

Hybrid parallel programming

Simon Scheidegger
simon.scheidegger@gmail.com

August 3rd, 2017

Open Source Macroeconomics Laboratory – BFI/UChicago

Including adapted teaching material from books, lectures and presentations by
B. Barney, B. Cumming, W. Gropp, G. Hager, M. Martinasso, R. Rabenseifner, O. Schenk, G. Wellein

Day 4, Thursday – Aug 3rd

1. Hybrid parallelism (8.00-9.00 – hands on).
2. Hybridize some of the projects together (9.15-10.00).
3. Advanced topics (10.15-11.00).
4. Start to present results from the projects (11.10 – 11.50).
5. Exercise sheet related to the day's topic (11.50-12.00 – hands on).

Outline

Hybrid parallelism in general

- Recap hardware & programming models
- Merging OpenMP & MPI
- “Hello World” in hybrid

Putting things together:

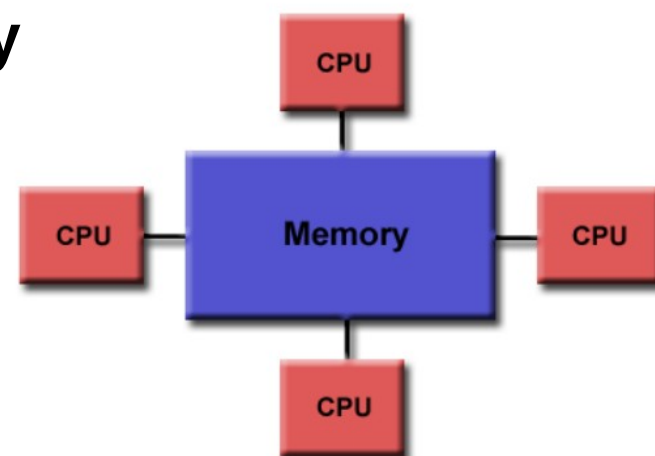
- Time Iteration, Adaptive Sparse Grids, HPC

Shared memory systems – OpenMP

- Process can access same GLOBAL memory

- **Uniform Memory Access (UMA)** model

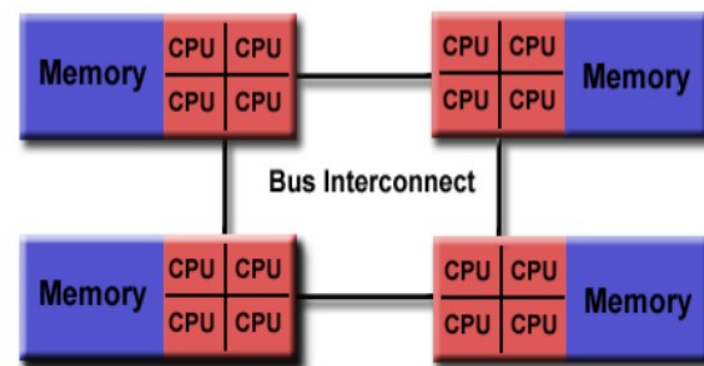
- Access time to memory is uniform.
- Local cache, all other peripherals are shared.



Shared Memory (UMA)

- **Non-Uniform Memory Access (NUMA)** model

- Memory is physically distributed among processors.
- Global virtual address spaces accessible from all processors.
- Access time to local and remote data is different.

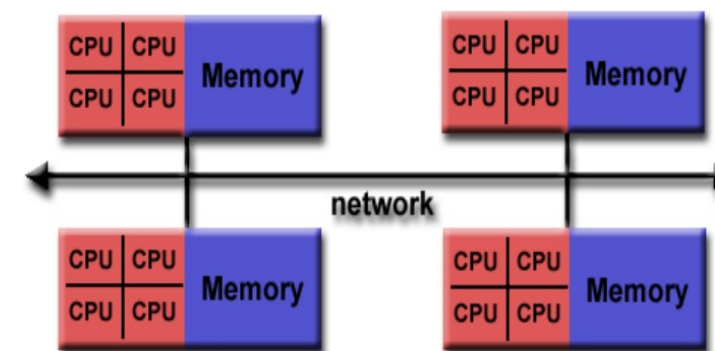
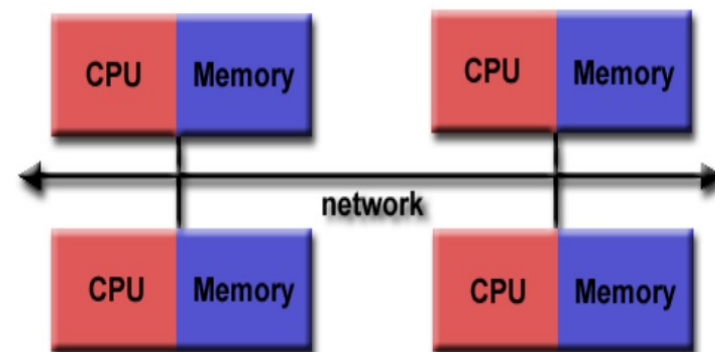


Shared Memory (NUMA)

→ **OpenMP**, but other solutions available (e.g. Intel's TBB).

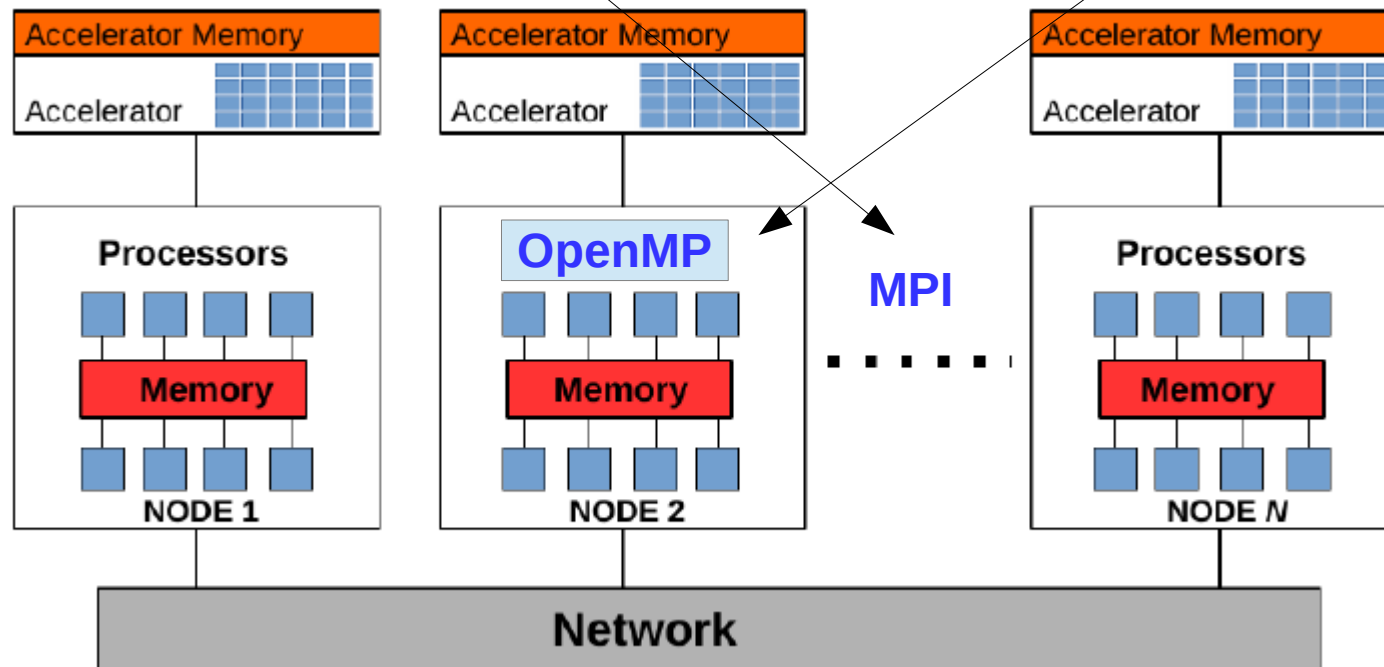
Distributed-memory parallel programming – MPI

- We need to use explicit message passing, i.e., communication between processes:
 - Most tedious and complicated but also the most flexible parallelization method.
 - Message passing is required if a parallel computer is of **distributed-memory** type, i.e., if **there is no way for one processor to directly access the address space of another**.
 - However, it can also be regarded as a programming model and used on shared-memory or **hybrid systems** as well.
- **Message Passing Interface (MPI)**.



Today's HPC systems

- Efficient programming of clusters of **shared memory (SMP)** nodes
- Hierarchical system layout
- Hybrid programming seems natural:
 - **MPI among the nodes.**
 - **Shared memory programming inside of each SMP node – OpenMP.**



Hybrid parallelism with MPI and OpenMP

When Does Hybridization Make Sense?

- When one wants to scale a shared memory OpenMP application for use on **multiple SMP nodes** in a cluster.
- When one wants to **reduce an MPI application's sensitivity to becoming communication bound**.
- **When one is designing a parallel program (nowadays) from the very beginning.**
- for 8/16/32/64/...ranks per multi-core node, this can have scaling problems with many nodes/MPI ranks.

