UG3.2.1: Other batch job examples on M100

- 1) OpenMP parallelization
- 2) MPI parallelization
- 3) Hybrid (MPI+OpenMP) parallelization
- Appendix) Pinning GPUs when launching a multiple serial execution within a job

Each node has 2 sockets, 16 cores per socket with SMT 4 (4 hardware threads, HTs, per core).

The HTs are enabled on all nodes, which expose themselves to SLURM has having 128 logical cpus. Since the nodes are not assigned in an exclusive way and can be shared by users, we configured the scheduler to always assign the requested cores in an exclusive way, that is from 1 to 128 as a multiple of 4. A job can ask resources up to a maximum to 128 cpus (hence, the upper limit for (ntasks-per-node) * (cpus-per-task) is 128). Even though you request --cpus-per-task<4, the 4 HTs per core will be assigned to your job.

You need to pay attention to the mapping and binding of MPI processes and of the OpenMP threads.

CAVEAT: the following discussion refers to XL 16.1.1, hpc-sdk 2021, gnu 8.4.0.

1) OpenMP parallelization

For threaded applications (pure OpenMP, no MPI), you obtain a full node by requesting --ntasks-per-node = 1 and --cpus-per-task=128. You can choose to exploit or not the SMT feature (it will depend on the value you assign to the OMP_NUM_THREADS variable) , but in any case switch the binding of the OMP threads on (for the XL , gnu, and hpc-sdk (ex-pgi) compilers it is off by default).

Different compilers abide by different default settings for the binding and placing of the threads:

- XL compilers
  - OMP_PROC_BIND = false (default)/true, close(default)/spread
  - OMP_PLACES = threads,cores (default)
  - Comments for **SMT configurations**:
    - in both the default setting for OMP_PLACES (=cores) and for OMP_PLACES=threads, the threads are bound to HW threads of the physical cores.
    - If OMP_NUM_THREADS < (cpus per task): for the default OMP_PLACES (=cores), the OMP threads are closely placed inside the physical cores, that is they are bound to the first HW threads of the physical core (cpu id: 0,1,4,5.....); setting OMP_PROC_BIND=close/spread makes no difference
    - If OMP_NUM_THREADS < (cpus per task): for OMP_PLACES=threads the OMP threads are by default closely bound to the first OMP_NUM_THREADS HW threads (cpu id: 0,1,2,3.....,OMP_NUM_THREADS-1); for OMP_PROC_BIND=spread the OMP threads are spread on the HW threads of the physical cores (cpu id: 0,2,4,6....2*(OMP_NUM_THREADS-1)).
- NVIDIA hpc-sdk compilers (ex PGI):
  - OMP_PROC_BIND = false (default)/true, close/spread(default)
  - OMP_PLACES = threads (default), cores
  - Comments for **SMT configurations**:
    - with the default OMP_PLACES (= threads) the threads are bound to HW threads of the physical cores.
    - If OMP_NUM_THREADS < (cpus per task): for the default OMP_PLACES (=threads) the OMP threads are always spread on the HW threads of the physical cores (cpu id: 0,2,4...); setting OMP_PROC_BIND=close/spread makes no difference
    - If OMP_NUM_THREADS < (cpus per task): for OMP_PLACES=cores, the OMP threads are bound to the 4 HW threads of the cores (cpu id: 0-3,4-7,8-11....); setting OMP_PROC_BIND=close/spread makes no difference
- GCC compilers
  - OMP_PROC_BIND = false (default)/true, close(default)/spread
  - OMP_PLACES = threads (default),cores
  - Comments for **SMT configurations**:
    - with the default OMP_PLACES (= threads) the threads are bound to HW threads of the physical cores.
    - If OMP_NUM_THREADS < (cpus per task): for the default OMP_PLACES (=threads) the OMP threads are by default closely bound to the first OMP_NUM_THREADS HW threads (cpu id: 0,1,2,3.....,OMP_NUM_THREADS-1); for OMP_PROC_BIND=spread the OMP threads are spread on the HW threads of the physical cores (cpu id: 0,2,4,6....2*(OMP_NUM_THREADS-1)).
    - If OMP_NUM_THREADS < (cpus per task): for OMP_PLACES=cores, the OMP threads are bound to the 4 HW threads of the cores (cpu id: 0-3,4-7,8-11....); setting OMP_PROC_BIND=close/spread makes no difference

For instance, with GCC compilers (gnu modules) if you want 32 OMP thread to be bound to one HW thread per physical, ask for the full node (--cpus-per-task=128) and

```
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=128
#SBATCH ........

export OMP_PROC_BIND=spread
export OMP_PLACES=threads      # not necessary, it's the default
export OMP_NUM_THREADS=32
<your exe>
```

Even though for different binding defaults, with XL and HPC-SDK compilers you simply need to set OMP_PROC_BIND=true, and you will have the OMP threads bound to the first HW thread of each physical core. (Explanation: for XL the default OMP_PLACES is cores, and no difference arises from the OMP_PROC_BIND set to close or spread. For HPC-SDK the default OMP_PLACES is threads,

2) MPI parallelization

For pure MPI applications (hence, no OpenMP threads as parallel elements), set the value of --ntasks-per-node to the number of MPI processes you want to run per node, and --cpus-per-task=4.
For instance, if you want to run an MPI application with 2 processes (no threads), each of them using 1 GPU:

```
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=2      # 2 MPI processes per node
#SBATCH --cpus-per-task=4        # 4 HTs per task
#SBATCH --gres=gpu:2

mpirun <your exe>
```

