JUWELS & JURECA
Tuning for the platform

Usage of ParaStation MPI

November 14th, 2024

Patrick Küven

ParTec AG

JUWELS & JURECA

Tuning for the platform

1. ParaStation MPI
2. Compiling your program
3. Running your program
4. Tuning parameters
5. Resources

History of ParaStation

- 1995: University project (→ University of Karlsruhe)
- 2005: Open source (→ ParaStation Consortium)
- Since 2004: Cooperation with JSC
  - various precursor clusters
  - DEEP-System (MSA prototype)
  - JuRoPA3 (J3)
  - JUAMS
  - JURECA (Cluster/Booster)
  - JUWELS (Cluster/Booster)
  - JURECA DC
  - JUPITER

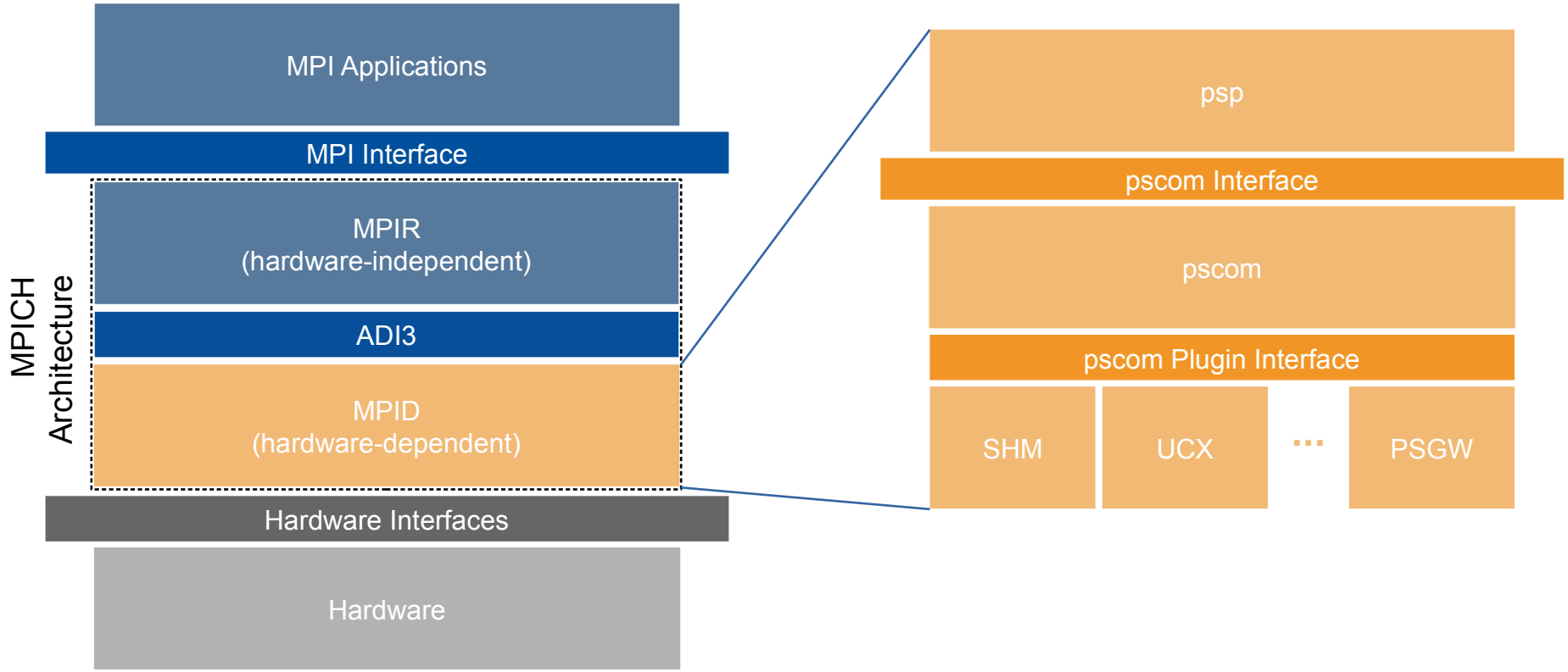
ParaStation MPI

- based on MPICH (4.1.1)
  - supports all MPICH tools (tracing, debugging, …)
- proven to scale up to 3,300 nodes and 136.800 procs per job running ParaStation MPI
  - JURECA DC: No. 102 (Top500 Jun 2024), No. 45 (Green500 Jun 2024)