Hybridization Using MPI and OpenMP

- facilitates cooperative shared memory (OpenMP) programming across clustered SMP nodes.
- MPI facilitates **communication** among SMP nodes.
- OpenMP manages the **workload** on each SMP node.
- **MPI and OpenMP are used in tandem** to manage the overall concurrency of the application.

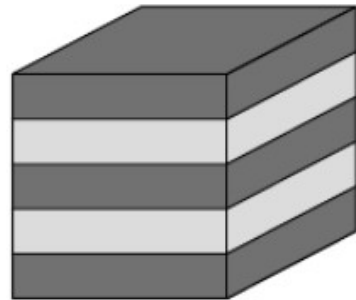
The hybrid MPI & OpenMP model

The MPI only model assigns **one process per core**:

- for 8/16/32/64/...ranks per multi-core node, this can have scaling problems with many nodes/MPI ranks.
- **the amount of data passed around in messages increases as number of ranks increases.**
- to take advantage of shared cache and DRAM on a socket, why not use threads on the socket/node, and pass messages between sockets/nodes? (it's a lot faster than sending messages around)

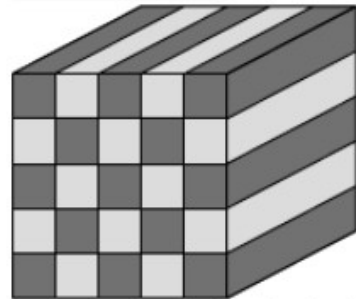
The hybrid MPI & OpenMP model has light-weight threads that share on-node memory.

Domain decomposition



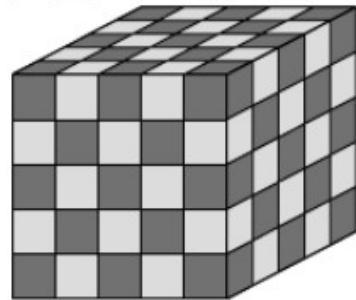
“Slabs”

$$\begin{aligned} c_{1d}(L, N) &= L \cdot L \cdot w \cdot 2 \\ &= 2wL^2 \end{aligned}$$



“Poles”

$$\begin{aligned} c_{2d}(L, N) &= L \cdot \frac{L}{\sqrt{N}} \cdot w \cdot (2 + 2) \\ &= 4wL^2N^{-1/2} \end{aligned}$$



“Cubes”

$$\begin{aligned} c_{3d}(L, N) &= \frac{L}{\sqrt[3]{N}} \cdot \frac{L}{\sqrt[3]{N}} \cdot w \cdot (2 + 2 + 2) \\ &= 6wL^2N^{-2/3} \end{aligned}$$

Figure 10.9: 3D domain decomposition of a cubic domain of size L^3 (strong scaling) and periodic boundary conditions: Per-process communication volume $c(L, N)$ for a single-site data volume w (in bytes) on N processes when cutting in one (top), two (middle), or all three (bottom) dimensions.

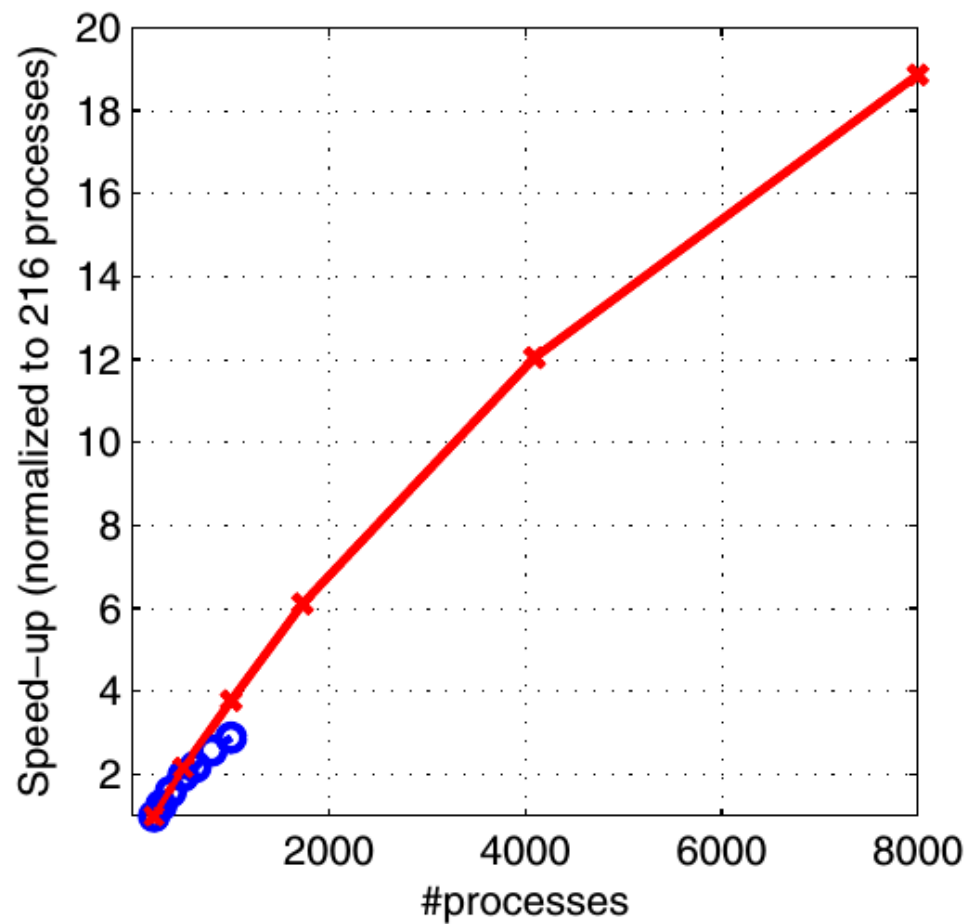
Example: Scaling “MPI” vs. “Hybrid”

See, e.g., Käppeli et al. (2011) <http://iopscience.iop.org/article/10.1088/0067-0049/195/2/20/pdf>

Hybrid parallelization:

- Reduces the amount of memory consumption.
- Reduces the amount of communication required.

Speed-up



Hybrid parallelism with MPI and OpenMP

Recall MPI

- Provides a familiar and explicit **means to use message passing on distributed memory clusters**.
- Has implementations on many architectures and topologies.
- Is the **de-facto standard** for distributed memory communications.
- Requires that program state synchronization must be handled explicitly due to the nature of distributed memory.
- data goes to the process.
- program correctness is an issue, but not big compared to those inherent to OpenMP.

OpenMP

- Allows for **implicit intra-node communication**, which is a shared memory paradigm.
- Provides for **efficient utilization of shared memory SMP systems**.
- Facilitates **relatively easy threaded programming**.
- Does not incur the overhead of message passing, since communication among threads is implicit.
- Is the de-facto standard, and is supported by most major compilers (Intel, IBM, gcc, etc).
- The process goes to the **data program correctness is an issue** since all threads can update shared memory locations.

The best from both worlds

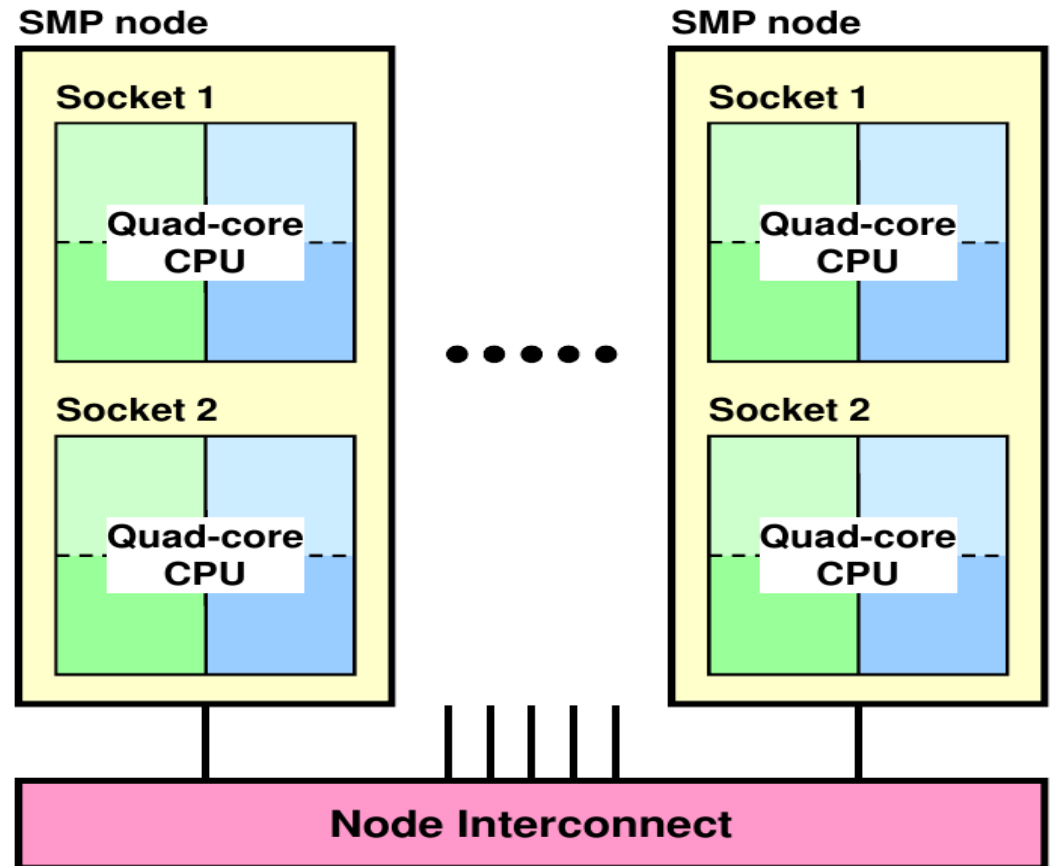
- MPI allows for **inter-node communication**.
- MPI facilitates efficient inter-node reductions and **sending of complex data structures**.
- Program synchronization is explicit.
- A Common execution scenario: a single MPI process is launched on each SMP node in the cluster.
- **Each process spawns N threads on each SMP node.**
- At some global sync point, the master thread on each SMP communicate with one another.
- **The threads belonging to each process continue until another sync point or completion.**

