

Advanced parallel programming – MPI+X

MPI + OpenMP + OpenMP offloading

Claudia Blas-Schenner and Ivan Vialov

VSC Research Center, TU Wien, Vienna, Austria

TREX Workshop: Code Tuning for the Exacale @ Bratislava, June 5, 2023

Abstract

TREX Workshop: Code Tuning for the Exascale **Slovak Academy of Sciences, Bratislava, Slovakia** **Day 1 - 05.06.2023**

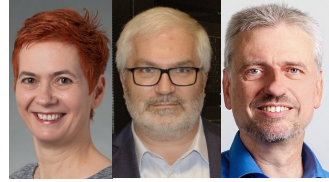
Claudia Blaas-Schenner and Ivan Vialov (VSC Research Center, TU Wien, Vienna, Austria)

Advanced parallel programming – MPI+X: Modern HPC systems are clusters of shared-memory nodes and especially the pre-exascale and exascale systems are accelerated with one to several GPUs per node. While the Message Passing Interface (MPI) is the dominant model to parallelize across nodes, there is a need to combine MPI with other programming paradigms such as OpenMP to fully exploit shared-memory within the nodes and to be able to offload heavy compute task to the GPUs.

In this one day tutorial, we will briefly cover MPI+OpenMP+OpenMP offloading.

We will explain how to properly tackle NUMA (non-uniform memory access) architectures and put a special focus on pinning. In the hands-on labs we will play around with affinity and the participants will get a good grasp about how pinning influences performance.

<https://trex-coe.eu/events/trex-workshop-code-tuning-exascale>



Hybrid Programming in HPC – MPI+X

Claudia Blas-Schenner¹⁾

claudia.blaas-schenner@tuwien.ac.at

Georg Hager²⁾

georg.hager@fau.de

Rolf Rabenseifner³⁾

rabenseifner@hls.de

¹⁾ VSC Research Center, TU Wien, Vienna, Austria (hands-on labs)

²⁾ Erlangen National High Performance Computing Center (NHR@FAU), FAU, Germany

³⁾ High Performance Computing Center (HLRS), University of Stuttgart, Germany

PTC ONLINE COURSE @ VSC Vienna, Dec 12-14, 2022

<http://tiny.cc/MPIX-VSC>

<https://doi.org/10.5281/zenodo.7566873>

General outline

Introduction

Programming Models

- MPI + OpenMP on multi/many-core [\(14\)](#) + Exercises
- MPI + Accelerators [\(88\)](#) + Exercises

Introduction

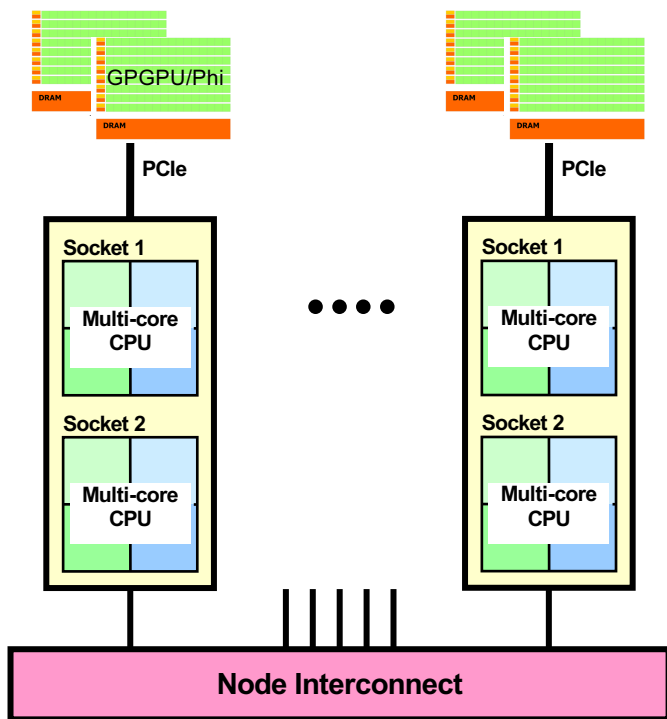
Hardware and programming models

Hardware Bottlenecks

Questions addressed in this tutorial

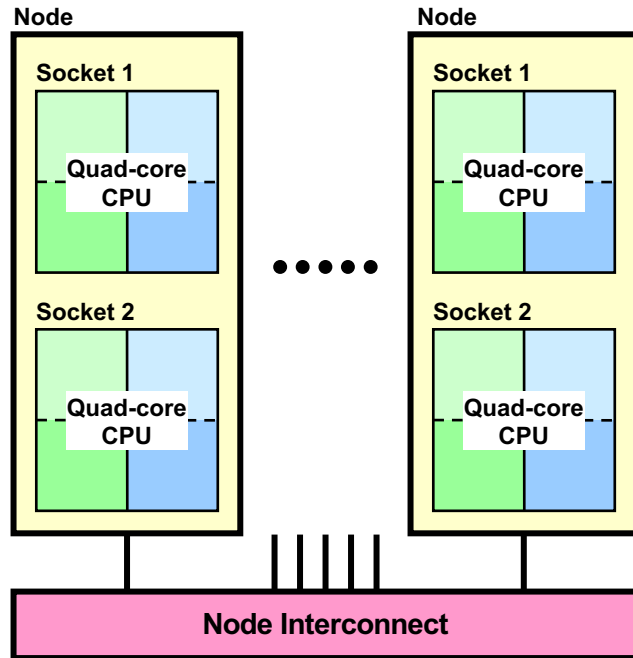
Remarks on Cost-Benefit Calculation

Hardware and programming models



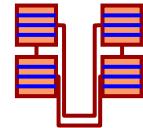
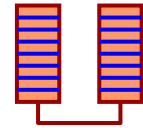
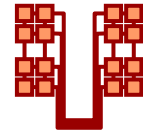
- MPI + threading
 - OpenMP
 - Cilk(+)
 - TBB (Threading Building Blocks)
- MPI + MPI shared memory
- MPI + accelerator
 - OpenACC
 - OpenMP accelerator support
 - CUDA
 - OpenCL, Kokkos, SYCL, ...
- Pure MPI communication

Options for running code on multicore clusters



■ Which programming model is fastest?

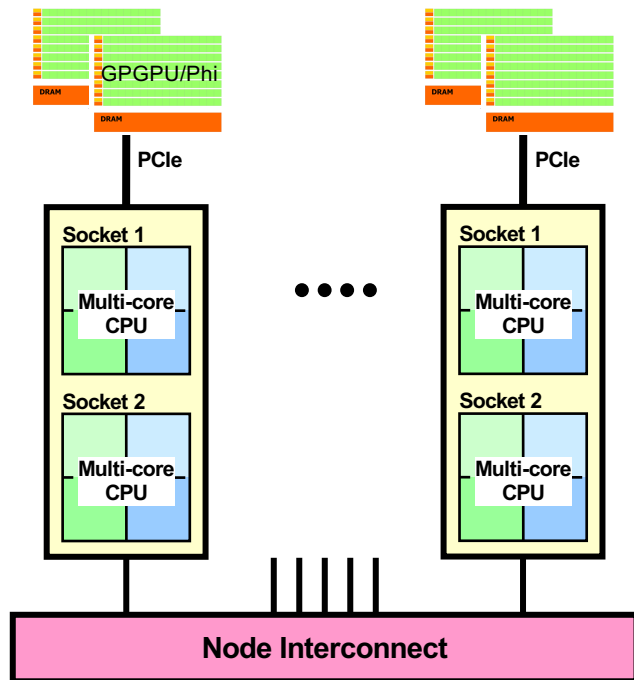
- MPI everywhere?
- Fully hybrid MPI & OpenMP?
- Something between? (Mixed model)
- Often hybrid programming **slower** than pure MPI
 - Examples, Reasons,



...



More Options with accelerators



Hierarchical hardware

- Many levels

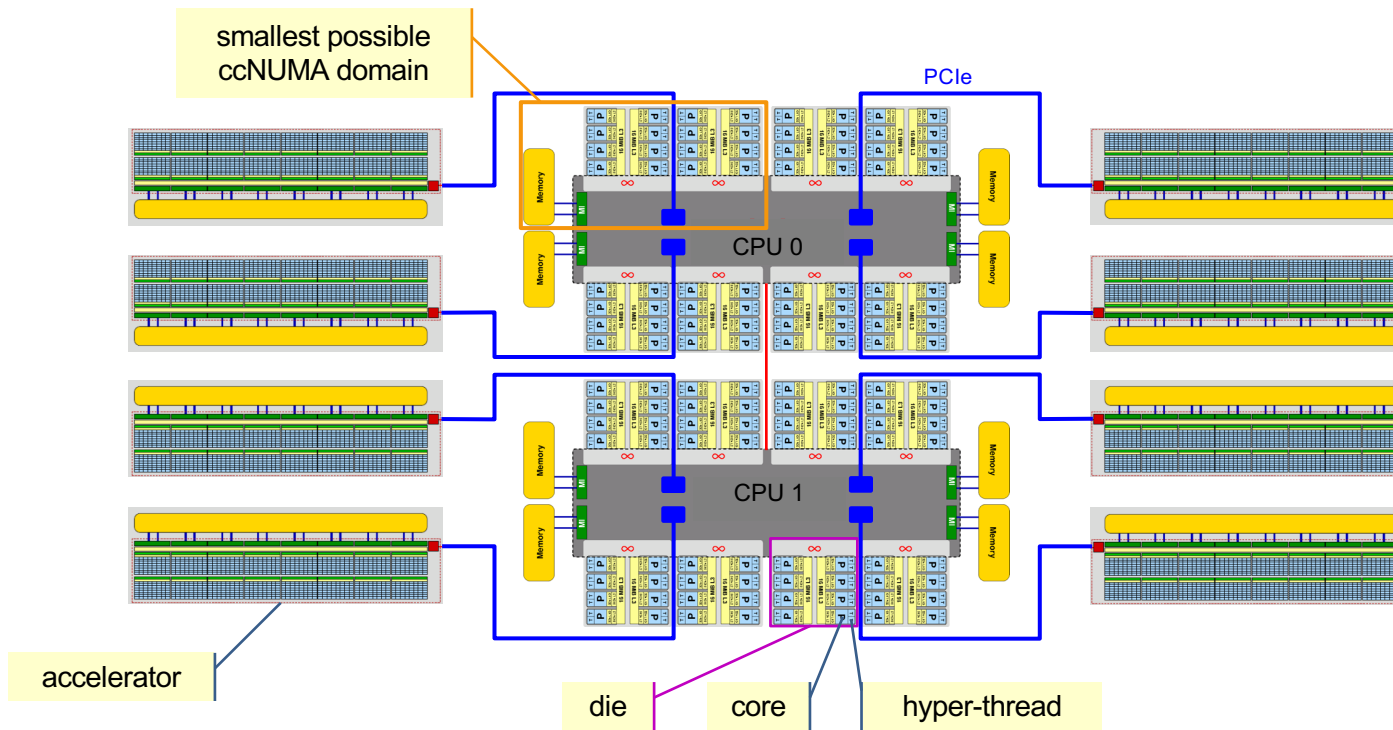
Hierarchical parallel programming

- Many options for MPI+X:
one MPI process per
 - node
 - CPU
 - ccNUMA domain
 - [...]
 - core
 - hyper-thread

Where is the bottleneck?

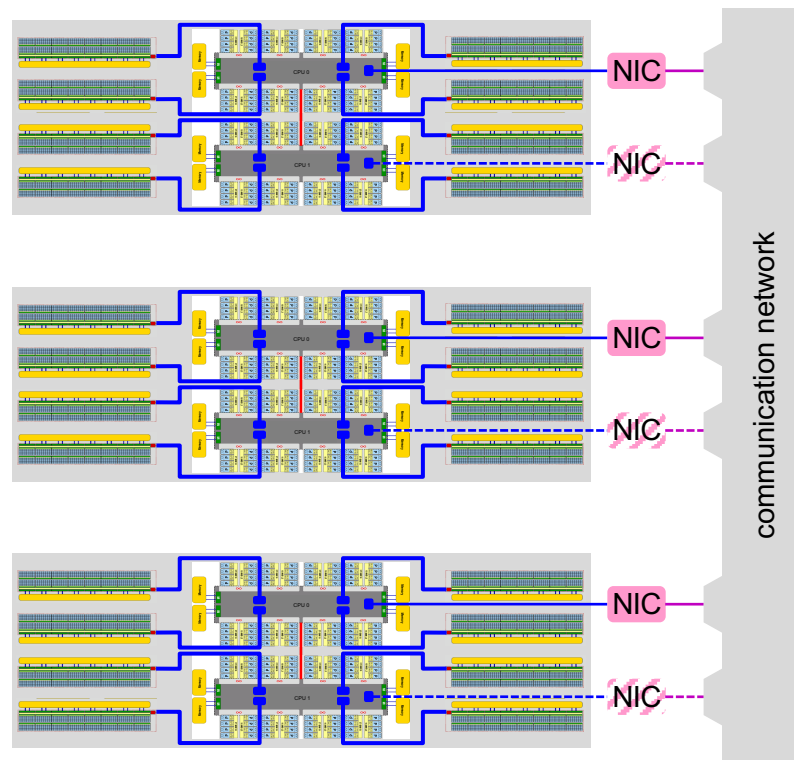
Dual-CPU ccNUMA + accelerator node architecture

Actual topology of a modern compute node



Hardware bottlenecks

- Multicore cluster
 - Computation
 - Memory bandwidth
 - Intra-CPU communication (i.e., core-to-core)
 - Intra-node communication (i.e., CPU-to-CPU)
 - Inter-node communication
- Cluster with CPU+Accelerators
 - Within the accelerator
 - Computation
 - Memory bandwidth
 - Core-to-Core communication
 - Within the CPU and between the CPUs
 - See above
 - Link between CPU and accelerator



Example: Hardware bottlenecks in SpMV

- Sparse matrix-vector-multiply with stored matrix entries

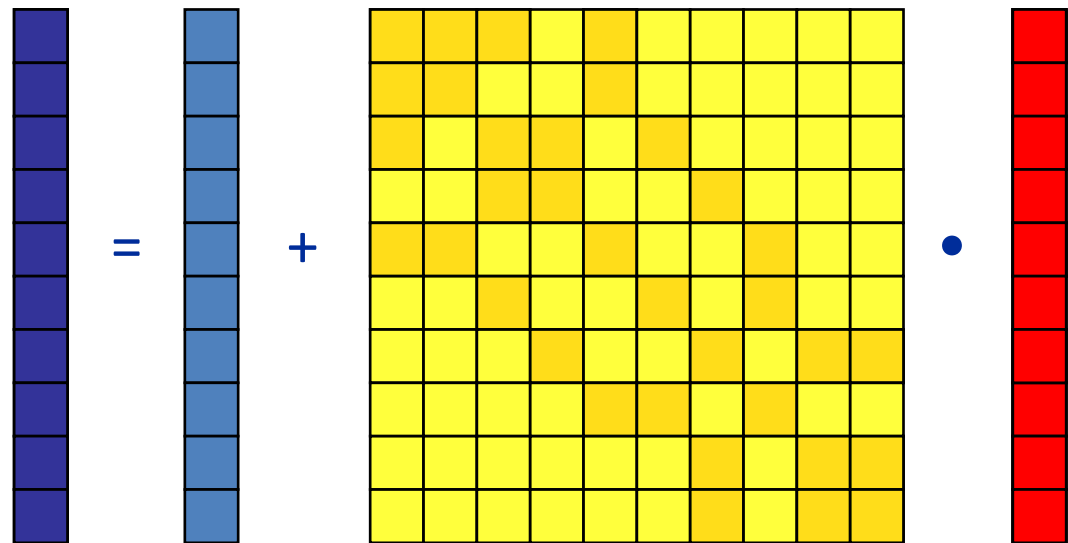
- **Bottleneck:** memory bandwidth of each CPU

- SpMV with calculated matrix entries
(many complex operations per entry)

- **Bottleneck:** computational speed of each core

- SpMV with highly scattered matrix entries

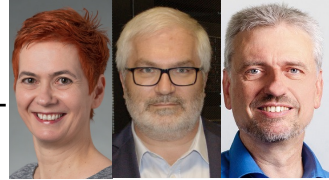
- **Bottleneck:** Inter-node communication



Questions addressed in this tutorial

- What is the **performance impact** of system **topology**?
- How do I **map my programming model** on the system to my advantage?
 - How do I do the **split** into MPI+X?
 - Where do my processes/threads run? How do I **take control**?
 - **Where** is my **data**?
 - How can I **minimize communication** overhead?
- How does hybrid programming help with **typical HPC problems**?
 - Can it **reduce communication** overhead?
 - Can it **reduce replicated data**?
- How can I leverage **multiple accelerators**?
 - What are typical **challenges**?





