UG3.4.x: D.A.V.I.D.E. UserGuide

Starting from January, 2020 the activity on D.A.V.I.D.E. has been stopped.

In this page:

---

hostname: login.davide.cineca.it

early availability: **July 2018**

start of production: (to be defined)

end of production: Jan 20, 2020

D.A.V.I.D.E. : (**D**evelopment of an **A**dded **V**alue **I**nfrastructure **D**esigned in **E**urope) is an Energy Aware Petaflops Class High Performance Cluster based on Power Architecture and coupled with NVIDIA Tesla Pascal GPUs with NVLink. The innovative design of D.A.V.I.D.E. has been developed by E4 Computer Engineering for PRACE, which has as its ultimate goal to produce a leading edge HPC cluster showing higher performance, reduced power consumption and ease of use.

D.A.V.I.D.E. is based on OpenPOWER platform and is among the harbingers of a new generation of HPC systems which deliver high performances while being environmentally conscious. It has been built using best-in-class components plus custom hardware and an innovative middleware system software.

A key feature of D.A.V.I.D.E. is an innovative technology for measuring, monitoring and capping the power consumption of the node and of the whole system, through the collection of data from the relevant components (processors, memory, GPUs, fans) to further improve energy efficiency. The technology has been developed in collaboration with the University of Bologna.

FEATURES

- Off-the-shelf components
- High speed and accurate per-node power sensing synchronized among the nodes
- Data accessible out-of-band and without processor intervention
- Out-of-Band and synchronized fine grain performance sensing
- Dedicated data-collection subsystem running on management nodes
- Predictive Power Aware job scheduler and power manager

Model: E4 Cluster Open rack

Architecture: OpenPower NViDIA NVLink
Nodes: 45 x (2 Power8+4Tesla P100) + 2 (service&login nodes)
Processors: OpenPower8 (8 cores per processor; 16 cores per node) + NVIDIA Tesla P100 SXM2
Internal Network: 2xIB EDR, 2x1GbE
Cooling: SoC and GPU with direct hot water
Cooling capacity: 40kW
Heat exchanger: Liquid-liquid, redundant pumpsModel: E4 Cluster Open rack
Storage: 1xSSD SATA
Max Performances: 22 TFLOPs (double precision), 44 TFLOPs single precision

Peak Performance: ~1 PFlop/s

System Architecture

Compute nodes: There are 45 nodes connected with an efficient Infiniband EDR 100 Gb/s networking, with a total peak performance of 990 TFlops and an estimated power consumption of less than 2kW per node. Each node is a 2 OU OCP form factor and hosts two IBM POWER8 Processors with NVIDIA NVLink and four Tesla P100 (Pascal) GPUs, with the intra-node communication layout optimized for best performance.

Access

DAVIDE can be accessed via SSH (Secure Shell) protocol using the hostname:

> login.davide.cineca.it

which establishes a connection to one of the available login nodes.

For informations about data transfer from other computers please follow the instructions and caveats on the dedicated section **Data storage**, or the document **Data Management**.

Accounting

For accounting information please consult our **dedicated section**.

The account_no (or project) is important for batch executions. You need to indicate an account_no to be accounted for in the scheduler, using the flag "-A"

#SBATCH -A <account_no>

With the "saldo -b" command you can list all the account_no associated to your username.

Disks and Filesystems

The storage organization conforms to the CINECA infrastructure (see Section **Data Storage and Filesystems**).

In addition to the home directory **$HOME**, for each user is defined a scratch area **$CINECA_SCRATCH**, a large disk for the storage of run time data and files.

No $WORK storage area is currently defined for active projects on DAVIDE.

The scratch storage is a BeeGFS parallel filesystem.

Modules environment

As usual, the software modules are collected in different profiles and organized by functional category (compilers, libraries, tools, applications,..).

"Base" profile is the default. It is automatically loaded after login and it contains basic modules for the programming activities (intel e gnu compilers, math libraries, profiling and debugging tools,..).

If you want to use a module placed under others profiles, for example an application module, you will have to load preventively the corresponding profile:

```
>module load profile/<profile name>
>module load autoload <module name>
```

For listing all profiles you have loaded use the following command:

```
>module list
```

In order to detect all profiles, categories and modules available on Davide the command “modmap” is available:

```
>modmap
```

Production environment

Roughly speaking, there are two different modes to use an HPC system: Interactive and Batch. For a general discussion see the section **Production Environment and Tools**.

