INF560
Algorithmique Parallèle et Distribuée

2021/2022

Patrick CARRIBAULT

CEA, DAM, DIF, F-91297 Arpajon

Lecture Outline - Hybrid

- Motivations
- Example of hybrid code
- Data Parallelism (Domain Decomposition)
- Taxonomy
- Granularity
- Placement

Hybrid Programming

- Motivations

Distributed-Memory Model (MPI)

- What are main advantages?
- Work on shared-memory systems & distributed-memory systems
  - Runtime implementation can be easily adapted
  - Good code may run on any platform
- Exploit whole cluster
  - Full MPI runs are mainly possible
  - Example: IBM machine with more than 1M MPI tasks
- Data locality
  - By default with execution model
- Performance loss with explicit actions
  - Non-parallel parts: inside MPI calls or related to MPI calls

Distributed-Memory Model (MPI)

- What are the main drawbacks?
- Memory consumption
  - No easy shared memory available
  - Necessity to duplicate some data
- No easy data sharing
- Difficult to perform load balancing
  - Load balancing would require communications to drive which task is taking which job (control)
  - Necessity to share or exchange data

Shared-Memory Model (OpenMP)

- What are main advantages?
- Incremental way
  - Starting from sequential version, incrementally update code with directives
  - Keep original semantics
- Data sharing
  - Easy to share data
- Memory consumption
  - Nothing to duplicate by default
  - No internal structures (like network buffers)
- Load balancing (within a node)
  - Easy to dynamically control work distribution

Shared-Memory Model (OpenMP)

- What are main drawbacks?

- Data locality
  - By default, data are mainly located on master memory banks
  - Possibility to spread memory pages, but require knowledge on future parallelism

- Not usable on multiple compute nodes
  - Work only on shared-memory system
  - Tentative on distributed-memory system (e.g., Cluster OpenMP from Intel), but low performance and large overhead due to page migration through network

- Fork/join model (explicit parallelism)
  - Amdahl's law

Hybrid Programming

- One possible solution
  - Mix MPI and OpenMP inside the same application!

- Hybrid programming
  - Application contains calls to MPI functions and OpenMP constructs
  - Both models are alive at the same time
  - Involves 2 runtime libraries (plus dedicated compiler for OpenMP)

This lecture focuses on MPI/OpenMP, but it may be extended to other models

Hybrid Programming

- Advantages
  - Memory consumption reduction
    - Factor equal to the number of core per node
  - Load balancing between set of cores
    - Depend on the core distribution between OpenMP teams and MPI processes
- Drawbacks
  - Might be complex to program, debug, profile and maintain
  - Runtime libraries may not be fully interoperable
- To be developed in this lecture…

Hybrid Programming

- Example

Hello Hybrid

- How to create and hybrid Hello World?

- Goal
  - Keep program simple
  - Print Hello string with
    1. MPI rank
    2. Total number of MPI processes
    3. OpenMP rank
    4. Size of OpenMP team

- Requirements
  - Notion of MPI rank
  - Notion of OpenMP rank
  - Need to understand organization of groups: communicator (MPI) and teams (OpenMP)

Hello Hybrid

```
#include <mpi.h>
#include <omp.h>
#include <stdio.h>

int main( int argc, char ** argv ) {
  int mpi_rank, mpi_size ;

  MPI_Init( &argc, &argv ) ;
  MPI_Comm_rank( MPI_COMM_WORLD, &mpi_rank );
  MPI_Comm_size( MPI_COMM_WORLD, &mpi_size ) ;

#pragma omp parallel
{
  printf( "Hello MPI %d (%d) & OpenMP %d (%d)\n",
    mpi_rank, mpi_size,
    omp_get_thread_num(), omp_get_num_threads() ) ;
}

  MPI_Finalize() ;
  return 0 ;
}
```

Hello Hybrid

```
#include <mpi.h>
#include <omp.h>
#include <stdio.h>

int main( int argc, char ** argv ) {
  int mpi_rank, mpi_size ;

  MPI_Init( &argc, &argv ) ;
  MPI_Comm_rank( MPI_COMM_WORLD, &mpi_rank );
  MPI_Comm_size( MPI_COMM_WORLD, &mpi_size ) ;

#pragma omp parallel
{
  printf( "Hello MPI %d (%d) & OpenMP %d (%d)\n",
    mpi_rank, mpi_size,
    omp_get_thread_num(), omp_get_num_threads() ) ;
}

  MPI_Finalize() ;
  return 0 ;
}
```

Both headers

MPI initialization

Calls to MPI functions in "sequential" parts

Regular OpenMP parallel region

Variable written by MPI are shared in OpenMP parallel region

Hello Hybrid: Compilation

- How to compile hybrid code?
- Need to combine information for MPI compilation and OpenMP compilation
- MPI
  - Put header and library directory/name for MPI
  - Or use directly provided script (like `mpicc`)
- OpenMP
  - Put right options to enable OpenMP lowering and transformations
  - Depends on the underlying compilers (`-fopenmp` for GNU compilers)

```
$ mpicc -fopenmp -o hello_hybrid hello_hybrid.c
$
```

Hello Hybrid: Execution

- Arguments required to launch hybrid application
  - Model parameters
  - Job manager parameters

- Model parameters
  - Number of MPI processes
    - Specified through job manager
  - Number of OpenMP threads → `OMP_NUM_THREADS` variable
    - Correspond to the number of threads created by each MPI process (i.e., for every OpenMP team alive in the program)
    - Default number available

- Job manager parameters
  - Number of instances (= number of MPI processes) → `-n`
  - Number of nodes → `-N`
  - Number of cores per MPI processes → `-c`
  - Executable binary

Hello Hybrid: Execution

```
#include <mpi.h>
#include <omp.h>
#include <stdio.h>

int main( int argc, char ** argv ) {
  int mpi_rank, mpi_size ;
  MPI_Init( &argc, &argv ) ;
  MPI_Comm_rank( MPI_COMM_WORLD, &mpi_rank );
  MPI_Comm_size( MPI_COMM_WORLD, &mpi_size ) ;

#pragma omp parallel
{
  printf( "Hello MPI %d (%d) & OpenMP %d (%d)\n",
    mpi_rank, mpi_size,
    omp_get_thread_num(),
    omp_get_num_threads() ) ;
}

  MPI_Finalize() ;
  return 0 ;
}
```

$ salloc –n 2 –N 2 mpirun ./hello_hybrid

```
Hello MPI 1 (2) & OpenMP 0 (8)
Hello MPI 1 (2) & OpenMP 4 (8)
Hello MPI 1 (2) & OpenMP 3 (8)
Hello MPI 1 (2) & OpenMP 5 (8)
Hello MPI 1 (2) & OpenMP 2 (8)
Hello MPI 1 (2) & OpenMP 6 (8)
Hello MPI 1 (2) & OpenMP 7 (8)
Hello MPI 1 (2) & OpenMP 1 (8)
Hello MPI 0 (2) & OpenMP 0 (8)
Hello MPI 0 (2) & OpenMP 5 (8)
Hello MPI 0 (2) & OpenMP 7 (8)
Hello MPI 0 (2) & OpenMP 6 (8)
Hello MPI 0 (2) & OpenMP 1 (8)
Hello MPI 0 (2) & OpenMP 2 (8)
Hello MPI 0 (2) & OpenMP 3 (8)
Hello MPI 0 (2) & OpenMP 4 (8)
```

By default: 8 OpenMP threads per MPI process

Hello Hybrid: Execution

```
#include <mpi.h>
#include <omp.h>
#include <stdio.h>

int main( int argc, char ** argv ) {
  int mpi_rank, mpi_size ;
  MPI_Init( &argc, &argv ) ;
  MPI_Comm_rank( MPI_COMM_WORLD, &mpi_rank );
  MPI_Comm_size( MPI_COMM_WORLD, &mpi_size ) ;

#pragma omp parallel
{
  printf( "Hello MPI %d (%d) & OpenMP %d (%d)\n",
    mpi_rank, mpi_size,
    omp_get_thread_num(),
    omp_get_num_threads() ) ;
}

  MPI_Finalize() ;
  return 0 ;
}
```

$ salloc –n 2 –N 1 mpirun ./hello_hybrid

```
Hello MPI 1 (2) & OpenMP 0 (8)
Hello MPI 1 (2) & OpenMP 4 (8)
Hello MPI 1 (2) & OpenMP 3 (8)
Hello MPI 1 (2) & OpenMP 5 (8)
Hello MPI 1 (2) & OpenMP 2 (8)
Hello MPI 1 (2) & OpenMP 6 (8)
Hello MPI 1 (2) & OpenMP 7 (8)
Hello MPI 1 (2) & OpenMP 1 (8)
Hello MPI 0 (2) & OpenMP 0 (8)
Hello MPI 0 (2) & OpenMP 5 (8)
Hello MPI 0 (2) & OpenMP 7 (8)
Hello MPI 0 (2) & OpenMP 6 (8)
Hello MPI 0 (2) & OpenMP 1 (8)
Hello MPI 0 (2) & OpenMP 2 (8)
Hello MPI 0 (2) & OpenMP 3 (8)
Hello MPI 0 (2) & OpenMP 4 (8)
```

No adaptation of OpenMP related to resources

Hello Hybrid: Execution

```
#include <mpi.h>
#include <omp.h>
#include <stdio.h>

int main( int argc, char ** argv ) {
  int mpi_rank, mpi_size ;
  MPI_Init( &argc, &argv ) ;
  MPI_Comm_rank( MPI_COMM_WORLD, &mpi_rank );
  MPI_Comm_size( MPI_COMM_WORLD, &mpi_size ) ;

#pragma omp parallel
{
  printf( "Hello MPI %d (%d) & OpenMP %d (%d)\n",
    mpi_rank, mpi_size,
    omp_get_thread_num(),
    omp_get_num_threads() ) ;
}

  MPI_Finalize() ;
  return 0 ;
}
```

$ OMP_NUM_THREADS=3 salloc –n 2 –N 1 mpirun ./hello_hybrid

```
Hello MPI 0 (2) & OpenMP 0 (3)
Hello MPI 0 (2) & OpenMP 2 (3)
Hello MPI 0 (2) & OpenMP 1 (3)
Hello MPI 1 (2) & OpenMP 0 (3)
Hello MPI 1 (2) & OpenMP 1 (3)
Hello MPI 1 (2) & OpenMP 2 (3)
```

Hello Hybrid: Execution

- Parameters from one model to another are independent

- Running the application in pure OpenMP

```
$ OMP_NUM_THREADS=8 salloc -n 1 -N 1 mpirun ./hello_hybrid
```

- Running the application in pure MPI

```
$ OMP_NUM_THREADS=1 salloc -n 8 -N 1 mpirun ./hello_hybrid
```

- Of course the application should support these modes!