Programming models

- **MPI + OpenMP on multi/many-core** + Exercises
- ~~MPI + MPI-3.0 shared memory~~ + Exercise
- ~~Pure MPI communication~~ + Exercise
- **MPI + Accelerators**

Programming models

- MPI + OpenMP

General considerations	slide 15
How to compile, link, and run	20
Hands-on: Hello hybrid!	29
System topology, ccNUMA, and memory bandwidth	31
Memory placement on ccNUMA systems	39
Topology and affinity on multicore	48
Hands-on: Pinning	59
Case study: The Multi-Zone NAS Parallel Benchmarks	
Hands-on: Masteronly hybrid Jacobi	61
Overlapping communication and computation	64
Communication overlap with OpenMP taskloops	70
Hands-on: Taskloop-based hybrid Jacobi	76
Main advantages, disadvantages, conclusions	77

Programming models

- MPI + OpenMP

General considerations

> General considerations

How to compile, link, and run

Hands-on: Hello hybrid!

System topology, ccNUMA, and memory bandwidth

Memory placement on ccNUMA systems

Topology and affinity on multicore

Hands-on: Pinning

~~Case study: The Multi-Zone NAS Parallel Benchmarks~~

Hands-on: Masteronly hybrid Jacobi

Overlapping communication and computation

Communication overlap with OpenMP taskloops

Hands-on: Taskloop-based hybrid Jacobi

Main advantages, disadvantages, conclusions

Potential advantages of MPI+OpenMP

Simple level

- **Leverage additional levels of parallelism**
 - Scaling to higher number of cores
 - Adding OpenMP with incremental additional parallelization
- **Enable flexible load balancing on OpenMP level**
 - Fewer MPI processes leave room for assigning workload more evenly
 - MPI processes with higher workload could employ more threads
 - Cheap OpenMP load balancing (tasking, dynamic/guided loops)
- **Lower communication overhead (possibly)**
 - Few “fat” MPI processes vs many “skinny” processes
 - Fewer messages and smaller amount of data communicated
- **Lower memory requirements due to fewer MPI processes**
 - Reduced amount of application halos & replicated data
 - Reduced size of MPI internal buffer space

Advanced level

- **Explicit communication/computation overlap**

MPI + any threading model

Special MPI init for multi-threaded MPI processes is required:

```
int MPI_Init_thread(    int * argc, char ** argv[],  
                      int  thread_level_required,  
                      int * thread_level_provided);  
  
int MPI_Query_thread( int * thread_level_provided);  
  
int MPI_Is_main_thread(int * flag);
```

may imply higher latencies due to some internal locks

• Possible values for **thread_level_required** (increasing order):

- **MPI_THREAD_SINGLE** Only one thread will execute
- **MPI_THREAD_FUNNELED** Only main¹⁾ thread will make MPI-calls
- **MPI_THREAD_SERIALIZED** Multiple threads may make MPI-calls, but only one at a time
- **MPI_THREAD_MULTIPLE** Multiple threads may call MPI, with no restrictions

• returned **thread_level_provided** may be less or more than **thread_level_required**

→ if (**thread_level_provided** < **thread_level_required**) **MPI_Abort(...)** ;

recommended directly after **MPI_Init_thread**

¹⁾ Main thread = thread that called `MPI_Init_thread`.

Recommendation: Start `MPI_Init_thread` from OpenMP master thread → OpenMP master = MPI main thread

Hybrid MPI+OpenMP masteronly style

```
for (iterations) {  
    #pragma omp parallel  
        numerical code  
    /*end omp parallel */  
  
    /* on master only */  
    MPI_Isend();  
    MPI_Irecv();  
    MPI_Waitall();  
} /* end for loop */
```

masteronly style:
MPI only outside of
parallel regions

Advantages

- Simplest possible hybrid model
- Thread-parallel execution and MPI communication strictly separate
- Minimally required MPI thread support level: `MPI_THREAD_FUNNELED`

Major Problems

- All other threads are sleeping while master thread communicates!
- Only one thread per process communicating → possible underutilization of network bandwidth

Masteronly style within large parallel region

```
#pragma omp parallel
for(iterations) {
  #pragma omp for
  for(i=0; ...) {
    // ... numerics
  } // barrier here
  #pragma omp single
  {
    MPI_Isend();
    MPI_Irecv();
    MPI_Waitall();
  } // Barrier here
} /* end iter loop */
```

- **Barrier** before MPI required
 - May be implicit
 - Prevent race conditions on communication buffer data
 - Between multi-threaded numerics
 - and MPI access by master thread
 - Enforce flush of variables
- **Barrier** after MPI required
 - May be implicit
 - Numerical loop(s) may need communicated data

Programming models

- MPI + OpenMP

How to compile, link, and run

General considerations

> **How to compile, link, and run**

Hands-on: Hello hybrid!

System topology, ccNUMA, and memory bandwidth

Memory placement on ccNUMA systems

Topology and affinity on multicore

Hands-on: Pinning

Case study: The Multi-Zone NAS Parallel Benchmarks

Hands-on: Masteronly hybrid Jacobi

Overlapping communication and computation

Communication overlap with OpenMP taskloops

Hands-on: Taskloop-based hybrid Jacobi

Main advantages, disadvantages, conclusions

How to compile, link and run

- Use appropriate **OpenMP compiler switch** (-openmp, -fopenmp, -mp, -qsmpt=openmp, ...) and MPI compiler script (if available)
- Link with **MPI library**
 - Usually wrapped in MPI compiler script
 - If required, specify to link against thread-safe MPI library
 - Often automatic when OpenMP or auto-parallelization is switched on
- **Running** the code
 - Highly **non-portable** – consult system docs (if available...)
 - Figure out **how to start fewer MPI processes than cores** per node
 - **Pinning** (who is running where?) is extremely **important** → see later



Compiling from a single source

Make use of pre-defined symbols

```
#ifdef _OPENMP # _OPENMP defined with -qopenmp
    // all that is special for OpenMP
#endif

#ifdef USE_MPI # USE_MPI defined with -DUSE_MPI
    // all that is special for MPI
#endif

#ifdef USE_MPI
    MPI_Init(...);
    MPI_Comm_rank(..., &rank);
    MPI_Comm_size(..., &size);
#else
    # recommended for non-MPI
    rank = 0;
    size = 1;
#endif
```

Compiling from a single source

Handling compilers

- Intel MPI + Intel C

```
mpicc      -DUSE_MPI -qopenmp  ...  
icc        -qopenmp  ...
```

- Intel MPI + Intel Fortran

```
mpiifort  -fpp -DUSE_MPI -qopenmp  ...  
ifort     -fpp           -qopenmp  ...
```

- OpenMPI + gcc

```
mpicc      -DUSE_MPI -fopenmp  ...  
gcc        -fopenmp  ...
```

- OpenMPI + gfortran

```
mpif90    -cpp -DUSE_MPI -fopenmp  ...  
gfortran  -cpp           -fopenmp  ...
```

Examples for compilation and execution

- **Cray XC40** (2 NUMA domains w/ 12 cores each), one process (12 threads) per socket
 - `ftn -h omp ...`
 - `OMP_NUM_THREADS=12 aprun -n 4 -N 2 \`
`-d $OMP_NUM_THREADS ./a.out`
- **Intel Ice Lake** (36-core 2-socket) cluster, **Intel MPI/OpenMP**, one process (36 threads) per socket
 - `mpiifort -qopenmp ...`
 - `mpirun -ppn 2 -np 4 \`
`-env OMP_NUM_THREADS 36`
`-env I_MPI_PIN_DOMAIN socket \`
`-env KMP_AFFINITY scatter ./a.out`

Examples for compilation and execution

- Intel Ice Lake (36-core 2-socket) cluster, Intel MPI/OpenMP + likwid-mpirun, one process (36 threads) per socket
 - `mpiifort -qopenmp ...`
 - `likwid-mpirun -np 4 -pin S0:0-35_S1:0-35 ./a.out`
- Intel Skylake (24-core 2-socket) cluster, GCC + OpenMPI 4.1, one process (24 threads) per socket
 - `mpif90 -fopenmp ...`
 - `OMP_NUM_THREADS=24 OMP_PLACES=cores OMP_PROC_BIND=close \mpirun --map-by ppr:1:socket:PE=24 ./a.out`
 - Dito, two processes per socket (12 threads each)
`OMP_NUM_THREADS=12 OMP_PLACES=cores OMP_PROC_BIND=close \mpirun --map-by ppr:2:socket:PE=12 ./a.out`

Learn about node topology

- A collection of tools is available
 - `numactl --hardware` (numatools)
 - `lstopo --no-io` (part of hwloc)
 - `cpuinfo -A` (part of Intel MPI)
 - **likwid-topology** (part of LIKWID tool suite <http://tiny.cc/LIKWID>)

```
$ likwid-topology -c -g
```

```
-----  
CPU name:      Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz
```

```
CPU type:      Intel Xeon IvyBridge EN/EP/EX processor
```

```
CPU stepping: 4
```

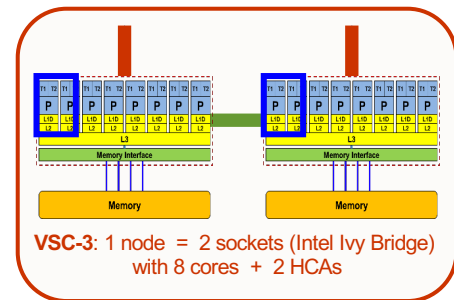
```
*****  
Hardware Thread Topology
```

```
*****  
Sockets:      2
```

```
Cores per socket: 8
```

```
Threads per core: 2
```

```
[... Some output omitted ...]
```



Learning about node topology

(...cont...)

Graphical Topology

Socket 0:

+-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+							
0 16	1 17	2 18	3 19	4 20	5 21	6 22	7 23
+-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+							
32kB	32kB	32kB	32kB	32kB	32kB	32kB	32kB
+-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+							
256kB	256kB	256kB	256kB	256kB	256kB	256kB	256kB
+-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+							
+-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+							
20MB							
+-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+							
+-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+							

Caveat:
Numbering may differ for
different setups of same CPU!

Learning about node topology

(...cont...)

Graphical Topology

Socket 1:

+-----+																									
	+-----+		+-----+		+-----+		+-----+		+-----+																
	8	24		9	25		10	26		11	27		12	28		13	29		14	30		15	31		
	+-----+		+-----+		+-----+		+-----+		+-----+		+-----+		+-----+		+-----+		+-----+		+-----+		+-----+				
	+-----+		+-----+		+-----+		+-----+		+-----+		+-----+		+-----+		+-----+		+-----+		+-----+		+-----+				
	32kB			32kB			32kB			32kB			32kB			32kB			32kB			32kB			
	+-----+		+-----+		+-----+		+-----+		+-----+		+-----+		+-----+		+-----+		+-----+		+-----+		+-----+				
	+-----+		+-----+		+-----+		+-----+		+-----+		+-----+		+-----+		+-----+		+-----+		+-----+		+-----+				
	256kB			256kB			256kB			256kB			256kB			256kB			256kB			256kB			
	+-----+		+-----+		+-----+		+-----+		+-----+		+-----+		+-----+		+-----+		+-----+		+-----+		+-----+				
	+-----+																								
	20MB																								
	+-----+																								
+-----+																									

Programming models

- MPI + OpenMP

Hands-On #1

Hello hybrid!

General considerations

How to compile, link, and run

> **Hands-on: Hello hybrid!**

System topology, ccNUMA, and memory bandwidth

Memory placement on ccNUMA systems

Topology and affinity on multicore

Hands-on: Pinning

Case study: The Multi-Zone NAS Parallel Benchmarks

Hands-on: Masteronly hybrid Jacobi

Overlapping communication and computation

Communication overlap with OpenMP taskloops

Hands-on: Taskloop-based hybrid Jacobi

Main advantages, disadvantages, conclusions

Hands-On #1

he-hy - Hello Hybrid! - compiling, starting

1. FIRST THINGS FIRST - PART 1: find out about a (new) cluster - login node
2. FIRST THINGS FIRST - PART 2: find out about a (new) cluster - batch jobs
3. MPI+OpenMP: **:TODO:** how to compile and start an application
how to do conditional compilation
4. MPI+OpenMP: **:TODO:** get to know the hardware - needed for pinning

→ see: **TODO.README**

Programming models - MPI + OpenMP

System topology, ccNUMA, and memory bandwidth

General considerations

How to compile, link, and run

Hands-on: Hello hybrid!