Interactive

A serial program can be executed in the standard UNIX way:

```
> ./program
```

This is allowed only for very short runs, since the **interactive environment has a 10 minutes time limit**: for longer runs please use the "batch" mode.

A parallel program can be executed interactively only within an "Interactive" SLURM batch job, using the "srun" command: the job is queued and scheduled as any other job, but when executed, the standard input, output, and error streams are connected to the terminal session from which srun was launched.

For example, to start an interactive session with the MPI program myprogram, using one node, one processors, one gpu, launch the command:

```
> salloc -N1 --ntasks-per-node=1 --gres=gpu:1 -A <account_name> -p dvd_usr_prod
```

SLURM will then schedule your job to start, and your shell will be unresponsive until free resources are allocated for you.

When the shell come back with the prompt, you can execute your program by typing:

```
> srun ./myprogram
```

or

```
> mpirun ./myprogram
```

The srun command will take by default PMI2 as MPI type.

SLURM automatically exports the environment variables you defined in the source shell, so that if you need to run your program myprogram in a controlled environment (i.e. specific library paths or options), you can prepare the environment in the origin shell being sure to find it in the interactive shell.

Batch

As usual on systems using SLURM, you can submit a script *script.x* using the command:

```
> sbatch script.x
```

You can get a list of defined partitions with the command:

```
> sinfo
```

You can simplify the output reported by the sinfo command specifying the output format via the "-o" option. A minimal output is reported, for instance, with:

> sinfo -o "%10D %20F %P"

which shows, for each partition, the total number of nodes and the number of nodes by state in the format "Allocated/Idle/Other/Total".

For more information and **examples of job scripts**, see section **Batch Scheduler SLURM**.

Submitting batch jobs on DAVIDE

On DAVIDE it is possible to submit jobs requiring different resources by specifying the corresponding partition.

Summary

In the following table you can find all the main features and limits imposed on the queue. In addition to user-based limits (max cores, memory, max walltime per job), partition-based limits are also imposed (see Max global resources per partition) on the maximum number of nodes which can be used by the running jobs on that partition.

The preemptable partition, dvd_usr_preempt has no such limitation: this partition allows to access additional nodes with respect to the allotted quantity for academic production (5 nodes), in a preemptable modality: the jobs submitted to the dvd_usr_preempt partition may be assigned nodes if available, and may be killed if the assigned resources are requested by jobs submitted to higher priority partitions (dvd_fua_prod and dvd_usr_prod); hence, we recommend its use only with restartable applications.

For EUROfusion users there are dedicated queues, please refer to the dedicated document.

SLURM Partition	**QOS**	max cores per job	**gpu per node**	**max walltime**	**max running jobs per user**	**max memory per job /node**	**priority**	**Max global resources per partition**	**notes**
dvd_all_serial (defalut partition)	noQOS	max = 1 (max mem = 15200MB)	0	04:00:00	-	15200MB	1	4 cores	defined on the login node
dvd_usr_prod	noQOS	min = 1 max = 32 (2 nodes)*	4	08:00:00	-	246000MB	100	5 nodes	--gres=gpu:N (N=1, 4)
dvd_usr_prod	dvd_qos_dbg	min = 1 max = 32 (2 nodes)*	4	00:30:00	-	246000MB	100	5 nodes	--gres=gpu:N (N=1, 4)
dvd_usr_preempt	noQOS	min = 1 max = 32 (2 nodes)*	4	08:00:00	-	246000MB	10	45 nodes	--gres=gpu:N (N=1, 4)

(*) 16 core per node, 8 threads per core

Programming environment

The programming environment of the DAVIDE cluster consists of a choice of compilers for the main scientific languages (Fortran, C and C++), debuggers to help users in finding bugs and errors in the codes, profilers to help in code optimization.

In general you must "load" the correct environment also for using programming tools like compilers, since "native" compilers are not available.

If you use a given set of compilers and libraries to create your executable, very probably you have to define the same "environment" when you want to run it. This is because, since by default linking is dynamic on Linux systems, at runtime the application will need the compiler shared libraries as well as other proprietary libraries. This means that you have to specify "module load" for compilers and libraries, both at compile time and at run time. To minimize the number of needed modules at runtime, use static linking to compile the applications.