Hello Hybrid: Execution

- Be careful to the default configuration of the software stack installed on the supercomputer

- Example on CEA supercomputer

```
$ srun -N 2 -n 2 -c 2 -p nehalem ./hello_hybrid
Hello MPI 1 (2) & OpenMP 0 (1)
Hello MPI 0 (2) & OpenMP 0 (1)
```

- Default number of OpenMP threads → 1
  - But 2 cores per task asked by SLURM
  - And number of cores per node: 8 (dual-socket quad-core Nehalem CPUs)

- Checking default environment variable

```
$ env | grep OMP_NUM_THREADS
OMP_NUM_THREADS=1
```

Hello Hybrid: Execution

- Relation between `-c` and number of allowed cores
  - Depends on SLURM & OpenMP runtime

- Example 1

```
$ unset OMP_NUM_THREADS
$ srun -N 2 -n 2 -c 2 -p nehalem ./hello_hybrid
Hello MPI 1 (2) & OpenMP 0 (2)
Hello MPI 1 (2) & OpenMP 1 (2)
Hello MPI 0 (2) & OpenMP 0 (2)
Hello MPI 0 (2) & OpenMP 1 (2)
```

- Example 2

```
$ srun -N 2 -n 2 -c 1 -p nehalem ./hello_hybrid
Hello MPI 1 (2) & OpenMP 0 (1)
Hello MPI 1 (2) & OpenMP 0 (1)
```

Hybrid Programming

- Domain Decomposition

Domain Decomposition

- One way to exploit data parallelism with different paradigm
  - Domain decomposition
- Main principle
  - Split data structures into multiple pieces
  - Each execution flow works on its own subset of data
- Advantages
  - Improve locality
  - Reduce communication and/or data sharing
- Available models
  - Can be exploited in MPI, OpenMP or hybrid

Domain Decomposition

- Domain decomposition in MPI
  - Each MPI owns a domain
    - E.g., tile of mesh, part of image, subpart of text…
  - Each MPI task processes (i.e., read/write) data inside its own domain
    - Write access is exclusive to this task

- How to access data which are outside domain?
  - E.g., blur filter in image project or pattern matching
  - Need to have a copy of those data and maintain them up-to-date
  - Notion of *ghost cells*

Domain Decomposition

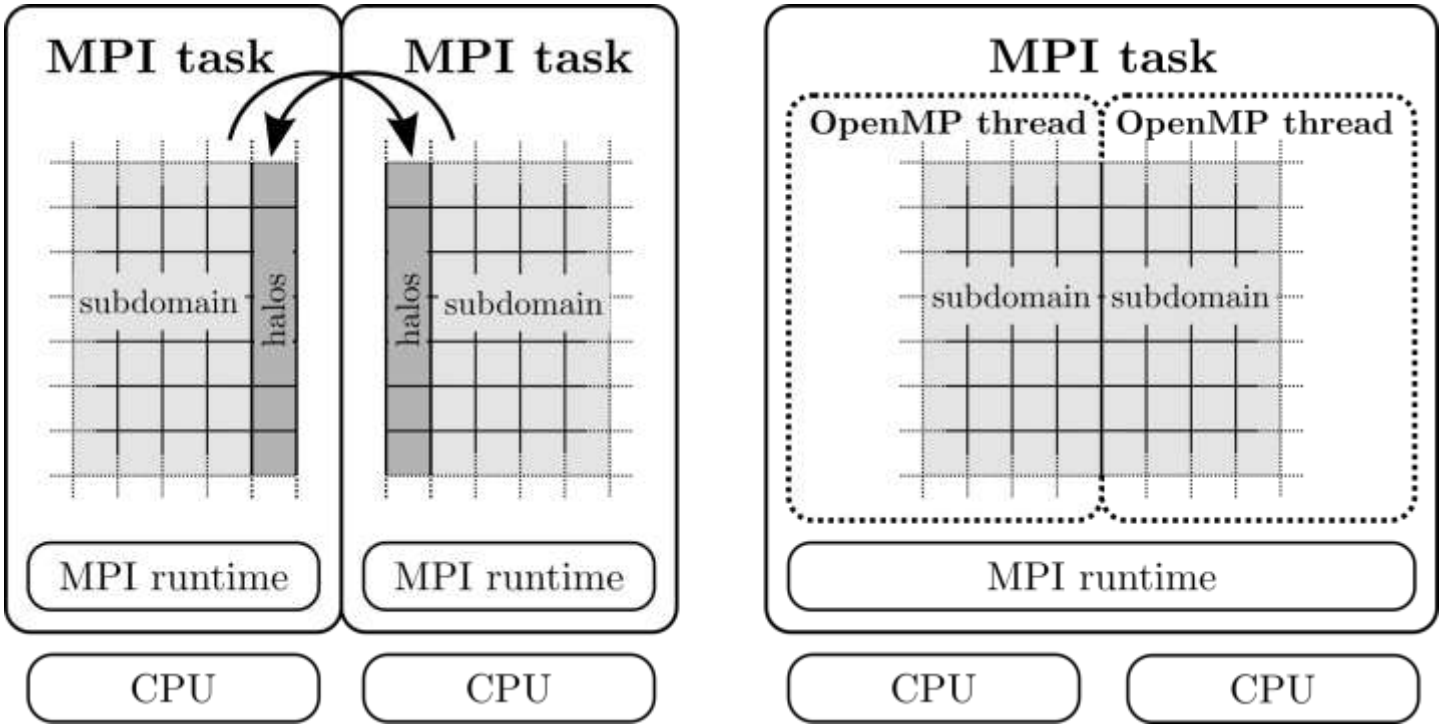
- Domain decomposition in OpenMP
  - Each OpenMP thread processes a subset of data
    - By hand through parallel regions
    - Automatically with for construct and static scheduling
  - OpenMP proposes constructs and clauses to perform automatic load balancing
    - Clause dynamic on workshare loop

- How to deal with data outside domain
  - Shared-memory model allows access to everything
  - Regular load operations

Domain Decomposition

- Domain decomposition in MPI+OpenMP
- First division for MPI task
  - MPI is the first model to be launched
  - Every MPI task starts the program by launching the main function
- Second division for OpenMP threads
  - Each MPI tasks create a team of OpenMP threads
  - Threads can be synchronized and driven inside one team (nothing between teams)

Domain Decomposition



Hybrid Programming

- Taxonomy

Difficulties of Hybrid Programming

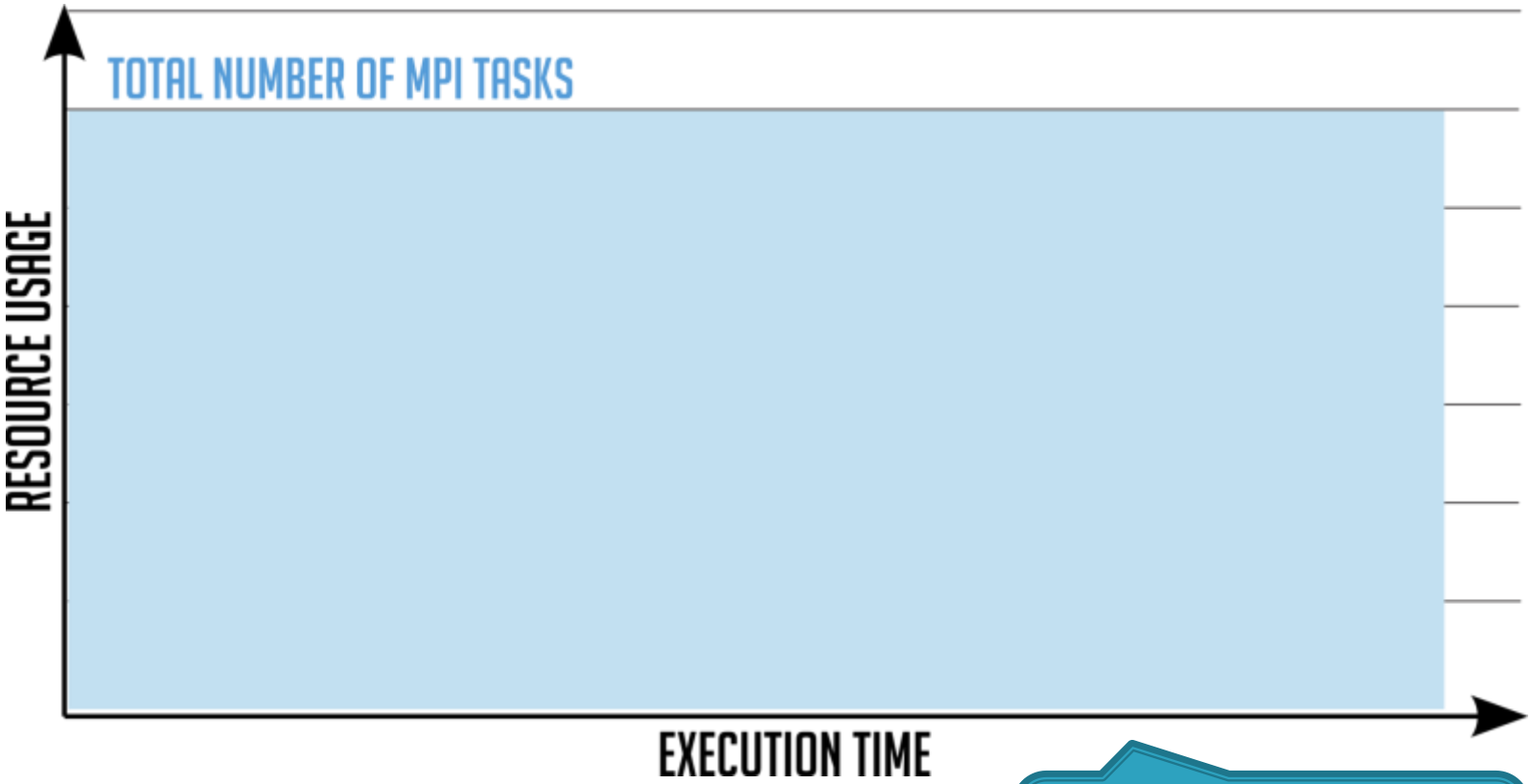
- Incremental
  - Scientific applications already parallelized with MPI and a shared-memory model is added to this existing code
  - Slow evolution of existing codes and a major difficulty to express this new level of parallelism.
- Mixing several interfaces
  - multiple models with completely different APIs or sets of directives can lead to difficulties because the underlying runtime systems may not be aware from each other
- Various ways of mixing
  - adding OpenMP can be as simple as putting some directives on a subset of loops or it can be trickier by trying to open one unique parallel region covering the whole program execution

