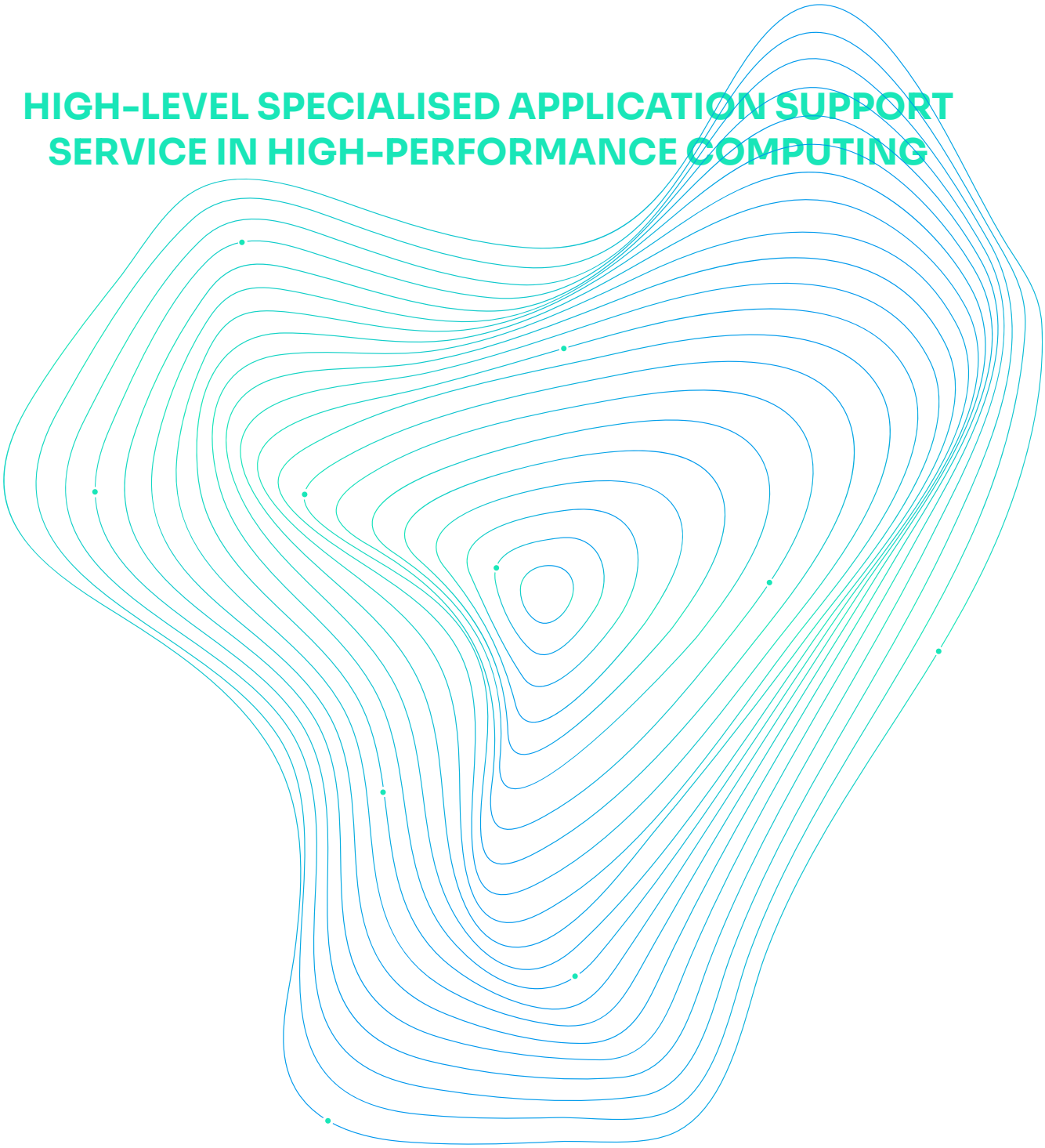
EPICURE

Unlocking European-level HPC Support

HIGH-LEVEL SPECIALISED APPLICATION SUPPORT SERVICE IN HIGH-PERFORMANCE COMPUTING



Co-funded by the European Union

EuroHPC Joint Undertaking

This project has received funding from the European High-Performance Computing Joint Undertaking under grant agreement No.101139786. Views and opinions expressed are, however, those of the author(s) only and do not necessarily reflect those of the European Union or EuroHPC Joint Undertaking. Neither the European Union nor the granting authority can be held responsible for them.