Memory consumption & mapping

- Memory consumption MPI & OpenMP with n threads per MPI process:
 - **Duplicated data may be reduced by factor n .**
- How many threads per MPI process?
SMP node = with m sockets (NUMA domains) and n cores/socket
- **How many threads (i.e., cores) per MPI process?**
 - Too few threads, too much memory consumption
- Optimum:
 - somewhere between 1 and $m \times n$ threads per MPI process.
- Typical optima:
 - 1 MPI process per socket.
 - 2 MPI processes per socket.
 - Seldom: 1 MPI process per whole SMP node.

A node

- Which programming model is fastest?
- MPI everywhere?
- Fully hybrid MPI & OpenMP?
- Something between? (Mixed model)



Mapping (2)

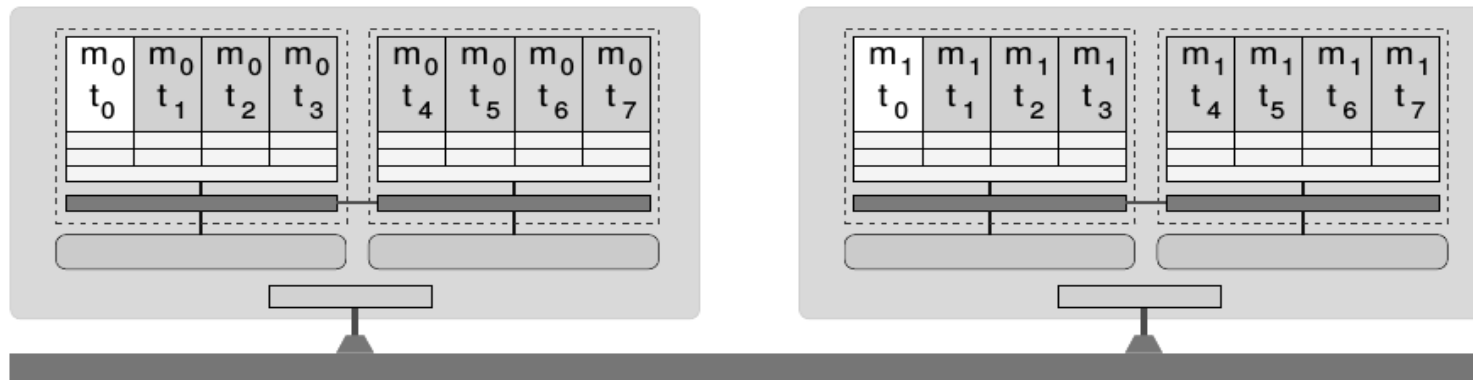


Figure 11.3: Mapping a single MPI process with eight threads to each node.

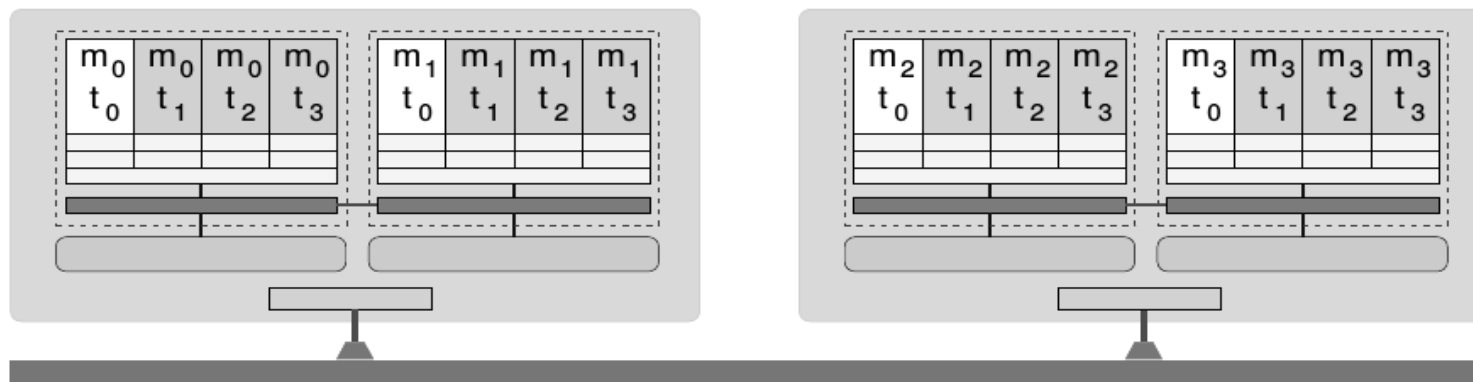


Figure 11.4: Mapping a single MPI process with four threads to each socket (L3 group or locality domain).

Mapping (3)

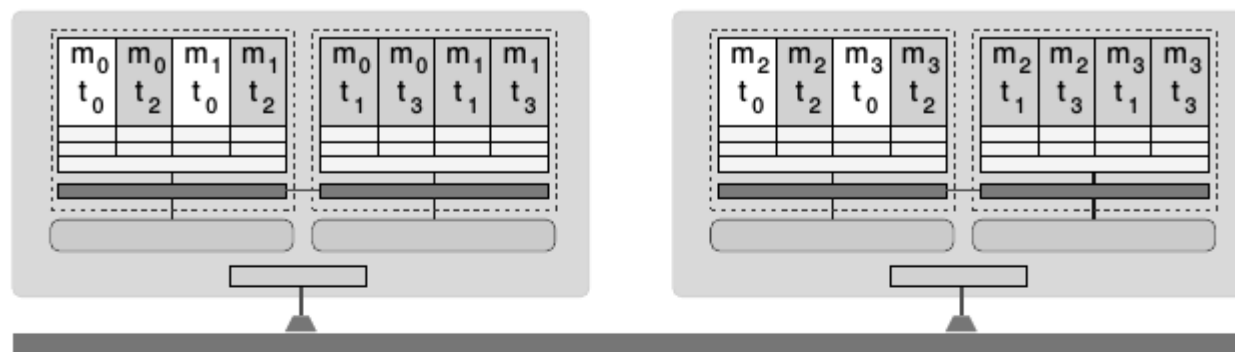


Figure 11.5: Mapping two MPI processes to each node and implementing a round-robin thread distribution.

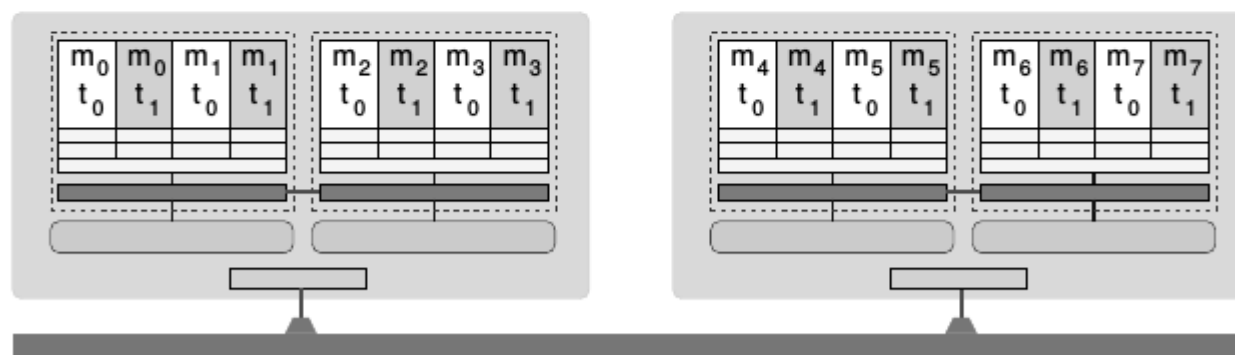


Figure 11.6: Mapping two MPI processes with two threads each to a single socket.