> **System topology, ccNUMA, and memory bandwidth**

Memory placement on ccNUMA systems

Topology and affinity on multicore

Hands-on: Pinning

Case study: The Multi-Zone NAS Parallel Benchmarks

Hands-on: Masteronly hybrid Jacobi

Overlapping communication and computation

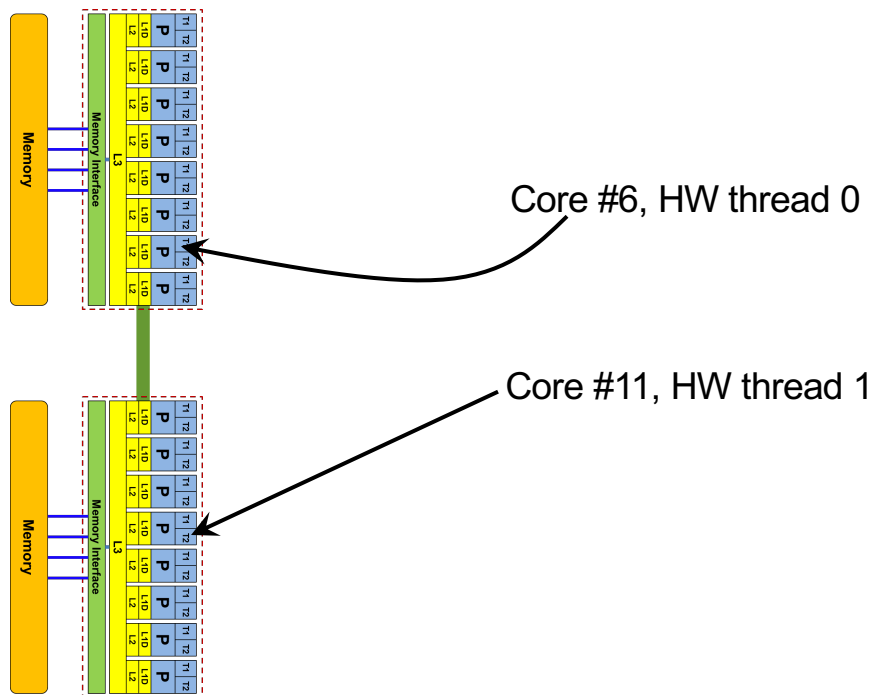
Communication overlap with OpenMP taskloops

Hands-on: Taskloop-based hybrid Jacobi

Main advantages, disadvantages, conclusions

What is “topology”?

Where in the machine does core (or hardware thread) #n reside?

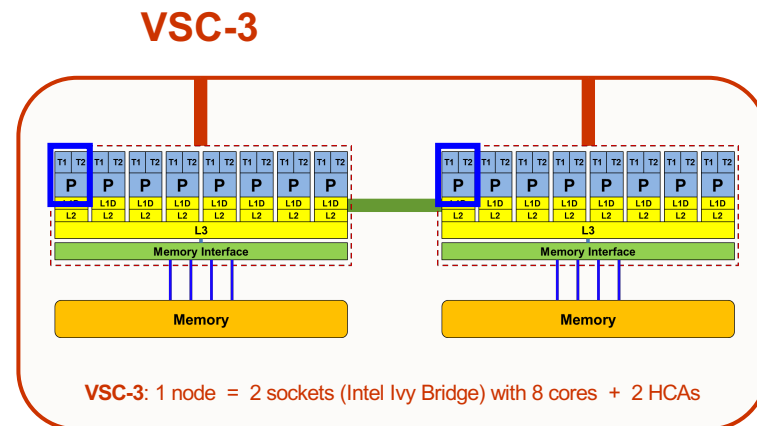


Why is this important?

- **Resource** sharing (cache, data paths)
- **Communication** efficiency (shared vs. separate caches, buffer locality)
- **Memory** access locality (ccNUMA!)

Compute nodes – caches

Latency	← typical →	Bandwidth
1–2 ns	L1 cache	200 GB/s
3–10 ns	L2/L3 cache	50 GB/s
100 ns	memory	20 GB/s (1 core)



Ping-Pong Benchmark – Latency

Intra-node vs. inter-node on VSC-3

- nodes = 2 sockets (Intel Ivy Bridge) with 8 cores + 2 HCAs
- inter-node = IB fabric = dual rail Intel QDR-80 = 3-level fat-tree (BF: 2:1 / 4:1)

Affinity matters!

```
myID = get_process_ID()
if(myID.eq.0) then
  targetID = 1
  S = get_walltime()
  call Send_message(buffer,N,targetID)
  call Receive_message(buffer,N,targetID)
  E = get_walltime()
  GBYTES = 2*N/(E-S)/1.d9 ! Gbyte/s rate
  TIME = (E-S)/2*1.d6      ! transfer time
else
  targetID = 0
  call Receive_message(buffer,N,targetID)
  call Send_message(buffer,N,targetID)
endif
```

Latency [μs]	MPI_Send(...)	
	OpenMPI	Intel MPI
intra-socket	0.3 μs	0.3 μs
inter-socket	0.6 μs	0.7 μs
IB -1- edge	1.2 μs	1.4 μs
IB -2- leaf	1.6 μs	1.8 μs
IB -3- spine	2.1 μs	2.3 μs

For comparison:
typical latencies

L1 cache	1–2 ns
L2/L3 c.	3–10 ns
memory	100 ns
HPC networks	1–10 μs

→ Avoiding slow data paths is the key to most performance optimizations!

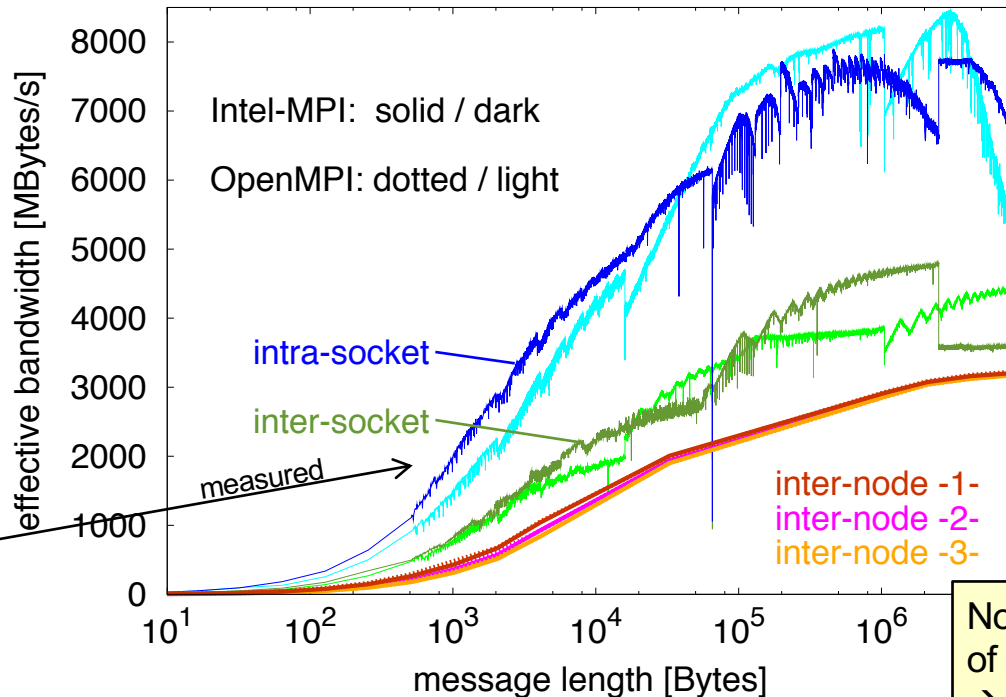
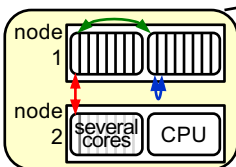
Ping-Pong 1-on-1 Benchmark – Effective Bandwidth

intra-node vs. inter-node on VSC-3

inter-node:
IB fabric
dual rail (2 HCAs)
Intel QDR-80
3-level fat-tree
BF: 2:1 / 4:1

QDR-80 (2 HCAs)
link: 80 Gbit/s
max 8 Gbytes/s
eff. 6.8 Gbytes/s

→ 1 HCA = 1/2 (2 HCAs)



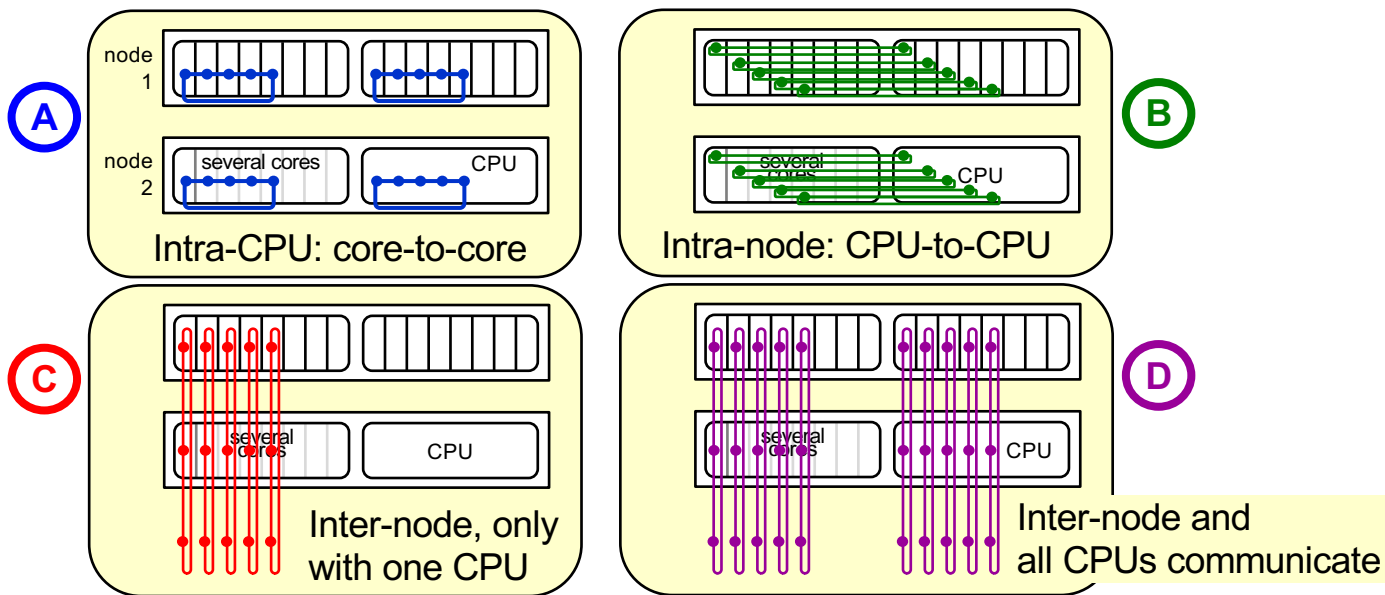
Not representative
of real applications
→ see next slide(s)

Multiple communicating rings

Benchmark halo_irecv_send_multiplelinks_toggle.c

- Varying message size,
- number of *communication cores per CPU*, and
- four communication schemes (example with 5 communicating cores per CPU)

See HLRS online courses
<http://www.hlrs.de/training/self-study-materials>
→ Practical → MPI.tar.gz
→ subdirectory MPI/course/C/1sided/

