First access to Deucalion:
Compilation, Execution and Profiling

Objectives

- First contact with Deucalion
- Compiling and running the program in the Arm partition
- Using profiler tools in Deucalion

Introduction

This session is designed to provide you with the first steps to access and use Deucalion in a usual HPC workflow. We will use the typical HPC case study – matrix multiplication.

We provide a basic implementation of matrix multiplication. In the first phase, you will compile and run the provided matrix multiplication program in the C language. After execution, you will use a profile tool to gather performance metrics about the program. Finally, you can compare the performance results with an optimized version of the code.

This guide is designed to enable the reader to compile, run and use profiling tools available in Deucalion. This guide provides all the necessary commands. Some commands may be optional or incomplete – users are encouraged to adapt them to their specific configuration. Any optional or incomplete command will have an asterisk (*) next to it.

We recommend reading this guide carefully, as it briefly explains the objectives and purpose of each command.

Exercise 1 – First access Deucalion

a) <u>Access Deucalion via ssh</u>: When you access Deucalion, you do it via the login nodes. These login nodes have the same architecture as the x86 compute nodes. In these login nodes, the users may only edit their files, make small compilations for x86 and check on their jobs.

```
ssh -i <path-to-private-key> <username>@login.deucalion.macc.fccn.pt
```

b) <u>*Use modules:</u> Modules allow users to load dependencies, such as GCC, OpenMPI, and others. At Deucalion, we use a standard module management system Lmod. We recommend using modules from the same toolchain (e.g., if possible, every module you load should be compiled with the same version of GCC) to ensure compatibility. Lmod provides a set of commands to manage module environments effectively.

Search for modules available: `ml spider <module>` or `ml av <module>`
Load modules: `ml <module>`
List all modules loaded: `ml list`
Unload modules: `ml unload <module>`
Unload all modules: `ml purge`

Search and load OpenMPI version 5.

c) <u>*Manager users' jobs:</u> Login nodes should be used to manage user jobs. Deucalion uses the typical HPC job scheduling SLURM. SLURM allows the users to analyse the

Co-funded by the European Union
EuroHPC Joint Undertaking
CNCA Centro Nacional de Computação Avançada

state of the queue, see which nodes are available in each queue, and manage their jobs:

View all jobs in the queue: `squeue`
View nodes and partitions available: `sinfo`
Submit your batch job: `sbatch <script.sh>`
Submit and execute via command line: `srun`
Cancel your job, use job id: `scancel <jobid>`
Allocate computes nodes: `salloc`

Exercise 2 – Compile the program in the ARM partition

In Deucalion, compilation for the arm architecture is available in two different variants: cross-compiler and native compilation. We suggest using the native compilation, for which the user must access an arm computing node

a) Allocate and access arm node:

```
salloc -N1 -p dev-arm -t 4:00:00 -A f202500001hpcvlabepicurea
```

The messages will tell you which node you took so you can ssh that node via

```
ssh cna[xxxx]
```

you can also do `squeue --me` to check what node you allocated.

b) Use modules: In the Arm partition, it will be necessary to reconfigure the MODULEPATH, before using the modules. For this, do:

```
source /share/env/module_select.sh
```

c) Copy Matrix Multiplication example: In Deucalion, users are encouraged to store their files within their project folder for better efficiency. For this session, we have prepared a set of code files to provide you with an initial experience in compiling on Arm. To proceed, access the project folder and copy these files into a new folder.

```
cd /projects/F202500001HPCVLABEPICURE/
cp -r Session1 epicurexx
```

You can make your folder private by using the following commands:

```
chown -R :<user> epicurexx
chmod -R 700 epicurexx
```

d) Load GCC and compile the program: Three C compilers are installed on the arm partition: Fujitsu, Arm and GNU. We suggest using the GNU compiler, preferably the latest version (in this case, we will use the latest version compatible with the profiler). To load and compile the problem, simply run**:**

```
ml GCC/13.3.0
gcc -O3 -ftree-vectorize -march=armv8.2-a+sve -o mm_arm mm.c
```

e) Run the program: As you already have an allocated node, you can run directly:

```
time ./mm_arm
```

Exercise 3 – Run profiler tool (Score-P)

The **profiler (example Score-P)** helps track, measure, and optimise the performance of users, processes, or systems within a specified cluster. It ensures the cluster operates efficiently, identifies potential issues, and enables decisions to improve overall system performance, whether in computing, networking, or data management contexts.

a) Define your environment: The first line loads Score-P, and the last two lines enable profiling and tracing.

```
ml Score-P/8.4-gompi-2024a
export SCOREP_ENABLE_PROFILING=true
export SCOREP_ENABLE_TRACING=true
```

b) Compile code with Score-P:

```
scorep gcc -O3 -ftree-vectorize -march=armv8.2-a+sve -o mm_arm mm.c
```

c) View metrics available (list a hardware counters available):

```
`papi_avail -a` OR `papi_native_avail`
```

d) Define metrics (taken from c)):

```
export SCOREP_METRIC_PAPI="PAPI_TOT_INS,SVE_INST_RETIRED"
```

e) Run the code (compiled with Score-P):

```
./mm_arm
```

f) View the results:

Create new ssh session with -X (repeat the process in exercise 1a):

```
ssh -X -i <path-to-private-key> <username>@login.deucalion.macc.fccn.pt
```

Load CUBE (GUI): `ml CubeGUI`

Use Score-P: `cube scorep-<session>/profile.cubex`

Exercise 4 – Run with sbatch (use a login node)

To optimise the test of your versions and use fewer computer resources, you should create a script to submit a job. The submission must contain the set of commands needed to measure your program. The script starts with the interpreter you want to use, in this case, the bash shell located in the `/bin/bash` folder. Lines beginning with `#SBATCH` contain information required by the queue management system. On Deucalion, you will always need to define your account and specify the partition of your run. The remaining lines will contain the commands that will be executed on the compute node.

a) Complete a bash script: complete the batch script by including the commands used in Exercise 3. To do this, we'll use Vim, the text editor available on Deucalion.

```
vim submit.sh
```

b) Submit the job:

```
sbatch submit.sh
```

c) View the results: run the cube with the previously generated file.

```
cube scorep-<session>/profile.cubex
```

Exercise 5 – Analyse performance

Matrix multiplication is a fundamental operation in linear algebra with graphics, physics, machine learning, and scientific computing applications. Efficient implementations often leverage vector instructions (SIMD - Single Instruction Multiple Data) to accelerate computations on modern CPUs and GPUs.

a) Analyse the total number of instructions and the number of SVE instructions. Use the *cube* to view the number of SVE instructions.

b) Comment on the efficiency of this version.

Exercise 6 – Optimise code

The Fujitsu A64FX processor features SVE 512 capabilities, meaning it can process 8 doubles (512 bits / 64 bits in a double=8 doubles) in a single instruction. This exercise aims to take advantage of this processor's capabilities. To achieve this, a simple optimisation is changing the order in which the elements of the C matrix are computed. Specifically, the order of the loops will be modified (compare the *mm.c* version with *mm_opt.c*).

a) Create a new script to compile and run the new version.

b) What performance gain did you achieve ($T_{mm.c}$ / $T_{mm\_opt.c}$)? Did this match your expectations?

c) Rewrite the script to also display the number of L1 cache misses (PAPI_L1_DCM).

d) Analyse the results of the new metric. Compare the number of cache misses between the *mm.c* and *mm_opt.c* versions. What conclusions can you draw?

Exercise 7 – Use multithreading (OpenMP)

The Fujitsu A64FX is a high-performance ARM CPU with 48 cores and Scalable Vector Extension (SVE) supporting 512-bit width. We already showed how SVE can impact the results, but now we combine OpenMP (for multi-threading) with SVE (for SIMD vectorization) to maximise parallelism and vectorisation. The OpenMP version (view source code *mm_omp.c*) takes advantage of the 48 cores. In this implementation, the rows of matrix C (AxB=C) are computed in parallel.

a) Create a new script to compile and run the multithreaded version.

   Use the same compilation command but replace the source code file with *mm_omp.c* and add the *-fopenmp* flag to the compile command.

   Define the variable *export OMP_NUM_THREADS=X*. Test the application with X= 1,2,4,12,24 threads.

b) What performance gain did you achieve ($T_{mm.c}$ / $T_{mm\_omp.c}$)? Was this gain in line with your expectations?

c) Does each thread process the same number of instructions? Verify this using the profiler.

Exercise 8 – Use multiprocessor (MPI)

The Arm partition in Deucalion consists of 1,632 A64FX nodes, interconnected by a high-performance network (Infiniband). To leverage this computing power, we can use MPI. MPI is a fundamental tool for developing parallel and distributed programs in high-performance computing environments, providing an efficient and scalable method for communication between processes and enabling optimal use of multicore systems or computer clusters. The file *mm_mpi.c* contains a basic implementation of matrix multiplication using MPI. This version was developed for use with a profiler and was designed to simplify the code. It assumes that the number of processes specified in *mpirun* is a multiple of the matrix size (default is 1024).

a) Create a new script to compile and run with MPI.

   This requires multiple nodes, and you must define the number of tasks per node. Modify the line #SBATCH --nodes=1 as follows:

```
#SBATCH --tasks-per-node=12

#SBATCH --nodes=2
```

   Load OpenMPI:

```
ml OpenMPI/5.0.3-GCC-13.3.0
```

   Compile the program:

```
scorep mpicc -O3 -ftree-vectorize -march=armv8.2-a+sve -o mm_profiler_mpi_arm mm_mpi.c
```

   Run the program:

```
mpirun -np $SLURM_NTASKS ./mm_profiler_mpi_arm
```

b) Use Cube to find out if both processes are load-balanced.