If you want the entire socket for your 2 MPI tasks, you need to request 32 cpus per task, and use the proper mapping configuration for the MPI tasks with the mpirun option --map-by <obj>:PE=N. For instance, you can map the processes on the object "socket", and indicate how many Parallel Elements (PE),
in terms of the next step of granularity, which for the object "socket" is the "physical" core (with its 4 HTs):

```
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=2      # 2 MPI processes per node
#SBATCH --cpus-per-task=32        # 4 HTs per task
#SBATCH --gres=gpu:2

mpirun  --map-by socket:PE=8 <your_exe>
```

In this case, each one of the two tasks per node can be mapped on 8 physical cores of the socket (the physical cpus being 16 per socket), hence the mapping is specified as
--map-by socket:PE=8. You can choose other mapping object, please verify that the result is correct in terms of MPI tasks binding (you can use the --report-bindings option of mpirun).
The number of mappable PEs for the socket object is given by 16 (= n. of physical cores per socket) / ntasks-per-node (= n. of MPI processes).
You can change the order in which the processors are assigned to the MPI ranks with the --rank-by option. If you want consecutive processes assigned to consecutive cores use --rank-by core.

If you want to exploit the SMT feature, request the number of tasks and 1 cpus-per-task, and bind (or map) the MPI processes to the "hwthread" element

```
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=128      # 128 MPI processes per node
#SBATCH --cpus-per-task=1       # 4 HTs per task

mpirun --map-by hwthread <your_exe>    (or mpirun --bind-to hwthread <your_exe>)
```

3) Hybrid (MPI+OpenMP) parallelization

Non-SMT: ask for a number of cpus-per-task equal to the number of OMP_NUM_THREADS you mean to use multiplied by 4.
Switch the binding of the OMP threads on, and correctly map the MPI processes with the --map-by option.

Example: 4 processes per node, 8 threads per process:

```
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=4
#SBATCH --cpus-per-task=32

export OMP_PROC_BIND=true
export OMP_NUM_THREADS=8

mpirun --map-by socket:PE=8 <your_exe>
```

SMT: set the value of --ntasks-per-node to the number of MPI processes you want to run per node, and --cpus-per-task = OMP_NUM_THREADS (if you want to exploit the SMT in terms of number of OMP threads) or to 128 / (ntasks-per-node) (if you want to exploit the SMT in terms of number of MPI processes).
Always switch the binding of the OMP threads on, and correctly map

Appendix) Pinning GPUs when launching a multiple serial execution within a job

Let's consider a situation like this:

```
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=4
#SBATCH --ntasks-per-socket=2
#SBATCH --gres=gpu:4

mpirun ./myscript.sh
```

where "myscript.sh" is a serial script that is equally executed by all the tasks allocated by the job, with no communication involved. You have requested 4 tasks, two for each socket, and you want each task to work within one of the 4 gpus available on the node, that are also two for each socket. So, at a first try your myscript.sh may result in something like this:

```
export CUDA_VISIBLE_DEVICES=${SLURM_LOCALID}
./my_actual_gpu_executable
```

In this way, task 0 will see GPU 0, task 1 will see GPU 1 and so on. This configuration, which may seem intuitively correct, can actually result in bad performance for some of the tasks running the executable. That is, because the IDs of the tasks and the GPUs are actually mismatched. While it is true that socket 1 hosts GPUs 0 and 1, and socket 2 hosts GPUs 2 and 3, the same can't be said for the cpu tasks, that are actually scattered: socket 1 hosts tasks 0 and 2, and socket 2 hosts tasks 1 and 3. To summarize:

```
FIRST SOCKET:
tasks 0 and 2 ---> GPUs 0 and 1
SECOND SOCKET:
tasks 1 and 3 ---> GPUs 2 and 3
```

Therefore, while working for GPU 1 and task 1, you are connecting a CPU and a GPU comung from different sockets, and this results in a slowdown in communication (the same happens with GPU 2 and task 2).

There are many ways to work around this unwanted behaviour. We present some of them:

1. **working with mpirun options**, specifically the flag --rank-by core (for Spectrum/OpenMPI) will compact the task assignment and rank the CPUs so that task 0 and 1 will belong to the first socket, and task 2 and 3 will belong to the second socket:

   ```
   mpirun --rank-by core ./myscript.sh
   ```

2. **pinning with Slurm directives**. There are a couple of possible options:

   ```
   #SBATCH --accel-bind=g
   ```

   From Slurm manual: "Bind each task to GPUs which are closest to the allocated CPUs."
   Another possible option is:

   ```
   #SBATCH --gpu-bind=closest
   ```

   "Bind each task to the GPU(s) which are closest" (useful when more than one process is attached to a single GPU).

3. A simple and intuitive way may be to **adapt your script** as follows:

   ```
   export CUDA_VISIBLE_DEVICES=${SLURM_LOCALID}

   if [ $SLURM_LOCALID -eq 1 ]
   then
      export CUDA_VISIBLE_DEVICES=2
   fi

   if [ $SLURM_LOCALID -eq 2 ]
   then
      export CUDA_VISIBLE_DEVICES=1
   fi

   ./my_actual_gpu_executable
   ```

   This way, the CPU tasks and the GPU IDs are matched in a way that they all belong to the same socket, and the communication is optimal.

PLEASE NOTE: the problem is related to the instance of serial executions working on different GPUs, launched via mpirun. When you submit an actual parallel execution, with MPI communications involved, the process pinning is automatically configured as a more intuitive pinning that keeps the task IDs compact and don't result in tag mismatching while compared with the ID of the GPUs. Your environment is safer in that case.