Maximizing GPU utilization with NVIDIA Multi-Process-Service for Quantum Monte Carlo simulations with the TurboRVB code

Authors
Laura Bellentani[1], Michele Casula[2], Tommaso Gorni[1]
[1]CINECA, Italy [2]Sorbonne Université, France

Abstract

This white paper describes the use of the NVIDIA Multi-Process Service (MPS) technology to enhance the performance of the Quantum Monte Carlo code TurboRVB on the Leonardo Booster partition at CINECA. The usage of NVIDIA MPS, allowing to efficiently schedule more than 1 MPI task per GPU, resulted in a factor-4 reduction in the GPU-hours needed to accomplish the simulations, without altering the wall time nor touching the source code. The reasons for this saving of resources are inspected with the NVIDIA profiling tools, and let us speculate that similar trends would be found in many scientific workloads other than Quantum Monte Carlo. This work was conducted as part of the EPICURE European initiative, providing high-level assistance to all the EuroHPC supercomputing projects.

1 Introduction

The EPICURE project [1], launched in February 2024, is a 4-year European initiative that aims to unify and strengthen the ecosystem of support services for the EuroHPC supercomputing projects. It includes sixteen research organizations, HPC centers, three Pre-Exascale and five Petascale EuroHPC machines, whose collaboration is a key aspect of the project. EPICURE introduces *Application Support Teams* (ASTs) dedicated to EuroHPC projects whose aim is to provide support for application porting, optimisation and scalability improvements, together with the skills and knowledge needed to fully utilize the EuroHPC clusters via webinars, hackathons, tutorials, and guidelines for users. By helping users make better use of these infrastructures, EPICURE supports progress across many areas, including artificial intelligence, scientific research, medicine, climate studies, and advanced engineering, thus contributing to strengthening Europe's competitiveness in supercomputing. To receive support from an EPICURE AST, the primary requirement is to have an active EuroHPC project, spanning from the larger regular and extreme projects to the smaller development and benchmarking ones. Specialists from the EPICURE consortium are selected to support users in Level 2 and Level 3 activities, including enabling the code on the machine, performance optimization, performance analysis, and scalability assessment. In addition to providing direct support, EPICURE shares HPC best practices with the broader supercomputing community through webinars and guidelines. This white paper describes the use of NVIDIA Multi-Process Service (MPS) technology [2] on

Table 1: Specifications of the Leonardo Booster partition.

Specification	Leonardo Booster
Nodes	3456
CPU per node	1× Intel Ice Lake 8358
Cores per node	32 cores
GPU per node	4× NVIDIA A100 64 GB
RAM per node	512 GiB DDR4 3200 MHz
Intra-node network	NVLink 3.0 (200 GB/s)
Network	200 Gbps Mellanox InfiniBand

the Leonardo Booster GPU partition at CINECA, see Tab. 1 and Refs. [3, 4], to enhance the performance of the Quantum Monte Carlo code TurboRVB [5, 6] for the EuroHPC Extreme allocation EHPC-EXT-2024E01-064. Specifically, the target workload has been optimized by reducing of about 4× the number of GPU-hours needed to accomplish it, without increasing the time-to-solution, thus saving energy and simulation costs. The results of our benchmarks show that NVIDIA MPS may also be beneficial for a wider range of scientific applications and workloads beyond the specific use case presented.

