PPMT: Parallel Power Measurements Tool

David Goz [a,*,a,b], Giovanni Lacopo [a,b], Cristiano Urban [a], Giuliano Taffoni [a,b] and Cristina Knapic [a]

[a] *INAF - Osservatorio Astronomico di Trieste, via Tiepolo 11, Trieste, 34143, TS, Italy*
[b] *ICSC – Centro Nazionale di Ricerca in High Performance Computing, Big Data e Quantum Computing, via Magnanelli 2, Bologna, 40033, BO, Italy*

Abstract

The Parallel Power Measurements Tool (PPMT) is a C++ library designed for real-time, hardware-aware energy profiling of HPC applications across CPU and GPU architectures, supporting both serial and MPI-distributed environments. Built on top of the PMT library, PPMT extends its capabilities to distributed-memory systems by introducing topology-aware MPI communication hierarchies (socket, node, world) and device-specific energy aggregation strategies: maximum energy per socket (for RAPL-based CPU measurements) and summed energy across GPUs (for NVML/rocm-smi counters). This enables accurate, scalable attribution of energy consumption to specific computational phases, such as data transfers, kernels, or idle periods, with a negligible impact on the application performance. PPMT delivers hierarchical reports (per-rank, socket, node, global) that reveal energy hotspots and imbalances in large-scale simulations, directly supporting optimization of energy-efficiency in astrophysical codes. This work is a direct implementation of the USC-C Computing strategy at INAF, which prioritizes *Green Computing* through systematic energy profiling, hardware-aware algorithm co-design, and sustainable HPC operations. By quantifying the energy cost of scientific computation per watt, PPMT empowers researchers to maximize science output while minimizing environmental and economic footprint, aligning with the global imperative for energy-efficient exascale computing.

Keywords: *HPC, energy-efficiency, energy-profiling, CPU, GPU, GitLab@INAF, Green Computing, ICSC, USC-C*

1. INTRODUCTION

In modern astrophysics, understanding the universe's most complex phenomena, such as galaxy formation and evolution, requires simulations of unprecedented scale and fidelity. These simulations run on exascale supercomputers composed of thousands of GPU-accelerated nodes, leveraging massive parallelism to solve nonlinear, multi-physics equations over billions of computational cells and trillions of particles. In this context, energy efficiency is not merely desirable, it is a non-negotiable enabler of scientific discovery.

1. **Scale Demands Efficiency:** While GPUs deliver orders-of-magnitude higher performance-per-watt than CPUs for parallel workloads, modern astrophysical codes can push them to their thermal and power limits. A single GPU node can consume 300–700 W under full load; in a system with 10,000+ nodes, this translates to 3–7 MW of power draw per machine (equivalent to a small town). Without energy-efficient design, the power budget is consumed by idle cycles, memory bottlenecks, or inefficient data movement, not scientific computation
2. **Simulation Fidelity vs. Energy Budget:** Astrophysical simulations are inherently expensive. Energy efficiency allows researchers to
   i) Run longer simulations;
   ii) Increase resolution without exceeding power caps;
   iii) Perform ensemble runs (parameter exploration) to quantify uncertainties, critical for interpreting next-generation observatories.

   Without efficiency gains, scientists are forced to compromise on physics fidelity or reduce sample sizes, potentially risking missed discoveries.
3. **Thermal and System Reliability Under Load**: GPU-intensive astrophysics codes run for days or weeks without interruption. Energy-efficient workloads ensure:
   i) Predictable runtime for time-critical simulations
   ii) Lower failure rates in long-running jobs.
4. **Economic and Institutional Sustainability**: Operating a GPU-cluster supercomputer is dominated by energy costs, often exceeding hardware procurement after 2–3 years. For public-funded astrophysics facilities, energy efficiency directly determines:
   i How many research teams can access the machine;
   ii Whether the facility can sustain operations under shrinking budgets;
   iii Whether funding agencies will approve future upgrades or new installations.
5. **The Path Forward: Efficiency as a Co-Design Principle**: In astrophysics HPC, energy efficiency must be co-designed with algorithms and hardware:
   - *Algorithmic efficiency:*
     i) Minimize data movement between memory levels;
     ii) Do the most scientific work with the fewest computational operations;
     iii) Maximize hardware utilization (e.g., GPU cores, HBM bandwidth).
   - *Hardware-aware optimization:* Tailoring computational kernels and data layouts to exploit the specific architectural features of modern CPUs and GPUs, such as SIMT execution model, cache hierarchies, NVLink / PCIe bandwidth for multi-GPUs communication.

For astrophysics on GPU-accelerated supercomputers, energy efficiency is the silent enabler of the next breakthrough. In this domain, the most important metric is no longer just how fast you compute, it's how much science you get per watt. Energy efficiency is no longer a constraint, it is the foundation of sustainable, scalable, and transformative astrophysical discovery.

