tipically:
- for all-to-all communications, the number of transfers decrease by a factor #threads^2
- the length of messages increases by a factor #threads
- Allow to overlap communication and computation.



MPI collective hybridization

Restrictions:

- In MPI_THREAD_MULTIPLE mode is forbidden at any given time two threads each do a collective call on the same communicator (MPI_COMM_WORLD)
- 2 threads calling each a MPI_Allreduce may produce wrong results
- **Use different communicators for each collective call**
- **Do collective calls only on 1 thread per process (MPI_THREAD_SERIALIZED mode should be fine)**



Multithreaded libraries

- Introduction of OpenMP into existing MPI codes includes OpenMP drawbacks (synchronization, overhead, quality of compiler and runtime…)
- A good choice (whenever possible) is to include into the MPI code a **multithreaded, optimized library suitable for the application**.
- **BLAS, LAPACK, MKL (Intel), FFTW** are well known multithreaded libraries available in the HPC ecosystem.
  - Some libraries create their own threads: must be called outside our “omp parallel” regions
  - Otherwise, check “how much” the the library is thread-safe, if at least can be called by the master thread of your *omp* region (MPI_THREAD_FUNNELED)
  - Look carefully at the doc of your library, e.g. http://www.fftw.org/doc/Usage-of-Multi_002dthreaded-FFTW.html

BGQ benchmark example

Up to 64 hardware threads per process are available on bgq (SMT)

Huge simulation, 30000x30000 points. Stopped after 100 iterations only for timing purposes.

Number of threads / processes	**MPI+OpenMP (TOT= 64 MPI, 1PPN) MPI_THREAD_MULTIPLE version Elapsed time (sec.)**	**MPI ONLY (TOT= 1024 MPI, 16,32,64 ppn Elapsed time (sec.)**
1	78.84	N.A
4	19.89	N.A
8	10.33	N.A
16	5.65	5.98
32	3.39	7.12
64	2.70	12.07

Conclusions

- Better scalability by a reduction of both the number of MPI messages and the number of processes involved in collective communications and by a better load balancing.
- Better adeguacy to the architecture of modern supercomputers while MPI is only a flat approach.
- Optimization of the total memory consumption (through the OpenMP shared-memory approach, savings in replicated data in the MPI processes and in the used memory by the MPI library itself).
- Reduction of the footprint memory when the size of some data structures depends directly on the number of MPI processes.
- It can remove algorithmic limitations (maximum decomposition in one direction for example).

Conclusions / 2

Applications that can benefit from hybrid approach:

- Codes having limited MPI scalability (through the use of MPI_Alltoall for example).
- Codes requiring dynamic load balancing
- Codes limited by memory size and having many replicated data between MPI processes or having data structures that depends on the number of processes.
- Inefficient MPI implementation library for intra-node communication.
- Codes working on problems of fine-grained parallelism or on a mixture of fine and coarse-grain parallelism.
- Codes limited by the scalability of their algorithms.

Conclusions / 3

- Hybrid programming is complex and requires high level of expertise.
- Both MPI and OpenMP performances are needed (Amdhal's law apply separately to the two approaches).
- Savings in performances are not guaranteed (extra additional costs).

Implementations and cluster usage notes

MPI implementations and thread support

- An implementation is not required to support levels higher than MPI_THREAD_SINGLE, that is, an implementation is not required to be thread safe
- Most MPI implementations support a very low default thread support
  - Usually no support (MPI_THREAD_SINGLE)
  - And probably MPI_THREAD_FUNNELED (even if not explicitly)
  - Which (usually) is ok for cases where MPI communications are called outside of OMP parallel regions (where MPI_THREAD_FUNNELED would be strictly required)
  - Implementations with thread support are more complicated, error-prone and sometimes slower
- Checking the MPI_Init_thread provided support is a good programming practice

MPI implementations: check thread support

```
required = MPI_THREAD_MULTIPLE;
ierr = MPI_Init_thread(&argc,&argv,required,&provided);
if(required != provided) {
    if(rank == 0) {  fprintf(stderr,"incompatible MPI thread support\n");
        fprintf(stderr,"required,provided: %d %d\n", required,provided); }
    ierr = MPI_Finalize();     exit(-1);
}
```

```
call MPI_Init_thread(required, provided, ierr)
if(provided /= required) then
    if(rank == 0) then
        print*,'Attention! incompatible MPI thread support'
        print*,'THREAD support required, provided:  ',required, provided
    endif
    call MPI_Finalize(ierr); STOP
endif
```

MPI implementations and thread support / 2

- Beware: the lack of thread support may result in subtle errors (not always clear)
- OpenMPI (1.8.3): when compiling OpenMPI there is a configure option to specify:
  - --enable-mpi-thread-multiple
  - [Enable MPI_THREAD_MULTIPLE support (default: disabled)]
- IntelMPI (5.0.2): both thread safe/non-thread safe versions are available:
  - Specify the option -mt_mpi when compiling your program to link the thread safe version of the Intel(R) MPI Library
  - E.g. mpif90 -mt_mpi main.f90