1.1 The TurboRVB code

TurboRVB is an open-source computational package for ab initio Quantum Monte Carlo (QMC) simulations of both molecular and bulk electronic systems [5, 6]. The code implements two types of QMC algorithms: Variational Monte Carlo (VMC) and the lattice-regularized variant of Diffusion Monte Carlo (LRDMC). A peculiarity of TurboRVB is the use of a resonating-valence-bond (RVB) type wave-function ansatz, which captures correlation effects beyond those captured by the Jastrow-Slater wave function commonly used in other QMC codes. Moreover, TurboRVB implements state-of-the-art algorithms to realize nodal-surface optimization at the VMC level, and to compute ionic forces, enabling structural optimization and molecular dynamics at the QMC level. Finally, TurboRVB supports twist-averaged calculations, allowing for a faster convergence to the thermodynamic limit of periodic systems [7].

TurboRVB is mainly written in Fortran, with a hybrid MPI+OpenMP parallelization. As routinely done in nearly all Monte Carlo codes, statistical sampling is performed using many independent Markov chains, called walkers. The MPI implementation of the sampling phase assigns one walker to each MPI task, resulting in an embarrassingly-parallel algorithm with nearly linear scaling. A more intensive communication between MPI tasks is needed instead during the optimization and branching steps. TurboRVB has been ported to NVIDIA GPUs within the TREX CoE [8], using an OpenMP offload + CUDA libraries framework. CUDA libraries calls (cuBLAS and cuSOLVER) are handled by a separate C-interface, so that porting to other GPU architectures would require very limited changes in the code base. TurboRVB is routinely deployed on HPC clusters worldwide for materials science and quantum chemistry research projects [9, 10, 11].

1.2 NVIDIA Multi Process Service (MPS)

Directive-based programming models for GPUs represent an interesting opportunity to port scientific workloads to accelerators without the burden of vendor-specific program-

ming languages, such as CUDA and HIP, which are known to provide greater control over performance but do not ensure the same level of readability and maintainability in the source code. These programming models require instructing the compiler about which loop need to be translated for the GPU architecture, while data movements between the host and the accelerator memories can be handled manually, to minimize CPU-GPU data transfers, which are the main bottlenecks in GPU-accelerated codes.

By using directive-based programming models, scientific applications can be ported to GPUs in a relatively straightforward manner, and eventually achieve significant speedups over CPU-only execution. However, such gains do not always translate into optimal efficiency, as GPUs may remain underutilized. This can arise from different factors, including latency overheads, non-optimal data movement patterns, partially serialized code regions, or simply kernels that are too small to fully exploit the computational capabilities of modern accelerators. To cope with this latter case, two options that do not require code refactoring are offered to the user on NVIDIA hardware: MPS [2] and MIG [12]. NVIDIA MIG allows to partition the same GPU device into up to 7 virtual devices, each with dedicated compute and memory resources, so that different processes can share the same GPU. On the other hand, NVIDIA MPS is a runtime feature that allows multiple CPU processes to share a single GPU concurrently, improving overall utilization and throughput by enabling concurrent kernel execution across multiple streams and reducing context-switch overhead compared to standard execution [13]. The two approaches have different pros and cons [14], and they can also be combined together [15]. In this paper, we focus on NVIDIA MPS due to its higher flexibility, since resources need not be assigned to each process in advance, and to its higher availability on HPC clusters, where MIG is often deactivated.

2 Benchmarks and analysis

The benchmark considered in this project is a twist-averaged Quantum Monte Carlo simulation of 128 hydrogen atoms with 32 independent twists, i.e. 32 different phase factors acquired by the wavefunction at the supercell boundary. For each twist, the wavefunction is sampled by 8 independent walkers, each walker providing 80 samples. After the sampling phase, an optimization step of the full wavefunction is performed, including the determinantal part. The sampling phase can be approximated to a highly embarrassing parallel case, whereas the optimization step is much more communication-intensive. For this specific benchmark, the optimization step takes roughly $1/16$ of the total wall time. This unit of work represents the most time-consuming part of the QMC simulations needed for the EuroHPC project EHPC-EXT-2024E01-064, and, more generally, for a typical QMC simulation, where hundreds or thousands of sampling+optimization steps are performed. The memory requirement per MPI process is estimated to be around 1 GiB, in line with the expected order of magnitude for simulations with hundreds of electrons with TurboRVB. Since the memory requirement per walker scales at most as $O(N^2)$, where N is the number of electrons, this workload is considered representative of a broad class of TurboRVB simulations, also from the memory point of view. By default, TurboRVB assigns each walker to a single MPI process, for a total of (32 twists $\times$ 8 walkers) $= 256$ MPI processes for the benchmark considered here. TurboRVB supports also multiple walkers per MPI process, but only with sequential processing, i.e., running one walker after the other; this configuration is then discarded, as it is always suboptimal in terms of performance and useful only when hardware resources are limited. In the following, we show

