On best practices with the system

Compiler man pages

- **The cc, CC, and ftn man pages contain information about the compiler driver commands**
- **Cray compiler: man craycc, crayCC, and crayftn**
- **GNU compiler: gcc, g++, and gfortran**
- **Intel: no man page available, but do `ifort/icc –help`**

- **To verify that you are using the correct version of a compiler, use:**
  - -V option on a cc, CC, or ftn command with CCE
  - --version option on a cc, CC, or ftn command with GNU

Recommended compiler optimization levels

- **Cray compiler**
  - The default optimization level (i.e. no flags) is equivalent of **`-O3`** of most other compilers
    - CCE optimizes rather aggressively by default, but this is also most thoroughly tested configuration
  - Try with **`-O3 -hfp3`**
    - We also test this thoroughly
    - -hfp3 gives you a lot more floating point optimization, esp. 32-bit
    - In case of precision errors, try a lower –hfp number (–hfp2 first,–hfp0 only if absolutely necessary)

Recommended compiler optimization levels

- **Intel compiler**
  - Always add **`-xAVX`**
  - The default optimization level (equal to -O2) is safe and gives usually good performance
  - Try with **-O3** (verify correctness & performance)
    - If that works still, you may try with **`-Ofast`**
  - Also setting **`-fp-model fast=2`** (or =1) may give some additional performance
  - Loop unrolling with **`-funroll-loops`** or **`-unroll-aggressive`** may also be beneficial

Recommended compiler optimization levels

- **GNU compiler**
  - Always add **`-mavx`**
    `-march=corei7-avx`
    `-mtune=corei7-avx`
  - Almost all HPC applications compile correctly with using **`-O3`**, so do that instead of the cautious default
    - **`-Ofast`** may give minor extra performance on top of -O3
  - **`-ffast-math`** may give some extra performance (verify results)
  - **`-funroll-loops`** benefits most applications

Loopmark: Compiler feedback

- **Compilers can generate annotated listing of your source code indicating important optimizations**
- **CCE**
  - `ftn -rm`
  - `cc -hlist=a`
- **Intel**
  - `ftn/cc -opt-report 3 -vec-report6`
  - If you want this into a file: add `-opt-report-file=filename`
  - See `ifort --help reports`
- **GNU**
  - `-ftree-vectorizer-verbose=9`

Loopmark: Compiler feedback

```
29.  b-------<  do i3=2,n3-1
30.  b b-----<     do i2=2,n2-1
31.  b b Vr--<        do i1=1,n1
32.  b b Vr            u1(i1) = u(i1,i2-1,i3) + u(i1,i2+1,i3)
33.  b b Vr    *         + u(i1,i2,i3-1) + u(i1,i2,i3+1)
34.  b b Vr            u2(i1) = u(i1,i2-1,i3-1) + u(i1,i2+1,i3-1)
35.  b b Vr    *         + u(i1,i2-1,i3+1) + u(i1,i2+1,i3+1)
36.  b b Vr-->        enddo
37.  b b Vr--<        do i1=2,n1-1
38.  b b Vr            r(i1,i2,i3) = v(i1,i2,i3)
39.  b b Vr    *         - a(0) * u(i1,i2,i3)
40.  b b Vr    *         - a(2) * ( u2(i1) + u1(i1-1) + u1(i1+1) )
41.  b b Vr    *         - a(3) * ( u2(i1-1) + u2(i1+1) )
42.  b b Vr-->        enddo
43.  b b----->     enddo
44.  b------->  enddo
```

Inlining & inter-procedural optimization

- **Cray compiler**
  - Inlining within a file is enabled by default
  - Command line options **`-OipaN`** (ftn) and **`-hipaN`** (cc/CC) where N=0..4, provides a set of choices for inlining behavior
    - 0 disables inlining, 3 is the default, 4 is even more elaborate
  - The **`-Oipafrom=`** (ftn) or **`-hipafrom=`** (cc/CC) option instructs the compiler to look for inlining candidates from other source files, or a directory of source files
- **Intel compiler**
  - Inlining within a file is enabled by default
  - Multi-file inlining enabled by the flag **`-ipo`**
- **GNU compiler**
  - Quite elaborate inlining enabled by **`-O3`**

Hyperthreads

- **Intel Hyper-Threading is a method of improving the throughput of a CPU by allowing two independent program threads to share the execution resources of one CPU**
  - When one thread stalls the processor can execute ready instructions from a second thread instead of sitting idle
  - Because only the thread context state and a few other resources are replicated (unlike replicating entire processor cores), the throughput improvement depends on whether the shared execution resources are a bottleneck
  - Typically much less than 2x with two hyperthreads
- **It may improve or degrade the performance of your application**
  - Try it, if it does not help, turn it off

Hyperthreads

- **With aprun, hyper-threading is controlled with the switch -j**
  - -j 1 = no hyper-threading (a node is treated to contain 16 cores)
  - -j 2 = hyper-threading enabled (a node is treated to contain 32 cores)
  - One has to oversubscribe the node for 32 HT tasks

```
...
# SBATCH --ntasks=128
# SBATCH --ntasks-per-node=32
# SBATCH –-ntasks-per-core=2

aprun -n 128  -N 32  -j 2 ./exe
```