MPI and resource allocation

- The problem: how to distribute and control the resource allocation and usage (MPI processes/OMP threads) within a resource manager
  - How to use all the allocated resources: if n cores per node have been allocated how to run my program using all of that cores
    - Not less, maximize performance
    - Not more, do not interact with jobs of other users
  - How to use at its best the resource
    - optimal mapping between MPI processes/OMP threads and physical cores

MPI and resource allocation / 2

- Most MPI implementations allow a tight integration with resource managers in order to ease the usage of the requested resources
- The integration may enforce some constraints or just give hints to the programmer
  - Strict: only the requested physical resources can be used by my job
  - Soft: the programmer may use resources not explicitly allocated but only forcing the default settings

Cineca Eurora: PBSPro and MPI

- Cineca Cineca cluster assign resources using PBSProfessional
  - Strict mode is not guaranteed
  - Beware: you can run using all the cores of you node even if you did not request all that cores (ncpus)

- OpenMPI and IntelMPI implementations available, both supporting PBS integration
  - **module load autoload openmpi/1.8.3—gnu--4.8.0**
  - **module load autoload openmpi/1.8.3-threadmultiple--gnu--4.8.0**
  - **module load autoload intelmpi/5.0.1—binary**

Cineca Galileo: PBSPro and MPI

- Cineca Galileo cluster assign resources using PBSProfessional
  - Strict mode is guaranteed through *cgroups* (control groups, a feature of Linux kernel which limits, isolate the usage of resources – CPU, memory, I/O, network, etc.. - of a process group). The processes can utilize only the resources assigned by the scheduler (minimizing the interaction with other jobs simultaneously running)

Review on PBS select option

- -lselect=<n_chunks>:ncpus:<n_cores>:mpiprocs:<n_mpi>
  - Beware: n_chunks usually means n_nodes but this is not guaranteed provided that two or more chunks of n_cores can be allocated on a single node
  - mpiprocs must be well understood, let us examine the file $PBS_NODEFILE created by PBS listing the allocated resources for some example cases
  - 6 MPI processes per chunk (node) and 6 cores reserved:useful for pure MPI runs

select=4
ncpus=6
mpiprocs=6

node007	node003	node001	node015
node007	node003	node001	node015
node007	node003	node001	node015
node007	node003	node001	node015
node007	node003	node001	node015
node007	node003	node001	node015

Review on PBS select option / 2

- 1 MPI process per chunk (node), but 6 cores reserved, useful for hybrid case (OMP_NUM_THREADS=6)

```
select=4
ncpus=6
mpiprocs=1
```

node007	node003	node001	node015

- 2 MPI processes per chunk (node), but 6 cores reserved, again for hybrid cases (OMP_NUM_THREADS=3)

```
select=4
ncpus=6
mpiprocs=2
```

node007	node003	node001	node015
node007	node003	node001	node015

OpenMPI and IntelMPI

- Resource allocation is not aware of the MPI implementation which will be used
  - But MPI implementations give a different meaning and enforce different constraints
  - We will discuss OpenMPI and IntelMPI implementations
  - Basically, mpirun <executable> runs MPI processes taken from $PBS_NODEFILE
  - But...
- IntelMPI allows to run a number of MPI processes even larger than the requested mpiprocs (I.e.the number of lines of the file $PBS_NODEFILE)
  - But if the PBS strict mode is active, the user will use only the reserved cores
  - If the number of MPI processes is larger than *ncpus*, we are experimenting the so called *oversubscribing*
- OpenMPI imposes mpiprocs as the upper limit for the number of MPI processes which can be run
  - Since mpiprocs<ncpus, no *oversubscribing* is possible

OpenMPI and IntelMPI

- It is possible to modify how the $PBS_NODEFILE is interpreted by mpirun
- OpenMPI:

  -npernode <n>

  – With -npernode it is possible to specify how many MPI processes to run on each node
- IntelMPI has the options :

  -perhost <n> || -ppn <n>

  – but these **do not work** at least on our installations

  – Since IntelMPI allows to run more processes than mpiprocs declared to PBS, you can specify mpiprocs=1 and then launch more processes which will be allocated round-robin

OpenMPI and IntelMPI

- It is also possible to completely override the $PBS_NODEFILE manually imposing a MPI machine file
  - Some limitations still apply
  - IntelMPI:

    -machine {name} | -machinefile {name}
  - OpenMPI

    -machinefile {machinefile} || --machinefile {machinefile} || -hostname {machinefile}

Where is my job running?