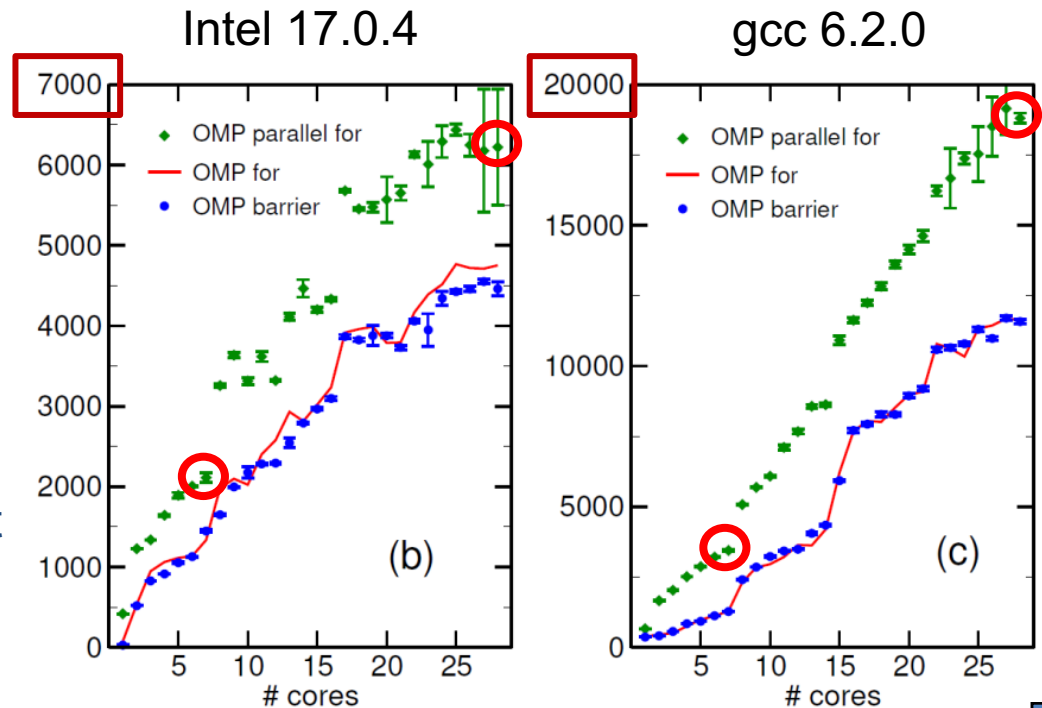


OpenMP barrier synchronization cost

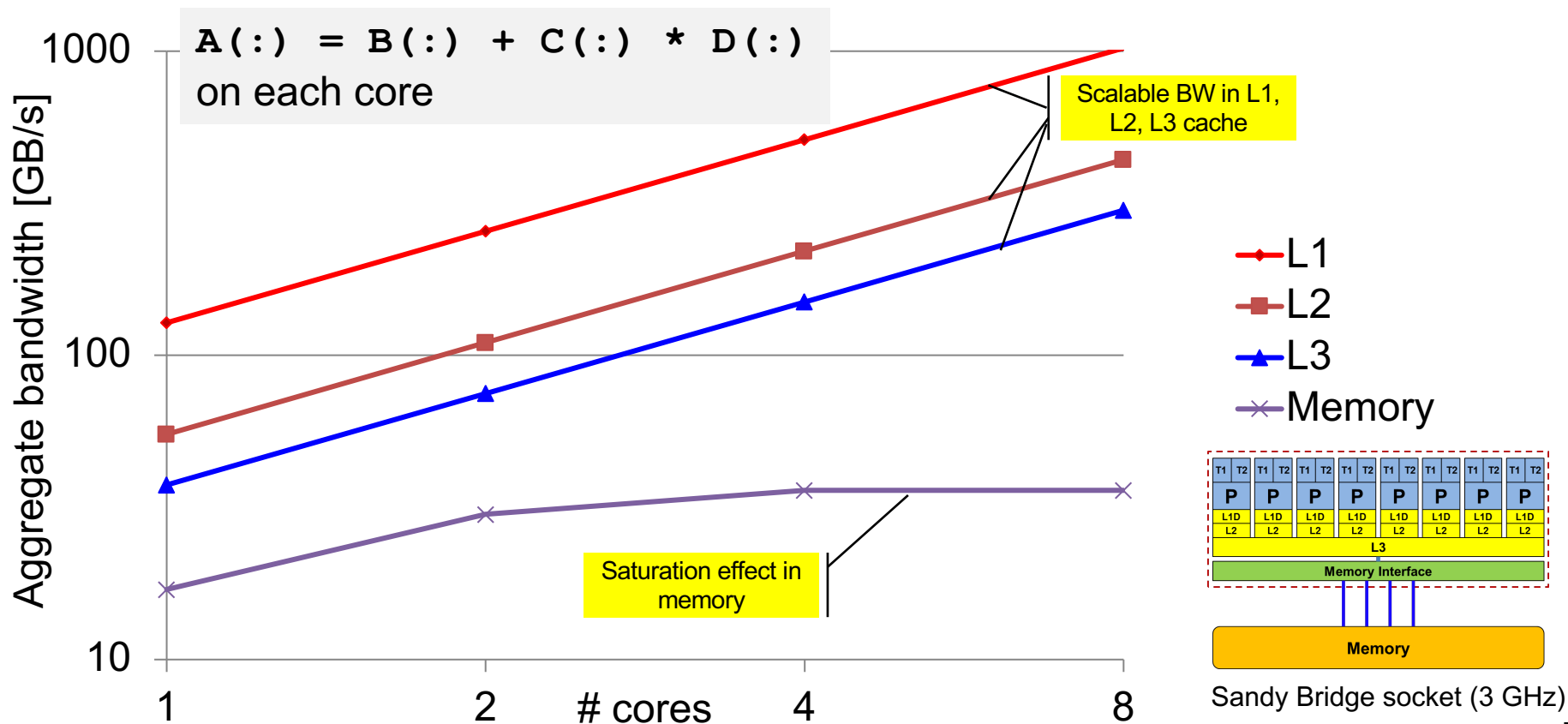
Comparison of **barrier synchronization** cost with increasing number of threads

- 2x Haswell 14-core (CoD mode)
- Optimistic measurements (repeated 1000s of times)
- No impact from previous activity in cache

→ Barrier **sync time** highly dependent on **system topology** & OpenMP runtime **implementation**



Accumulated bandwidth saturation vs. # cores



Programming models

- MPI + OpenMP

Memory placement on ccNUMA systems

General considerations

How to compile, link, and run

Hands-on: Hello hybrid!

System topology, ccNUMA, and memory bandwidth

> Memory placement on ccNUMA systems

Topology and affinity on multicore

Hands-on: Pinning

Case study: The Multi-Zone NAS Parallel Benchmarks

Hands-on: Masteronly hybrid Jacobi

Overlapping communication and computation

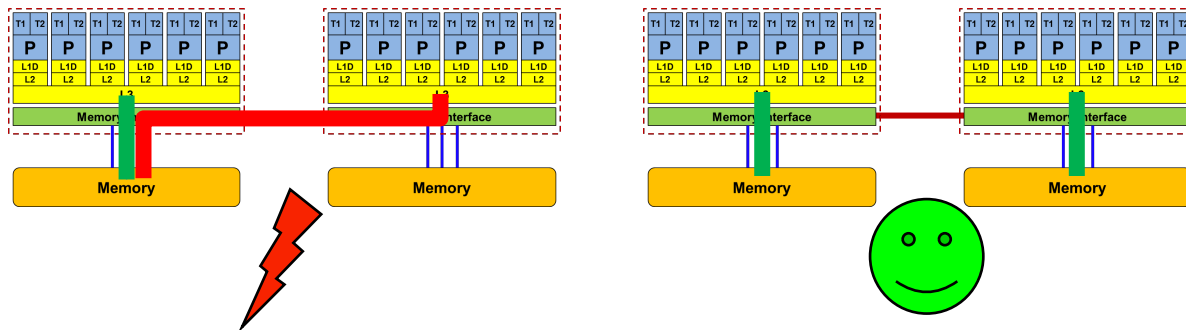
Communication overlap with OpenMP taskloops

Hands-on: Taskloop-based hybrid Jacobi

Main advantages, disadvantages, conclusions

A short introduction to ccNUMA

- ccNUMA:
 - whole memory is **transparently accessible** by all processors
 - but **physically distributed**
 - with **varying bandwidth and latency**
 - and **potential contention** (shared memory paths)
 - Memory placement occurs with **OS page granularity** (often 4 KiB)



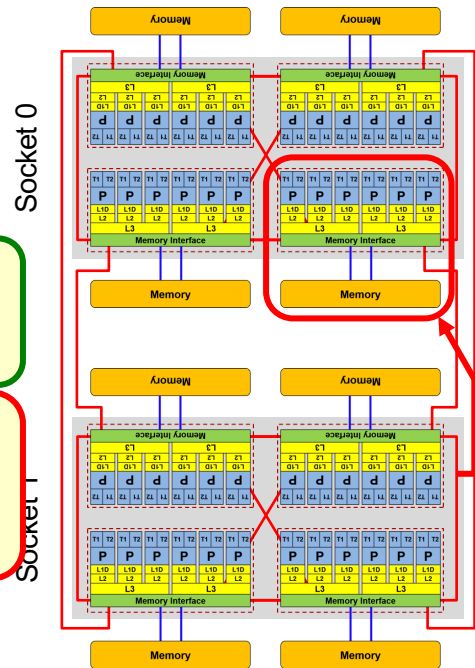
How much bandwidth does non-local access cost?

- Example: AMD “Naples” 2-socket system (8 chips, 2 sockets, 48 cores):
STREAM Triad bandwidth measurements [Gbyte/s]

CPU node		0	1	2	3	4	5	6	7
Memory node	0	32.4	21.4	21.8	21.9	10.6	10.6	10.7	10.8
	1	21.5	32.4	21.9	21.9	10.6	10.5	10.7	10.6
	2	21.8	21.9	32.4	21.5	10.6	10.6	10.7	10.6
	3	21.9	21.9	21.5	32.4	10.6	10.6	10.6	10.7
	4	10.6	10.7	10.6	10.6	32.4	21.4	21.9	21.9
	5	10.6	10.6	10.6	10.6	21.4	32.4	21.9	21.9
	6	10.6	10.7	10.6	10.6	21.9	21.9	32.3	21.4
	7	10.7	10.6	10.6	10.6	21.9	21.9	21.4	32.5

Highest bandwidth between memory and cores of one NUMA domain

Do you want to run your application 3 times slower?
(If your appl. is memory bandwidth bound)



Avoiding locality problems

- How can we make sure that memory ends up where it is close to the CPU that uses it?
 - See next slides (first-touch initialization)
- How can we make sure that it stays that way throughout program execution?
 - See later in the tutorial (pinning)
- **Taking control** is the key strategy!

Solving Memory Locality Problems: First Touch

- "Golden Rule" of ccNUMA:
A memory page gets mapped into the local memory of the processor that first touches it!
- Consequences
 - Process/thread-core **affinity** is decisive!
 - With **OpenMP**, **data initialization code** becomes important even if it takes little time to execute (“**parallel first touch**”)
 - **Parallel first touch is automatic for pure MPI**
 - If thread team does not span across NUMA domains, memory mapping is not a problem
- **Automatic page migration** may help if memory is used long enough

Important

Solving Memory Locality Problems: First Touch

- "Golden Rule" of ccNUMA:

A memory page gets mapped into the local memory of the processor that first touches it!

Important

- Except if there is not enough local memory available
- Some OSs allow to influence placement in more direct ways
 - → libnuma (Linux)
- **Caveat:** "touch" means "write," not "allocate" or "read"
- Example:

```
double *huge = (double*)malloc(N*sizeof(double));  
// memory not mapped yet  
for(i=0; i<N; i++) // or i+=PAGE_SIZE  
    huge[i] = 0.0; // mapping takes place here!
```

▪

Most simple case: explicit initialization

```
integer,parameter :: N=10000000
double precision A(N), B(N)
```

A=0.d0



```
!$OMP parallel do
do i = 1, N
  B(i) = function ( A(i) )
end do
!$OMP end parallel do
```

```
integer,parameter :: N=10000000
double precision A(N),B(N)
```

```
!$OMP parallel
!$OMP do schedule(static)
```

```
do i = 1, N
```

```
  A(i)=0.d0
```

```
end do
```

```
!$OMP end do
```

```
...
```

```
!$OMP do schedule(static)
```

```
do i = 1, N
```

```
  B(i) = function ( A(i) )
```

```
end do
```

```
!$OMP end do
```

```
!$OMP end parallel
```



Handling ccNUMA in practice

- Solution A
 - One (or more) MPI process(es) per ccNUMA domain
 - **Pro:** optimal page placement (perfectly local memory access) for free
 - **Con:** higher number (>1) of MPI processes on each node
- Solution B
 - One MPI process per node or one MPI process spans multiple ccNUMA domains
 - **Pro:** Smaller number of MPI processes compared to Solution A
 - **Cons:**
 - Explicitly parallel initialization needed to “bind” the data to each ccNUMA domain
→ otherwise loss of performance
 - Dynamic/guided schedule or tasking → loss of performance
- Thread binding is mandatory for A and B! – Never trust the defaults! ■

Conclusions from the observed topology effects

- **Know your hardware** characteristics:
 - Hardware topology (use tools such as likwid-topology)
 - Typical hardware bottlenecks
 - These are independent of the programming model!
 - Hardware bandwidths, latencies, peak performance numbers
- **Know your software** characteristics
 - Typical numbers for communication latencies, bandwidths
 - Typical OpenMP overheads
- Learn how to **take control**
 - See next chapter on affinity control
- **Leveraging topology effects is a part of code optimization!**



Programming models

- MPI + OpenMP

Topology and affinity on multicore

General considerations

How to compile, link, and run

Hands-on: Hello hybrid!

System topology, ccNUMA, and memory bandwidth

Memory placement on ccNUMA systems

> **Topology and affinity on multicore**

Hands-on: Pinning

Case study: The Multi-Zone NAS Parallel Benchmarks

Hands-on: Masteronly hybrid Jacobi

Overlapping communication and computation

Communication overlap with OpenMP taskloops

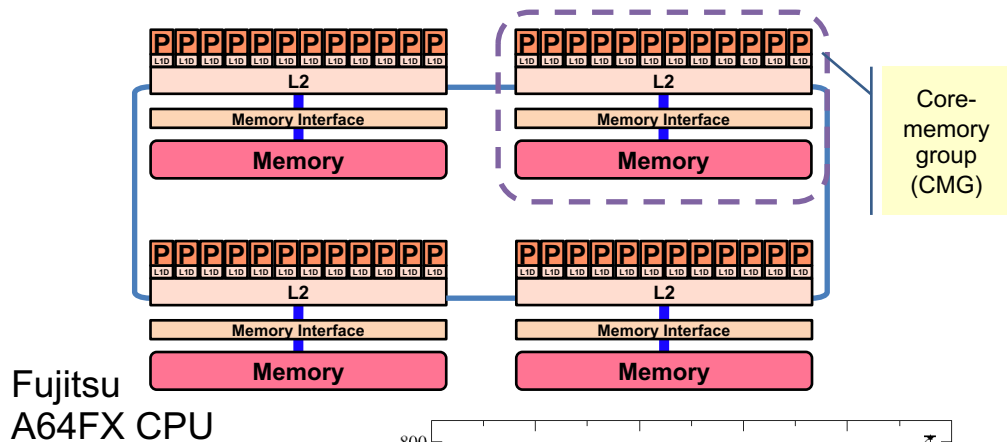
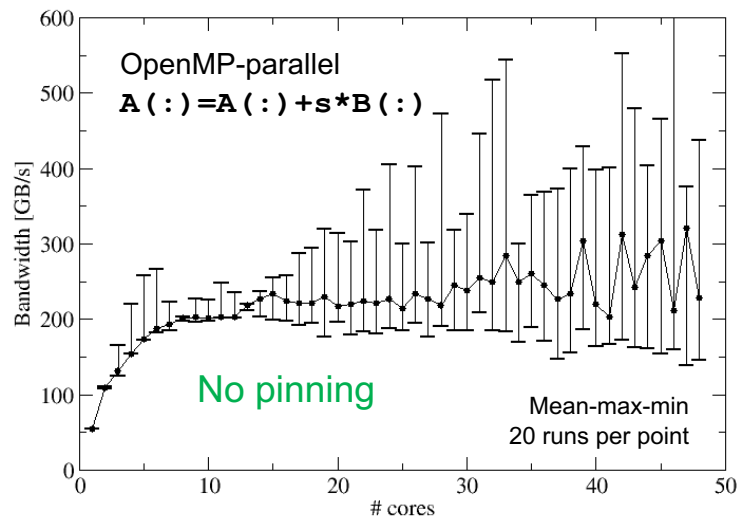
Hands-on: Taskloop-based hybrid Jacobi

Main advantages, disadvantages, conclusions

Thread/Process Affinity (“Pinning”)

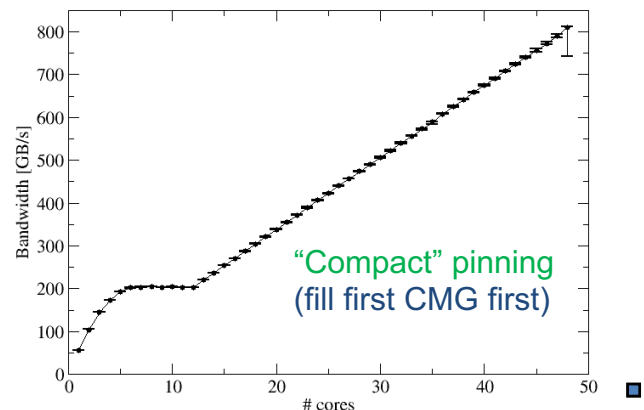
- **Highly OS-dependent** system calls
 - But available on all OSs
 - Non-portable
- Support for **user-defined pinning for OpenMP** threads in all compilers
 - Compiler specific
 - **Standardized in OpenMP** (places)
 - Generic Linux: `taskset`, `numactl`, `likwid-pin`
- **Affinity** awareness in all **MPI** libraries
 - **Not defined** by the MPI **standard** (as of 4.0)
 - Necessarily non-portable feature of the startup mechanism (`mpirun`, ...)
- Affinity awareness in batch **scheduler**
 - Batch scheduler must work with MPI + OpenMP affinity
 - Difficult, non-portable, every combination is different

Anarchy vs. affinity with OpenMP STREAM



There are several reasons for caring about affinity:

- Eliminating performance variation
- Making use of architectural features
- Avoiding resource contention



OMP_PLACES and Thread Affinity (see OpenMP-4.0 page 7 lines 29-32, p. 241-243)

A *place* consists of one or more *processors*.