how the low-memory requirements and high independence of the MPI processes have been leveraged with NVIDIA MPS to use the computational resources more efficiently.

TurboRVB has been compiled on the Leonardo Booster partition with the software stack specified in the `modules-turborvb.sh` file, as shown in listing 2.1.

Listing 2.1: Software stack

```
module purge
module load intel-oneapi-mkl/2023.2.0
module load nvhpc/24.3
module load openmpi/4.1.6--nvhpc--24.3
```

In listings 2.2–2.3, we provide a prototype of the Slurm job adopted on Leonardo Booster to perform QMC simulations with TurboRVB and activating NVIDIA MPS with 2 MPI processes per GPU. These listings are specific to MPI-based simulations launched with the Slurm `srun` command; if `mpirun`/`mpiexec` is used instead, some adjustments may be needed, with particular attention to the propagation of the MPI environment variables setting the node and process IDs. Multithreading is disabled because it did not prove to be beneficial for this benchmark. The Leonardo Booster partition has a quite symmetric node architecture [4], hence process binding to GPUs is not crucial for performance. Still, an even division of the CPU cores per process is enforced via Slurm to divide cache resources equally among processes.

Listing 2.2: Slurm job

```
#!/bin/bash
#SBATCH --nodes=32
#SBATCH --ntasks-per-node=8
#SBATCH --cpus-per-task=4
#SBATCH --gres=gpu:4
#SBATCH --time=00:30:00
#SBATCH --exclusive
#SBATCH --partition=boost_usr_prod

source modules-turborvb.sh

export OMP_NUM_THREADS=1; export OMP_PLACES=cores; export OMP_PROC_BIND
    =close

TASKS_PER_GPU=$((SLURM_NTASKS_PER_NODE/SLURM_GPUS_ON_NODE))

srun -n $SLURM_NTASKS --cpu-bind=cores --cpus-per-task=
    $SLURM_CPUS_PER_TASK \
      ./mps-wrapper.sh ${TASKS_PER_GPU} turborvb-mpi.x < datasmin.input
          > datasmin.output
```

Listing 2.3: MPS wrapper

```
#!/usr/bin/env bash

TASKS_PER_GPU=$1
NODEID=${SLURM_NODEID}

export CUDA_MPS_LOG_DIRECTORY=./pipe${NODEID}
export CUDA_MPS_PIPE_DIRECTORY=./pipe${NODEID}
if [ $SLURM_LOCALID -eq 0 ]; then
        nvidia-cuda-mps-control -d
fi

MY_GPU=$((SLURM_LOCALID/TASKS_PER_GPU))
export CUDA_VISIBLE_DEVICES=$MY_GPU

BINDINGS="task ${SLURM_LOCALID}, bound to GPU ${CUDA_VISIBLE_DEVICES} :
    $(taskset -cp $$)"
echo "# [BINDING REPORT, NODE(${NODEID}) $(hostname)] ${BINDINGS}"

${@:2}

if [ $SLURM_LOCALID -eq 0 ]; then
        echo quit | nvidia-cuda-mps-control
        echo "Stopped MPS deamon on node ${NODEID}"
fi
```

The Slurm job prototypes to run TurboRVB with NVIDIA MPS (listings 2.2–2.3) have been integrated in a JUBE framework [16] to standardize the job submission and analysis. The total execution time has been gathered from the `elapsedraw` time of step zero via the Slurm `sacct` utility, which yields the whole `srun` wall time with the precision of one second. It must be noted that this time includes the MPS start and stop times on purpose, so to keep track of their overhead with respect to the standard execution without MPS. The QMC workload described above has been run on the Leonardo Booster partition, varying only the number of MPI tasks per GPU, hence the total number of nodes.

