Performance Best Practices

For POWER8 (and other CPUs)

08.03.2017, EuroHACK Jülich

Topics

- Compilers & Libraries
- NUMA and Pinning

- This is just a guide, not the one fits all package -> Try it, and measure the effect.
- This is by no means complete.
- This focuses on IBM POWER8 LE, but is also applicable to other architectures
- Specific to HPC / FLOP-heavy codes
- Without: Speedup GPU vs. CPU is meaningless

Compilers & Libraries

Compilers: XL

- Use a good compiler optimization level: `-O2` or better
- Use vendor compilers wherever possible
  - In case of IBM POWER this is XL: xlc, xlc++, xlf
    - Usefull flags you should use:
      `CFLAGS=-O3 -qarch=pwr8 -qtune=pwr8 -qaltivec`
    - Flags you can try:
      `-qhot`
      `-O4`
      `-O5`
    - → but check your results (non IEEE-floating optimizations, different interpretations of C standard)
  - If your code is not compiling with XL: Tell us (with a short example code demonstrating the error)!

Compilers: GCC

- Second choice, if other compilers are not working (but you might want to get your code standard-compliant at some time...)
- Use the AT or most recent version installed on the system (old version lack support for current architectures)
- Flags you want to use:
  ```
  CFLAGS=-O3 -flto -mcpu=power8 -mtune=power8 -mpower8-fusion -mpower8-vector -mvsx -maltivec
  ```
- Flags you might try (and check your results and performance)
  ```
  CFLAGS+= -Ofast -mdirect-move -ffast-math
  ```

Compilers: PGI

- `CFLAGS=-fast`
- If you want to know what it does:
  `-help -fast`
- If you want to know what the compiler is able/unable to do, use
  `-Minfo`
- `-acc` enables OpenACC code generation
- `-mp` enables OpenMP code generation

- *Thanks to Sebastien for providing this input*

Libraries

- Don‘t use distro packages of math libraries or MPI (compiled to work everywhere, not optimized)
- Use packages provided on the cluster (module system), ideally vendor-package if available.
  – BLAS/LaPACK: ESSL
- Don‘t compile common libraries yourself (except you know what you are doing = know all config options of the package)
- Bonus: IBM Advance Toolchain (AT)
  – Standard Linux toolchain (gcc, gdb, glibc, ...) with POWER-specific optimizations and additional tools/libs (tcmalloc, boost, openssl)

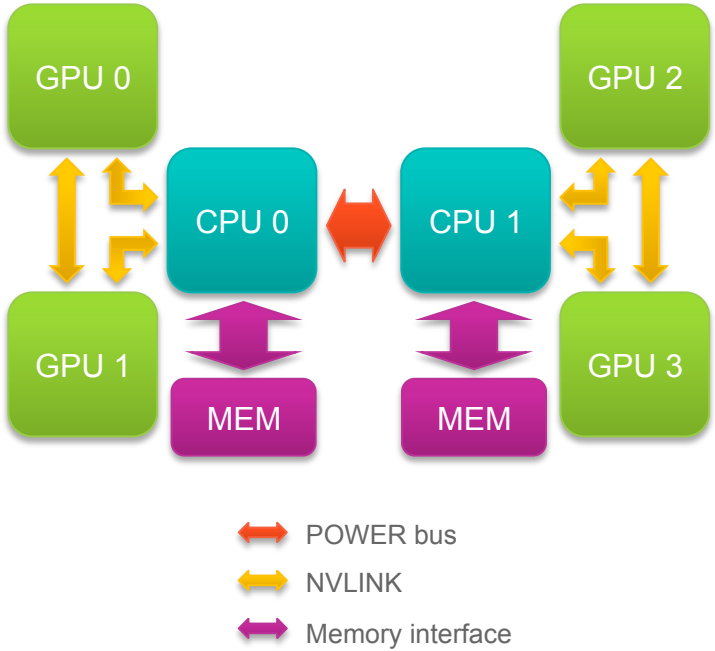
NUMA & Pinning

Thread migration / Cache thrashing

- Linux kernel process scheduler moves processes from thread to thread to get optimal balancing
  - Usually: good thing!
  - For HPC: not so much...
    - Processes / ranks fill all cores to the brim (but get moved due to kernel and system deamons)
    - Migrating takes time (sleep-move-resume).
    - Migrated threads loose their cached data (adding latency and waste bandwidth)
    - This problem increases with higher number of threads and higher code efficiency
    - Kernel typically does not take much care of system architecture

NUMA (non-uniform memory access): The problem

- Two CPU sockets in the system, connected by a cache-coherent bus
  - Communication adds latency and has a limited bandwidth
  - → Try to avoid remote socket memory accesses



The solution: Process pinning!

- Tell the kernel to not move your processes to different threads

- There are multiple ways:
  - Command line tools
  - Via OpenMP environment variables
  - Via MPI launchers
  - Via the cluster batch scheduler
- Check binding for your jobs (report them, experiment in interactive sessions and check with `htop`)

Process pinning: Command line tool `numactl`

- Simple tool, to bind all threads / memory accesses of the comand following to hardware
  - Nomenclature: node ~= socket, cpu = smt-thread
- Show NUMA setup of the server:
  `numactl -H`
- Just use socket (=node) local memory:
  `numactl --localalloc ./myapp`
- Bind process to socket 0:
  `numactl --cpunodebind=0 ./myapp`
- Bind processes to threads 0-8 (first physical core; but: no 1-1 mapping, all bound to all)
  `numactl --physcpubind=0-8 ./myapp`
- Documentation: `man numactl`
- Complex to use with OpenMP / MPI, especially with batch schedulers

Process pinning: OpenMP

- If GCC, only works with GOMP version >= 5
- Set environment variables
  `export OMP_PROC_BIND=true`
  to pin threads.
- Use
  `OMP_PLACES={0,8,16,24,32,40,48,56,64,72,80} OMP_NUM_THREADS=10 ./my-omp-app`
  to explicitly bind one thread per core on one socket.
- Debugging: Show binding with
  `export OMP_DISPLAY_ENV=true`
- Documentation:
  https://gcc.gnu.org/onlinedocs/libgomp/Environment-Variables.html#Environment-Variables
- Usually: Don‘t use OpenMP across sockets

Process pinning: OpenMPI

- The OpenMPI mpirun wrapper has neat binding options for the ranks
- Nomenclature:
  - hwthread = (SMT-)thread
  - core = hardware core
  - socket = socket
- Use these with
  `mpirun --map-by <...> ./myMPIapp`
  and
  `mpirun --bind-to <...> ./myMPIapp`
- Show binding with
  `mpirun --report-bindings <...>`
- Documentation: https://www.open-mpi.org/doc/v2.0/man1/mpirun.1.php

Process pinning: LSF / batch scheduler

- Can use affinity strings in combination if LSF-supporting MPI
- Documentation & Examples: https://www.ibm.com/support/knowledgecenter/SSETD4_9.1.2/lsf_admin/affinity_support_lsf_submit.html https://www.ibm.com/support/knowledgecenter/en/SSETD4_9.1.2/lsf_admin/affinity_res_req_string.html

```
#!/bin/bash

#BSUB -J myHPCjob                    # job name
#BSUB -W 04:30                       # wall-clock time (hrs:mins)
#BSUB -n 640                         # total number of mpi ranks

#BSUB -R "span[ptile=160] affinity[thread(1):cpubind=thread:membind=localprefer]"

#BSUB -q myqueue

module load theModuleIneed theOtherModule
mpirun ./myMPIapp
```