Opportunities

Algorithmic opportunities due to larger physical domains inside of each MPI process:

- If MPI domain decomposition is based on physical zones:
Nested Parallelism Outer loop with MPI / inner loop with OpenMP.
- **Load-Balancing**: Using OpenMP dynamic and guided work sharing
- **Memory consumption**: Significantly reduction of replicated data on MPI level.
- **Reduced MPI scaling problems**: Significantly reduced number of MPI processes
- Opportunities, if MPI speed-up is limited due to algorithmic problem
- **Significantly reduced number of MPI processes.**

Basic MPI/OpenMP programming models

- The basic idea of a hybrid OpenMP/MPI programming model is to **allow any MPI process to spawn a team of OpenMP threads** in the same way as the master thread does in a pure OpenMP program.
- **Inserting OpenMP compiler directives into an existing MPI code** is a straightforward way to build a first hybrid parallel program.
- Following the guidelines of good OpenMP programming, **compute intensive loop constructs are the primary targets for OpenMP parallelization in a naive hybrid code.**
- **Before launching the MPI processes one has to specify the maximum number of OpenMP threads per MPI process in the same way as for a pure OpenMP program.**
- At execution time each MPI process activates a team of threads (being the master thread itself) whenever it encounters an OpenMP parallel region.
- **There is no automatic synchronization between the MPI processes for switching from pure MPI to hybrid execution**, i.e., at a given time some MPI processes may run in completely different OpenMP parallel regions, while other processes are in a pure MPI part of the program.
- Synchronization between MPI processes is still restricted to the use of appropriate MPI calls.

A common way to implement hybrid parallelism

Fortran

```
include 'mpif.h'
program hybsimp

call MPI_Init(ierr)
call MPI_Comm_rank (... , irank, ierr)
call MPI_Comm_size (... , isize, ierr)
! Setup shared mem, comp. & Comm

!$OMP parallel do
  do i=1,n
    <work>
  enddo
! compute & communicate

call MPI_Finalize(ierr)
end
```

C/C++

```
#include <mpi.h>
int main(int argc, char **argv){
  int rank, size, ierr, i;

  ierr= MPI_Init(&argc,&argv[]);
  ierr= MPI_Comm_rank (...,&rank);
  ierr= MPI_Comm_size (...,&size);
  //Setup shared mem, compute & Comm

  #pragma omp parallel for
  for(i=0; i<n; i++){
    <work>
  }
  // compute & communicate

  ierr= MPI_Finalize();
}
```

Example: "hello world hybrid"

1. go to OSM_Lab/HPC_day4/code_day4/hybrid:

```
> cd OSM_Lab/HPC_day4/code_day4/hybrid
```

2. Have a look at the code

```
> vi 1a.hello_world_hybrid.cpp
```

3. compile by typing:

```
> make
```

4. Experiment with different numbers of threads/MPI Processes

```
> export OMP_NUM_THREADS=4
```

```
> mpirun -np 2 ./1a.hello_world_hybrid.exec
```

Example

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

```
int main(int argc, char *argv[]) {
    int numprocs, rank, namelen;
    int iam = 0, np = 1;
```

```
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
    #pragma omp parallel default(shared) private(iam, np)
```

```
    {
        np = omp_get_num_threads();
        iam = omp_get_thread_num();
        printf("Hello from thread %d out of %d from process %d out of %d\n",
               iam, np, rank, numprocs);
    }
```

```
    MPI_Finalize();
```

```
}
```

MPI

OpenMP

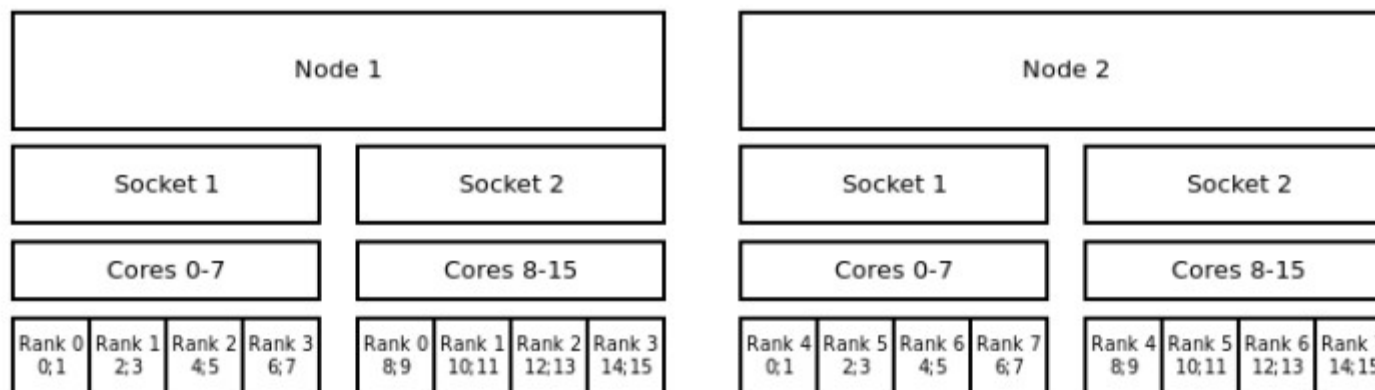
Slurm with Hybrid Jobs

In the illustration below the default binding of a Hybrid-job is shown. In which 8 global ranks are distributed onto 2 nodes with 16 cores each. Each rank has 4 cores assigned to it.

```
#!/bin/bash
#SBATCH --nodes=2
#SBATCH --tasks-per-node=4
#SBATCH --cpus-per-task=4

export OMP_NUM_THREADS=4

srun --ntasks 8 --cpus-per-task $OMP_NUM_THREADS ./application
```



Example 2: Slurm on Midway

1. go to OSM_Lab/HPC_day4/code_day4/hybrid:

```
> cd OSM_Lab/HPC_day4/code_day4/hybrid
```

2. Have a look at the code

```
> vi submit_hybrid_midway.sh
```

3. compile by typing:

```
> make
```

4. Experiment with different numbers of nodes/threads/MPI Processes and look at the output.

```
> sbatch submit_hybrid_midway.sh
```

Slurm – Hybrid

```
#!/bin/bash
# a sample job submission script to submit a hybrid MPI/OpenMP job to the sandyb
# partition on Midway1 please change the --partition option if you want to use
# another partition on Midway1

# set the job name to hello-hybrid
#SBATCH --job-name=hello-hybrid

# send output to hello-hybrid.out
#SBATCH --output=hello-hybrid.out

# this job requests 4 MPI processes
#SBATCH --ntasks=4

# and request 8 cpus per task for OpenMP threads
#SBATCH --cpus-per-task=8

# this job will run in the sandyb partition on Midway1
#SBATCH --partition=sandyb

# load the openmpi default module
module load openmpi

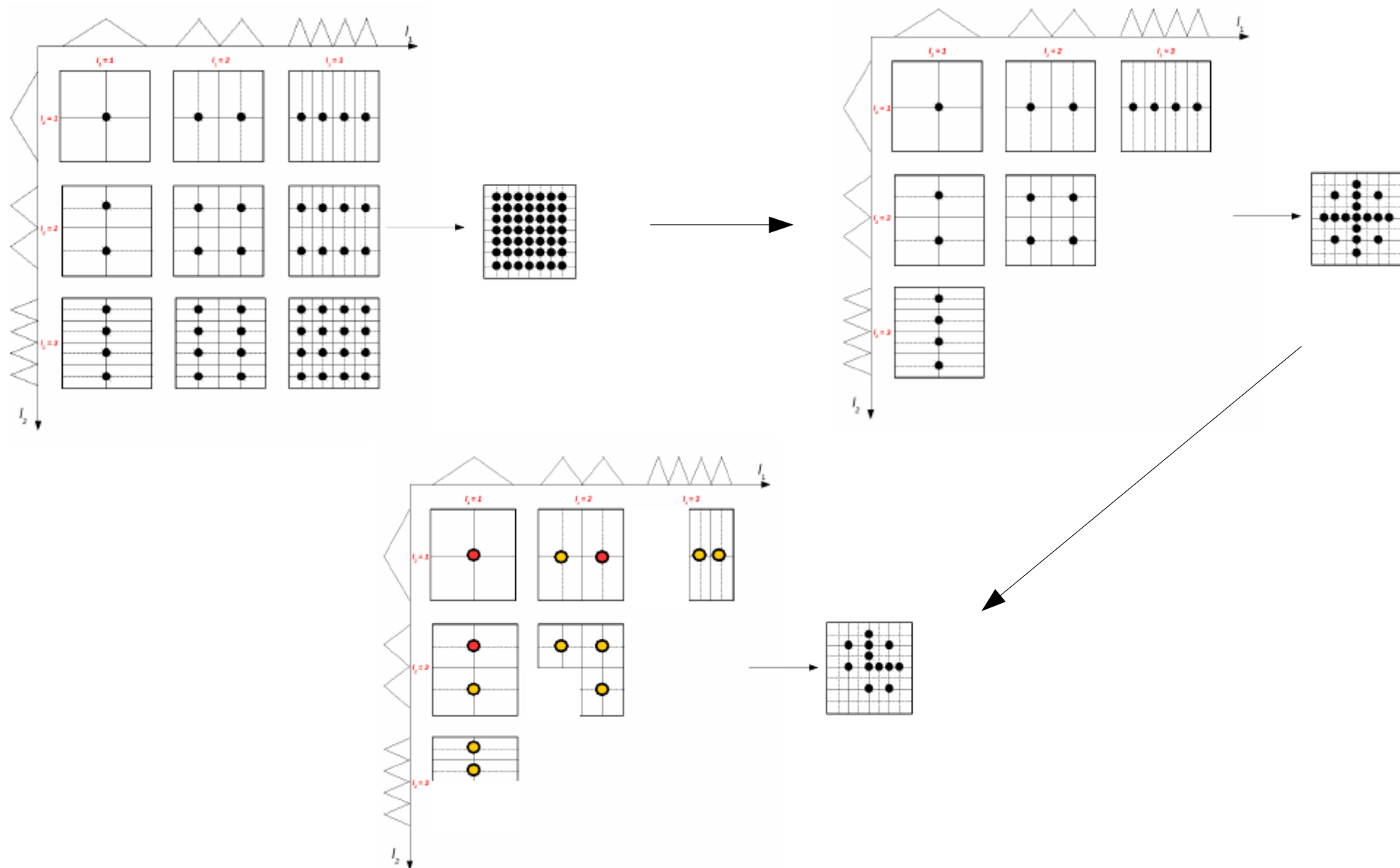
# set OMP_NUM_THREADS to the number of --cpus-per-task we asked for
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK

# Run the process with mpirun. Notice -n is not required. mpirun will
# automatically figure out how many processes to run from the slurm options
mpirun ./1a.hello_world_hybrid.exed
```