Compilers

You can check the complete list of available compilers on MARCONI with the command:

```
> module available
```

and checking the "compilers" section. In general, the available compilers are currently:

GNU (gcc, g77, g95):

> module load gnu/6.4.0

LLVM Compiler Infrastructure:

> module load llvm/6.0.0

CUDA:

```
> module load cuda/9.2.88
```

After loading the appropriate module, use the "man" command to get the complete list of the flags supported by the compiler, for example:

```
> module load gnu
> man gfortran
```

There are some flags that are common for all these compilers. Others are more specifics. The most common are reported later for each compiler.

1. If you want to use a specific library or a particular include file, you have to give their paths, using the following options

```
-I/path_include_files specify the path of the include files
-L/path_lib_files -l<xxx> specify a library lib<xxx>.a in /path_lib_files
```

2. If yout want to debug your code you have to turn off optimisation and turn on run time checkings: these flags are described in the following section.

If you want to compile your code for normal production you have to turn on optimisation by choosing a higher optimisation level:

```
-O2 or -O3 Higher optimisation levels
```

Other flags are available for specific compilers and are reported later.

GNU compilers

The gnu compilers are available by loading the corresponding module, but they are not the best optimizing compilers.

The name of the GNU compilers are:

- **g77**: Fortran77 compiler
- **gfortran**: Fortran95 compiler
- **gcc**: C compiler
- **g++**: C++ compiler

The documentation can be obtained with the **man** command:

```
> man gfortan
```

NOTE: keep in mind that D.A.V.I.D.E. is based on Power8, so the compiling flags specific for the architecture should be searched in the "PowerPC" section of man documentation. Flags like "-mavx", used in other CINECA clusters based on x86, won't work in this one.

XL compiler

Not available.

Optimized Scientific Libraries

IBM ESSL are not available.

Parallel programming

OpenMPI

The OpenMPI libraries and wrappers are available on Davide. The package has been configured to support Mellanox Hierarchical Collectives (hcoll), and Mellanox Messaging support (mxm).

Additional information

Process/thread affinity:

Each node: 2 Power8 sockets (CoresPerSocket=8, ThreadsPerCore=8), Gres=gpu:tesla:4

The multithreading is active with 8 threads per physical core (128 logical cpus):

$ ppc64_cpu --info

```
Core   0:    0*    1*    2*    3*    4*    5*    6*    7*
Core   1:    8*    9*   10*   11*   12*   13*   14*   15*
```

```
Core   2:   16*   17*   18*   19*   20*   21*   22*   23*
Core   3:   24*   25*   26*   27*   28*   29*   30*   31*
Core   4:   32*   33*   34*   35*   36*   37*   38*   39*
Core   5:   40*   41*   42*   43*   44*   45*   46*   47*
Core   6:   48*   49*   50*   51*   52*   53*   54*   55*
Core   7:   56*   57*   58*   59*   60*   61*   62*   63*
Core   8:   64*   65*   66*   67*   68*   69*   70*   71*
Core   9:   72*   73*   74*   75*   76*   77*   78*   79*
Core  10:   80*   81*   82*   83*   84*   85*   86*   87*
Core  11:   88*   89*   90*   91*   92*   93*   94*   95*
Core  12:   96*   97*   98*   99*  100*  101*  102*  103*
Core  13:  104*  105*  106*  107*  108*  109*  110*  111*
Core  14:  112*  113*  114*  115*  116*  117*  118*  119*
Core  15:  120*  121*  122*  123*  124*  125*  126*  127*
```

Due to how the hardware is detected on a Power8 architecture, the numbering of (logical) cores follows the order of threading.

Since the nodes can be shared by users, Slurm has been configured to allocate one (physical) task per core by default.

NOTE: Without this option, by default one task will be allocated per thread on nodes with more than one ThreadsPerCore configured (as it is on Davide).

As a result of such configuration, for each requested task a physical core with all its 8 threads will be allocated to the task.

Since a physical core (8 HTs) is assigned to one task, a maximum of 16 tasks per node can be asked (--ntasks-per-node), corresponding (as mentioned) to receive 8 logical cpus per task. The use of --cpus-per-task is hence discouraged as a sbatch directive, potentially leading to incorrect allocation.