  - JUWELS Booster: No. 21 (Top500 Jun 2024), No. 44 (Green500 Jun 2024)
  - JEDI: No. 189 (Top500 Jun 2024), No. 1 (Green500 Jun 2024)
- supports a wide range of interconnects, even in parallel
  - InfiniBand on JURECA DC and JUWELS
  - Omni-Path on JURECA Booster (deprecated)
  - Extoll on DEEP projects research systems (deprecated)
- tight integration with Cluster Management (e.g. healthcheck)
- MPI libraries for several compilers
  - especially for GCC and Intel

ParaStation MPI: Modularity

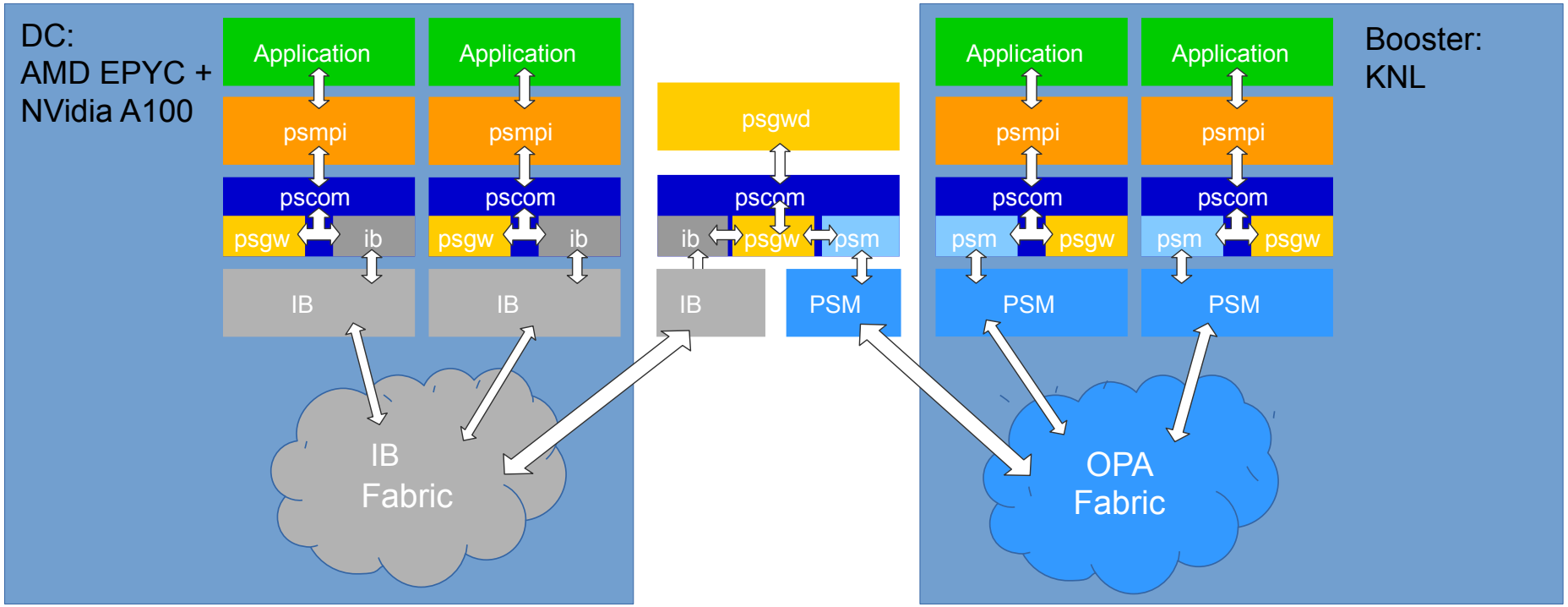
- 2 or more different modules with different hardware
- a job can execute dynamically on all modules
- you can pick the best out of all the worlds in a single job
- e.g. JURECA:
  - DC: AMD EPYC + Nvidia A100 + Infiniband
  - Booster: Intel KNL + Omni-Path
- how do these modules communicate with each other?