The complexity of modern HPC nodes further complicates energy profiling. A typical computational node in today's HPC clusters comprises four CPU sockets, organized into two NUMA regions, each with independent power domains and memory controllers, and is equipped with four GPU accelerators, each with its own dedicated High Bandwidth Memory (HBM), direct NVLink interconnects for peer-to-peer communication, and independent power/thermal management units, as shown in Fig. 1. This heterogeneous, multi-domain architecture means that energy consumption is not uniform across the node: CPU cores, GPU kernels, memory transfers, and inter-GPU communication each draw power from distinct electrical circuits. Without explicit knowledge of this topology, energy measurements become spatially ambiguous or temporally misaligned, leading to misleading conclusions about which code regions are truly energy-intensive.

*Corresponding author: david.goz@inaf.it (David Goz).

In this context, we present the **Parallel Power Measurement Toolkit (PPMT)**, a parallel high-level library capable of collecting power measurements on various architectures, such as CPUs and GPUs. PPMT is meant to be easily used to probe specific application regions to measure in detail power consumption and to enable scientifically rigorous, hardware-aware energy profiling.

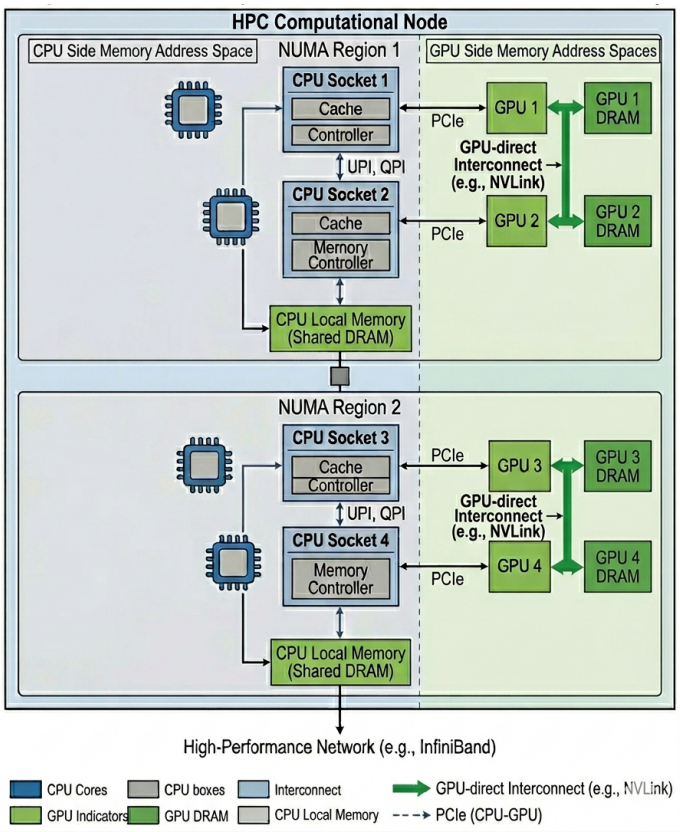

Figure 1. Hardware topology of a representative HPC computational node: 4 CPU sockets (2 NUMA regions), 4 GPUs with dedicated HBM, interconnected via NVLink and PCIe Gen5. Power sensors are located at the socket and GPU level. PPMT maps measurements to these physical domains to enable accurate attribution.

2. PMT BACKGROUND

The PMT is a lightweight, C++-based library designed for real-time monitoring of power consumption on HPC systems, supporting CPUs and GPUs across multiple architectures [2]. It interfaces directly with vendor-specific hardware APIs, such as NVML for NVIDIA GPUs, rocm-smi for AMD GPUs, and RAPL for Intel CPUs, to collect power data. The core of PMT operates via a background thread that continuously samples power usage from the selected backend, with sampling frequency constrained by hardware capabilities.

PMT is designed for easy integration into C++ and Python applications, is installable via CMake or Spack, and is Linux-only, making it suitable for deployment on typical HPC environments. The library, released under Apache License Version 2.0, is available at the link: https://git.astron.nl/RD/pmt.

PPMT, the subject of this report, extends PMT into the distributed-memory domain by enabling concurrent, synchronized power measurements across multiple MPI processes, transforming PMT from a single-process tool into a parallel, scalable framework for monitoring energy consumption in MPI-based HPC applications.

3. PPMT OVERVIEW

PPMT is a C++ library that extends the PMT to support real-time, synchronized energy and performance profiling in distributed-memory applications, particularly those using MPI. It enables multi-process, multi-node measurement of CPU/GPU power consumption while preserving PMT's low-overhead, sensor-accurate sampling.