Pinning on the level of *places*.

Free migration of the threads on a place between the *processors* of that place.

processor is the smallest unit to run a thread or task

▪ **OMP_PLACES=threads**

abstract_name

→ Each place corresponds to the single *processor* of a single hardware thread (hyper-thread)

▪ **OMP_PLACES=cores**

→ Each place corresponds to the processors (one or more hardware threads) of a single core

▪ **OMP_PLACES=sockets**

→ Each place corresponds to the processors of a single socket (consisting of all hardware threads of one or more cores)

▪ **OMP_PLACES=abstract_name(num_places)**

→ In general, the number of places may be explicitly defined

<lower-bound>:<number of entries>[:<stride>]

▪ Or with explicit numbering, e.g. 8 places, each consisting of 4 processors:

- `setenv OMP_PLACES "{0,1,2,3},{4,5,6,7},{8,9,10,11}, ... {28,29,`
- `setenv OMP_PLACES "{0:4},{4:4},{8:4}, ... {28:4}"`
- `setenv OMP_PLACES "{0:4}:8:4"`

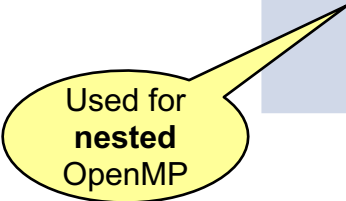
CAUTION:

The numbers highly depend on hardware and operating system, e.g.,
{0,1} = hyper-threads of 1st core of 1st socket, or
{0,1} = 1st hyper-thread of 1st core of 1st and 2nd socket, or ...

OMP_PROC_BIND variable / proc_bind() clause

Determines how places are used for pinning:

OMP_PROC_BIND	Meaning
FALSE	Affinity disabled
TRUE	Affinity enabled, implementation defined strategy
CLOSE	Threads bind to consecutive places
SPREAD	Threads are evenly scattered among places
MASTER	Threads bind to the same place as the master thread that was running before the parallel region was entered



Used for
nested
OpenMP

Some simple OMP_PLACES examples

- Intel Xeon w/ SMT, 2x36 cores, 1 thread per physical core, fill 1 socket

```
OMP_NUM_THREADS=36  
OMP_PLACES=cores  
OMP_PROC_BIND=close
```

- Intel Xeon Phi with 72 cores,
32 cores to be used, 2 threads per physical core

```
OMP_NUM_THREADS=64  
OMP_PLACES=cores(32)  
OMP_PROC_BIND=close # spread will also do
```

- Intel Xeon, 2 sockets, 4 threads per socket (no binding within socket!)

```
OMP_NUM_THREADS=8  
OMP_PLACES=sockets  
OMP_PROC_BIND=close # spread will also do
```

- Intel Xeon, 2 sockets, 4 threads per socket, binding to cores

```
OMP_NUM_THREADS=8  
OMP_PLACES=cores  
OMP_PROC_BIND=spread
```

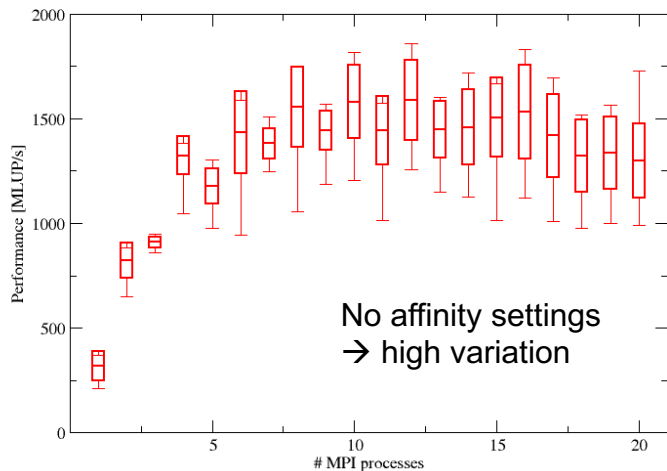
Always prefer abstract places
instead of HW thread IDs! ■

Pinning of MPI processes

- Highly system dependent!
- **Intel MPI**: env variable `I_MPI_PIN_DOMAIN`
- **OpenMPI**: choose between several mpirun options, e.g.,
-bind-to-core, -bind-to-socket, -bycore, -byslot ...
- Cray's **aprun**: pinning by default

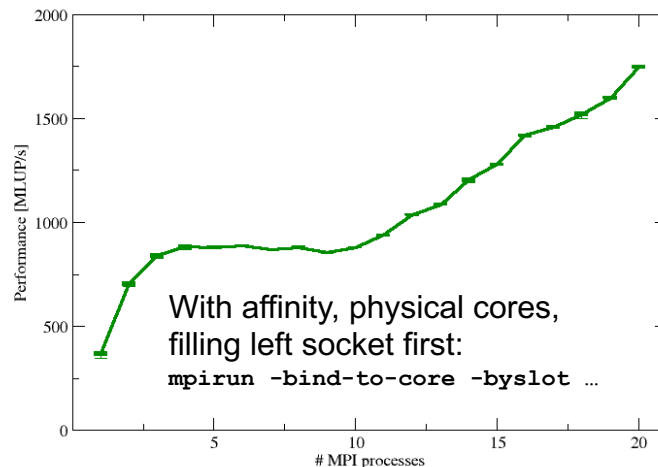
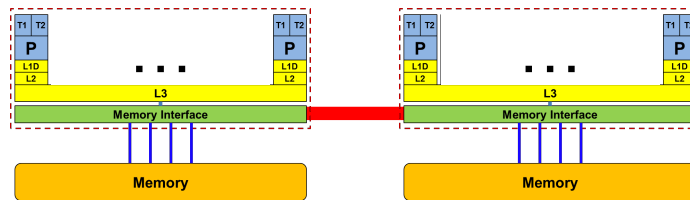
- Platform-independent tools: likwid-mpirun
(likwid-pin, numactl)

Anarchy vs. affinity with a heat equation solver



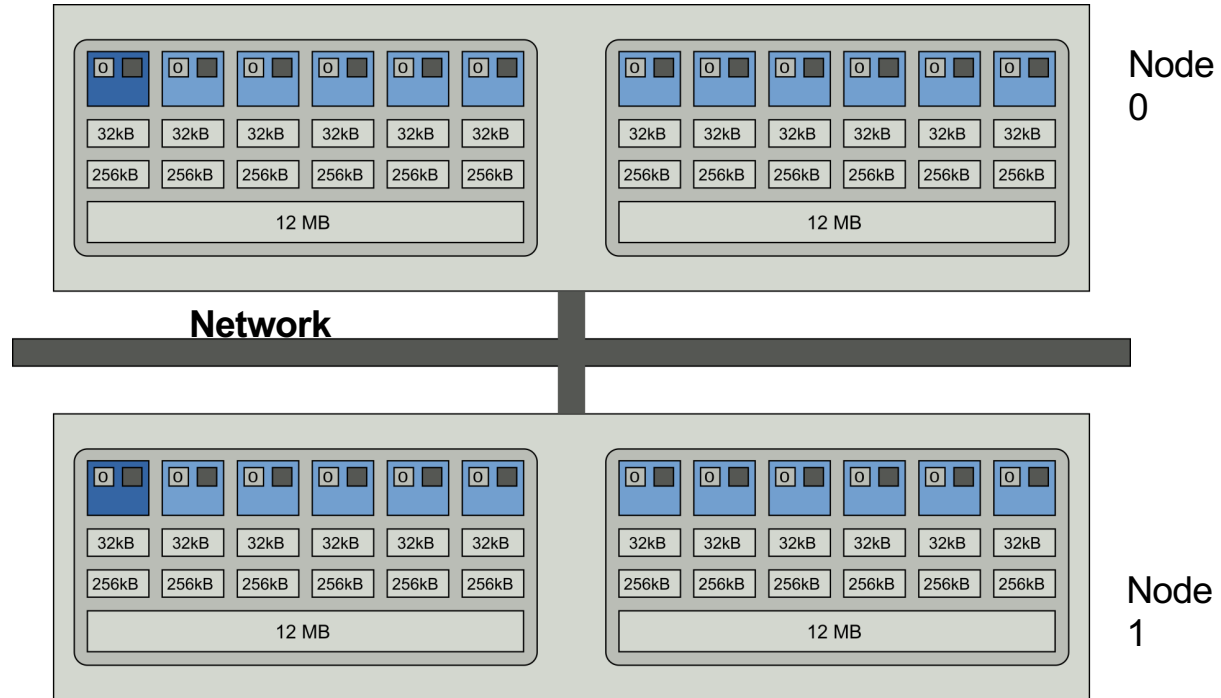
Reasons for caring about affinity:

- Eliminating performance variation
- Making use of architectural features
- Avoiding resource contention



likwid-mpirun: 1 MPI process per node

```
likwid-mpirun -np 2 -pin N:0-11 ./a.out
```

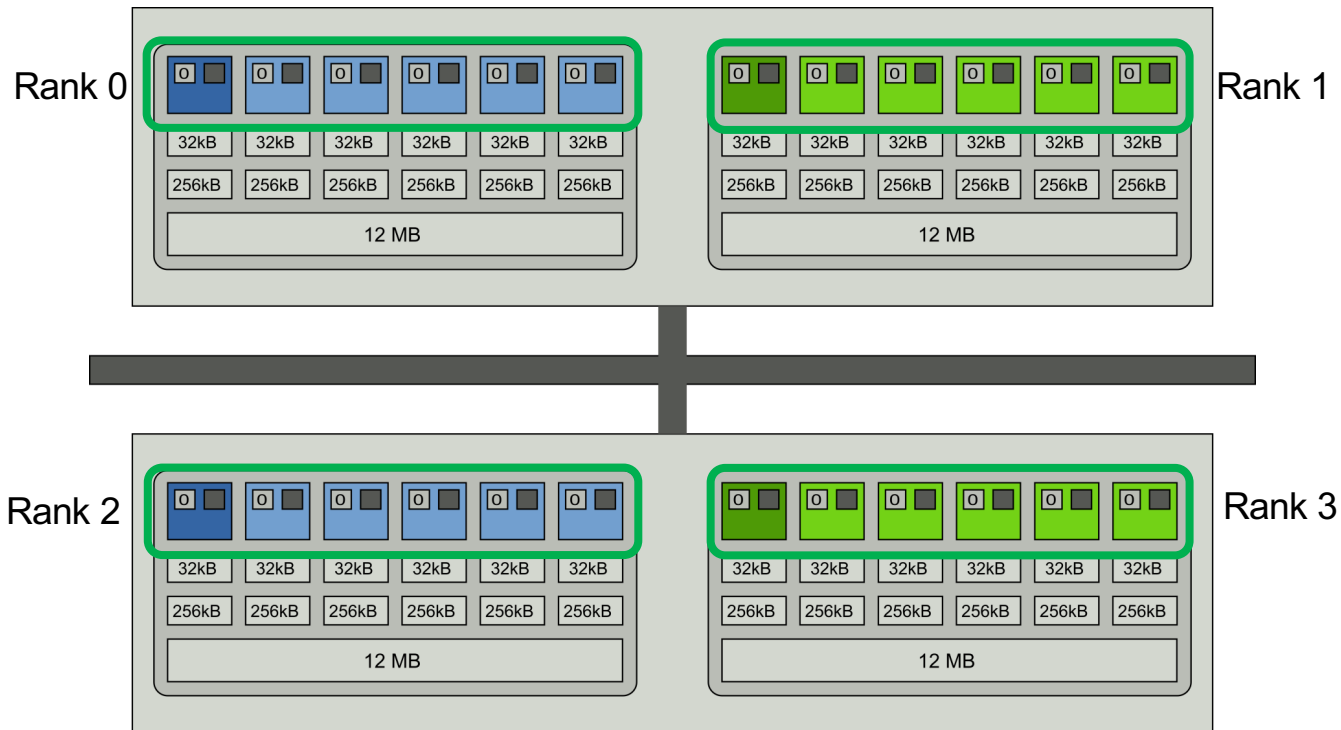


Intel MPI+compiler:

```
OMP_NUM_THREADS=12 mpirun -ppn 1 -np 2 -env KMP_AFFINITY scatter ./a.out
```


likwid-mpirun: 1 MPI process per socket

```
likwid-mpirun -np 4 -pin s0:0-5_s1:0-5 ./a.out
```



Intel MPI+compiler:

```
OMP_NUM_THREADS=6 mpirun -ppn 2 -np 4 \  
-env I_MPI_PIN_DOMAIN socket -env KMP_AFFINITY scatter ./a.out
```

MPI/OpenMP affinity: Take-home messages

- Learn how to take control of hybrid execution!
 - Almost all performance features depend on topology and thread placement! (especially if SMT/Hyperthreading is on)
- Always observe the topology dependence of
 - Intranode MPI performance
 - OpenMP overheads
 - Saturation effects / scalability behavior with bandwidth-bound code
- Enforce proper thread/process to core binding, using appropriate tools (→ whatever you use, but use SOMETHING)
- Memory page placement on ccNUMA nodes
 - Automatic optimal page placement for one (or more) MPI processes per ccNUMA domain (solution A)
 - Explicitly parallel first-touch initialization only required for multi-domain MPI processes (solution B)

Programming models

- MPI + OpenMP

Hands-On #2

Pinning

General considerations

How to compile, link, and run

Hands-on: Hello hybrid!

System topology, ccNUMA, and memory bandwidth

Memory placement on ccNUMA systems

Topology and affinity on multicore

> **Hands-on: Pinning**

Case study: The Multi-Zone NAS Parallel Benchmarks

Hands-on: Masteronly hybrid Jacobi

Overlapping communication and computation

Communication overlap with OpenMP taskloops

Hands-on: Taskloop-based hybrid Jacobi

Main advantages, disadvantages, conclusions

Hands-On #1

he-hy - Hello Hybrid! - pinning

5. MPI-pure MPI: compile and run the MPI "Hello world!" program (pinning)
6. MPI+OpenMP:: **:TODO:** compile and run the Hybrid "Hello world!" program
7. MPI+OpenMP: **:TODO:** how to do pinning

→ see: **TODO.README**

Programming models

- MPI + OpenMP

Hands-On #3

Masteronly hybrid Jacobi

General considerations

How to compile, link, and run

Hands-on: Hello hybrid!