ParaStation MPI: pscom



- low-level communication layer supporting various transports and protocols
- applications may use multiple transports at the same time

ParaStation MPI: pscom



- for the JURECA DC-Booster System, the ParaStation MPI Gateway Protocol bridges between Mellanox IB and Intel Omni-Path
- in general, the ParaStation MPI Gateway Protocol can connect any two low-level networks supported by pscom
- implemented using the *psgw* plugin to pscom, working together with instances of the *psgwd*

ParaStation MPI: Modular Jobs

- two processes communicate through a gateway, if they are not directly connected by a high-speed network (e.g. IB or OPA)
- static routing to choose a common gateway
- high-speed connections between processes and gateway daemons
- virtual connection between both processes through the gateway, transparent for application
- virtual connections are multiplexed through gateway connections
- further information: apps.fz-juelich.de/jsc/hps/jureca/modular-jobs.html

ParaStation MPI: CUDA awareness

- CUDA awareness supported by the following MPI APIs
  - Point-to-point (e.g. `MPI_SEND`, `MPI_RECV`, …)
  - Collectives (e.g. `MPI_Allgather`, `MPI_Reduce`, …)
  - One-sided (e.g. `MPI_Put`, `MPI_Get`, …)
  - Atomics (e.g. `MPI_Fetch_and_op`, `MPI_Accumulate`, …)
- CUDA awareness for all transports via staging
- CUDA optimization: UCX
- ability to query CUDA awareness at compile- and runtime

ParaStation MPI: CUDA awareness

- activate CUDA awareness by meta modules
  - default configurations

- query CUDA awareness:

```
#if defined(MPIX_CUDA_AWARE_SUPPORT) && MPIX_CUDA_AWARE_SUPPORT
printf("The MPI library is CUDA-aware\n");
#endif
```

```
if (MPIX_Query_cuda_support())
    printf("The MPI library is CUDA-aware\n");
```

```
MPI_Info_get(MPI_INFO_ENV, "cuda_aware",
                           sizeof(is_cuda_aware)-1, is_cuda_aware,
                           &api_available);
```

Compiling on JUWELS

- currently MPI-4 version (5.10.0-1) available
- single thread tasks
  - `module load Intel ParaStationMPI`
  - `module load GCC ParaStationMPI`
- multi-thread tasks (mt)
  - `module load Intel ParaStationMPI/5.10.0-1-mt`
  - no multi-thread GCC version available
- ChangeLog available with
  - `less $(dirname $(which mpicc))/../ChangeLog`
- Gnu and Intel compilers available
- module spider for getting current versions
- see also the previous talk JUWELS - Introduction

Wrapper

- Wrappers
  - `mpicc` (C)
  - `mpicxx` (C++)
  - `mpif90` (Fortran 90)
  - `mpif77` (Fortran 77)
- when using OpenMP and the need to use the „mt“ version, add
  - `-fopenmp` (GNU)
  - `-qopenmp` (Intel)

Did I use the wrapper correctly?

- libaries are linked at runtime according to **`LD_LIBRARY_PATH`**
- `ldd` shows the libraries attached to your binary
- look for ParaStation libraries

```
ldd hello_mpi:
...
libmpi.so.12 => /p/software/juwels/stages/2020/
software/psmpi/5.10.0-1-iccifort-2020.2.254-GCC-9.3.0/
lib/libmpi.so.12 (0x000015471ea43000)
...

vs.

...
libmpi.so.12 => /p/software/juwels/stages/2020/
software/psmpi/
5.10.0-1-iccifort-2020.2.254-GCC-9.3.0-mt/lib/
libmpi.so.12 (0x000014f110e58000)
...
```

JUWELS: start via srun

- use **`srun`** to start MPI processes
- **`srun -N <nodes> -n <tasks>`** spawns task
  - directly (**`-A <account>`**)
  - via **`salloc`**
  - from batch script via **`sbatch`**
- exports full environment
- stop interactive run with (consecutive) **`^C`**
  - passed to all tasks
- no manual clean-up needed
- you can log into nodes which have an allocation/running job step
  - **`squeue -u <user>`**
  - **`sgoto <jobid> <nodenumber>`**
    - **e.g.** **`sgoto 2691804 0`**

hello_mpi.c

```c
/* C Example */
#include <stdio.h>
#include <mpi.h>

int main (int argc, char **argv) {

  int numprocs, rank, namelen;

  char processor_name[MPI_MAX_PROCESSOR_NAME];

  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
  MPI_Get_processor_name (processor_name, &namelen);
  printf ("Hello world from process %d of %d on %s\n",
          rank, numprocs, processor_name);
  MPI_Finalize ();
  return 0;
}
```

