JURECA – Tuning for the platform

Usage of ParaStation MPI

2015-11-26

Frank Robel
Outline

- ParaStation MPI
- Compiling your program
- Running your program
- Tuning parameters
- Resources
ParaStation MPI

- Based on MPICH
  - supports all MPICH tools (tracing, debugging, …)
- Proven to scale up to 3,000 nodes and 40,000 procs per job
  - JuRoPA running ParaStation MPI: 0.274 PFLOPS (2009)
  - JURECA running ParaStation MPI: 1.42 PFLOOPS (2015)
- Supports a wide range of interconnects, even in parallel
  - e.g. InfiniBand EDR on JURECA cluster in Jülich
ParaStation MPI

- MPI libraries for several compilers
  - especially for GCC and Intel
- Recently added features include:
  - *Improved InfiniBand bandwidth performance*
  - *Improved (dynamic) process management*
  - *MPI-3 compliance (e.g. non-blocking collectives)*
ParaStation History

- 1995: University project (→ University of Karlsruhe)
- 2004: Open source (→ ParaStation Consortium)
- 2004: Cooperation with JSC
  - various precursor clusters
  - JUDGE
  - DEEP
  - JuRoPA2 (J2)
  - JuRoPA3 (J3)
  - JURECA
Recent Versions

**JURECA**

- **ParaStation MPI**
  - → *psmpi-5.1.4-1* (MPI-3)
- **Intel Compilers** → v 15.0.3
- **Gnu gcc** → v 5.2.0
Compiling on JURECA

- Only MPI-3 version (5.1.4-1) available
- `module avail 2>&1 | grep psmpi`
- `[i|g]psmpi/2015.7[–mt]`
  - `single thread task/multi-thread tasks (mt)`
  - `mt currently only available for intel`
- ChangeLog available with
  - `less $(dirname $(which mpicc))/../ChangeLog`
- Gnu and Intel compilers available
  - `gcc-5.2.0 (gpsmpi)`
  - `intel-2015.0.3 (ipsmpi)`
- see also the previous talk about Toolchains / Lmod usage...
Wrapper vs. Manual Compilation

- Wrappers
  - `mpicc (C)`
  - `mpicxx (C++)`
  - `mpif90 (Fortran 90)`
  - `mpif77 (Fortran 77)`
- `mpi<LANG> --show`
  - shows what would happen
  - useful for legacy Makefiles
  - allows to tweak compiler
- When using the “mt” version (and using OpenMP), add
  - `--fopenmp (gcc)`
  - `--openmp (intel)`
Wrapper vs. Manual Compilation

- Intel C-Compiler + ParaStation MPI
  - *module load ipsmpi*
  - *mpicc -show*

```
```
Did the wrapper link correctly?

- Libraries are linked at runtime according to `LD_LIBRARY_PATH`
- `ldd` shows the libraries attached to your binary
- Look for ParaStation libraries
  
  e.g. `ldd hello_mpi_default`:
  
  - `linux-vdso.so.1` => `(0x00007fff96ffe000)`
  - `libmpi.so.12` => `/usr/local/software/jureca/Stage3/software/Toolchain/iccifort/2015.3.187-GCC-bare-4.9.3/psmpi/5.1.4-1-mt/lib/libmpi.so.12` (0x00002b7574971000)

Wrong Module !?!
(e.g. if application is not multi-threaded)
JURECA: start via srun

- **WARNING:** DO NOT USE `mpiexec`
  - *Unexpected results may happen!*
  - *Use `srun` instead!*

- `srun -N <nodes> -n <tasks>` spawns the tasks, either:
  - *interactively via `salloc` (to be called previously)*
  - *from batch job (inside an `sbatch` script)*

- Full environment is exported via `srun`
- Pressing `^C` (in interactive) will be passed to all tasks
- No manual clean-up needed
/* 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 JURECA

- `module load ipsmpi`
- `mpicc -O3 -o hello_mpi hello_mpi.c`

Interactive:
- `salloc -N 2 # get an allocation`
- `srun -n 2 ./hello_mpi`
  - `Hello world from process 0 of 2 on jrc0491`
  - `Hello world from process 1 of 2 on jrc0492`