Putting things together: Time Iteration, Adaptive Sparse Grids, HPC



Hybrid parallelism in Sparse grids



Algorithm for time iteration & SG

Scheidegger et al. (2017)

Data: Initial guess $p_{next} = (p_{next}(1), \dots, p_{next}(N_s = 16))$ for next period's policy function. Approximation accuracy $\bar{\eta}$. Maximal refinement level L_{max} . Starting refinement level $L_0 \leq L_{max}$. Refinement threshold ϵ .

Result: The (approximate) equilibrium policy function $p = (p(1), \dots, p(N_s = 16))$.

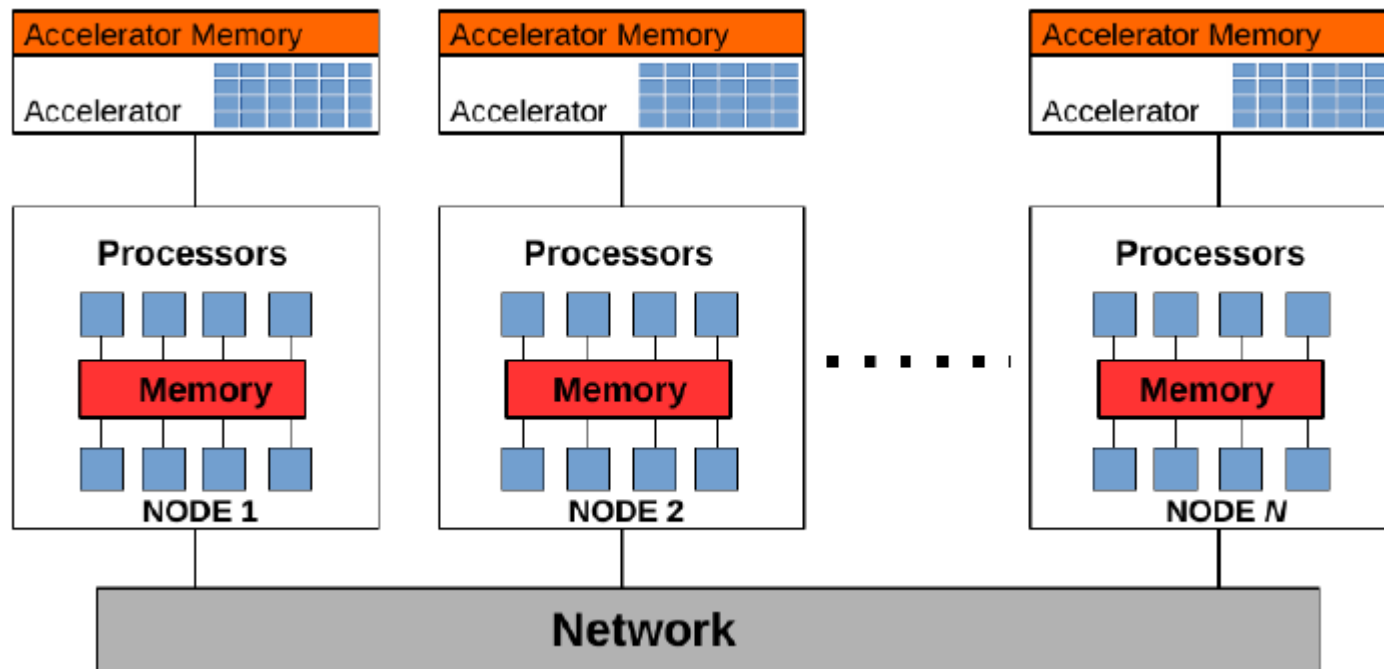
```

while  $\eta > \bar{\eta}$  do
  Set  $z = 1$ .
  for  $z \leq N_s$  do
    Set  $l = 1$ , set  $G(z) \subset S(z)$  to be the level 1 grid on  $S(z)$ , and set  $G_{old}(z) = \emptyset, G_{new}(z) = \emptyset$ .
    while  $G(z) \neq G_{old}(z)$  do
      for  $g(z) \in G(z) \setminus G_{old}(z)$  do
        Compute the optimal policies  $p(g(z))$  by evaluating (5) to (15) given next period's policy  $p_{next}$ .
        Define the policy  $\tilde{p}(g(z))$  by interpolating  $\{p(g(z))\}_{g(z) \in G_{old}(z)}$ .
        if ( $l < L_{max}$  and  $\|p(g(z)) - \tilde{p}(g(z))\|_\infty > \epsilon$ ) or  $l < L_0$ , then
          | Add the neighboring points (sons) of  $g(z)$  to  $G_{new}(z)$ .
        end
      end
      Set  $G_{old}(z) = G(z)$ , set  $G = G_{old}(z) \cup G_{new}(z)$ , set  $G_{new}(z) = \emptyset$ , and set  $l = l + 1$ .
    end
    Define the policy function  $p(z)$  as the sparse grid interpolation of  $\{p(g(z))\}_{g(z) \in G(z)}$ .
    Calculate (an approximation for) the error within a state:
     $\eta(z) = \|p(z) - p_{next}(z)\|$ . Set  $p_{next}(z) = p(z)$ .
    set  $z = z + 1$ .
  end
  error:  $\eta = \max(\eta(1), \dots, \eta(N_s))$ 
end

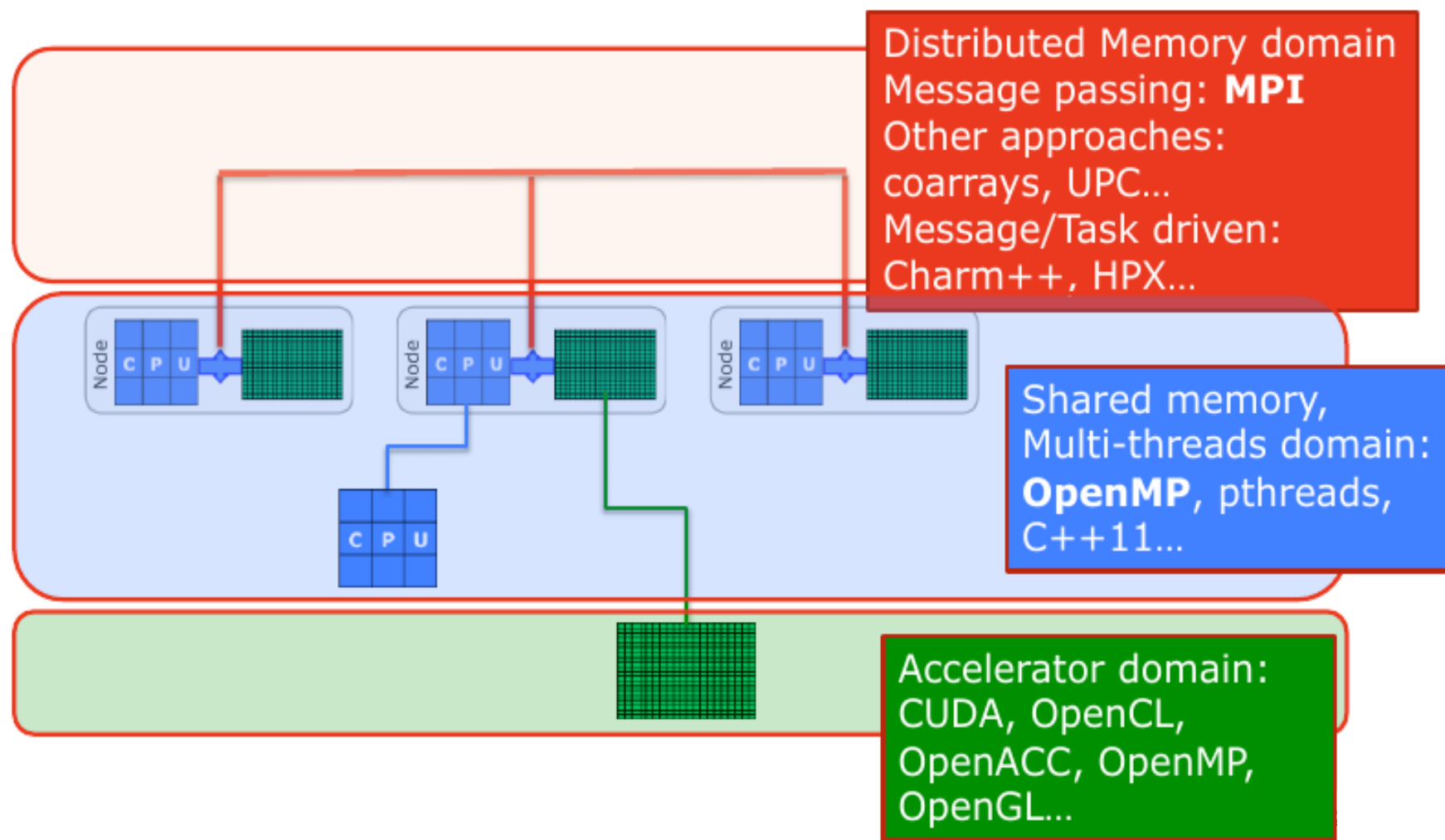
```

Algorithm 1: Overview of the crucial steps of the time iteration algorithm.