Running on JUWELS (Intel chain)

- `module load Intel`
- `module load ParaStationMPI`
- `mpicc -O3 -o hello_mpi hello_mpi.c`
- **Interactive:**
- `salloc -N 2 -A partec # get an allocation`
- `srun -n 2 ./hello_mpi`

```
Hello world from process 0 of 2 on jwc08n188.juwels
Hello world from process 1 of 2 on jwc08n194.juwels
```

- **Batch:**
- `sbatch ./hello_mpi.sh`

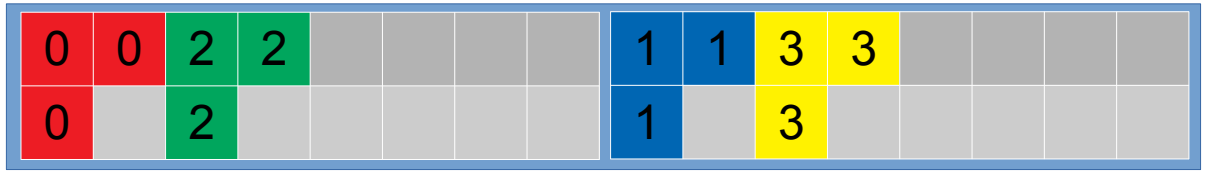
- Increase verbosity:
  - `PSP_DEBUG=[1,2,3,…] srun -n 2 ./hello_mpi`

Process Placement

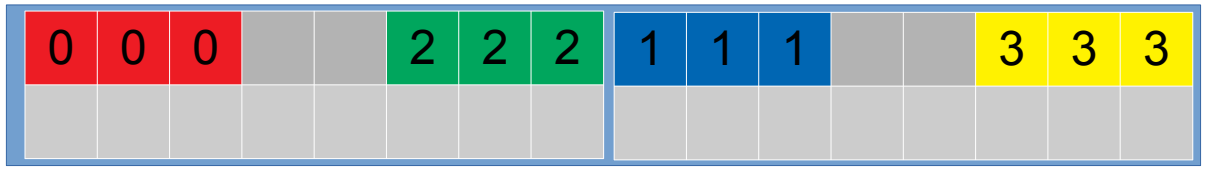
- ParaStation process pinning:
  - avoid task switching
  - make better use of CPU cache and memory bandwidth
- JUWELS is pinning by default:
  - so `--cpu-bind=threads` may be omitted
- manipulate pinning:
  - e.g. for „large memory / few task“ applications
- manipulate via
  - `--cpu-bind=threads|sockets|cores|mask_cpu:<mask1>,<mask2>,…`
    - CPU masks are always interpreted as hexadecimal values
  - `--distribution=*|block|cyclic|arbitrary|plane=<options> [:*|block|cyclic|fcyclic[:*|block|cyclic|fcyclic]][,Pack|NoPack]`
- further information: https://apps.fz-juelich.de/jsc/hps/juwels/affinity.html

Process Placement

- Example:
  - `--ntasks-per-node=4`
  - `--cpus-per-task=3`
- `--cpu-bind=threads`



- `--cpu-bind=mask_cpu:0x7,0x700,0xE0,0xE000`



Process Placement

- best practice depends not only on topology, but also on characteristics of application:

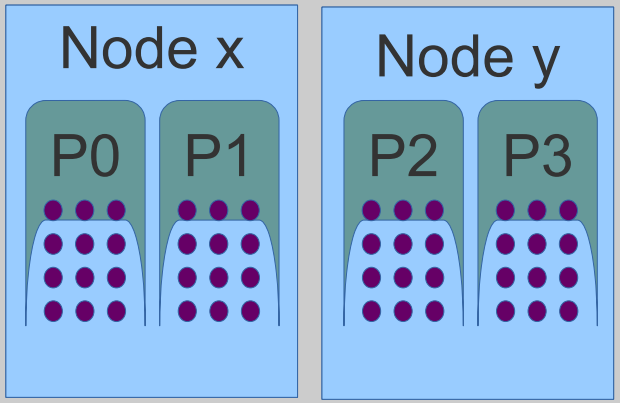
- putting threads far apart is
  - improving the aggregated memory bandwidth available to your application
  - improving the combined cache size available to your application
  - decreasing the performance of synchronization constructs