Graphical Representation

- Main goal
  - Find major parameters shaping performance of hybrid application
- For this purpose
  - Study of evolution of resource utilization over time with full MPI code and hybrid version
- Constraints
  - Consider no overhead related to programming models
  - Consider only if a resource is exploited or not
    - No notion of single-core performance

Graphical Representation

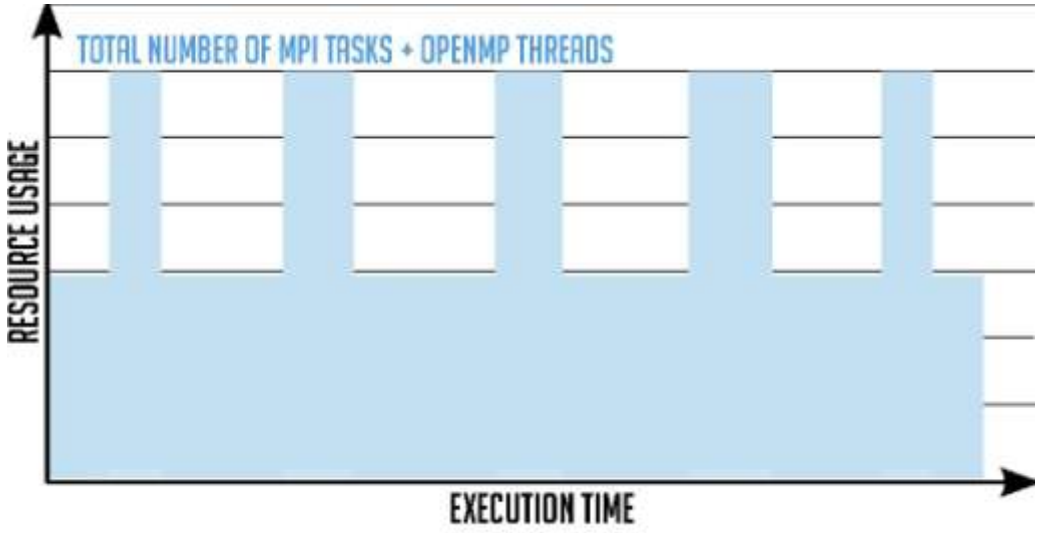
- Full MPI application



Do not consider performance of compute part

Graphical Representation

- MPI/OpenMP hybrid code
  - Small parallel regions (e.g., only loops)



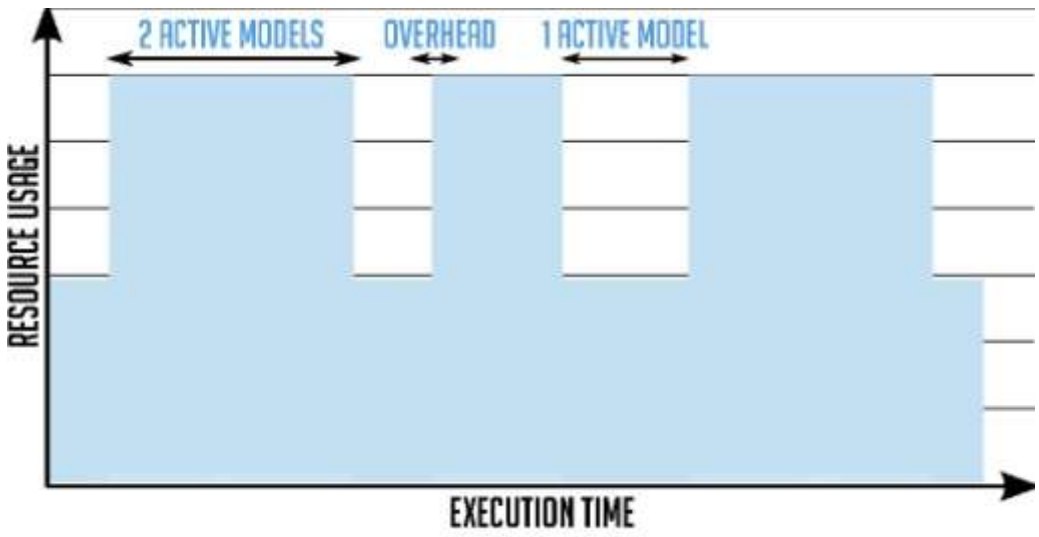
Remarks
- 2 OpenMP threads per team

Constraints
- No runtime overhead
- Bulk synchronous

Conclusion
- Amdahl's law

Graphical Representation

- MPI/OpenMP hybrid code
  - Larger parallel regions



Remarks
- 2 OpenMP threads per team
- Fewer parallel regions

Constraints
- No runtime overhead
- Bulk synchronous

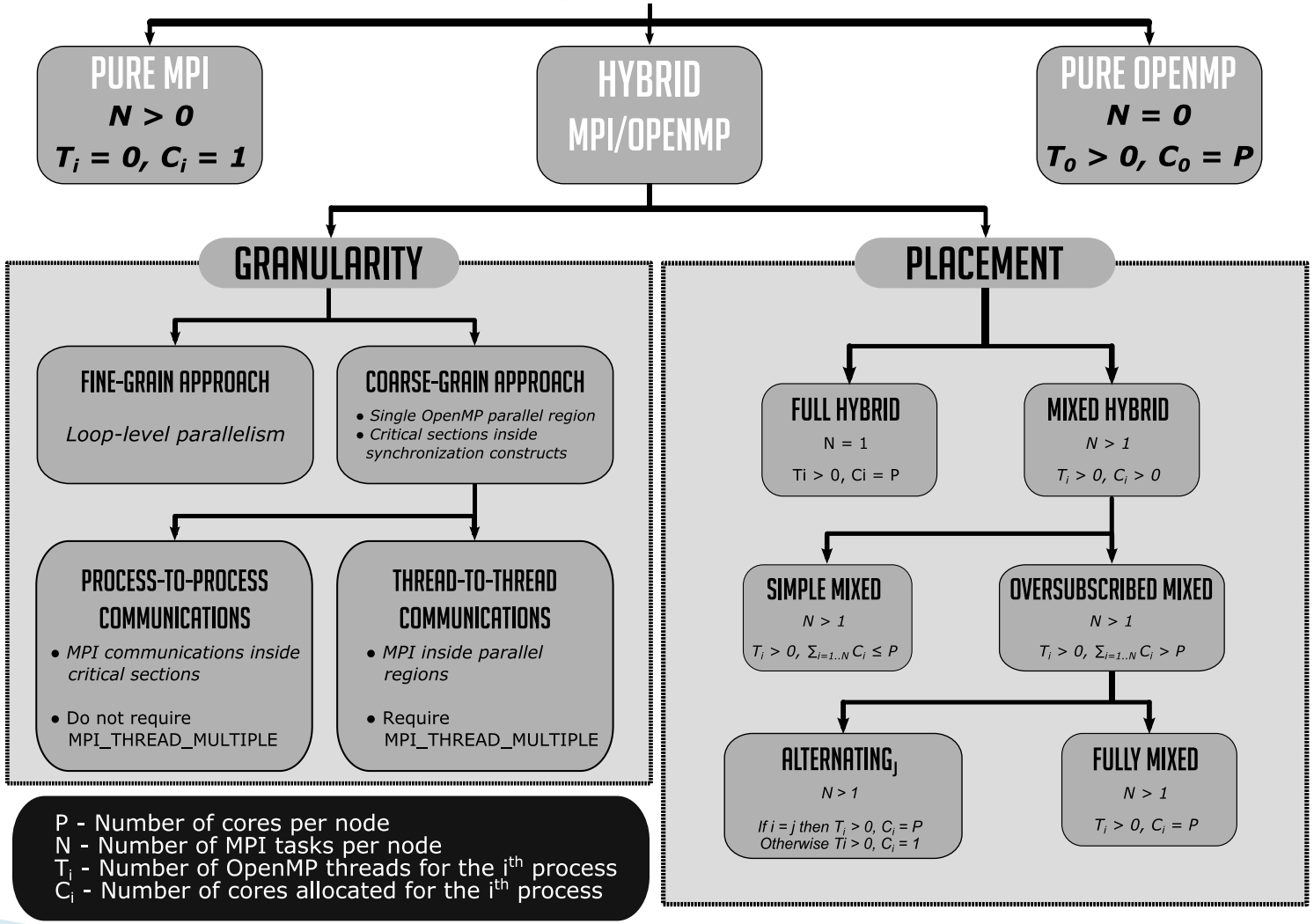
Conclusion
- Less spare resources

Towards a Taxonomy

- How models are used may shape the overall performance
  - Simple representation but enough to highlight some issues
  - It is important to exploit all resources as often as possible
    - Either when both models are used
    - Or if models are spread differently among cores/resources

- Deducing two parameters shaping performance
  - Granularity
  - Placement

Hybrid Taxonomy



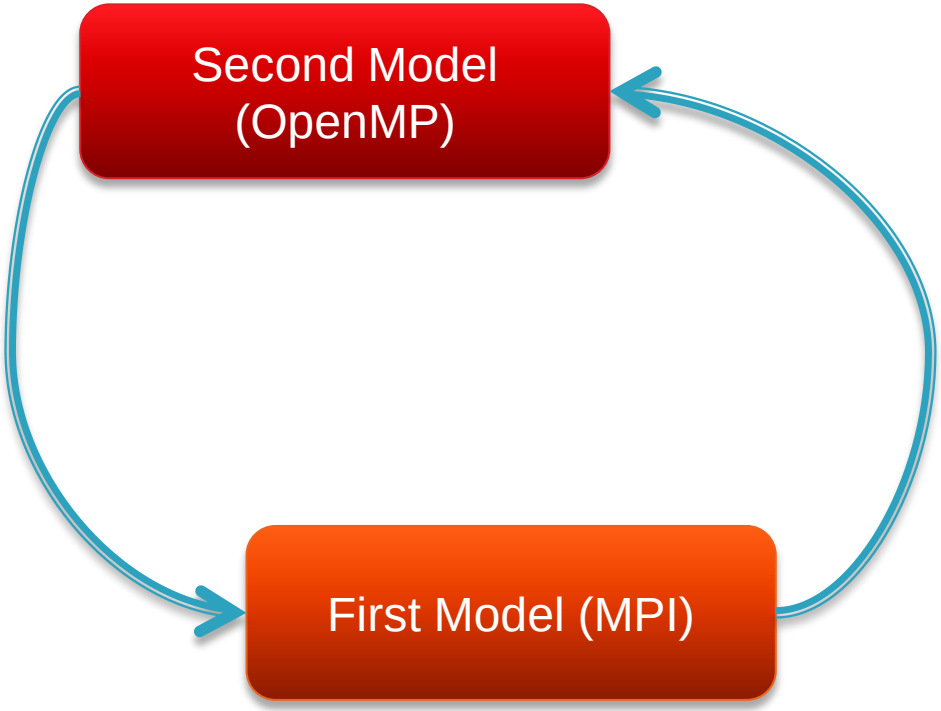
P - Number of cores per node
N - Number of MPI tasks per node
T_i - Number of OpenMP threads for the ith process
C_i - Number of cores allocated for the ith process