- It may seem a silly question but it is not
- How to check which resources have been allocated for my job
  - qstat -n1 <job_id> (for completed jobs add the -x flag)
  - qstat -f <job_id> |grep exec_host (for completed jobs add the -x flag)
  - Cat $PBS_NODEFILE (during the job execution, to be run by the master assigned node, e.g. in the batch script)
- How to check where am I actually running? Yes, it may be different from the previous point
  - mpirun <mpirun options> hostname
  - change the source code calling MPI_Get_processor_name
- Am I really using all the requested cores?
  - See more later

NUMA architectures

- Many current architectures used on HPC clusters are NUMA
  - Non-uniform memory access (NUMA) is a computer memory design used in multiprocessing, where the memory access time depends on the memory location relative to the processor. Under NUMA, a processor can access its own local memory faster than non-local memory (memory local to another processor or memory shared between processors)
- How to optimize my code and my run?
  - Numactl – Linux command to control NUMA policy for processes or shared memory
  - OpenMPI thread assignment
  - MPI process assignement

OpenMP thread assignment

- Intel compiler: set KMP_AFFINITY to compact/scatter (other options available)
  - Specifying compact assigns the OpenMP* thread <n>+1 to a free thread context as close as possible to the thread context where the <n> OpenMP* thread was placed.
  - Specifying scatter distributes the threads as evenly as possible across the entire system. scatter is the opposite of compact
  - add the modifier “verbose” to have a list of the mapping (threads vs core). For example: export KMP_AFFINITY=verbose,compact
  - More info at: https://software.intel.com/en-us/node/522691

- GCC OpenMP uses the lower level variable GOMP_CPU_AFFINITY.
  - For example, GOMP_CPU_AFFINITY="0 3 1-2 4-15:2"

OpenMP thread assignment / 2

- OpenMP v4.5 provides OMP_PLACES and OMP_PROC_BIND (it's standard!)
  - OMP_PLACES: Specifies on which CPUs the threads should be placed. The thread placement can be either specified using an abstract name or by an explicit list of the places. Allowed abstract names: threads, cores and sockets
  - OMP_PROC_BIND: specifies whether theads may be moved between CPUs. If set to TRUE, OpenMP theads should not be moved; if set to FALSE they may be moved. Use a comma separated list with the values MASTER, CLOSE and SPREAD to specify the thread affinity policy for the corresponding nesting level.
  - More from OpenMP recent standards
- Thread assignment can be crucial for manycore architectures (Intel PHI)

OpenMP thread assignment / 4

- Example (on a machine with 2 10-core sockets):

  export OMP_PLACES=cores;

• export OMP_PROC_BIND= master	• export OMP_PROC_BIND= close	• export OMP_PROC_BIND= spread
• t0 -> c0	• t0 -> c0	• t0 -> c0
• t1 -> c0	• t1 -> c1	• t1 -> c10
• t2 -> c0	• t2 -> c2	• t2 -> c1
• t3 -> c0	• t3 -> c3	• t3 -> c11

MPI process mapping/binding

- OpenMPI: tuning possible via options of mpirun
  - main reference: https://www.open-mpi.org/doc/v1.8/man1/mpirun.1.php
  - beware: mpirun automatically binds processes as of the start of the v1.8 series. Two binding patterns are used in the absence of any further directives:
    - Bind to core: when the number of processes is <= 2
    - Bind to socket: when the number of processes is > 2
  - more intuitive syntax still works but deprecated

    -npernode <n> / -npersocket <n>

MPI process mapping-binding / 2

- But --map-by function is more complete and can reproduce the above cases

  --map-by ppr:<n>:node / --map-by ppr:<n>:socket
- For process binding:

  --bind-to <node/socket/core/...>
- To order processes (rank in round-robin fashion according to the specified object)

  --rank-by <foo>
- --report-bindings: useful option to see what is happening wrt bindings

```
mpirun --map-by ppr:1:socket --rank-by socket --bind-to socket
--report-bindings -np $PROCS ./bin/$PROGRAM
```

MPI process mapping-binding / 3

- “If your application uses threads, then you probably want to ensure that you are either not bound at all (by specifying --bind-to none), or bound to multiple cores using an appropriate binding level or specific number of processing elements per application process.”
- mpirun --map-by ppr:1:socket:PE=10 --rank-by socket --bind-to socket --report-bindings <PROGRAM> (hybrid, 1 process per socket, 10 threads per process)
- mpirun --map-by ppr:2:socket:PE=5 --rank-by socket --bind-to socket --report-bindings <PROGRAM>
  - r0 -> c0-4
  - r1 -> c10-14
  - r2 -> c5-9
  - r3 -> c15-19

MPI process mapping-binding / 4

- IntelMPI: tuning possible via environment variables
  - main reference: https://software.intel.com/en-us/node/528816
- I_MPI_PIN: turn on/off process pinning. Default is on
- I_MPI_PIN_MODE: choose the pinning method
- I_MPI_PIN_PROCESSOR_LIST: Define a processor subset and the mapping rules for MPI processes within this subset.
- I_MPI_PIN_DOMAIN: additional environment variable to control process pinning for hybrid Intel MPI Library applications
- mpirun *-print-rank-map* to know the process binding

References

- MPI documents
  - http://www.mpi-forum.org/docs/

- OpenMP specifications
  - http://openmp.org/wp/openmp-specifications/

- **SuperComputing 2015:**
  - **http://www.mcs.anl.gov/~thakur/sc15-mpi-tutorial/ l**

- Intel & GNU compilers