- putting threads close together is
  - improving the performance of synchronization constructs
  - decreasing the available memory bandwidth and cache size

Hybrid MPI/OpenMP

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

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

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Get_processor_name(processor_name, &namelen);

#pragma omp parallel default(shared) private(iam, np)
  {
    np = omp_get_num_threads();
    iam = omp_get_thread_num();
    printf("Hello from thread %02d out of %d from process %d out of %d on %s\n",
           iam, np, rank, numprocs, processor_name);
  }

  MPI_Finalize();
}
```

Example:
2 Nodes, 2x2 Procs,
2x2x24 Threads



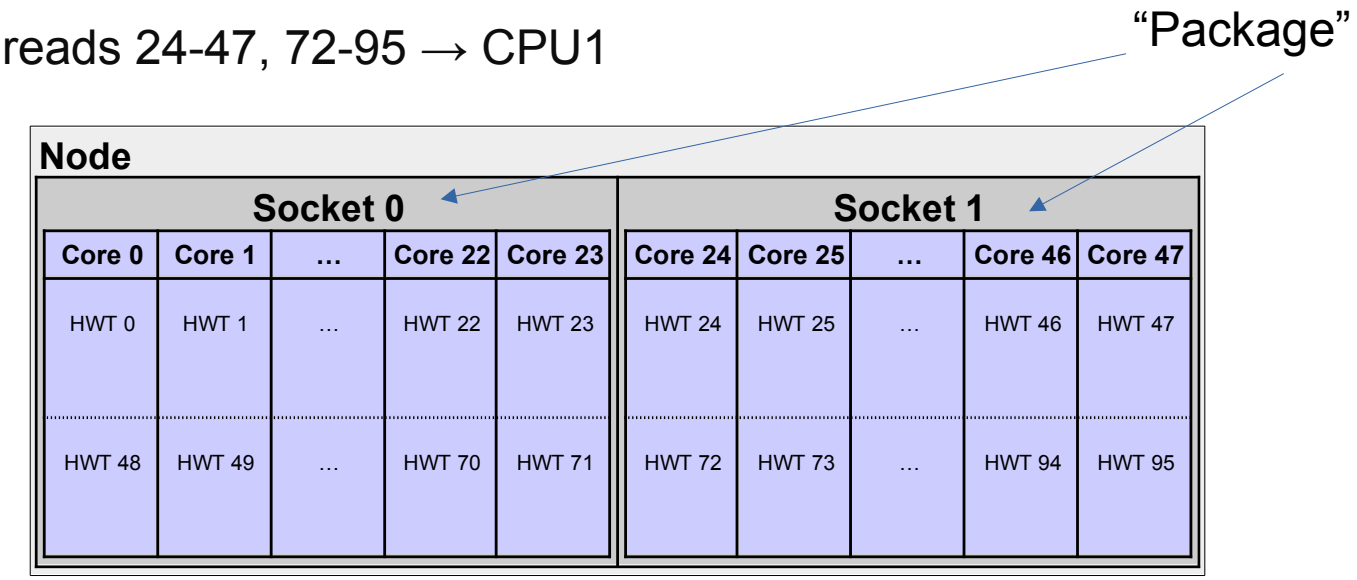
On JUWELS

- `module load Intel ParaStationMPI/5.10.0-1-mt`
- `mpicc -O3 -qopenmp -o hello_hybrid hello_hybrid.c`
- `salloc -N 2 -A partec -cpus-per-task=24`
- `export OMP_NUM_THREADS=${SLURM_CPUS_PER_TASK}`
- `srun -n 4 ./hello_hybrid | sort`