Hybrid Programming

- Granularity

Granularity

- Mixing two models may involve different ways to include the second model inside the first one
- Example:
  - How are the OpenMP parallel region regarding MPI functions?
- Impact on the first model in order to consider different contexts when performing parallel actions (like sending and receiving messages in MPI)



Fine Grain w/ MPI

- Fine grain mode
  - OpenMP parallel regions contains no calls to MPI
  - MPI functions are called only in regular context
    - As for full MPI application
- Consequences
  - Almost no impact
  - Be careful of spare cores because of Amdahl's law
- Example
  - Hybrid code to find max inside an array

Fine Grain w/ MPI

```
MPI_Comm_size( MPI_COMM_WORLD, &mpi_size)

/* global_tab contains all elements in rank 0 */

MPI_Scatter( global_tab, n/mpi_size,
  MPI_INT, local_tab, n/mpi_size, MPI_INT,
  0, MPI_COMM_WORLD);

max = local_tab[0];
#pragma omp parallel for private(i) reduction(max:max)
for ( i = 1 ; i < n/mpi_size ; i++ ) {
  if ( local_tab[i] > max ) {
    max = local_tab[i] ;
  }
}

MPI_Allreduce( &max, &global_max, 1,
  MPI_INT, MPI_MAX, MPI_COMM_WORLD);
```

Distribution of array element over MPI ranks

Processing of each subpart in OpenMP (no MPI calls inside)

Eventually, each MPI task has the max value in `global_max` variable

Coarse Grain w/ MPI

- Coarse grain mode
  - May involve MPI calls inside OpenMP parallel region
  - Can be different contexts
    - If inside master construct, single region or just done by any thread inside parallel region
- Consequences
  - MPI should support calls in multithreaded context
  - Each OpenMP thread belonging to the same team will have the same MPI rank
- How to know if MPI library support threads?
  - Notion of initialization of MPI with thread support
  - Will imply a different process inside the MPI implementation
    - For example: require locking structure to avoid data concurrency

Thread Safety in MPI

- MPI 2 defines 4 levels of thread safety
  - `MPI_THREAD_SINGLE`: one thread exists
  - `MPI_THREAD_FUNNELED`: process can be multithreaded by only master thread is allowed to perform MPI calls
  - `MPI_THREAD_SERIALIZED`: process can be multithreaded, but one thread at a time can perform MPI calls
  - `MPI_THREAD_MULTIPLE`: no restriction

- To choose the thread-safety level
  - No call to `MPI_Init` anymore
  - Call to `MPI_Init_thread`

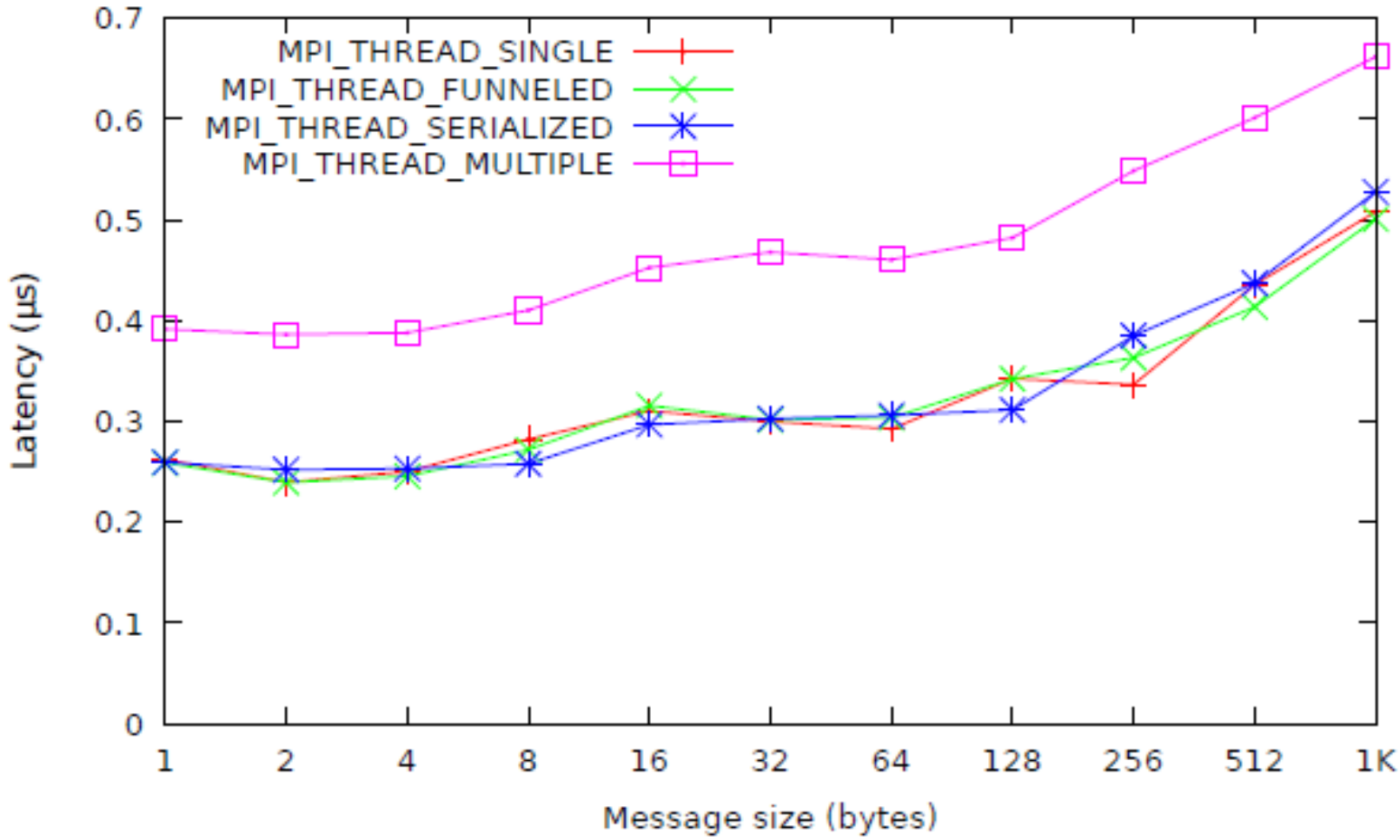
```
int MPI_Init_thread( int *argc,
     char ***argv, int required, int *provided )
```

- Warning
  - Required contains the level you want
  - Provided contains (as output) the level chosen by the runtime library
  - Both parameters may be different!

Thread Safety in MPI

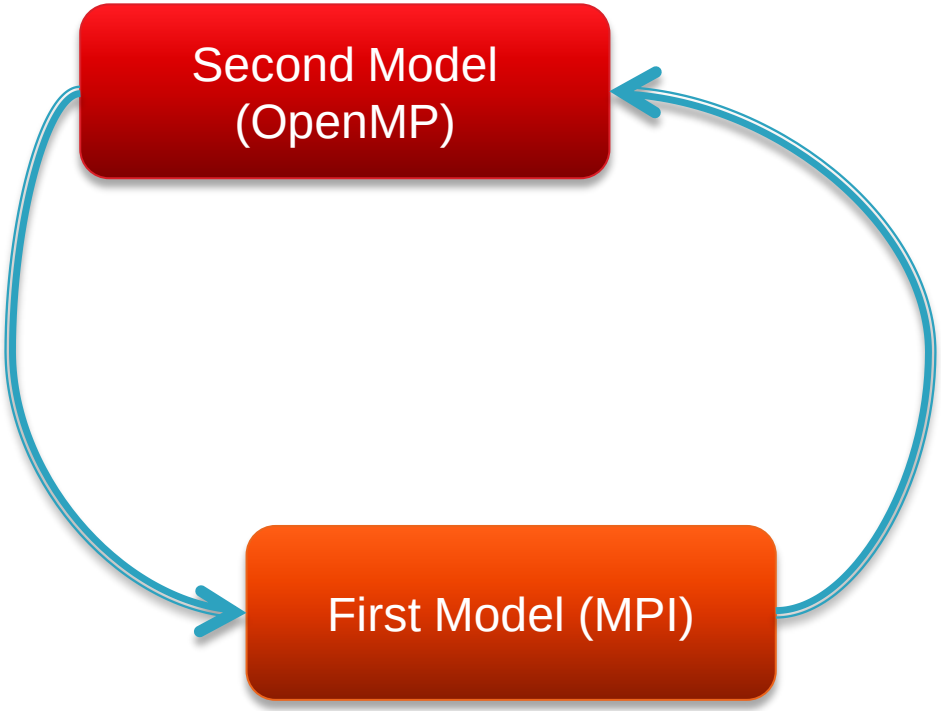
- Impact on MPI implementation and behavior
- Thread-safety requires some sort of locking mechanisms (or maybe data duplication)
- MPI_THREAD_SINGLE → no impact because application should not call MPI functions in parallel regions
- MPI_THREAD_FUNNELED → be careful about calling an external library which is non thread safe
- MPI_THREAD_SERIALIZED → reentrant code (no thread-specific variables)
- MPI_THREAD_MULTIPLE → data structure accesses should be protected by any sort of mechanism

Thread Safety in MPI



Granularity

- Granularity has a large impact on the first model
  - Need to handle multithreaded calls
  - Mandatory to avoid performance penalty
- What about the second model?