- **Or alternatively**

```
...
# SBATCH --nodes=4
# we have 2x16 HT cores per node

aprun -n 128 -j 2 ./exe
```

OpenMP thread placement

- **When running a hybrid MPI+OpenMP application, the optimal number of threads/MPI task depends on the application and even input**
  - On Sisu, one should experiment with 1x16, 2x8, perhaps also with 4x4, even 8x2 (MPI tasks x OpenMP threads per node)
- **The XC system is able to place OpenMP threads appropriately when the code is compiled with the Cray or GNU compiler**
  - Just do e.g. ”aprun -n 64 -d 16 -N 1 ./a.out” (for a 64x16=1024 core job)
  - You can use the aprun switch -S to force a certain number of MPI tasks per a numa node (=CPU) and -ss to have the threads to allocate memory only in the local numa node

OpenMP thread placement

- **With Intel compiler, this becomes more difficult**
  - Only 8 or 16 threads per node works properly and we have still control the affinity explicitly
  - One needs to set additional environment variable KMP_AFFINITY to certain values for avoiding performance problems
  - For a 1x16 job, have the following in your batch script

```
export OMP_NUM_THREADS=16
export KMP_AFFINITY="granularity=fine,compact,1"
aprun -n 8 -d 16 -N 1 -cc none ./intel_hyb
```

  - For a 2x8 job, do the following

```
export OMP_NUM_THREADS=8
export KMP_AFFINITY="compact,1"
aprun -n 8 -d 8 -N 2 -cc numa_node -S 2 -ss ./intel_hyb
```

MPI rank placement across nodes

- **The default rank ordering can be changed using the following environment variable:**

```
export MPICH_RANK_REORDER_METHOD=N
```

- **These are the different values (N) that you can set it to:**
  - 0: Round-robin placement – Sequential ranks are placed on the next node in the list.
  - 1: (DEFAULT) SMP-style placement
  - 2: Folded rank placement
  - 3: Custom ordering. The ordering is specified in a file named MPICH_RANK_ORDER.

MPI - Async Progress Engine Support

- **Used to improve communication/computation overlap**
  - Each MPI rank starts a “helper thread” during MPI_Init
- **Helper threads progress MPI engine while application computes**
- **Only large inter-node messages are progressed**
- **To enable on XC when using 1 stream per core, add to the batch job script**
  - `export MPICH_NEMESIS_ASYNC_PROGRESS=1`
  - `export MPICH_MAX_THREAD_SAFETY=multiple`
  - `export MPICH_GNI_USE_UNASSIGNED_CPUS=enabled`
  - Run application: `aprun –n XX  a.out`
- **10% or more performance improvements with some apps**

Huge pages

- **The Aries performs better with sc. huge pages than with 4K pages. The Aries can map more pages using fewer resources meaning communications may be faster**
- **Huge pages will also affect TLB performance:**
  - Your code may run with fewer TLB misses (hence faster)
  - However, your code may load extra data and so run slower
  - Only way to know is by experimentation
- **Use modules to change default page sizes (`man intro_hugepages`):**
  - `e.g. module load craype-hugepages#`
    - `craype-hugepages128K`
    - `craype-hugepages512K`
    - **`craype-hugepages2M`**
    - **`craype-hugepages8M`**
    - `craype-hugepages16M`
    - `craype-hugepages64M`

Most commonly successfully on Cray XC

Huge pages – usage

- **Link with the correct library:**
  `-lhugetlbfs`
- **Activate the library at run time:**
  `export HUGETLB_MORECORE=yes`
- **Launch the program with aprun pre-allocating huge pages**
  - request <size> MBytes per PE **`-m<size>h`** (advisory mode)
  - request <size> MBytes per PE **`-m<size>hs`** (required mode)
  - Example: **`aprun -m700hs -N2 -n8 ./my_app`**
    - Requires 1400 MBytes of huge page memory on each node

DMAPP collectives

- **On Cray XE and XC systems, the sc. DMAPP collectives will (usually significantly) improve the performance of the expensive collectives**
  - Enabled by the variable:
    `export  MPICH_USE_DMAPP_COLL=1`
  - Can be used selectively, e.g.
    `export  MPICH_USE_DMAPP_COLL=mpi_allreduce`
- **Features some restrictions and requires explicit linking with the corresponding library and using huge pages; consult ’man mpi’**

I/O optimization

- **Tune filesystem (Lustre) parameters**
  - Lustre stripe counts & sizes, see ”man lfs”
  - Rule of thumb:
    - # files > # OSTs => Set stripe_count=1
      You will reduce the lustre contention and OST file locking this way and gain performance
    - #files==1 => Set stripe_count=#OSTs
      Assuming you have more than 1 I/O client
    - #files<#OSTs => Select stripe_count so that you use all OSTs
- **Use I/O buffering for all sequential I/O**
  - IOBUF is a library that intercepts standard I/O (stdio) and enables asynchronous caching and prefetching of sequential file access
  - No need to modify the source code but just
    - Load the module iobuf
    - Rebuild your application