System topology, ccNUMA, and memory bandwidth

Memory placement on ccNUMA systems

Topology and affinity on multicore

Hands-on: Pinning

Case study: The Multi-Zone NAS Parallel Benchmarks

> **Hands-on: Masteronly hybrid Jacobi**

Overlapping communication and computation

Communication overlap with OpenMP taskloops

Hands-on: Taskloop-based hybrid Jacobi

Main advantages, disadvantages, conclusions

Example: MPI+OpenMP-Hybrid Jacobi solver

- Source code: See <http://tiny.cc/MPIX-VSC>
- This is a Jacobi solver (2D stencil code) with domain decomposition and halo exchange
- The given code is MPI-only. You can build it with make (take a look at the `Makefile`) and run it with something like this (adapt to local requirements):

```
$ <mpirun-or-whatever> -np <numprocs> ./jacobi.exe < input
```

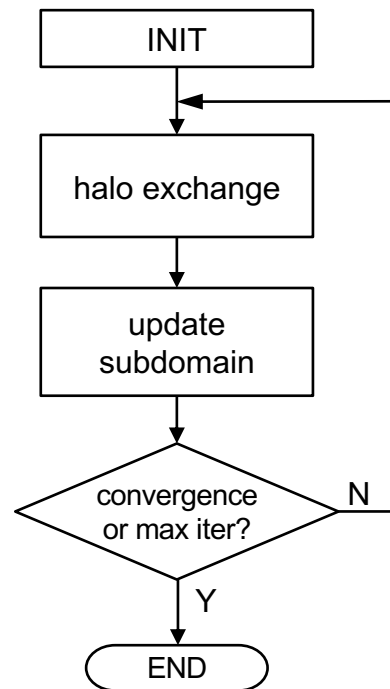
Task: parallelize it with OpenMP to get a hybrid MPI+OpenMP code, and run it effectively on the given hardware.

- Notes:
 - The code is strongly memory bound at the problem size set in the input file
 - Learn how to take control of affinity with MPI and especially with MPI+OpenMP
 - Always run multiple times and observe performance variations
 - If you know how, try to calculate the maximum possible performance and use it as a “light speed” baseline

<http://tiny.cc/MPIX-VSC>

Example cont'd

- Tasks (we assume N_c cores per CPU socket):
 - Run the MPI-only code on one node with $1, \dots, N_c, \dots, 2 * N_c$ processes (1 full node) and observe the achieved performance behavior
 - Parallelize appropriate loops with OpenMP
 - Run with OpenMP and 1 MPI process (“OpenMP-only”) on $1, \dots, N_c, \dots, 2 * N_c$ cores, compare with MPI-only run
 - Run hybrid variants with different MPI vs. OpenMP ratios
- Things to observe
 - Run-to-run performance variations
 - Does the OpenMP/hybrid code perform as well as the MPI code? If it doesn't, fix it!



<http://tiny.cc/MPIX-VSC>

Programming models

- MPI + OpenMP

Overlapping Communication and Computation

General considerations

How to compile, link, and run

Hands-on: Hello hybrid!

System topology, ccNUMA, and memory bandwidth

Memory placement on ccNUMA systems

Topology and affinity on multicore

Hands-on: Pinning

Case study: The Multi-Zone NAS Parallel Benchmarks

Hands-on: Masteronly hybrid Jacobi

> **Overlapping communication and computation**

Communication overlap with OpenMP taskloops

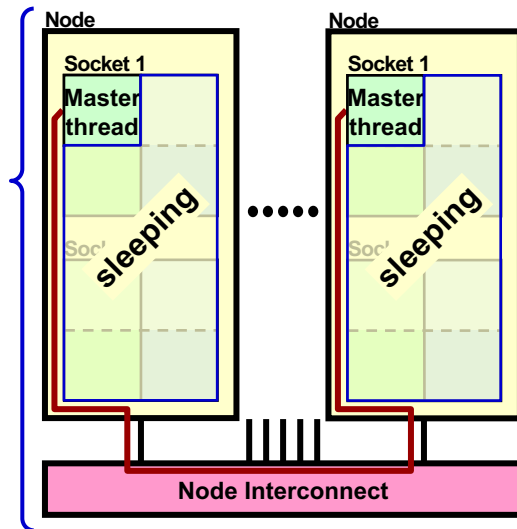
Hands-on: Taskloop-based hybrid Jacobi

Main advantages, disadvantages, conclusions

Sleeping threads with masteronly style

```
for (iteration ....)
{
  #pragma omp parallel
  numerical code
  /* end parallel */

  /* on master only */
  MPI_Send(halos);
  MPI_Recv(halos);
} /*end for loop*/
```

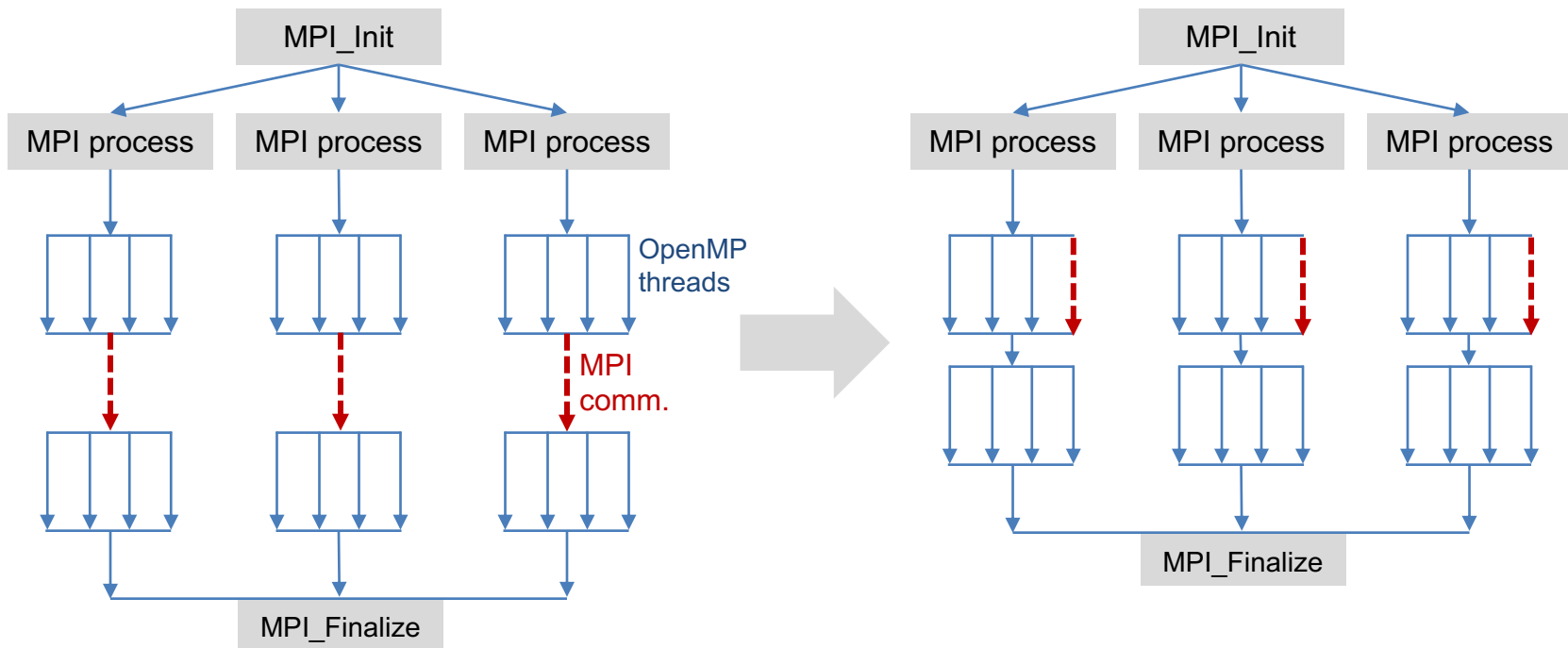


- **Problem:**
 - Sleeping threads are wasting CPU time
- **Solution:**
 - **Overlapping** of computation and communication
- **Limited benefit:**
 - **Best case:** reduces communication overhead from 50% to 0%
→ speedup of **2x**
 - Usual case of 20% to 0%
→ speedup of **1.25x**
 - Requires significant work → later

Nonblocking vs. threading for overlapped comm.

- Why not use **nonblocking** calls?
 - **Asynchronous progress not guaranteed**
 - **Options** (implementation dependent):
 - Communication offload to NIC
 - Additional internal progress thread (MPI_ASYNC... with MPICH)
 - Intranode and internode communication may be handled very differently
- **Using threading** for communication **overlap**
 - One or more threads/tasks handles communication, rest of team “do the work”
 - **How to organize the work** sharing among all threads?
 - Non-communicating threads
 - Communicating threads after communication is over
 - Not all of the work can usually be overlapped → see next slide

Using threading/tasking for comm. overlap

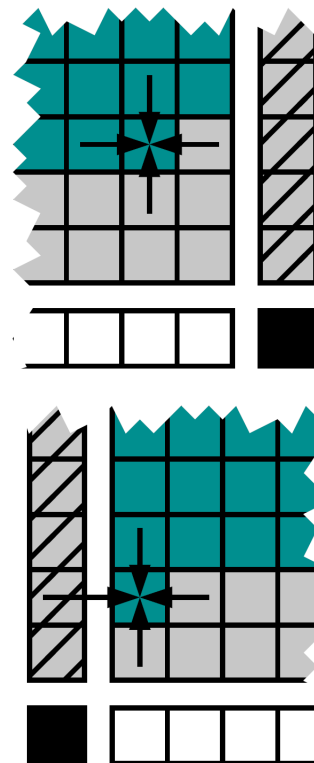


Explicit overlapping of communication and computation

The basic principle appears simple:

```
#pragma omp parallel
{
  // ... do other parallel work
  if (thread_ID < 1) {
    MPI_Send/Recv ... // comm. halo data
  } else {
    // Work on data that is independent
    // of halo data
  }
} // end omp parallel

// Now work on data that needs the
// halo data (all threads)
```



Overlapping communication with computation

Three problems:

- **Application problem:** separate application into
 - code that can run before the halo data is received
 - code that needs halo data
 - **May be hard to do**
- **Thread-rank problem:** distinguish comm. / comp. via thread ID
 - Work sharing and load balancing is harder
 - Options
 - Fully manual work distribution
 - Nested parallelism
 - Tasking & taskloops
 - Partitioned comm (MPI-4.0)
- **Optimal memory placement** on ccNUMA may be difficult

error-prone & clumsy

```
if (my_thread_ID < 1) {
    MPI_Send/Recv
} else {
    my_thread_range=(high-low-1)/(num_threads-1)+1;
    my_thread_low=low+(my_thread_ID-1)*my_thread_range;
    my_thread_high=low+(my_thread_ID-1+1)
        *my_thread_range;
    my_thread_high=min(high, my_thread_high);
    for (i=my_thread_low; i<my_thread_high; i++) {
        ...
    }
}
```

Programming models

- MPI + OpenMP

Communication overlap with OpenMP taskloops

General considerations

How to compile, link, and run

Hands-on: Hello hybrid!

System topology, ccNUMA, and memory bandwidth

Memory placement on ccNUMA systems

Topology and affinity on multicore

Hands-on: Pinning

Case study: The Multi-Zone NAS Parallel Benchmarks

Hands-on: Masteronly hybrid Jacobi

Overlapping communication and computation

> **Communication overlap with OpenMP taskloops**

Hands-on: Taskloop-based hybrid Jacobi

Main advantages, disadvantages, conclusions

OpenMP `taskloop` Directive – Syntax

- Immediately following loop executed in **several tasks**

- **Not a work-sharing directive!**

- Should be executed only by one thread!

A task can be run by any thread, across NUMA nodes
→ 😞 **perfect first touch impossible!**

- Fortran:

```
!$OMP taskloop [ clause [ [ , ] clause ] ... ]
```

```
do_loop
```

```
[ !$OMP end taskloop [ nowait ] ]
```

Loop iterations must be independent, i.e., they can be executed in parallel

- If used, the `end do` directive must appear immediately after the end of the loop

- C/C++:

```
#pragma omp taskloop [ clause [ [ , ] clause ] ... ] new-line
```

```
for-loop
```

- The corresponding *for-loop* must have canonical shape → next slide

OpenMP `taskloop` Directive – Details

- *clause* can be one of the following:

- `if ([taskloop:] scalar-expr)` [a task clause]
- `shared (list)` [a task clause]
- `private (list) , firstprivate (list)` [a do/for clause] [a task clause]
- `lastprivate (list)` [a do/for clause]
- `default (shared | none | ...)` [a task clause]
- `collapse (n)` [a do/for clause]
- `grainsize (grain-size)` ← Mutually exclusive
- `num_tasks (num-tasks)` ← Mutually exclusive
- `untied, mergeable` [a task clause]
- `final (scalar-expr) , priority (priority-value)` [a task clause]
- `nogroup`
- `reduction (operator:list)` ← [a do/for clause]

Since
OpenMP 5.0!