Granularity w/ OpenMP

- Fine-grain mode
  - OpenMP used with small parallel regions
  - For example: loop-based parallelism with combined constructs like parallel for
  - Only few OpenMP constructs and clauses are involved

- Consequences
  - Performance of OpenMP mainly relies on the capacity to activate and deactivate OpenMP threads
  - Basically, performance of entering and exiting parallel regions is the key
  - Optimization of loop scheduling (e.g., dynamic schedule) is important as well

Granularity w/ OpenMP

- Coarse-grain mode
  - OpenMP used with large parallel regions
  - Ideal situation: only one parallel region, opened after entering `main` function and exited before `MPI_Finalize`
  - Need to rely on many other constructs to drive the execution of threads inside this parallel region

- Consequences
  - Performance of performing a parallel region is not crucial
  - But performance of other constructs are important
    - Barrier
    - Single
    - …
  - Of course, loop scheduling should be optimized as well in this context

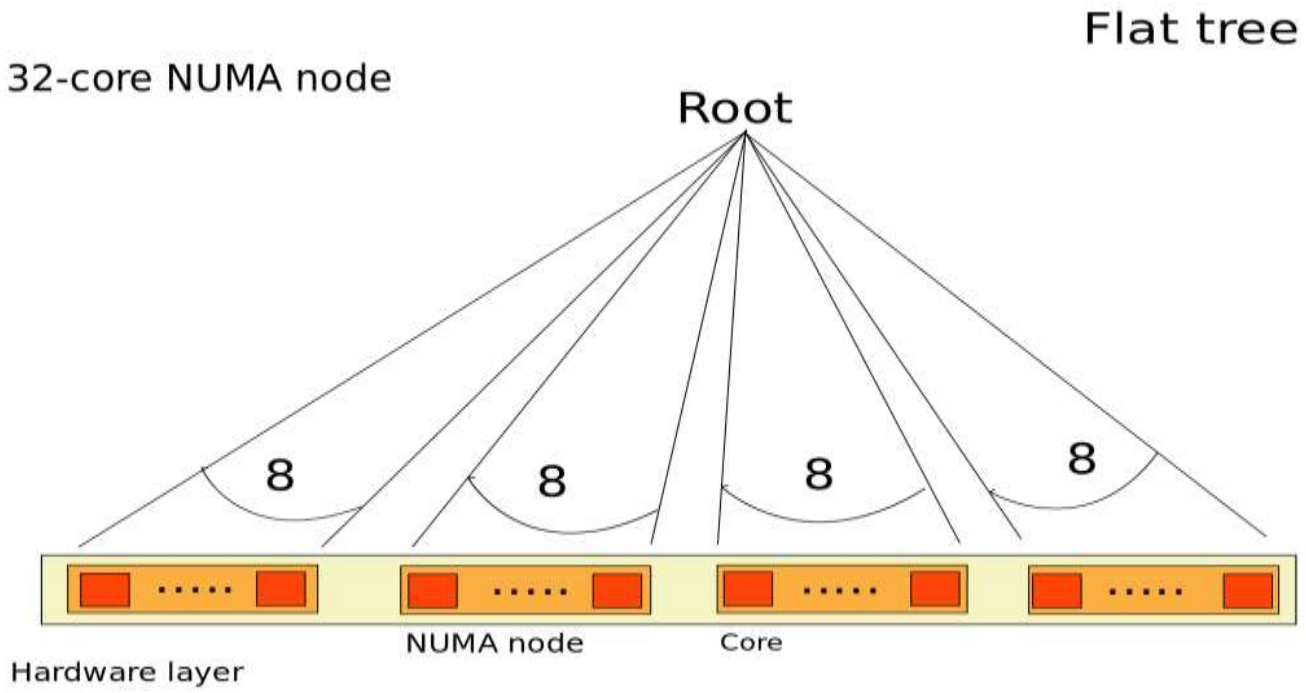
Granularity w/ OpenMP

- Impact of runtime stacking on OpenMP Layer
  - Require strong performance in fine-grain and coarse-grain approaches
  - Fine-grain
    - Optimization of launching/stopping a parallel region
    - Optimization of performing loop scheduling
  - Coarse-grain
    - Optimization of synchronization constructs (barrier, single, nowait…)

- OpenMP runtime design and implementation
  - Goal: design of OpenMP runtime fully integrated into MPI runtime dealing with Granularity and Placement
  - Implementation in MPC (Multi-Processor Computing) unified with optimized MPI layer
    - MPC: Thread-based MPI (CEA & ECR development with Marc Pérache and Julien Jaeger)
    - Open source: available at http://mpc.paratools.com/ (current version 4.0.0)

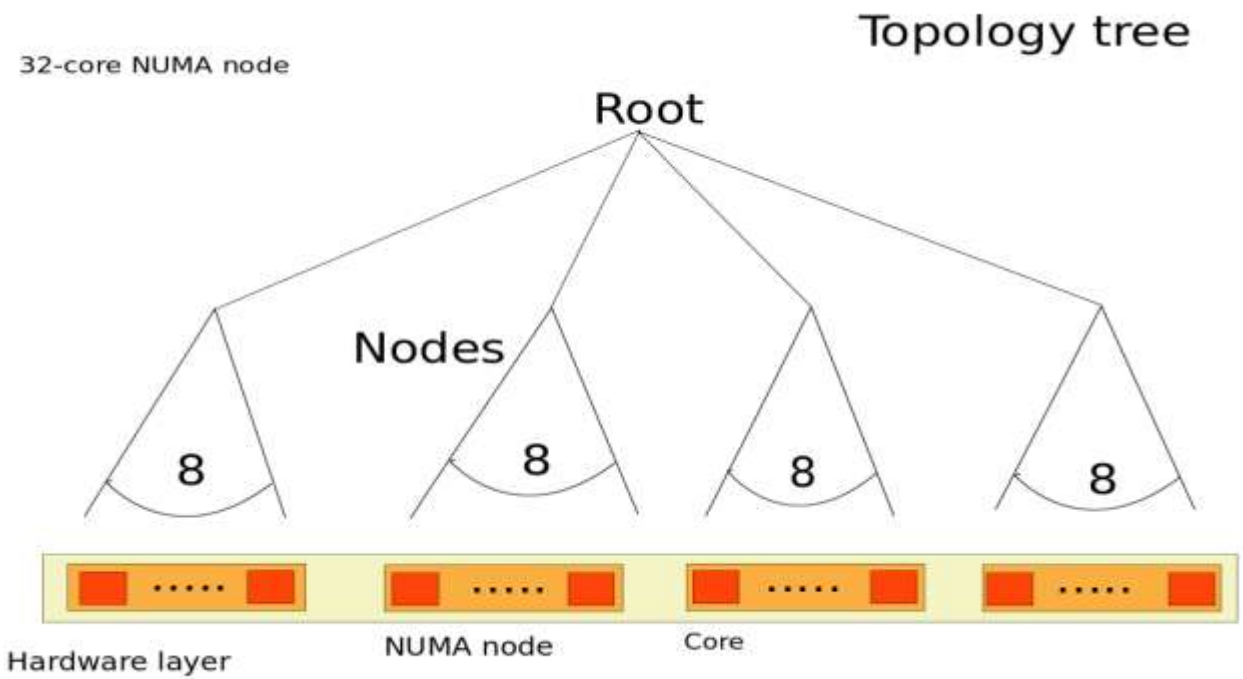
OpenMP Main Structure

- Most simple structure
  - Fast to wake few threads
  - Large overhead for numerous threads
- Example w/ 32-core node (TERA 100)



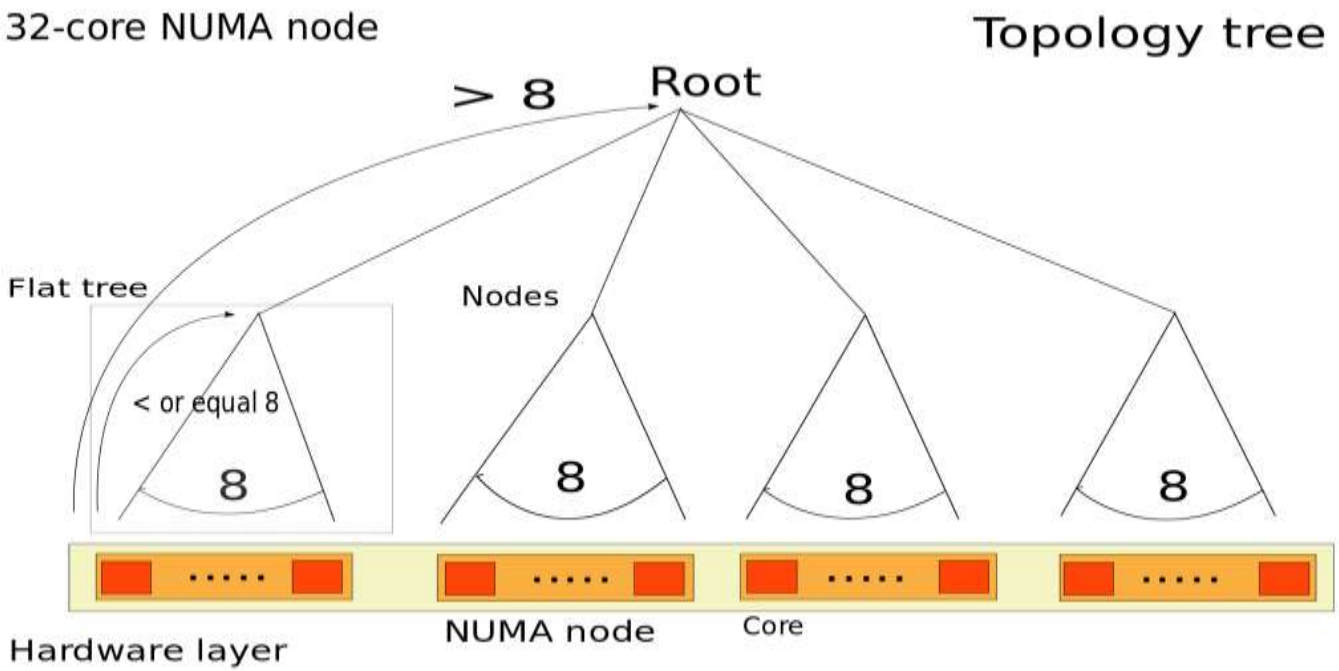
OpenMP Main Structure

- Tree following the architecture topology
  - 4 NUMA nodes with 8 cores ➔ 4-8 tree
  - More parallelism to wake up large number of threads
  - Overhead for few threads (tree height)