The results of our benchmark are reported in Fig. 1. The time-to-solution is practically unaltered up to MPI 4 tasks per GPU, while less than a 30% increase is observed with 8 MPI tasks per GPU, indicating that a reduction in the required resources by a factor of 4 is possible without any impact on the time-to-solution. Looking at the GPU-hours consumption, the minimum consumption is instead reached with the slower 8-tasks-per-GPU configuration, as the 30% increase in wall time is compensated by the factor-2 reduction in the required resources, globally lowering the needed GPU-hours from 7.9 to 1.3. Finally, comparing the wall times for one MPI task per GPU with and without MPS shows that enabling MPS does not add any significant overhead.

3 Profiling

To further inspect the cause of this efficiency gain, we use the `nvidia-smi` utility to monitor GPU utilization at runtime. We consider the following metrics, as defined in the `nvidia-smi --help-query-gpu` section of `nvidia-smi`:

- `GPU utilization`: percent of time over the past sample period during which one or more kernels were executing on the GPU.

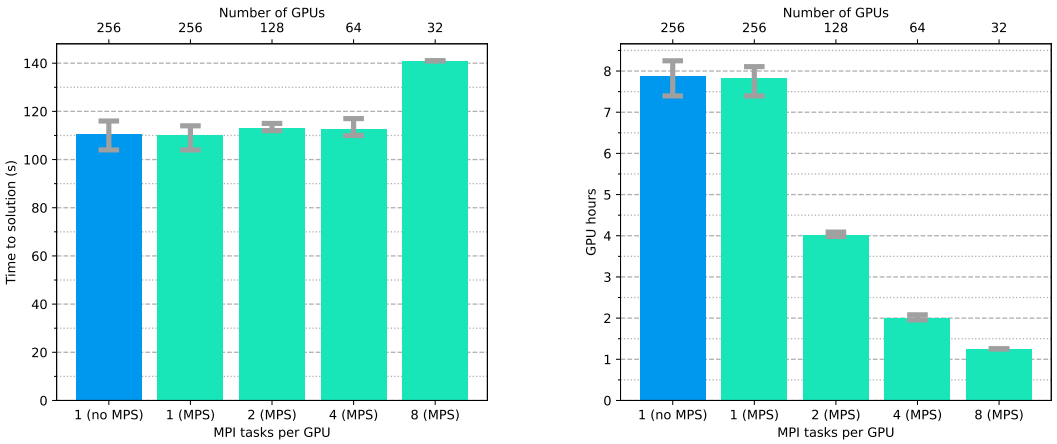


Figure 1: Time to solution (left) and GPU hours (right) for the TurboRVB workload considered in this paper, launched with 256 MPI ranks. The reference time-to-solution (no MPS, blue bar) has been averaged over 9 repetitions of the same run, while the MPS time-to-solution (light-blue bars) has been averaged over 3 repetitions per configuration. The error bars represent the maximum and minimum values obtained for each configuration. Each time has been obtained with a precision of one second using Slurm's `sacct` command.