Batch:
- `sbatch -N 2 ./hello_mpi.sh`
Process Placement

- ParaStation process pinning:
  - Avoid task switching
  - Make better use of CPU cache
- Manipulate pinning:
  - e.g. for "large memory / few task" applications
- `psiadmin -c 'show cpumap <node>'`
  - `<node number>` 0 1 2 3 4 5 ...
- Manipulate via `__PSI_CPUMAP="C_{1-C_i},C_{j-C_k},C_{l-C_n}"`
- For example on JURECA:
  - `__PSI_CPUMAP="0,4,8,12,..." srun --cpu_bind=rank -n 4 ./testcore`
- JURECA is pinning by default:
  - So `--cpu_bind=rank` may be omitted
```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 Proc, 2x2x12 Threads

Node x
- P0
- P1

Node y
- P2
- P3
On JURECA

- module load ipsmpi/2015.07-mt
- mpicc -O3 -openmp -o hello_hybrid hello_hybrid.c
- salloc -N 2 --cpus-per-task=12
- export OMP_NUM_THREADS=${SLURM_CPUS_PER_TASK}
- srun -n 4 ./hello_hybrid

Hello from thread 00 out of 12 from process 0 out of 4 on jrc0491
Hello from thread 01 out of 12 from process 0 out of 4 on jrc0491
Hello from thread 02 out of 12 from process 0 out of 4 on jrc0491
Hello from thread 03 out of 12 from process 0 out of 4 on jrc0491
.
.
.
Hello from thread 09 out of 12 from process 3 out of 4 on jrc0492
Hello from thread 10 out of 12 from process 3 out of 4 on jrc0492
Hello from thread 11 out of 12 from process 3 out of 4 on jrc0492
Pinning: Which core for a thread?

- **JURECA:**
  - 2 Sockets, 12 Cores per Socket
  - 2 HW-Threads per Core
  - → 48 Threads possible

- Normally (SMT):
  - Threads 0-11, 24-35 → CPU0
  - Threads 12-23, 36-47 → CPU1

<table>
<thead>
<tr>
<th>Node</th>
<th>Socket 0</th>
<th>Socket 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core 0</td>
<td>Core 1</td>
<td>...</td>
</tr>
<tr>
<td>HWT 0</td>
<td>HWT 1</td>
<td>...</td>
</tr>
<tr>
<td>HWT 24</td>
<td>HWT 25</td>
<td>...</td>
</tr>
<tr>
<td>Core 12</td>
<td>Core 13</td>
<td>...</td>
</tr>
<tr>
<td>HWT 12</td>
<td>HWT 13</td>
<td>...</td>
</tr>
<tr>
<td>HWT 36</td>
<td>HWT 37</td>
<td>...</td>
</tr>
</tbody>
</table>
Pinning: Which core for a thread?

- No thread pinning by default on JURECA

- Allow the Intel OpenMP library thread placing
  - `export KMP_AFFINITY=[verbose,]...`
  - `compact`: place threads as close as possible
  - `scatter`: as evenly as possible

- Full environment is exported via `srun` on JURECA

- For GCC: set `GOMP_CPU_AFFINITY` (see manual)
PMI Barrier Timeout

- If you see ...
  - *PSI*logger: Timeout: Not all clients joined the first pmi barrier:
    
    \[ \text{joined}=36765 \quad \text{left}=99 \quad \text{round}=10 \]

- Typically in large applications (-n > 10,000)
- ... you should **not** increase the timeout ...
  - *The timeout is calculated from the requested job size*
- ... but it is possible to increase the number of rounds
  - `export PMI_BARRIER_ROUND=<N>`
  - `default is 10`

- **Max job size: 128 x 48 = 6144**
  - you should not see any PMI Barrier Timeout
Large Job Considerations

- Every MPI process talks to all others:
  - \((N-1) \times 0.55 \text{ MB communication buffer space per process!}\)

- Example 1 on JURECA:
  - \(\text{max job size } 128 \times 48 = 6144\) processes
  - \(6143 \times 0.55 \text{ MB } \rightarrow \sim 3379\) MB / process
  - \(\times 48 \text{ process / node } \rightarrow \sim 158\) GB communication buffer space
  - But there are only \(128\) GB main memory per node

- Example 2 on JURECA:
  - Max job size with only one process per core
  - \((128 \times 24 - 1) \times 0.55 \text{ MB } \times 24 \rightarrow \sim 40\) GB / node
  - Main memory – OS – GPFS – 40 GB \(\rightarrow \sim 60\) GB
  - 60 GB / 24 procs \(\rightarrow 2.5\) GB left for each process
On Demand / Buffer Size

Two possible solutions:

1. Create buffers on demand only:
   - `export PSP_ONDEMAND=[0|1]`

2. 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!*
   - *Tradeoff: Performance penalty*
     - small queue lengths may lead to low bandwidths
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...
Non Uniform Memory Access (NUMA)
NUMA Policies

- If memory is bound to processes, only local memory is accessible → and can get exhausted:
  - `srun -n 1 --cpu_bind=rank ./blockmem_mpi`
  - `srun: error: jrc0075: task 0: Killed`
  - `srun: Force Terminated job step 1505858.15`

- If memory is not bound to processes, all memory is accessible:
  - `srun -n 1 --cpu_bind=none ./blocknen_mpi`

- On JURECA is `--cpu_bind=rank` used by default so it can be omitted

- But: membind off → data is crossing CPUs (NUMA) → ~15–20% performance drop!
Resources

- www.parastation.com
- www.fz-juelich.de/ias/jsc/jureca
- /opt/parastation/doc/pdf
- by mail: support@par-tec.com
- by mail: sc@fz-juelich.de
- Download ParaStation MPI at github:
  - https://github.com/ParaStation/psmgmt
  - https://github.com/ParaStation/pscom
  - https://github.com/ParaStation/psmpi2
  - git clone https://github.com/ParaStation/psmpi2.git
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!
Why ParaStation MPI?

- Three aspects:
  - Scalability
  - Performance
  - Robustness
Why ParaStation

- Scalability
  - “Desirable property of system, network or process to handle growing amounts of work in a graceful manner [...]”
  - Runtime proportional to number of operations
  - Runtime inverse proportional to amount of resources

- Proven to scale up to 3,000 nodes and 40,000 processes per job
Performance

The graph shows the global performance in Gflops as a function of the number of processes for two different MPI implementations:

- **psmpi2-intel** (red line)
- **openmpi-intel** (green line)

The performance increases linearly with the number of processes, indicating good scalability for both implementations.
Robustness

- “Ability of a computer system to cope with errors during execution [...] despite abnormalities in input [...]”

- Crash of a parallel job:
  - Must not crash master control
  - Must release resources

- ParaStation solution: daemon concept
  - No single point of failure
  - Management of entire process lifecycle
  - Resource monitoring

- Job clean up after
  - process dies (intentionally or unintentionally)
  - node dies (e.g. due to HW failure)
Why ParaStation MPI?

- Three aspects:
  - **Scalability** ✓
  - **Performance** ✓
  - **Robustness** ✓
Thank you!