Recall – Today's HPC systems



Overall picture of programming models



Parallel time iteration/DP algorithm

Brumm et al. (2015), Brumm & Scheidegger (2017)

-Our implementation:

Hybrid parallel

(MPI & Intel TBB & GPU (CUDA/THRUST)).

-newly generated points are distributed via MPI

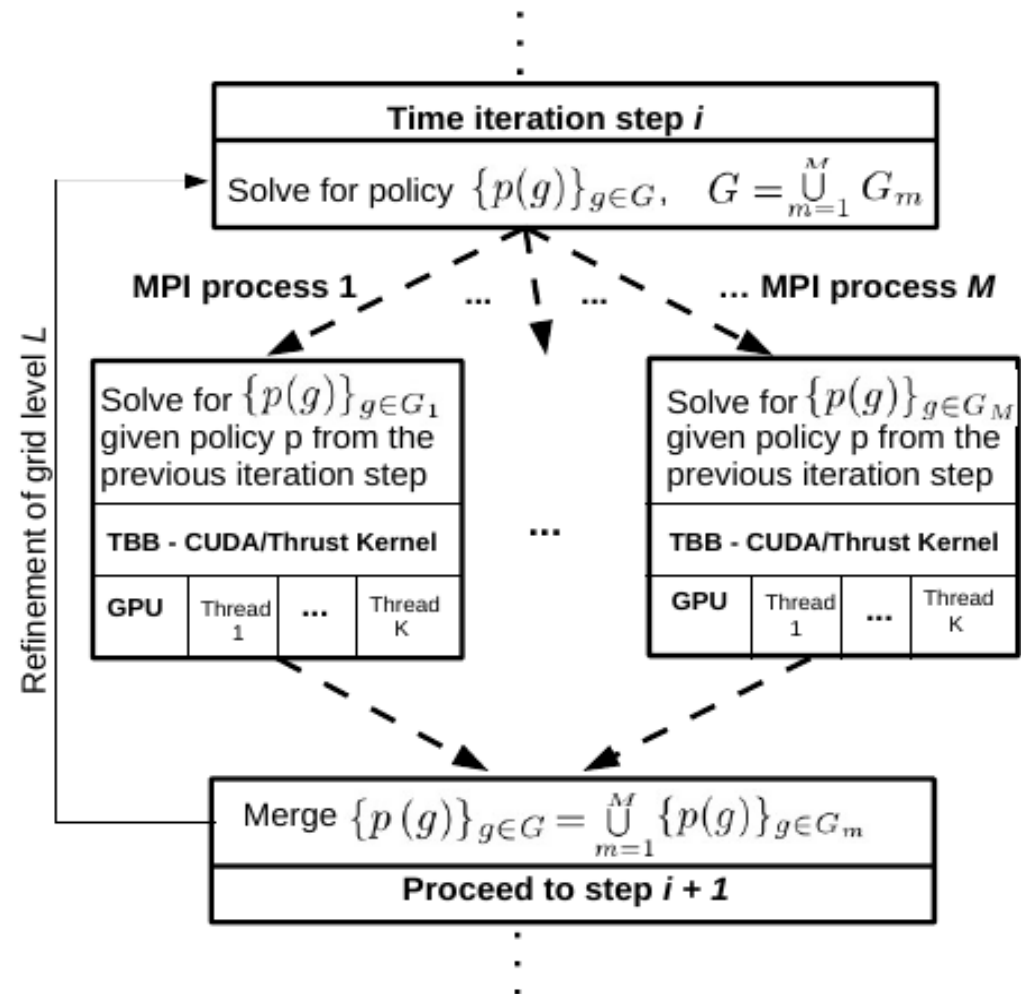
**Solve optimizations/
nonlinear equations locally**

(e.g. IPOPT (Wächter & Biegler (2006))).

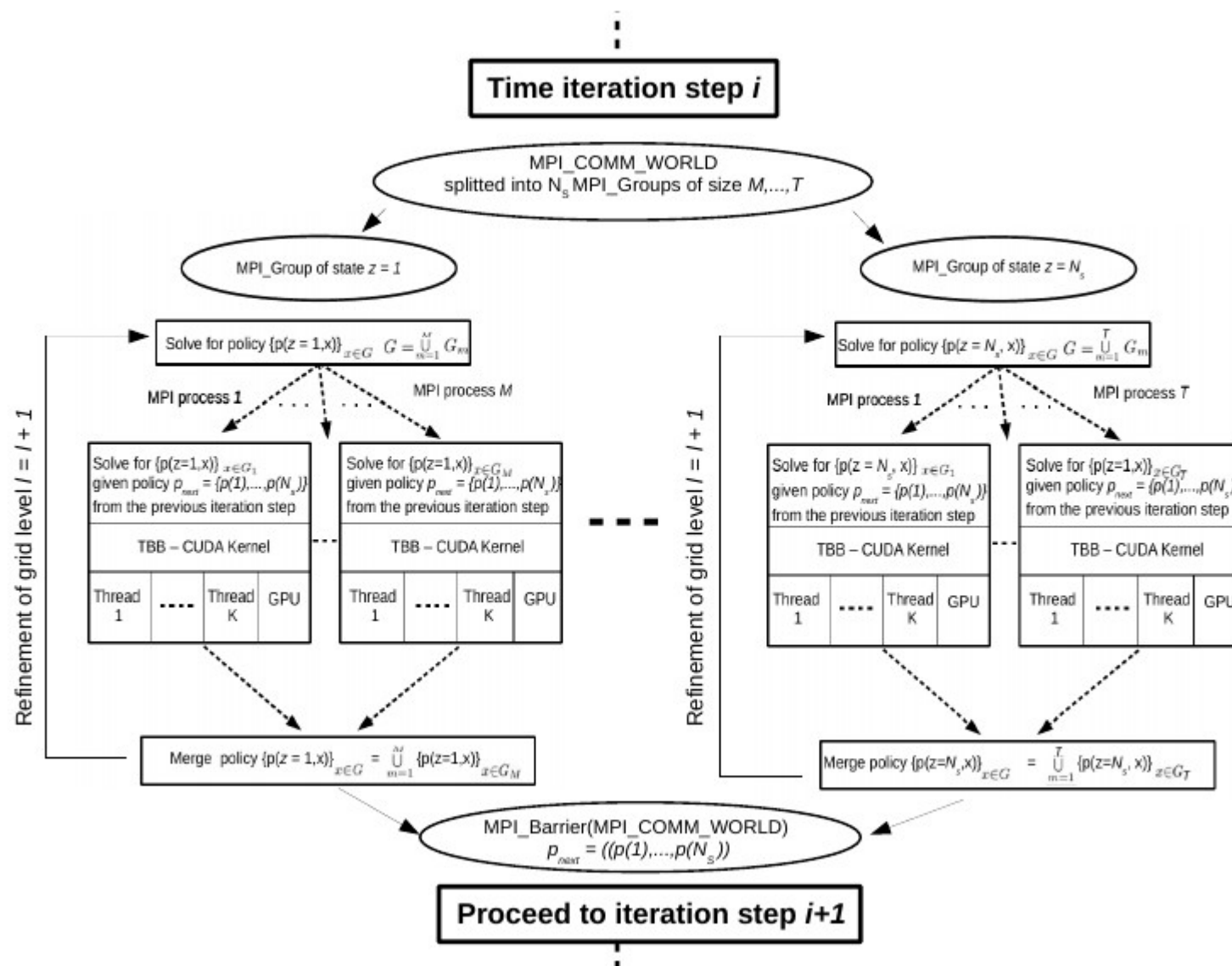
In parallel: 'messy' !