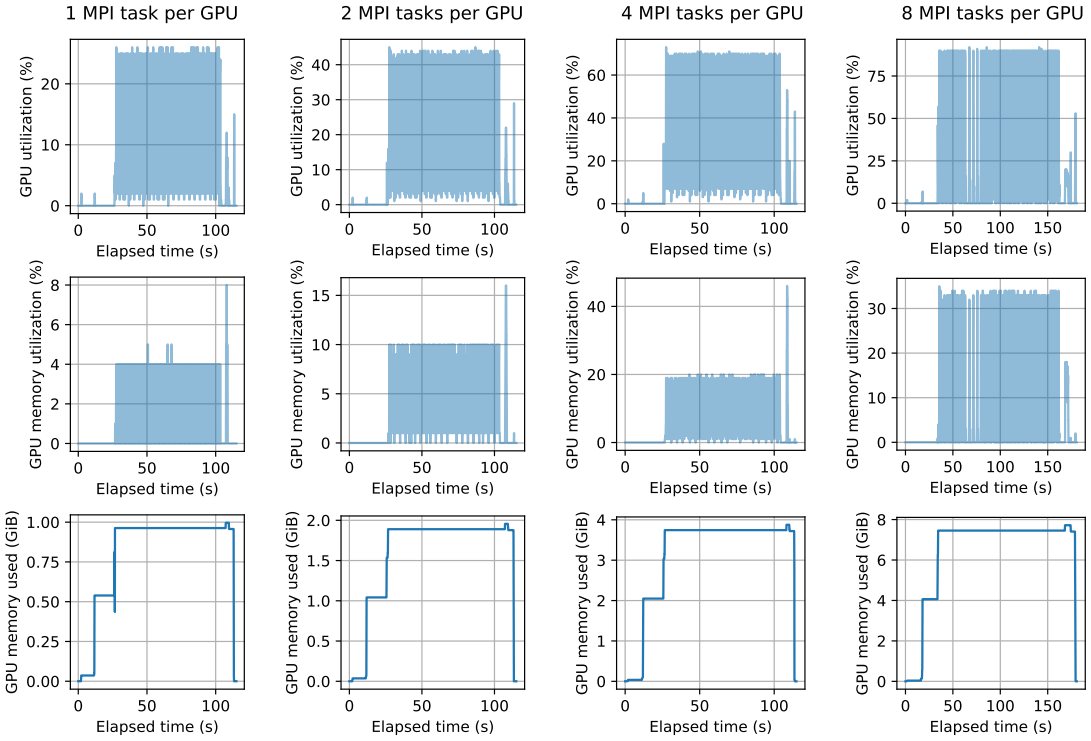


Figure 2: GPU overall and memory utilization for a different number of MPI tasks per GPU and MPS enabled. All the reported values have been gathered with `nvidia-smi` with a 10 ms resolution. Values are reported for a single GPU, as the workload is evenly distributed across all MPI tasks.

- `Memory utilization`: percent of time over the past sample period during which global (device) memory was being read or written.
- `Memory used`: total memory allocated by active contexts.

One `nvidia-smi` process per node is launched in the background with a sampling period of 10 ms as shown in listing 3.1.

Listing 3.1: nvidia-smi invokation

```
nvidia-smi \
--query-gpu=timestamp,index,utilization.memory,utilization.gpu,memory.
    used \
--format=csv -lms 10 \
--filename=node_${SLURM_NODEID}.out
```

We notice that the considered TurboRVB workload is particularly favorable for this setup, as only 1 thread per MPI process is used, so that, except for the 8-MPI-tasks-per-GPU case, a physical CPU core can be completely dedicated to the monitoring task. Indeed, less than 10% overhead is reported for the 1-,2-, and 4-MPI-tasks-per-GPU cases, while it exceeds 20% in the 8-MPI-tasks-per-GPU case. The above metrics are reported as a function of time in Fig. 2. The TurboRVB workload is evenly distributed across all MPI tasks, thus results for a single GPU device are reported in the figure.

All three metrics show a uniform increase in GPU activity as the TurboRVB processes are packed into the same GPU. As anticipated, the memory burden is quite limited in this kind of workload, with `nvidia-smi` reporting slightly less than 1 GiB per MPI process. Memory is unlikely to be the cause of saturation at 8 MPI tasks per GPU, as Leonardo A100 GPUs have 64 GiB of memory each. On the contrary, saturation seems to occur at the computing level, where the GPU utilization peaks at 25 %, 45 %, 70 % and 90 % for the 1-,2-,4- and 8-MPI-tasks-per-GPU cases, respectively. This trend points to a saturation of the available streaming multiprocessors. Moreover, in the 8-MPI-tasks-per-GPU case, some idle gaps become visible in both memory and GPU utilization, pointing again to difficulties for the MPS scheduler to accommodate kernels on the compute units.