For instance:

#SBATCH -N 1

#SBATCH --ntasks-per-node=4

#SBATCH --gres=gpu:4

#SBATCH -A <account_name>

#SBATCH -p dvd_usr_prod

will result in having 32 logical cpus for the job cpuset, for instance 0-15, 64-79. Slurm automatically assigns the cpuset most appropriate to access the (2-per-socket) GPUs

$ nvidia-smi topo -m

```
        GPU0   GPU1   GPU2   GPU3   mlx5_0  mlx5_1   CPU Affinity
GPU0     X     NV2    SYS    SYS    NODE    SYS       0-15
GPU1    NV2     X     SYS    SYS    NODE    SYS       0-15
GPU2    SYS    SYS     X     NV2    SYS    NODE      64-79
GPU3    SYS    SYS    NV2     X     SYS    NODE      64-79
mlx5_0 NODE    NODE    SYS    SYS     X     SYS
mlx5_1  SYS    SYS    NODE    NODE    SYS     X


Legend:

  X    = Self
```

```
  SYS  = Connection traversing PCIe as well as the SMP interconnect between NUMA nodes (e.g., QPI/UPI)
  NODE = Connection traversing PCIe as well as the interconnect between PCIe Host Bridges within a NUMA node
  PHB  = Connection traversing PCIe as well as a PCIe Host Bridge (typically the CPU)
  PXB  = Connection traversing multiple PCIe switches (without traversing the PCIe Host Bridge)
  PIX  = Connection traversing a single PCIe switch
  NV#  = Connection traversing a bonded set of # NVLinks
```

If you have an hybrid MPI/OpenMP application, ask for a number of tasks so to obtain a number of logical cores equal to the product of the number of MPI processes * the number of OMP threads per task. For instance, for 4 MPI processes and 16 OMP threads per task, you need 64 logical cores, hence 8 physical cores:

#SBATCH -N 1

#SBATCH -ntasks-per-node=8

#SBATCH -A <account_name>

#SBATCH -p dvd_usr_prod

Once allocated the resource (with for instance a cpuset given by 0-15,64-79), the user can run their job specifying the desired OMP_NUM_THREADS and an adequate number of MPI processes:

export OMP_NUM_THREADS=16 (or set the --cpus-per-task as below)

srun -N 1 -n 4 (--cpu-bind=core) --cpus-per-task=16 -m block:block <exe>

The -m flag allows to specify the desired process distribution between nodes/socket/cores (the default is block:cyclic). Please refer to srun manual for more details on the processes distribution and binding. Note that the binding flag is required in order to obtain the correct process binding in case the -m flag is not used.

You can then set the OMP affinity to threads exporting the OMP_PLACES variable.

GPU Environment

NVIDIA GPUDirect RDMA technology is not supported on Davide because of hardware limitations: the Minsky processor does not have PCIe switches, and any attempt to use GPU Direct RDMA will incur system crashes.

Hence, cuda-aware MPI is not supported (CUDA APIs have to be used by MPI or serial processes to copy data to/from GPUs to/from Host via NVLink interconnect (intra-socket - GPU Direct PeertoPeer) or PCIe/IB (inter-socket/nodes). GPU Direct PeerToPeer is supported (via CUDA APIs) on the NVLinks.

In case of CUDA aware MPI, GPU buffers can be passed directly:

```
//MPI rank 0

MPI_Send(s_buf_d,size,MPI_CHAR,1,100,MPI_COMM_WORLD);

//MPI rank n-1

MPI_Recv(r_buf_d,size,MPI_CHAR,0,100,MPI_COMM_WORLD, &status);
```

On DAVIDE you need to do some recoding as follows:

```
//MPI rank 0

cudaMemcpy(s_buf_h,s_buf_d,size,cudaMemcpyDeviceToHost);

MPI_Send(s_buf_h,size,MPI_CHAR,1,100,MPI_COMM_WORLD);

//MPI rank 1

MPI_Recv(r_buf_h,size,MPI_CHAR,0,100,MPI_COMM_WORLD,&status);

cudaMemcpy(r_buf_d,r_buf_h,size,cudaMemcpyHostToDevice);
```