- policy from previous iteration **visible on all MPI processes**.
- we have to ensure some sort of 'load balancing'.
- **Now a lot better with TBB**

One single time-step

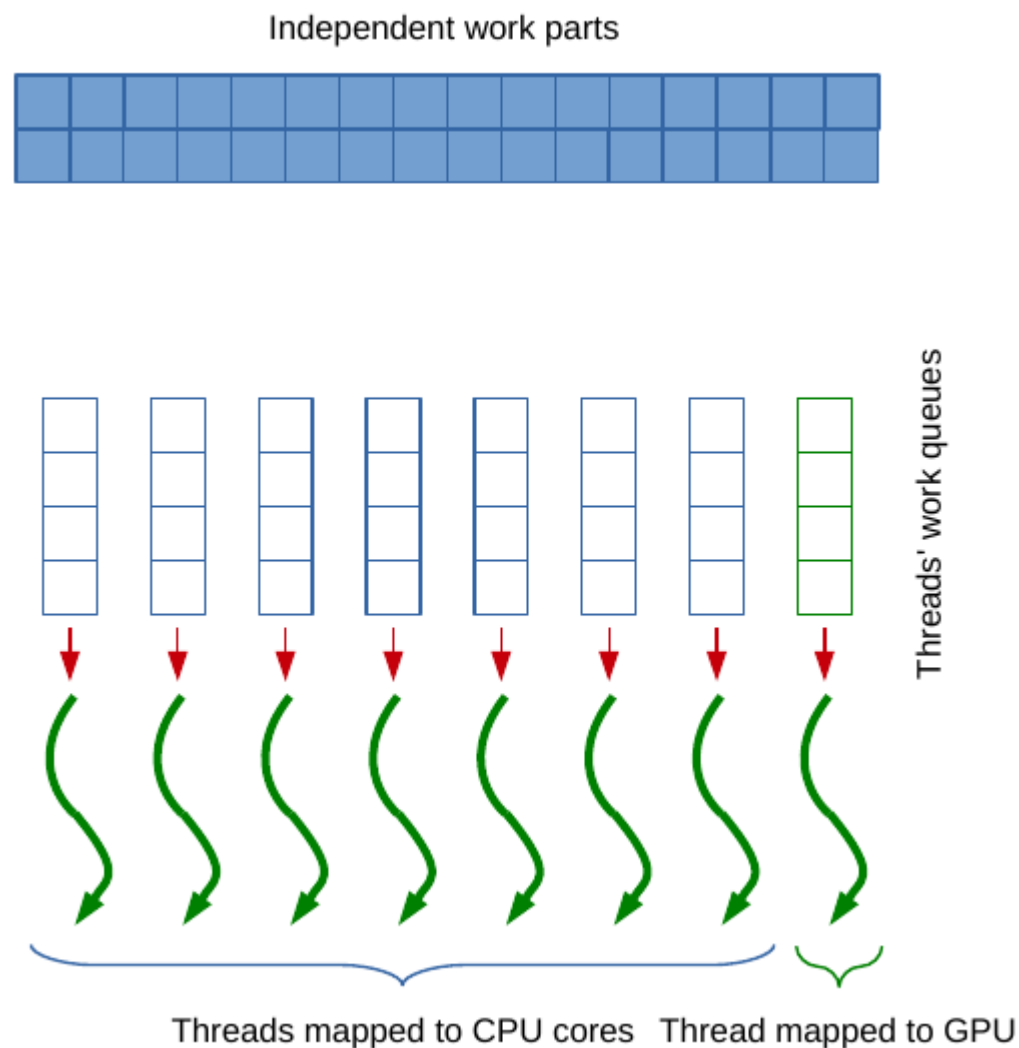


With discrete states



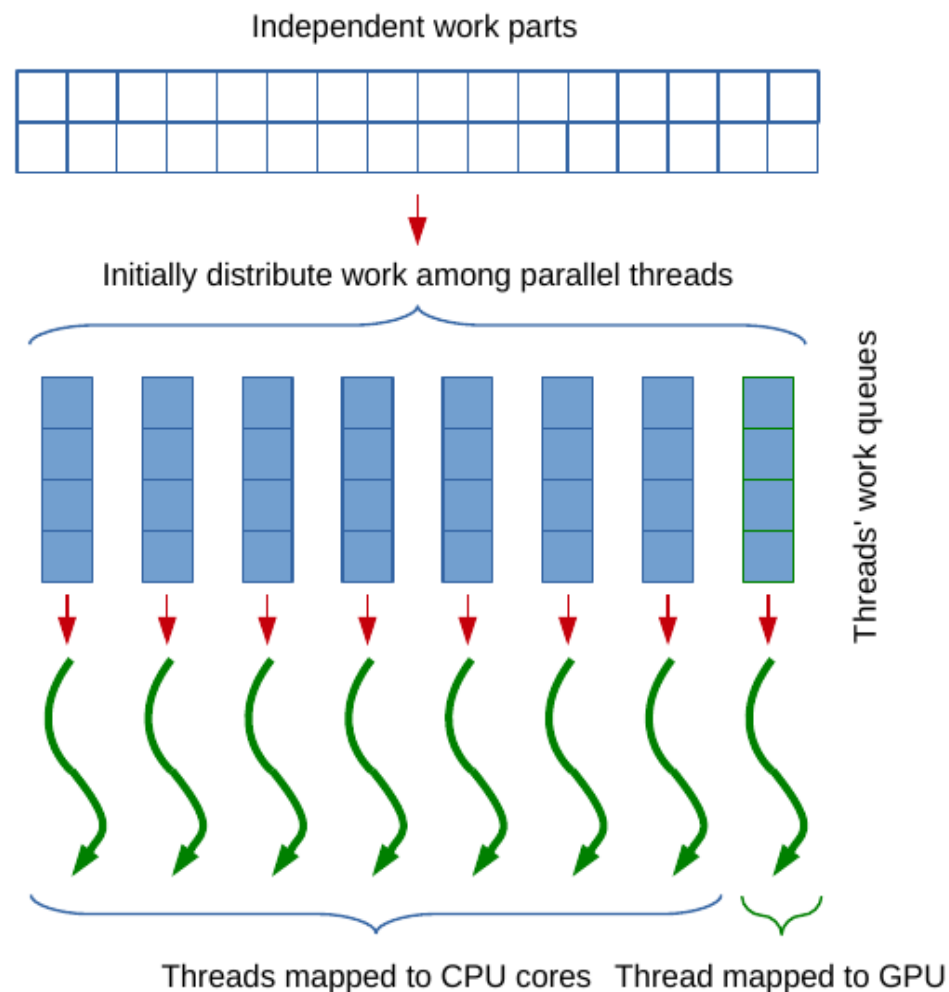
Intel® Threading Building Blocks (TBB)

-TBB maps different threads, similar to OpenMP.



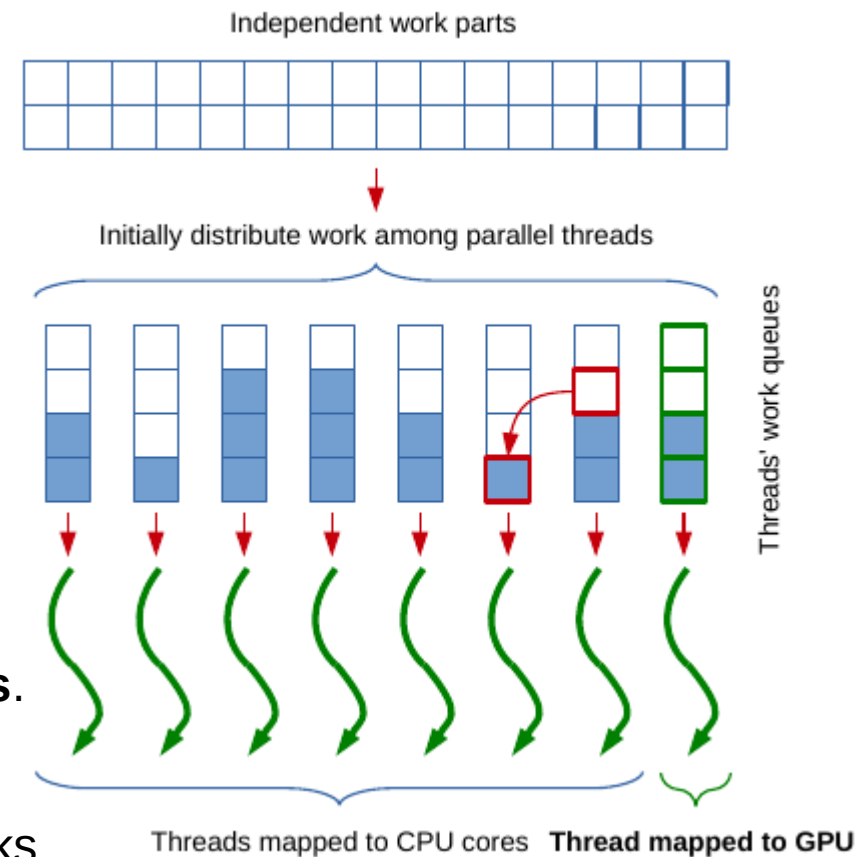
Intel® Threading Building Blocks (TBB)

- TBB maps different threads, similar to OpenMP.
- Every thread is initially assigned an equal logical queue of tasks.