This scenario is confirmed by profiling with NVIDIA Nsight Systems all the MPI processes residing on the same GPU, with a wrapper placed between the `srun` invocation and the MPS wrapper to profile a selected subset of MPI processes. The wrapper is shown in listing 3.2. Notice that the `MPI_MAX_RANK` can be exported by the user in the Slurm job, or passed to the wrapper as a parameter.

Listing 3.2: Nsight Systems wrapper

```
#!/usr/bin/env bash

export procid=$SLURM_PROCID
if ((procid >= 0 && procid <= $MPI_MAX_RANK)); then
    echo "Running on process $procid profiler $NSYS"
    nsys profile -e NSYS_MPI_STORE_TEAMS_PER_RANK=1 -t mpi,cuda -o
        profile_%q{procid} "$@"
else
    "$@"
fi
```

Figure 3: Snapshot of the timeline view from the NVIDIA NSight System GUI. The x-axis refers to the same 5 ms window of time, while the y-axis reports the GPU kernel and memory operations. The color blue indicates the execution of computational kernels, while green, red, and purple indicate data movements.

The results are shown in Fig. 3, with all four traces displaying a 5-ms interval where GPU offloading occurs. As it can be seen, roughly 7 chunks of equivalent operations occur in the 1-MPI-task-per-GPU case, with idle gaps of roughly 0.5 ms between them. These gaps are successfully exploited by the MPS scheduler to accommodate many more similar chunks on the same GPU, as shown in the 2- and 4-MPI-tasks-per-GPU cases. On the contrary, in the 8-MPI-tasks-per-GPU case, only 7 processes out of 8 are able to run in parallel; moreover, a single process completes only 4 or 5 chunks of operations in the same 5-ms windows, which confirms the saturation of the GPU computing resources observed with `nvidia-smi`.

4 Conclusions

We highlighted how MPS can be used to optimize the use of resources on NVIDIA GPU architecture for a standard material-science workflow, such as ab initio Quantum Monte Carlo. Together with summarizing the activities carried out by the EPICURE application support team on Quantum Monte Carlo simulations with TurboRVB code, this paper serves as a guideline to enable the tool on the Leonardo supercomputing cluster and to assess the performance of its execution with NVIDIA NSight Systems and `nvidia-smi`.

Here, we demonstrate that, with MPS enabled, the program can perform the same

256-MPI-task simulation with a significant reduction in terms of GPU hours and without modifying the source code. An efficient concurrent use of the same GPU by different MPI task has been observed up to 4 MPI tasks per GPU, with a net factor-4 gain in terms of GPU-hours for the same time-to-solution. Some serialization occurs with 8 MPI tasks per GPU causing a 30% increase in the time-to-solution, still leading to an over factor-6 gain in term of GPU-hours.

We remark that the resulting benefit, apart from an easier scheduling of the job and consequently a faster project pace, is also a considerable reduction of the energy consumed by the simulations, which we aim at quantifying in future work.

Data reproducibility

The input files needed to reproduce these data on Leonardo Booster can be found in the EPICURE repository at the URL: `https://opencode.it4i.eu/epicure/epicure-sw/-/tree/main/white-papers/NvidiaMPS-TurboRVB`.

Acknowledgements

The authors thank the EPICURE project team and all partner institutions for their contributions. Computing time was provided by CINECA under project allocation EHPC-EXT-2024E01-064.

This project has received funding from the European High-Performance Computing Joint Undertaking under grant agreement No. 101139786. Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or EuroHPC Joint Undertaking. Neither the European Union nor the granting authority can be held responsible for them.

This work is co-funded by the European Union's Horizon Europe research and innovation program under grant agreement No. 101093475.

How to cite this document

Laura Bellentani, Michele Casula, Tommaso Gorni (April 15, 2026). *Maximizing GPU utilization with NVIDIA Multi-Process-Service for Quantum Monte Carlo simulations with the TurboRVB code* EPICURE White Paper EPICURE-WP-2025-001.