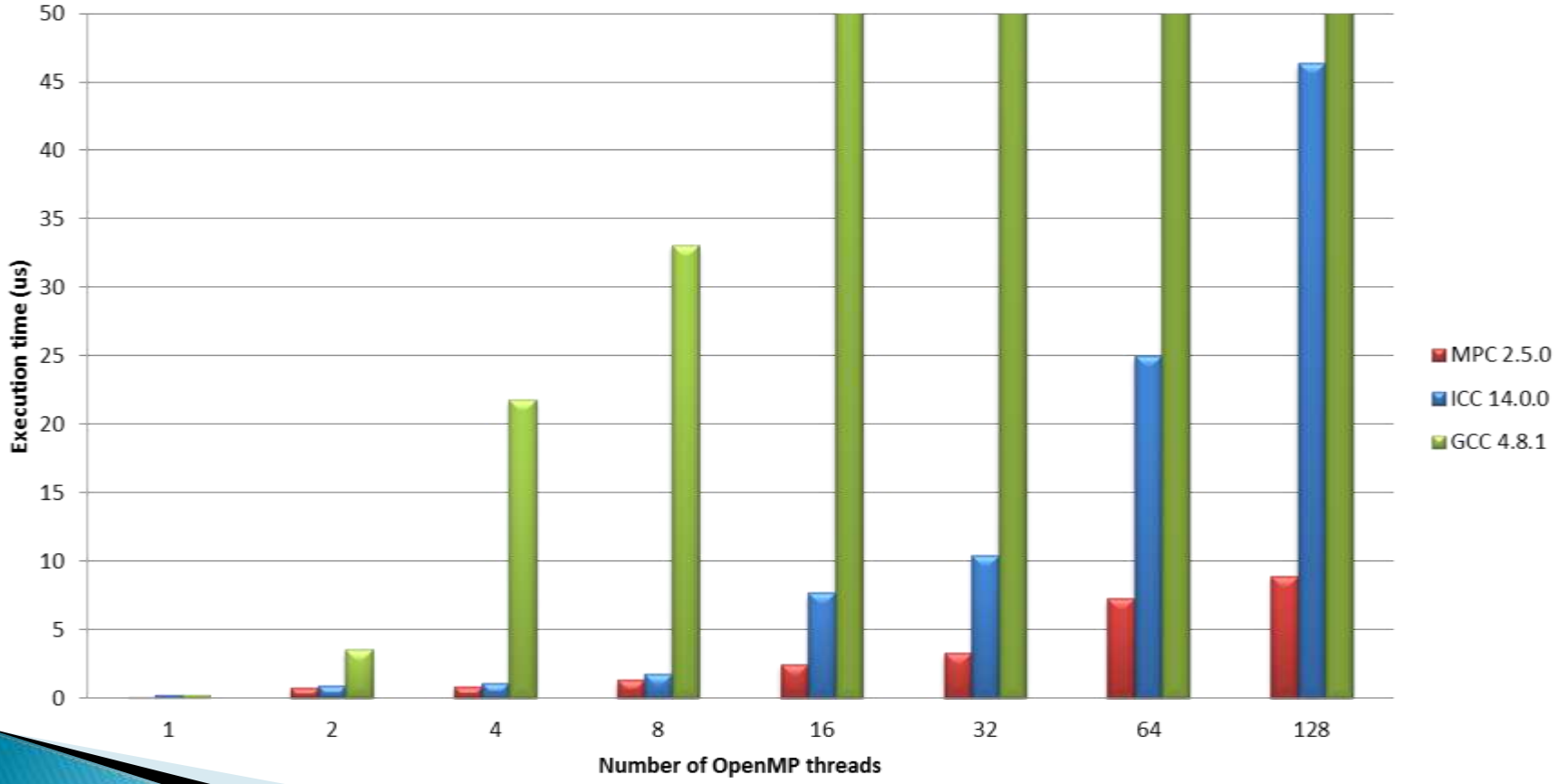
OpenMP Main Structure

- MPC implementation: dynamically choose the right root
  - Exploit sub-trees inside the topology tree for efficient thread activation and synchronization
  - Depending on the number of threads, use the smallest sub-trees



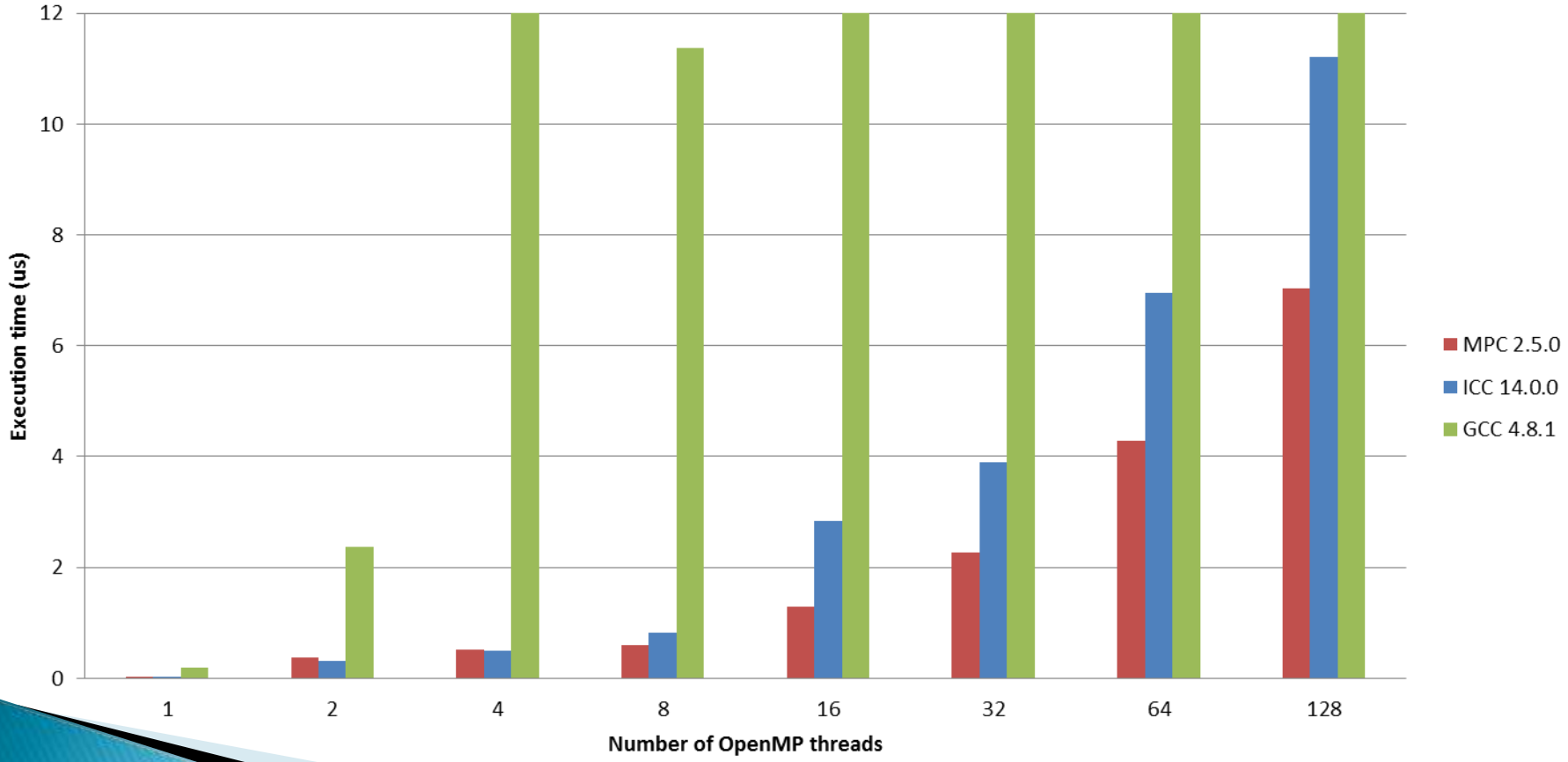
OpenMP Experimental Results

- EPCC Microbenchmark: Parallel Region Overhead
  - Experiments on 16-sockets Nehalem EX (8 cores), 2 NUMA levels ➔ 128 cores



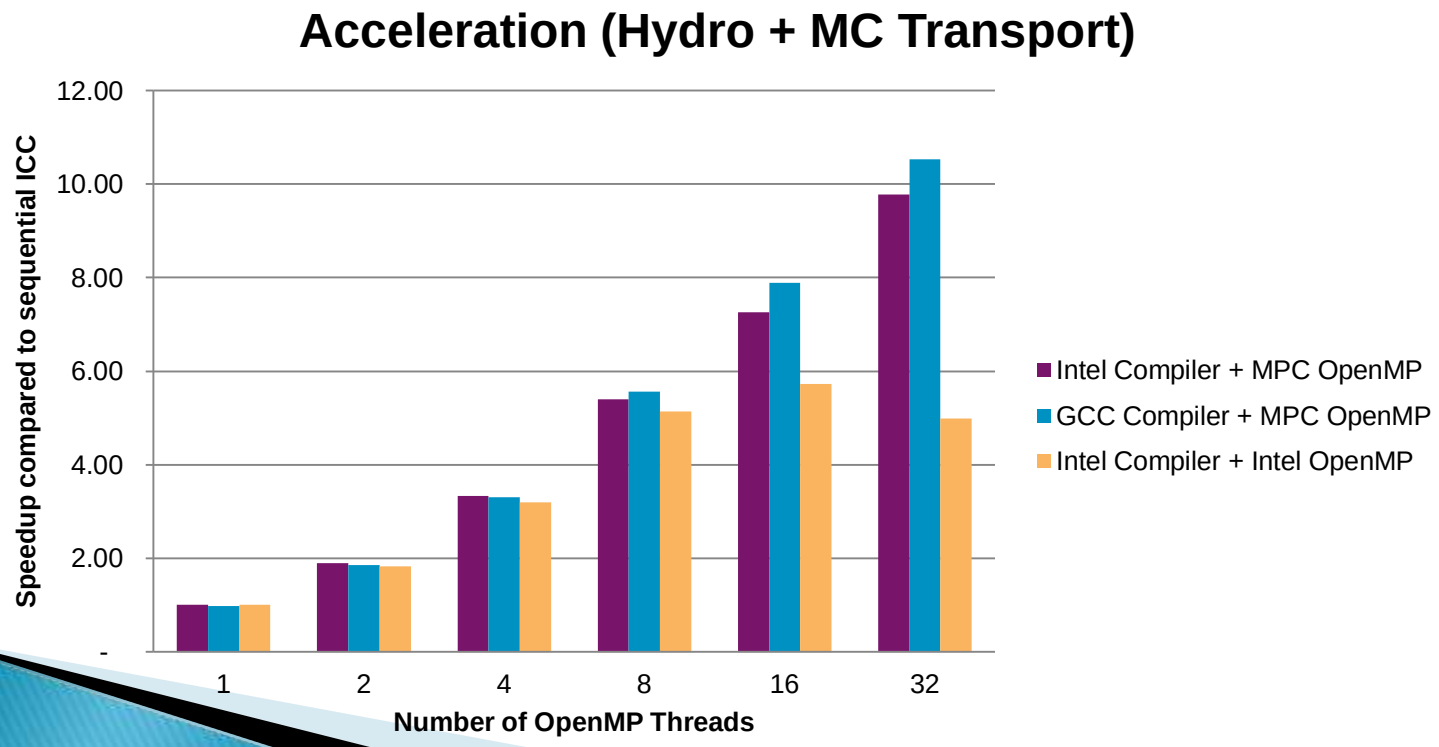
OpenMP Experimental Results

- EPCC Microbenchmark: Barrier Overhead
  - Experiments on 16-sockets Nehalem EX (8 cores), 2 NUMA levels ➔ 128 cores