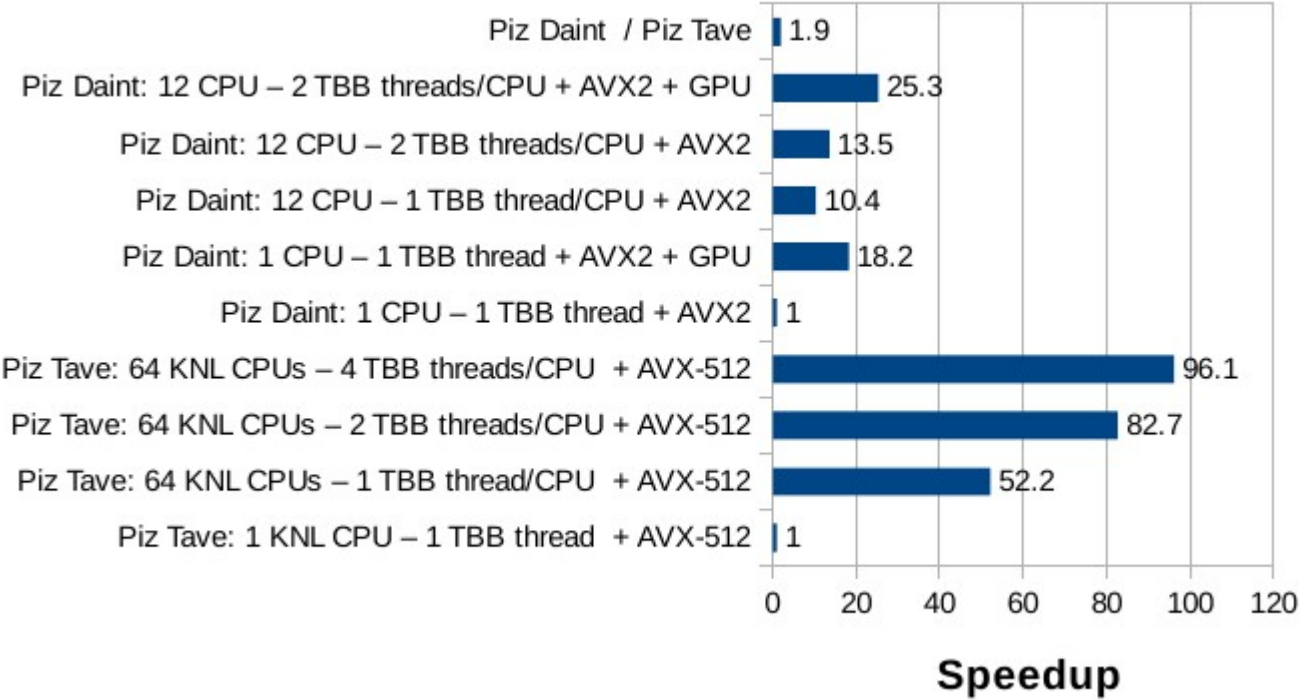


Intel® Threading Building Blocks (TBB)

- TBB maps different threads, similar to OpenMP.
- Every thread is initially assigned an equal logical queue of tasks.
- However, different tasks may be processed faster or slower, due to differences between tasks and/or compute cores
- TBB approach to work balancing: once one thread runs out of tasks, **“steal” a task from another thread, which makes slower progress.**
- We map one extra thread onto **GPU**
→ **CPU** cores and **GPU** process interpolation tasks together.

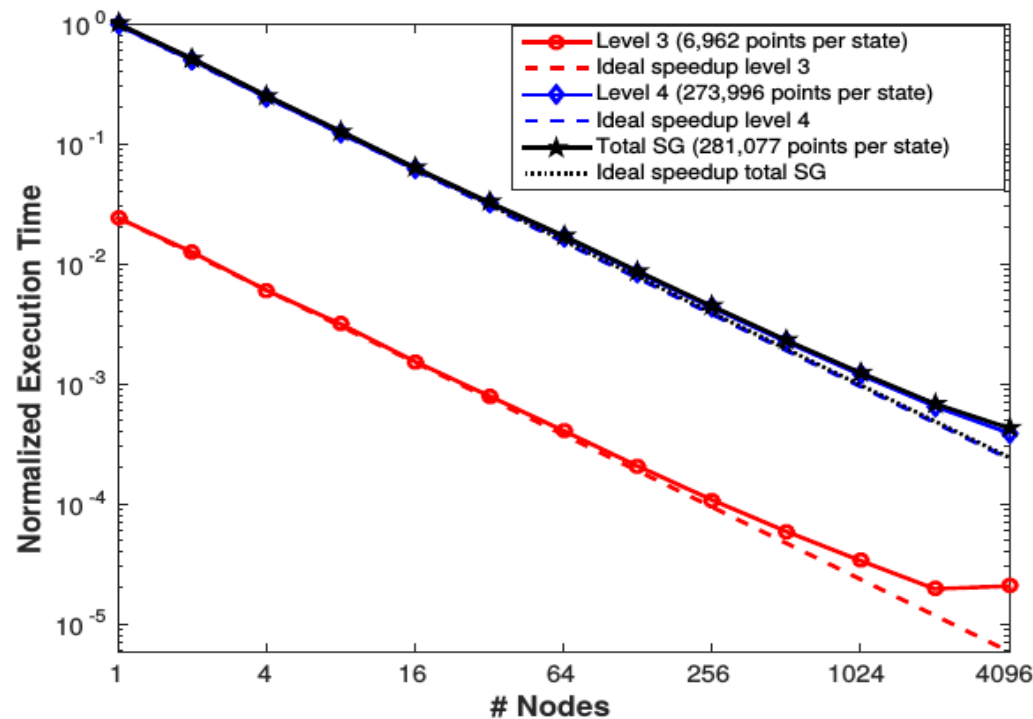


Strong scaling – 1 node



Strong scaling – intra-node

Scheidegger et al. (2017)



- Annually calibrated OLG model.
- 16 discrete states (stochastic tax rates on labor and capital).
- solve this model in few hours.

Figure 6: Strong scaling on Piz Daint for an OLG model using 4 levels of grid refinements, 16 discrete states, and $16 \cdot 281,077 = 4,497,232$ points and 265,336,688 unknowns in total. “Total SG” shows the entire, normalized simulation time up to 4,096 nodes. We also show normalized execution times for the computational sub-components on different levels, e.g., for level 3 using 6.962 points. Dashed and dotted lines show the ideal speedup.

Time to solution

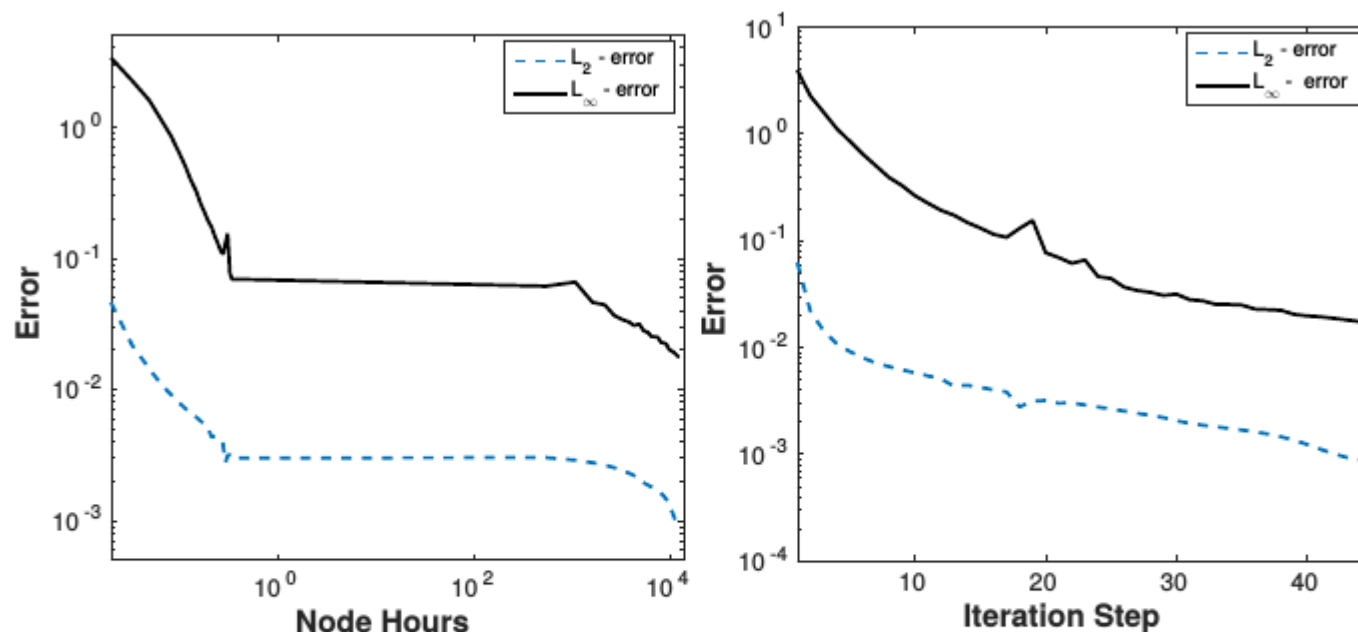


Figure 7: Comparison of the L_2 and L_∞ -error for adaptive sparse grid solutions of the 59-dimensional OLG model as a function of compute time or iterations spent on Piz Daint.

Questions?

?