References

[1] *EPICURE webpage*. URL: `https://epicure-hpc.eu`.

[2] *Nvidia MPS documentation*. URL: `https://docs.nvidia.com/deploy/mps/index.html`.

[3] *CINECA's Leonardo documentation*. URL: `https://docs.hpc.cineca.it/hpc/leonardo.html`.

[4] Matteo Turisini, Giorgio Amati, and Mirko Cestari. "Leonardo: A pan-european pre-exascale supercomputer for hpc and ai applications". In: *JLSRF* 8 (2023), A186. DOI: `https://doi.org/10.17815/jlsrf-8-186`. eprint: `https://arxiv.org/abs/2307.16885`.

[5] *TurboRVB github repository*. URL: https://github.com/sissaschool/turborvb.

[6] Kousuke Nakano et al. “TurboRVB: A many-body toolkit for ab initio electronic simulations by quantum Monte Carlo”. In: *The Journal of Chemical Physics* 152.20 (May 2020), p. 204121. ISSN: 0021-9606. DOI: 10.1063/5.0005037. eprint: https://pubs.aip.org/aip/jcp/article-pdf/doi/10.1063/5.0005037/16745553/204121_1_online.pdf. URL: https://doi.org/10.1063/5.0005037.

[7] Mario Dagrada et al. “Exact special twist method for quantum Monte Carlo simulations”. In: *Phys. Rev. B* 94 (24 Dec. 2016), p. 245108. DOI: 10.1103/PhysRevB.94.245108. URL: https://link.aps.org/doi/10.1103/PhysRevB.94.245108.

[8] *TREX CoE*. URL: https://trex-coe.eu/.

[9] Kousuke Nakano et al. “TurboGenius: Python suite for high-throughput calculations of ab initio quantum Monte Carlo methods”. In: *The Journal of Chemical Physics* 159.22 (Dec. 2023), p. 224801. ISSN: 0021-9606. DOI: 10.1063/5.0179003. eprint: https://pubs.aip.org/aip/jcp/article-pdf/doi/10.1063/5.0179003/18281773/224801_1_5.0179003.pdf. URL: https://doi.org/10.1063/5.0179003.

[10] Giacomo Tenti, Kousuke Nakano, and Michele Casula. “Self-Consistency Error Correction for Accurate Machine Learning Potentials from Variational Monte Carlo”. In: *Journal of Chemical Theory and Computation* 21.19 (2025). PMID: 40990331, pp. 9335–9346. DOI: 10.1021/acs.jctc.5c00715. eprint: https://doi.org/10.1021/acs.jctc.5c00715. URL: https://doi.org/10.1021/acs.jctc.5c00715.

[11] Marco Cherubini, Abhishek Raghav, and Michele Casula. “Ferroelectric quantum critical point in superconducting hydrides: The case of H _3 S”. In: *arXiv preprint arXiv:2602.00833* (2026).

[12] *Nvidia MIG documentation*. URL: https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html.

[13] Darren Hsu, David Clark, and Janet Paulsen. *Maximizing OpenMM Molecular Dynamics Throughput with NVIDIA Multi-Process Service*. 2025. URL: https://developer.nvidia.com/blog/maximizing-openmm-molecular-dynamics-throughput-with-nvidia-multi-process-service/.

[14] Jorge Villarrubia et al. “A comprehensive evaluation of spatial co-execution on GPUs using MPS and MIG technologies: J. Villarrubia et al.” In: *The Journal of Supercomputing* 82.4 (2026), p. 175.

[15] Alan Gray and Szilárd Páll. *Maximizing GROMACS Throughput with Multiple Simulations per GPU Using MPS and MIG*. 2021. URL: https://developer.nvidia.com/blog/maximizing-gromacs-throughput-with-multiple-simulations-per-gpu-using-mps-and-mig/.

[16] *The JUBE benchmark framework*. URL: https://apps.fz-juelich.de/jsc/jube/docu/introduction.html.