OpenMP Experimental Results

- Multi-physics input set
  - Mesh: 500,000 AMR cells and 200,000 Monte-Carlo particles (200 time steps)
  - Results on one node of CEA TERA-100 (4 x 8-core Nehalem CPUs = 32 cores)
  - Comparison of Intel/GCC compilers and Intel/MPC OpenMP runtime
  - Better scalability with MPC

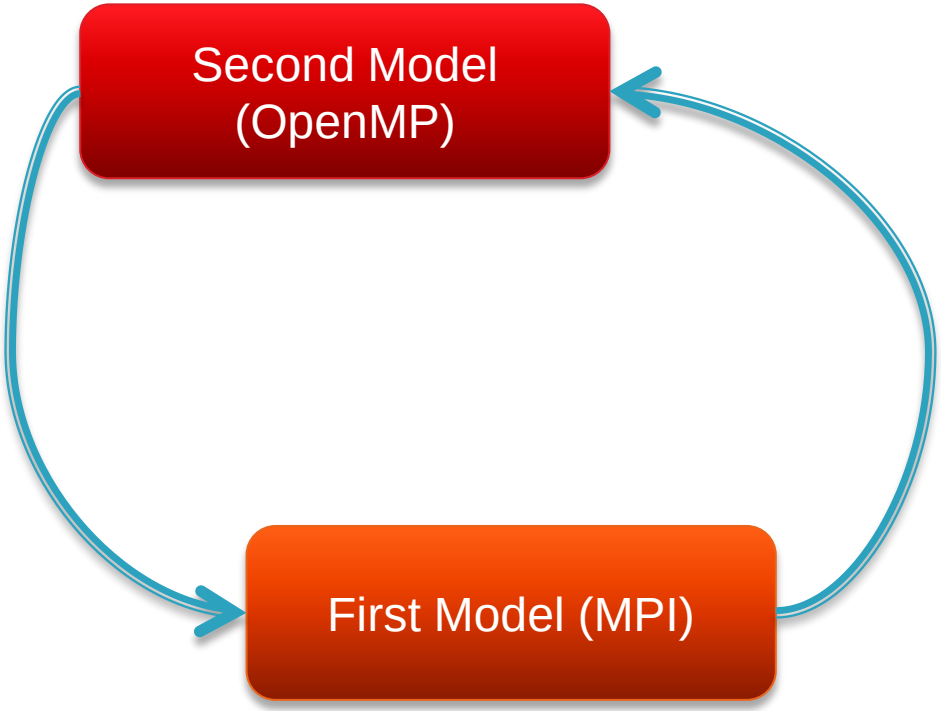


Hybrid Programming

- Placement

Placement

- Second parameter from taxonomy shaping overall performance
- Mixing multiple programming models may lead to concurrency for resources
  - Thread/process binding on cores/hyperthreads
- Need to book resources according to other models
  - First model will spawn first
  - Second model should adapt



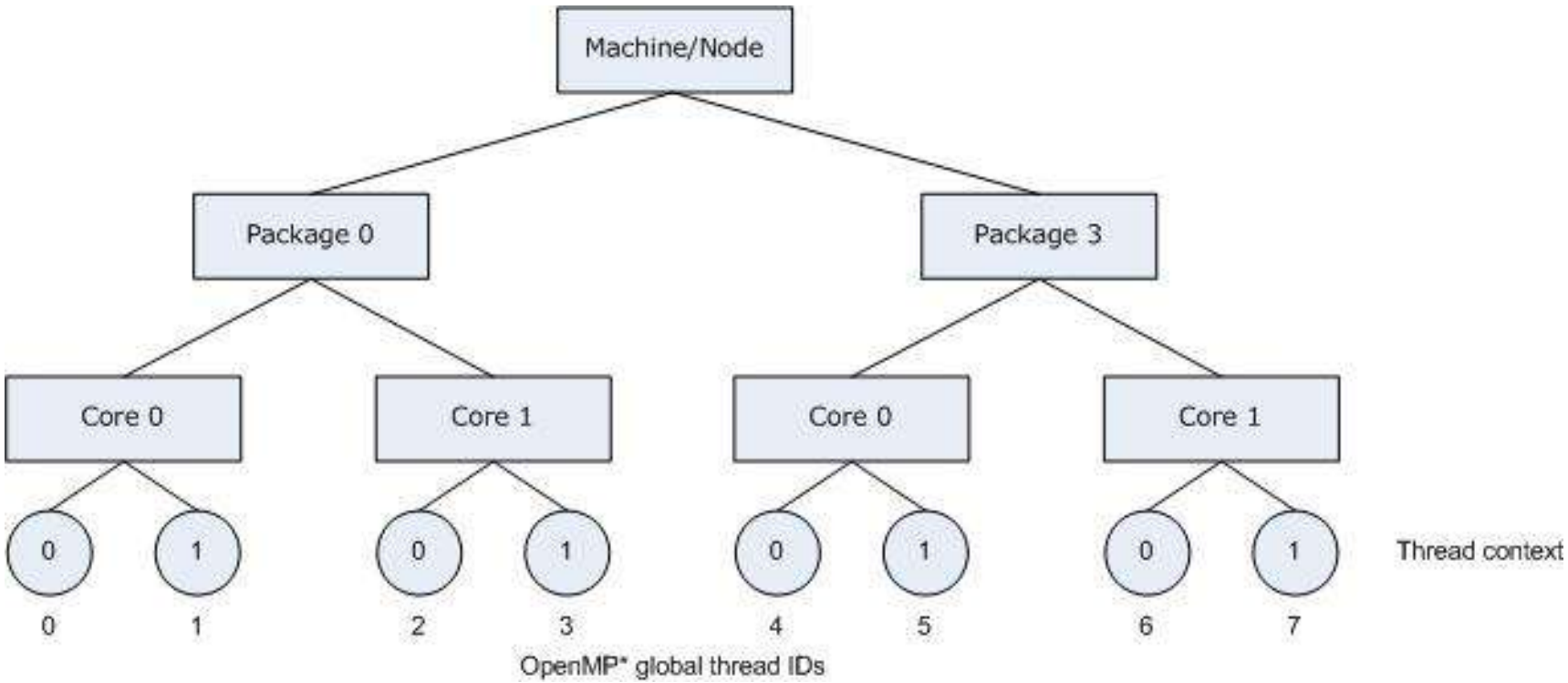
OpenMP Placement

- Thread placement according to available cores (job manager + MPI runtime)
  - Fully Hybrid/Simple Mixed → no oversubscribing
  - Fully Mixed/Alternating → oversubscribing → need to avoid busy waiting

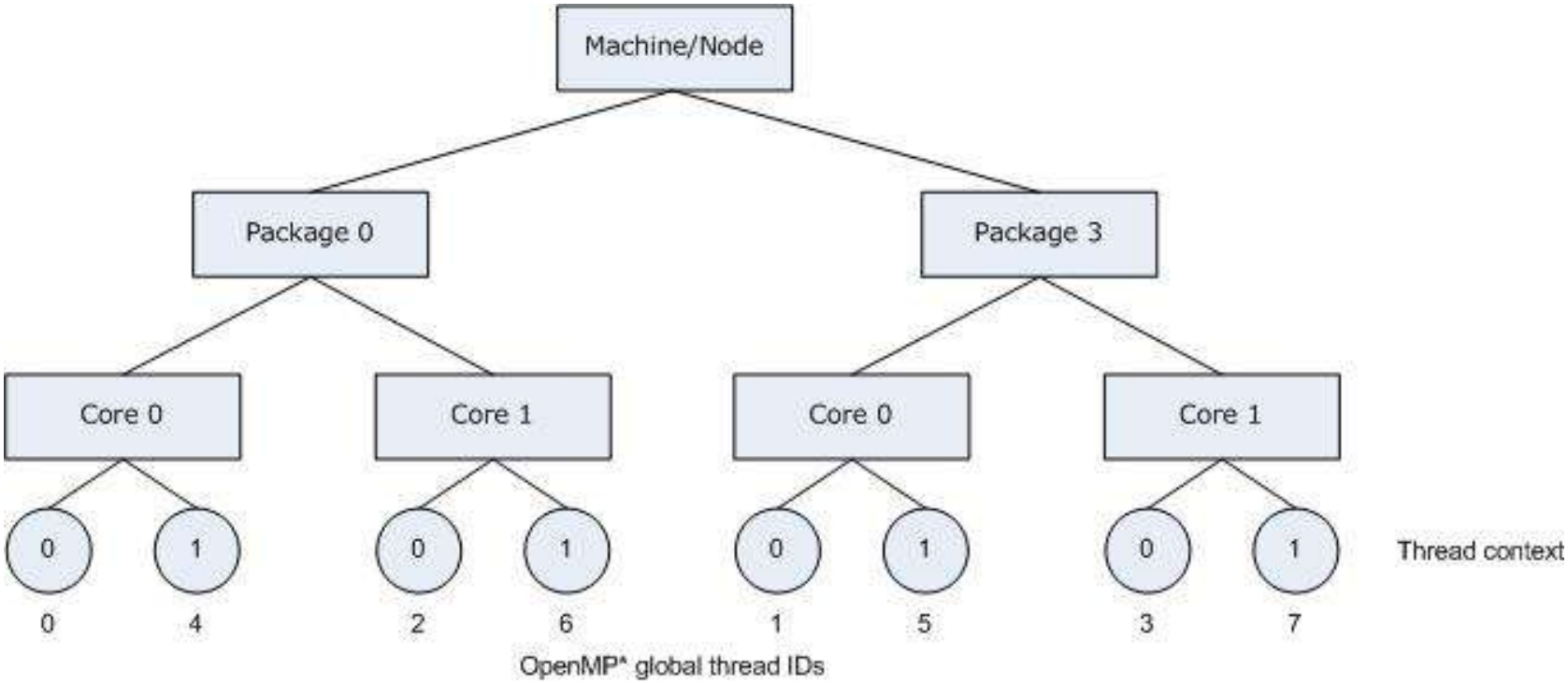
- Existing API to bind OpenMP threads
  - Intel compiler:

    ```
    KMP_AFFINITY=[<modifier>,...] <type> [,<permute>] [,<offset>]
    ```
  - Since OpenMP 4: OMP_PROC_BIND w/ OMP_PLACES

OpenMP Compact Binding



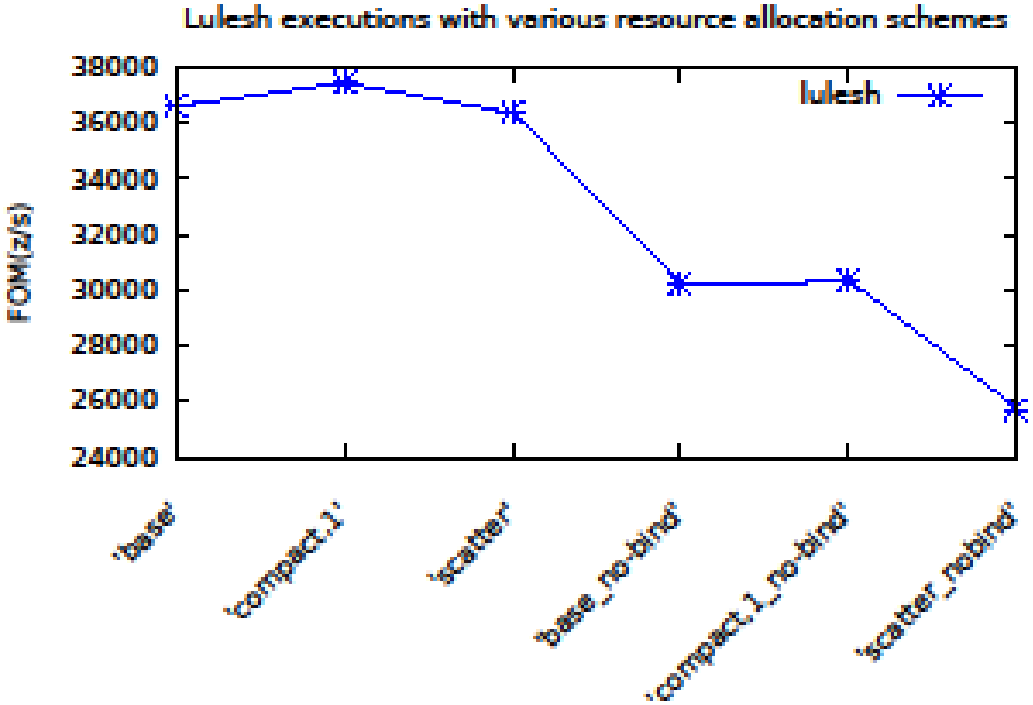
OpenMP Scatter Binding



Placement Experiments

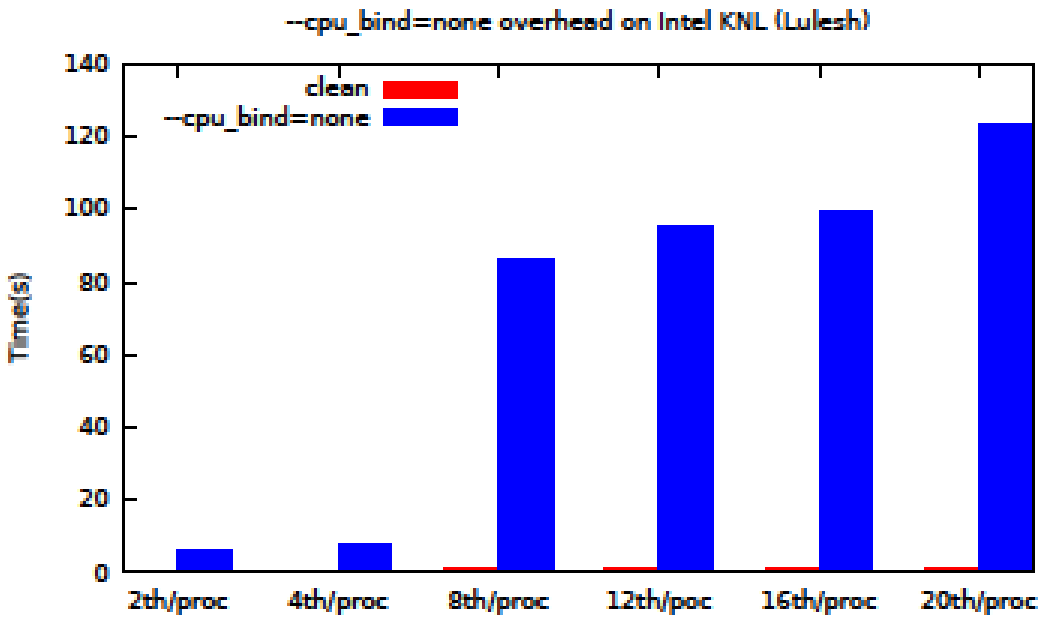
- Experiments of running LULESH benchmark on 4 nodes w/ dual-socket 16-core Haswell (w/ hyperthreading)
  - W/ and w/out KMP_AFFINITY (OpenMP)
  - W/ and w/ binding for MPI (from SLURM)

- Results in Figure of Merit (higher is better)



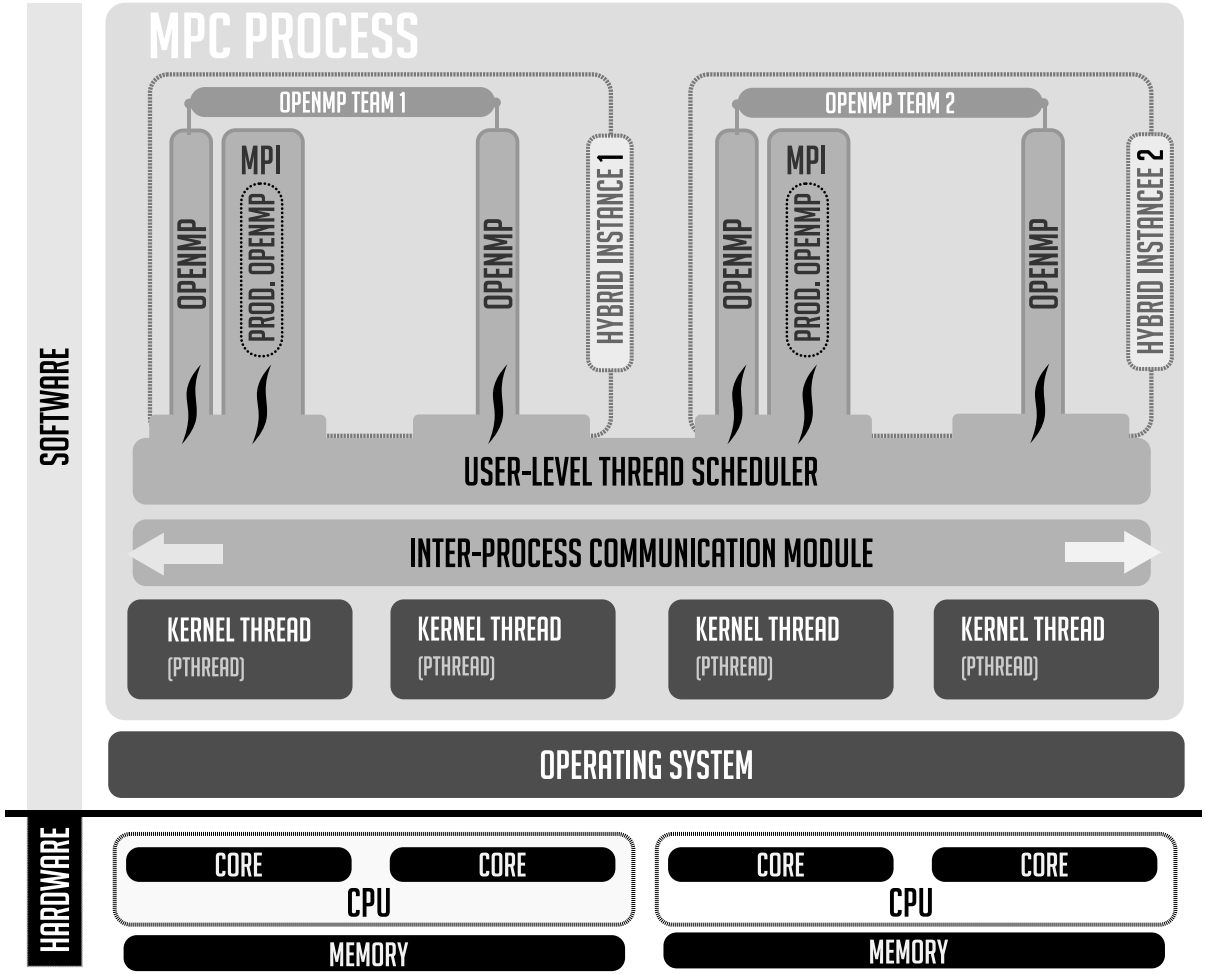
Placement Experiments

- Experiments of running LULESH benchmark on Intel Xeon Phi KNL
  - W/ and w/ binding for MPI (from SLURM)
- Results in time (lower is better)



Automatic Placement in MPC

- MPI/OpenMP integration
  - Automatic MPI task placement on the node
  - Automatic OpenMP thread placement
    - Topology inheritance
- Example
  - Node with 2 CPUs
  - 2 cores per CPU
  - 2 MPI tasks per node
  - Default: 2 OpenMP threads per team



Hybrid Programming

- Project

Hybrid Project

- Main goal
  - Exploit available parallelism with MPI and OpenMP programming models
- Main steps
  - Start with MPI
    - Prepare a mode with only 1 MPI task
  - Add OpenMP
    - Driven by profiling
    - Be careful of best practices for OpenMP
- Cost model
  - According to inputs
    - Input set, number of MPI tasks, number of threads (either default or set through environment variable
  - Enable best parallelism

Hybrid Programming

- Conclusion

Conclusion

- Hybrid programming model
  - Mix of MPI and OpenMP inside the same application
  - Compilation through MPI script and OpenMP option (to activate OpenMP support)
  - Depending on granularity: may require an MPI implementation supporting thread (different initialization)

- Be careful about granularity and placement when designing and launching an hybrid program

Best Practices of Runtime Stacking

- Focused on one specific runtime stacking
  - MPI as lower layer
  - OpenMP as upper layer

- Best Practices for Runtime Stacking
  - Consider arbitrary nested parallelism
    - Composition of application and parallel libraries
    - Composition of multiple solvers in simulation codes
  - Best practices for lower layer
    - Thread support
    - Generation of restricted topology
  - Best practices for upper layer
    - Consideration of restricted hierarchical topology
    - Busy waiting removal
    - Optimization of raw performance