- do/ for clauses that are **not** valid on a `taskloop`:

- `schedule (type [, chunk]) , nowait`
- `linear (list [: linear-step]) , ordered [(n)]`

OpenMP single & taskloop Directives

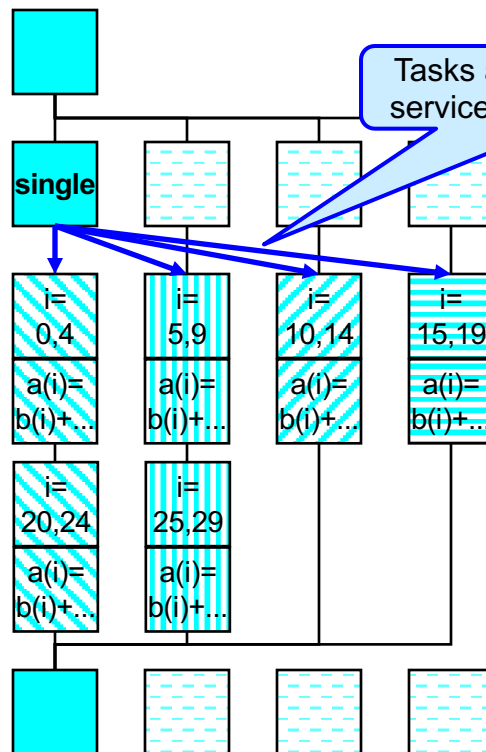
C/C++

C / C++:

```
#pragma omp parallel
{
    #pragma omp single
    {
        #pragma omp taskloop
        for (i=0; i<30; i++)
            a[i] = b[i] + f * (i+1);
    }
} /*omp end single*/
} /*omp end parallel*/
```

A lot more tasks than threads may be produced to achieve a good load balancing

Tasks are queued and then serviced by team of threads



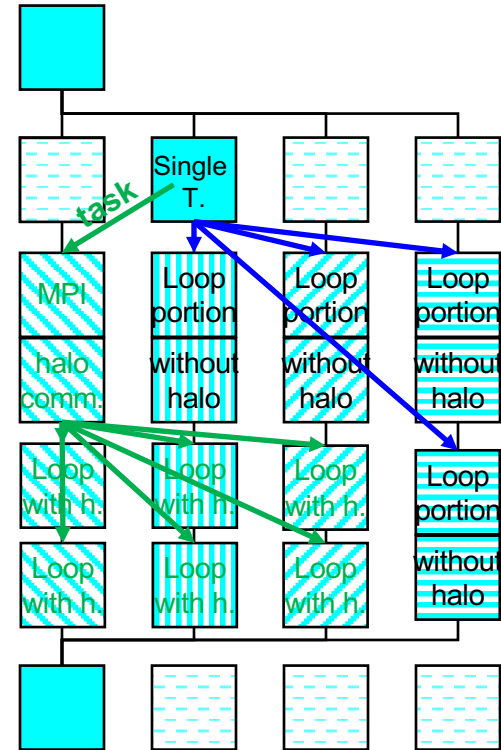
Comm. overlap with task & taskloop Directives – C/C++

C/C++

```
#pragma omp parallel
{
  #pragma omp single
  {
    #pragma omp task
    { // MPI halo communication:
      MPI_Send/Recv...
      // numerical loop using halo data:
      #pragma omp taskloop
      for (i=0; i<100; i++)
        a[i] = b[i] + b[i-1] + b[i+1] + b[i-2]...;
    } /*omp end of halo task */

    // numerical loop without halo data:
    #pragma omp taskloop
    for (i=100; i<10000; i++)
      a[i] = b[i] + b[i-1] + b[i+1] + b[i-2]...;
    ...
  } /*omp end single */
} /*omp end parallel*/
```

Number of tasks may be influenced with grainsize or num_tasks clauses



Partitioned Point-to-Point Communication

- New in MPI-4.0:
Partitioned communication is “partitioned” because it allows for multiple contributions of data to be made, potentially, from multiple actors (e.g., threads or tasks) in an MPI process to a single communication operation.
- A **point-to-point** operation (i.e., send or receive)
 - can be **split** into **partitions**,
 - and each **partition** is **filled** and then “sent” with **MPI_Pready** by a thread;
 - same for receiving
- Technically provided as a **new form** of **persistent** communication.

Programming models

- MPI + OpenMP

Hands-On #4

Taskloop-based hybrid Jacobi

→ optional...

General considerations

How to compile, link, and run

Hands-on: Hello hybrid!

System topology, ccNUMA, and memory bandwidth

Memory placement on ccNUMA systems

Topology and affinity on multicore

Hands-on: Pinning

Case study: The Multi-Zone NAS Parallel Benchmarks

Hands-on: Masteronly hybrid Jacobi

Overlapping communication and computation

Communication overlap with OpenMP taskloops

> **Hands-on: Taskloop-based hybrid Jacobi**

Main advantages, disadvantages, conclusions

Programming models

- MPI + OpenMP

Main advantages,
disadvantages,
conclusions

General considerations

How to compile, link, and run

Hands-on: Hello hybrid!

System topology, ccNUMA, and memory bandwidth

Memory placement on ccNUMA systems

Topology and affinity on multicore

Hands-on: Pinning

~~Case study: The Multi-Zone NAS Parallel Benchmarks~~

Hands-on: Masteronly hybrid Jacobi

Overlapping communication and computation

Communication overlap with OpenMP taskloops

Hands-on: Taskloop-based hybrid Jacobi

> **Main advantages, disadvantages, conclusions**

Load Balancing with hybrid programming

- On same or different level of parallelism
- **OpenMP** enables
 - cheap **dynamic** and **guided** load-balancing
 - via a parallelization option (clause on `omp for / do` directive)
 - without additional software effort
 - without explicit data movement
- On **MPI** level
 - **Dynamic load balancing** requires moving of parts of the data structure through the network
 - Significant runtime overhead
 - Complicated software → rarely implemented
- **MPI & OpenMP**
 - Simple static load balancing on MPI level, dynamic or guided on OpenMP level

```
#pragma omp parallel for schedule(dynamic)
for (i=0; i<n; i++) {
    /* poorly balanced iterations */ ...
}
```

} medium-quality,
} cheap implementation

MPI+OpenMP: Main advantages

- **Increase parallelism**
 - Scaling to higher number of cores
 - Adding OpenMP with incremental additional parallelization
- **Lower memory requirements** due to smaller number of MPI processes
 - Reduced amount of application halos & replicated data
 - Reduced size of MPI internal buffer space
 - Very important on systems with many cores per node
- **Lower communication overhead (possibly)**
 - Few multithreaded MPI processes vs many single-threaded processes
 - Fewer number of calls and smaller amount of data communicated
 - Topology problems from pure MPI are solved
(was application topology versus multilevel hardware topology)
- Provide for **flexible load-balancing** on coarse and fine levels
 - Smaller #of MPI processes leave room for assigning workload more evenly
 - MPI processes with higher workload could employ more threads

Additional advantages when overlapping communication and computation:

- No sleeping threads

MPI+OpenMP: Main disadvantages & challenges

- **Non-Uniform Memory Access:**
 - Not all memory access is equal: ccNUMA locality effects
 - Penalties for access across NUMA domain boundaries
 - First touch is needed for *more than one NUMA domain per MPI process*
 - Alternative solution:
One MPI process on each NUMA domain (i.e., chip)
- **Multicore / multsocket anisotropy effects**
 - Bandwidth bottlenecks, shared caches
 - Intra-node MPI performance: Core ↔ core vs. socket ↔ socket
 - OpenMP loop overhead
- **Amdahl's law** on both, MPI and OpenMP level
- Complex thread and process **pinning**

Masteronly style (i.e., MPI outside of parallel regions)

- **Sleeping threads**

Additional disadvantages when overlapping communication and computation:

- **High programming overhead**
- **OpenMP is only partially prepared for this programming style → taskloop directive**

Questions addressed in this tutorial

- What is the **performance impact** of system topology? It's massive
- How do I **map my programming model** on the system to my advantage?
 - How do I do the **split** into MPI+X? Problem dependent
 - Where do my processes/threads run? How do I **take control**?
 - **Where is my data**? ccNUMA first-touch placement Process/thread affinity
 - How can I **minimize communication overhead**?
- How does hybrid programming help with **typical HPC problems**?
 - Can it **reduce communication overhead**?
 - Can it **reduce replicated data**?
- How can I leverage **multiple accelerators**?
 - What are typical **challenges**?

Conclusions

Major advantages of hybrid MPI+OpenMP

In principle, none of the programming models perfectly fits to clusters of SMP nodes

Major advantages of MPI+OpenMP:

- Only one level of sub-domain “surface-optimization”:
 - SMP nodes, or
 - Sockets or NUMA domains
- **Second level of parallelization**
 - Application may scale to more cores
- Smaller number of MPI processes implies:
 - Reduced size of MPI internal buffer space
 - Reduced space for replicated user-data

**Most important arguments
on many-core systems**

- **Reduced communication overhead**
 - No intra-node communication
 - Longer messages between nodes and fewer parallel links may imply better bandwidth
- **“Cheap” load-balancing methods on OpenMP level**
 - Application developer can split the load-balancing issues between course-grained MPI and fine-grained OpenMP

Disadvantages of MPI+OpenMP

- Using OpenMP
 - may prohibit compiler optimization
 - **may cause significant loss of computational performance**
- Thread fork / join overhead
- On ccNUMA SMP nodes:
 - Loss of performance due to **missing memory page locality** or **missing first touch strategy**
 - E.g., with the MASTERONLY scheme:
 - One thread produces data
 - Master thread sends the data with MPI
 - data may be internally communicated from one NUMA domain to the other one
- Amdahl's law for **each** level of parallelism
- Using MPI-parallel application libraries? → **Are they prepared for hybrid?**
- Using thread-local application libraries? → **Are they thread-safe?**

MPI+OpenMP versus MPI+MPI-3.0 shared memory

MPI+3.0 shared memory

- **Pro:** Thread-safety is not needed for libraries.
- **Con:** No work-sharing support as with OpenMP directives.
- **Pro:** Replicated data can be reduced to one copy per node:
May be helpful to save memory, **if pure MPI scales in time, but not in memory**
- Substituting intra-node communication by shared memory loads or stores has only limited benefit (and only on some systems), especially if the communication time is dominated by inter-node communication
- **Con:** No reduction of MPI ranks
→ no reduction of MPI internal buffer space
- **Con:** Virtual addresses of a shared memory window may be different in each MPI process
→ no binary pointers
→ i.e., linked lists must be stored with offsets rather than pointers

Conclusions

- Future hardware will be more complicated
 - Heterogeneous → GPU, FPGA, ...
 - Node-level ccNUMA is here to stay, but will only be one of your problems
 -
- High-end programming → more complex → many pitfalls
- Medium number of cores → more simple (**#cores / SMP-node** still grows)
- **MPI + OpenMP → workhorse on large systems**
 - Major pros: **reduced memory needs** and **second level of parallelism**
- **MPI + MPI shared memory → only for special cases and medium #processes**
- Pure MPI communication → still viable if it does the job
- OpenMP only → on large ccNUMA nodes (almost gone in HPC)

Thank you for your interest

Q & A

Please fill out the feedback sheet – Thank you



Programming models

- MPI + Accelerator

General considerations [88](#)

OpenMP offloading [95](#)

Advantages & main challenges [106](#)

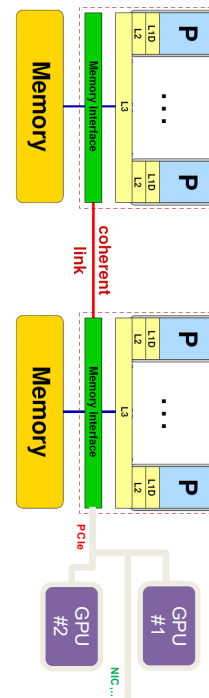
Accelerator programming: Bottlenecks reloaded

Example: 2-socket Intel “Ice Lake” (2x36 cores) node with two NVIDIA A100 GPGPUs (PCIe 4)

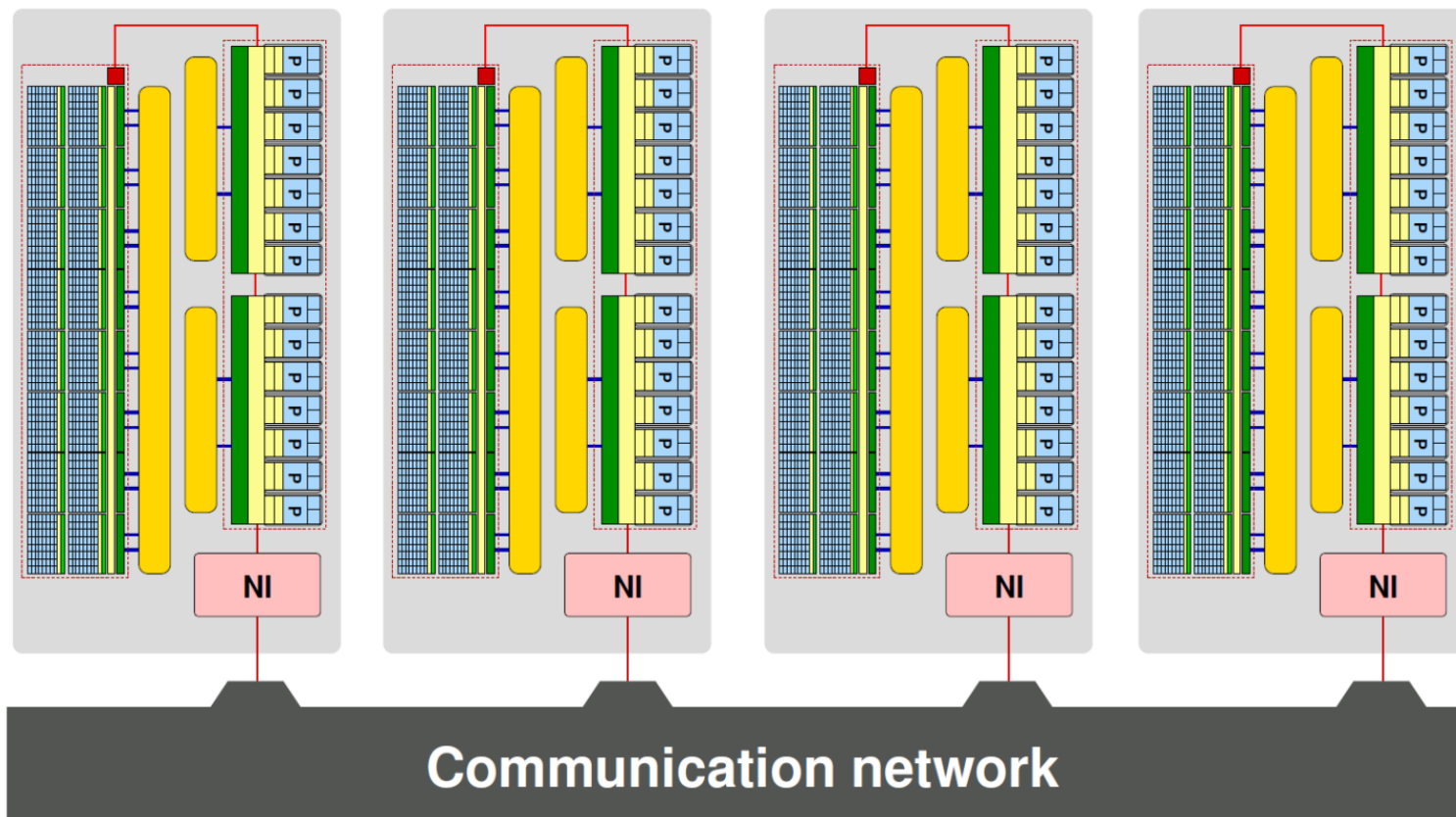
	per GPGPU	per CPU
DP peak performance	9.7 Tflop/s ← 4x	2.3 Tflop/s
Machine balance	0.11 B/F	0.10 B/F
eff. memory (HBM) bandwidth	1300 Gbyte/s ← 8x	170 Gbyte/s
inter-device bandwidth (PCIe)	≈ 30 Gbyte/s	
inter-device bandwidth (NVlink)	> 500 Gbyte/s	

→ Speedups can only be attained if communication overheads are under control

→ Basic estimates help



Accelerator + MPI: How does the data get from A to B?