```
Hello from thread 00 out of 24 from process 0 out of 4 on jwc01n238.juwels
Hello from thread 00 out of 24 from process 1 out of 4 on jwc01n238.juwels
Hello from thread 00 out of 24 from process 2 out of 4 on jwc01n247.juwels
Hello from thread 00 out of 24 from process 3 out of 4 on jwc01n247.juwels
Hello from thread 01 out of 24 from process 0 out of 4 on jwc01n238.juwels
Hello from thread 01 out of 24 from process 1 out of 4 on jwc01n238.juwels
Hello from thread 01 out of 24 from process 2 out of 4 on jwc01n247.juwels
Hello from thread 01 out of 24 from process 3 out of 4 on jwc01n247.juwels
                                        .
                                        .
                                        .
Hello from thread 23 out of 24 from process 0 out of 4 on jwc01n238.juwels
Hello from thread 23 out of 24 from process 1 out of 4 on jwc01n238.juwels
Hello from thread 23 out of 24 from process 2 out of 4 on jwc01n247.juwels
Hello from thread 23 out of 24 from process 3 out of 4 on jwc01n247.juwels
```

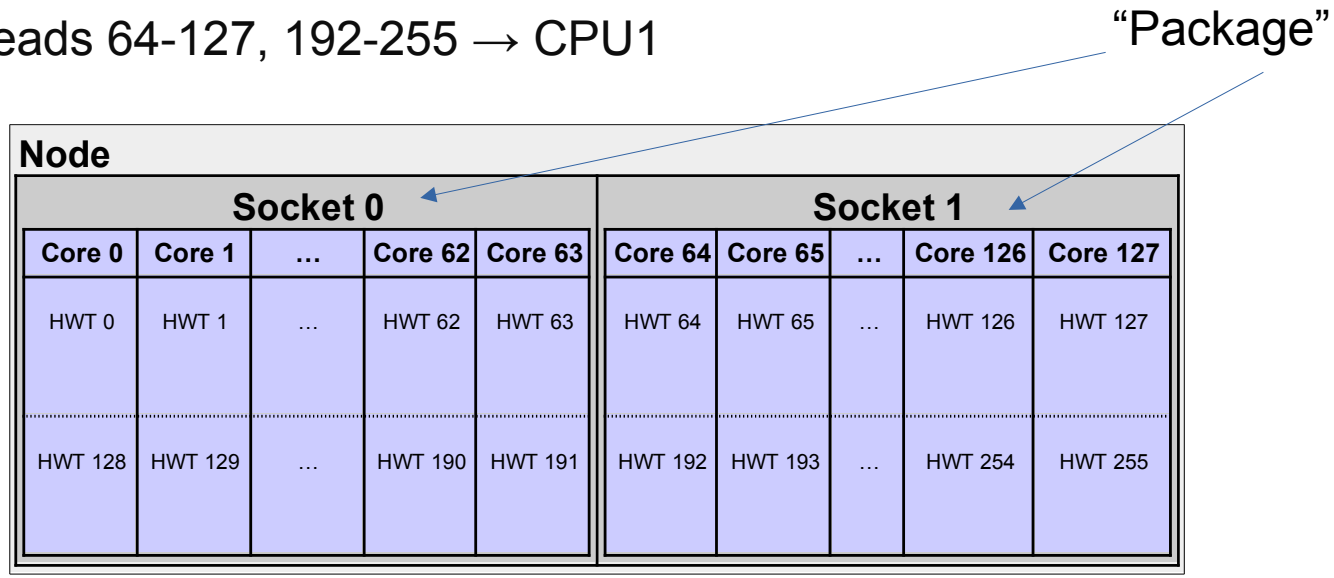
Pinning

- JUWELS:
  - 2 Sockets, 24 Cores per Socket
  - 2 HW-Threads per Core
  - → 96 HW-Threads possible
- normally (SMT):
  - HW-Threads 0-23, 48-71 → CPU0
  - HW-Threads 24-47, 72-95 → CPU1



Pinning

- JURECA DC:
  - 2 Sockets, 64 Cores per Socket
  - 2 HW-Threads per Core
  - → 256 HW-Threads possible
- normally (SMT):
  - HW-Threads 0-63, 128-191 → CPU0
  - HW-Threads 64-127, 192-255 → CPU1



Pinning

- no thread pinning by default on JURECA and JUWELS
- allow the Intel OpenMP library thread placing
  - `export KMP_AFFINITY=[verbose,modifier,…]`
    - `compact`: place threads as close as possible
    - `scatter`: as evenly as possible
- full environment is exported via srun on JURECA and JUWELS
- for GCC: `set GOMP_CPU_AFFINITY` (see manual)

Large Job Considerations

- every MPI process talks to all others:
  - (N-1) x 0.55 MB communication buffer space per process!

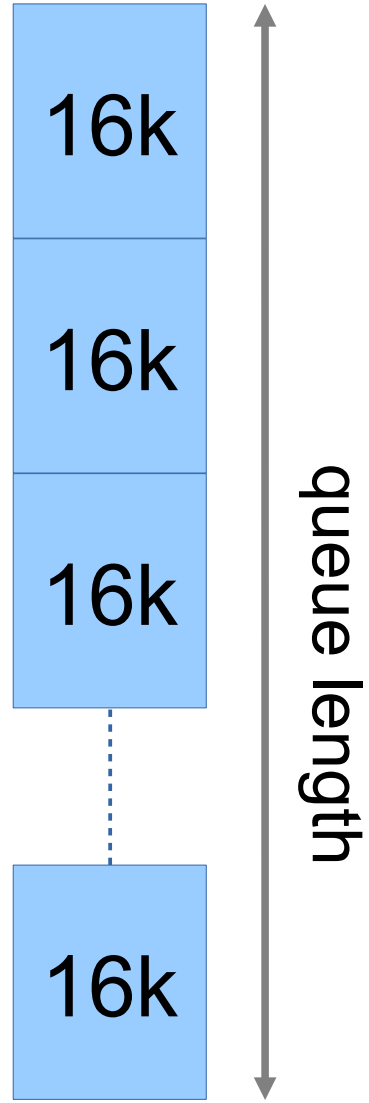