DEVANA's Multi-GPU nodes: nvidia-smi tool

```
+-----+
| NVIDIA-SMI 525.85.05   Driver Version: 525.85.05   CUDA Version: 12.0   |
+-----+
| GPU  Name                Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf    Pwr:Usage/Cap|      Memory-Usage | GPU-Util  Compute M. |
|                                           MIG M.         |
+-----+-----+
|   0   NVIDIA A100-SXM...  On      | 00000000:17:00:0 Off  |           0         |
| N/A   29C    P0     48W / 400W |  0MiB / 40960MiB |    0%      Default  |
|                                           Disabled        |
+-----+-----+
|   1   NVIDIA A100-SXM...  On      | 00000000:31:00:0 Off  |           0         |
| N/A   30C    P0     54W / 400W |  0MiB / 40960MiB |    0%      Default  |
|                                           Disabled        |
+-----+-----+
|   2   NVIDIA A100-SXM...  On      | 00000000:B1:00:0 Off  |           0         |
| N/A   28C    P0     52W / 400W |  0MiB / 40960MiB |    0%      Default  |
|                                           Disabled        |
+-----+-----+
|   3   NVIDIA A100-SXM...  On      | 00000000:CA:00:0 Off  |           0         |
| N/A   29C    P0     50W / 400W |  0MiB / 40960MiB |    0%      Default  |
|                                           Disabled        |
+-----+-----+

+-----+
| Processes:                                                       |
| GPU  GI  CI           PID   Type   Process name                      GPU Memory |
|      ID  ID                                   |             Usage          |
+-----+-----+
| No running processes found                                     |
+-----+-----+
```

DEVANA's Multi-GPU nodes: topology and i/connect

```
trainer2@n141 ~ > nvidia-smi topo -m
      GPU0 GPU1 GPU2 GPU3 NIC0 NIC1 NIC2 CPU Affinity NUMA Affinity
GPU0      X  NV4  NV4  NV4  NODE NODE NODE   0-31         0
GPU1     NV4   X  NV4  NV4  NODE NODE NODE   0-31         0
GPU2     NV4  NV4   X  NV4  SYS  SYS  SYS   32-63        1
GPU3     NV4  NV4  NV4   X  SYS  SYS  SYS   32-63        1
NIC0     NODE NODE SYS  SYS   X  NODE NODE
NIC1     NODE NODE SYS  SYS  NODE  X  PIX
NIC2     NODE NODE SYS  SYS  NODE PIX   X
```

Legend:

X = Self

SYS = Connection traversing PCIe as well as the SMP interconnect between NUMA nodes (e.g., QPI/UPI)

NODE = Connection traversing PCIe as well as the interconnect between PCIe Host Bridges within a NUMA node

PHB = Connection traversing PCIe as well as a PCIe Host Bridge (typically the CPU)

PXB = Connection traversing multiple PCIe bridges (without traversing the PCIe Host Bridge)

PIX = Connection traversing at most a single PCIe bridge

NV# = Connection traversing a bonded set of # NVLinks

Questions to ask

- Is the MPI implementation CUDA aware?
 - **Yes:** Can use device pointers in MPI calls
 - **No:** Explicit DtoH/HtoD buffer transfers required
 - Copying to consecutive halo buffers may still be necessary
- Is NVLink available?
 - **Yes:** Direct GPU-GPU MPI communication with MPI
 - Supported by: P100, V100, A100, H100
 - **No:** copies via host (even with NVIDIA GPUDirect)
- **Unified Memory** or explicit DtoH/HtoD transfers?
 - UM: Transparent sharing of host and device memory
- Actual bandwidths and latencies?
 - Highly system and implementation dependent!

See also:
<https://www.fz-juelich.de/en/ias/isc/news/events/seminars/msa-seminar/2020-01-21-cuda-aware-mpi>

Options for hybrid accelerator programming

multicore host
MPI
MPI+MPI3 shmem ext.
MPI+threading (OpenMP , pthreads, TBB,...)
threading only
PGAS (CAF, UPC,...)
...

accelerator
CUDA, HIP
OpenCL
OpenACC
OpenMP 4.0++
special purpose
...

Which model/combination is the best?

→ the one that allows you to address the relevant hardware bottleneck(s)

Programming models - MPI + Accelerator

General considerations [88](#)

OpenMP offloading [95](#)

Advantages & main challenges [106](#)

What is OpenMP offloading?

- “Everybody knows OpenMP”
- API that supports offloading of loops and regions of code (e.g. loops) from a host CPU to an attached accelerator in C, C++, and Fortran
- Set of **compiler directives**, **run-time routines**, and **environment variables**
- Simple programming model for using accelerators (focus on GPGPUs)
- **Memory model:**
 - Host CPU + Device may have completely separate memory; Data movement between host and device performed by host via runtime calls; Memory on device may not support memory coherence between execution units or need to be supported by explicit barrier
- **Execution model:**
 - Compute intensive code regions offloaded to the device, executed as kernels ; Host orchestrates data movement, initiates computation, waits for completion; Support for multiple levels of parallelism, including SIMD



A very simple OpenMP example (nvc 23.1-0): Vector Triad

```
int main ()
{
    double* restrict a = malloc(nsize * sizeof(double));
    double* restrict b = malloc(nsize * sizeof(double));
    double* restrict c = malloc(nsize * sizeof(double));
    double* restrict d = malloc(nsize * sizeof(double));
;
    ...
    #pragma omp target enter data map(to:a[0:nsize], b[0:nsize], c[0:nsize])
        compute(a ,b , c ,d ,N);
    ...
}
```

data
mgmt

```
void compute (double *restrict a , double *b,...) {
    #pragma omp target teams distribute\
        parallel for simd
    for(int i=0; i<N ; ++i) {
        a[i] = b[i] + c [i] * d[i];
    }
}
```

execution

```
nvc -g -O3 -mp=gpu -gpu=managed -Minfo -c triad.F90
17, #omp target teams distribute parallel for simd
17, Generating "nvkernel_main_F1L17_2" GPU kernel
19, Loop parallelized across teams and threads(128),
schedule(static)
17, Generating target enter data map(to:
c[:nsize],b[:nsize],a[:nsize])
25, #omp target teams distribute parallel for simd
25, Generating "nvkernel_main_F1L25_4" GPU kernel
28, Loop parallelized across teams and threads(128),
schedule(static)
38, Generating target exit data map(from:
c[:nsize],b[:nsize],a[:nsize])
```

Example: 2D Laplace equation

We want to solve this:

$$\partial_{xx}u(x, y) + \partial_{yy}u(x, y) = 0,$$

$$u(x, y) \in [0,1] \times [0,1] \setminus \partial\Omega$$

subject to the boundary conditions:

$$u(x, 0) = u(x, 1) = x$$

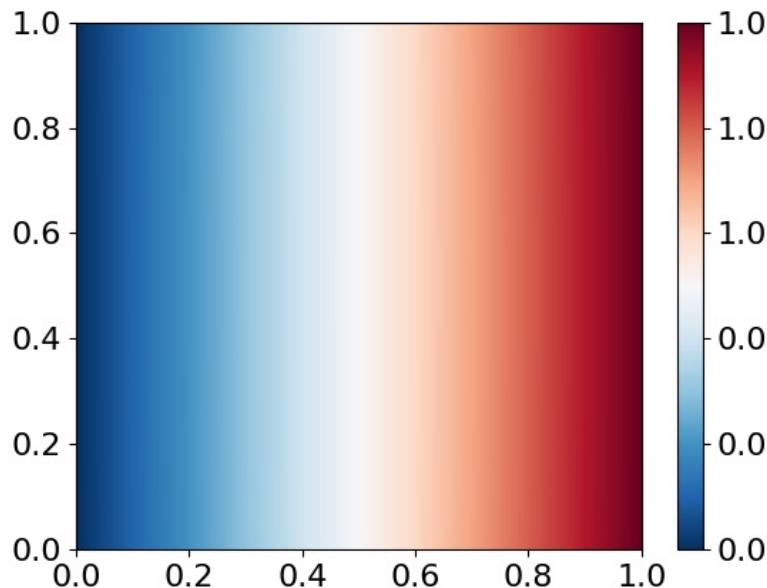
$$u(0, y) = 0$$

$$u(1, y) = 1$$

numerically, using finite differences:

$$(\partial_{xx}u(x, y))_{ij} \approx \frac{u_{i+1,j} - 2u_{ij} + u_{i-1,j}}{\Delta x^2}.$$

Converged solution:



Example: Fortran 2D Jacobi solver offloading

Basic step:

```
allocate(a(0:ni+1,0:nj+1), b(0:ni+1,0:nj+1))
!$omp target enter data map(to:a(0:ni+1,0:nj+1), b(0:ni+1,0:nj+1))
!$omp target teams distribute parallel do
do j = 1, nj
  do i = 1, ni
    b(i,j) = (a(i,j-1) + a(i,j+1) + a(i-1,j) + a(i+1,j)) / 4d0
  end do
end do
call swap(b,a)
```

And check for the convergence:

```
error = 0d0
!$omp target teams distribute parallel do simd reduction(max:error)
do j = 1, nj
  do i = 1, ni
    error = max(error, abs(a(i,j)-b(i,j)))
  end do
end do
```

Example: multi-GPU offloading with MPI; one node

Typical MPI 1D domain decomposition: distribute **a** and **b** over MPI ranks

```
allocate(a(0:ni+1,s-1:e+1), b(0:ni+1,s-1:e+1))
!$omp target enter data map(to:a(0:ni+1,s-1:e+1), b(0:ni+1,s-1:e+1))
!$omp target teams distribute parallel do
do j = s, e
  do i = 1, ni
    b(i,j) = (a(i,j-1) + a(i,j+1) + a(i-1,j) + a(i+1,j)) / 4d0
  end do
end do
call swap(b,a)
```



Example: multi-GPU offloading with MPI; one node

Typical MPI 1D domain decomposition: distribute **a** and **b** over MPI ranks and send the rank's portion of the data to the corresponding GPU

```
gpuid = mpirank
allocate(a(0:ni+1,s-1:e+1), b(0:ni+1,s-1:e+1))
!$omp target enter data map(to:a(0:ni+1,s-1:e+1), b(0:ni+1,s-1:e+1)) device(gpuid)
!$omp target teams distribute parallel do device(gpuid)
do j = s, e
  do i = 1, ni
    b(i,j) = (a(i,j-1) + a(i,j+1) + a(i-1,j) + a(i+1,j)) / 4d0
  end do
end do
```



Example: multi-GPU offloading with MPI; one node

Exchange halos (`MPI_SENDRECV` or whatever you like):

```
call MPI_CART_CREATE( MPI_COMM_WORLD, 1, [mpisize], [.false.], .true.,  
comm1d, mpierr)  
call MPI_COMM_RANK(comm1d, mpirank, mpierr)  
call MPI_CART_SHIFT( comm1d, 0, 1, left, right, mpierr)  
8<-----  
call MPI_SENDRECV(                                     &  
    a(1,e), nx, MPI_DOUBLE_PRECISION, right, 0, &  
    a(1,s-1), nx, MPI_DOUBLE_PRECISION, left, 0, &  
    comm1d, MPI_STATUS_IGNORE, ierr)  
call MPI_SENDRECV(                                     &  
    a(1,s), nx, MPI_DOUBLE_PRECISION, left, 1, &  
    a(1,e+1), nx, MPI_DOUBLE_PRECISION, right, 1, &  
    comm1d, MPI_STATUS_IGNORE, ierr)  
8<-----
```

Example: multi-GPU offloading with MPI; multi-node

Each compute node sees only its own GPUs (4 on DEVANA). We split the communicator further to get node's local ranks:

```
call MPI_COMM_SPLIT_TYPE(comm1d, MPI_COMM_TYPE_SHARED, mpirank, &
                             MPI_INFO_NULL, commlocal, mpierr)
call MPI_COMM_RANK(commlocal, lrank, mpierr)
gpuid = lrank
```



Job submission on multi-GPU clusters

```
trainer2@login02 ~ > cat onenode.sh
#!/bin/bash
#BATCH --time=00:05:00

#SBATCH --nodes=1
#SBATCH --ntasks-per-node=4
#SBATCH --cpus-per-task=1

#SBATCH --partition=ngpu
#SBATCH --job-name=mpiompgpu_onenode
#SBATCH --err=mpiompgpu_onenode.err
#SBATCH --out=mpiompgpu_onenode.out
#SBATCH --gres=gpu:4

module load nvhpc/23.1 GCC/11.3.0
mpirun -np 4 ./jacobi_mpi_gpu
```

```
trainer2@login02 ~ > cat twonodes.sh
#!/bin/bash
#SBATCH --time=00:05:00

#SBATCH --nodes=2
#SBATCH --ntasks-per-node=4
#SBATCH --cpus-per-task=1

#SBATCH --partition=ngpu
#SBATCH --job-name=mpiompgpu_twonodes
#SBATCH --err=mpiompgpu_twonodes.err
#SBATCH --out=mpiompgpu_twonodes.out
#SBATCH --gres=gpu:4

module load nvhpc/23.1 GCC/11.3.0
mpirun -np 8 ./jacobi_mpi_gpu
```


Example: multi-GPU multi-node benchmarking

A word of caution: sometimes we have to run the benchmark for some time, discarding timings of the first half of iterations.

Benchmarking 2D Laplace, 9600^2 points on DEVANA (4 A100 per node):

N GPUs	Execution time, s
1	12.81
2	6.78
4	4.01
8	2.71

Programming models - MPI + Accelerator

General considerations [88](#)

OpenMP offloading [95](#)

Advantages & main challenges [106](#)

MPI+Accelerators: Main advantages

- Hybrid **MPI/OpenMP** can leverage accelerators and yield performance increase over pure MPI on multicore
- Compiler/**pragma-based API** provides relatively easy way to use co-processors
- OpenMP 4.0/4.5/5.1 extensions provide flexibility to use a wide range of heterogeneous co-processors (GPU, APU, heterogeneous many-core types)

MPI+Accelerators: Main challenges

- Considerable **implementation effort for basic usage**, depending on complexity of the application
- Efficient usage of pragmas requires **good understanding of performance issues**
 - Performance is not only about code; **data structures** can be decisive as well
- Support for accelerator pragmas still restricted to certain environments
 - **NVIDIA GPUs** have best support

Questions addressed in this tutorial

- What is the **performance impact** of system **topology**?
- How do I **map my programming model** on the system to my advantage?
 - How do I do the **split** into MPI+X?
 - Where do my processes/threads run? How do I **take control**?
 - **Where** is my **data**?
 - How can I **minimize communication** overhead?
- How does hybrid programming help with **typical HPC problems**?
 - Can it **reduce communication** overhead?
 - Can it **reduce replicated data**?
- How can I leverage **multiple accelerators**?
 - What are typical **challenges**?

Data structures are decisive,
inter-device communication
support varies

Thank you for your interest!

TREX Workshop: Code Tuning for the Exacale @ Bratislava, June 5, 2023