- example 1 on JUWELS:
  - job size 256 x 96 = 24,576 processes
  - 24,575 x 0.55 MB → ~ 13,516 MB / process
  - x 96 processes / node → ~ 1,267 GB communication buffer space
  - but there is only 96 GB of main memory per node
- example 2 on JURECA DC:
  - job size 256 x 256 = 65,536 processes
  - 65,535 x 0,55 MB → ~ 36,044 MB / process
  - x 256 processes / node → ~ 9,011 GB communication buffer space
  - but there is only 512 GB of main memory per node

On Demand / Buffer Size

- Three possible solutions:
- 1. Try using alternative meta modules

- 2. Create buffers on demand only:
  - `export PSP_ONDEMAND=1`
  - activated by default!

- 3. Reduce the buffer queue length:
  - (default queue length is 16)
  - `export PSP_OPENIB_SENDQ_SIZE=3`
  - `export PSP_OPENIB_RECVQ_SIZE=3`
  - do not go below 3, deadlocks might occur!
  - trade-off: performance penalty
    - (sending many small messages)



On Demand / Queue Size Guidelines

- On-Demand works best with nearest neighbor communications
  - (Halo) Exchange
  - Scatter/Gather
  - All-reduce
  - …
- but for *All-to-All* communication:
  - queue size modification only viable option…
- example

```
rank 0: for ( ; ; ) MPI_Send ()
rank 1: for ( ; ; ) MPI_Recv ()
```

- **PSP_OPENIB_SENDQ/RECVQ_SIZE=4: 1.8 seconds**
- **PSP_OPENIB_SENDQ/RECVQ_SIZE=16: 0.6 seconds**
- **PSP_OPENIB_SENDQ/RECVQ_SIZE=64: 0.5 seconds**

Resources

- www.par-tec.com
- www.fz-juelich.de/en/ias/jsc/systems/supercomputers
- /opt/parastation/doc/pdf
- by mail: sc@fz-juelich.de
- by mail: support@par-tec.com
- download ParaStation MPI at github:
  - https://github.com/ParaStation/psmgmt
  - https://github.com/ParaStation/pscom
  - https://github.com/ParaStation/psmpi

Summary

- you now should be able to
  - compile
  - run your application
  - tune some runtime parameters
  - diagnose and fix specific errors
  - know where to turn to in case of problems

Thank you for your attention!

ParTec
MODULAR SUPERCOMPUTING

Questions?