1. **Core API (`ppmt.hpp`, `ppmt.inl`)**
   The central interface is the templated, 64-byte-aligned PPMT class, designed for low-overhead instrumentation:
   - provides **`startCPU()`/`stopCPU()`** and **`startGPU()`/`stopGPU()`** methods to delimit regions of interest;
   - uses aligned allocators (**`utility::AlignedAllocator<..., 64>`**) for all internal buffers to prevent false sharing and ensure cache-efficient, low-overhead profiling in multi-core environments. This design choice is particularly important in HPC, where precise power sampling must not interfere with application performance due to cache-line contention. It is worth noting that, although this alignment strategy anticipates potential future expansion into multi-threaded contexts (e.g., OpenMP), PPMT v0.1.0 (presented in the current technical report) does not yet support threaded execution models. In the current implementation, each MPI rank runs profiling operations in a single-threaded manner, relying on PMT's background thread only for sensor sampling, while the PPMT instrumentation itself remains synchronous and rank-local. Support for explicit multi-threading (e.g., per-thread profiling within a rank) is therefore a plausible direction for future development, and the use of aligned data structures already places the codebase on a path toward such enhancements;
   - dynamically resizes internal tag and iteration buffers, supporting flexible profiling workloads;
   - internally stores per-device **`EnergyState`** (joules, watts, seconds) and metadata **`ProfilerTag`** for post-processing;
   - supports optional FLOP counting integration (via PAPI [3]) for GFLOP/s/W efficiency metrics.
   - **C ABI layer**: a thin C wrapper (**`ppmt_c.h`**, **`ppmt_c.cpp`**) exposes stable foreign-function interface (FFI) symbols, enabling integration with legacy Fortran/C/Python codes.
2. **MPI infrastructure (`mpi_impl.hpp`, `parallel_interface.hpp`, `mpi_utils.hpp`)**
   PPMT embeds a hierarchical MPI topology management layer, essential for coarse- and fine-grained power aggregation:
   - **Communicator Hierarchy**: at initialization, PPMT builds three nested communicators:
     - **`worldComm`**: full MPI communicator (global scope);
     - **`nodeComm`**: ranks residing on the same physical node (detected via **`MPI_Comm_split`** on hostname or socket topology);
     - **`socketComm`**: groups ranks within the same CPU socket, using NUMA-aware detection via **`/proc/cpuinfo`** or similar.
   - **Leadership Model**: each socket elects a socket master (rank 0 in **`socketComm`**). Socket masters coordinate intra-node aggregation and report to node-level masters, and ultimately to world rank 0 for global reporting;
   - **Error Handling**: a dedicated **`MPIException`** class in **`mpi_utils.hpp`** translates low-level MPI return codes into human-readable messages (e.g., **`MPI_ERR_COMM`**);
   - **C ABI layer**: a thin C wrapper (**`parallel_c_api.h`**, **`c_wrapper.cpp`**) exposes stable foreign-function interface (FFI) symbols, enabling integration with legacy Fortran/C/Python codes.
3. **Energy Reporting Strategy** PPMT employs distinct energy aggregation strategies for CPU and GPU devices, reflecting fundamental differences in their power monitoring granularity and hardware architecture. These strategies are designed to preserve

measurement accuracy while minimizing reporting overhead in distributed environments.

- **CPU Energy (RAPL-based)**: On x86 CPUs, power consumption is measured via the Running Average Power Limit (RAPL) interface, which provides per-socket energy readings, not per-core or per-process. Consequently, when multiple MPI ranks co-reside on the same CPU socket, they share the same RAPL counter. `PPMT` v0.1.0 assumes that MPI processes on the same socket are workload-balanced and execute synchronized computational phases (e.g., in a well-balanced MPI domain decomposition). Under this assumption, the energy consumed for a given profiling label (e.g., a kernel or loop) is reported as the **maximum** energy reading observed across all ranks on that socket during the labeled interval. This choice reflects the conservative interpretation that the socket's total power draw is dominated by the most active rank, and summing would overcount shared resources (e.g., memory controller, uncore).
  Future Enhancement: This assumption may break under load imbalance or asynchronous execution. To relax it, future versions of `PPMT` will track precise `start` and `stop` timestamps for each profiling label per rank, enabling time-aligned energy integration across co-located ranks, thereby allowing accurate, temporally-resolved aggregation even under unbalanced workloads.
- **GPU Energy (NVML/rocm-smi-based)**: GPU power monitoring is performed via vendor-specific APIs (NVML for NVIDIA, rocm-smi for AMD), which provide per-GPU energy counters. Since each GPU is a physically distinct device with independent power rails, energy consumption is inherently additive across GPUs. Therefore, for a given profiling label, `PPMT` reports the **sum** of energy consumed across all GPUs visible to the MPI ranks on a node. This approach is both physically accurate and scalable: even if multiple MPI ranks are assigned to the same GPU (e.g., via multi-process service or CUDA context sharing), the underlying hardware counter still reflects total energy usage, and summing across GPUs yields the correct aggregate for the application phase.
  Note: `PPMT` does not currently distinguish between GPU-sharing and GPU-exclusive MPI ranks; it assumes that each GPU is either exclusively used by one rank or that shared usage is accounted for by the hardware counter's inherent summation semantics. `PPMT` currently supports NVIDIA GPUs exclusively, with AMD support planned for future releases.

`PPMT` supports hierarchical energy reporting across multiple levels of the MPI topology:

- **Per-rank** [`PPMT_report_all_processes.txt`]:
  Raw CPU and GPU energy (in joules), duration (seconds), and average power (watts) for each labeled region, recorded locally by each MPI rank.
- **Socket-level** [`PPMT_report_sockets.txt`]:
  For CPU, the maximum energy across all ranks on the same socket; for GPU, the sum of energy across all GPUs accessible from that socket.
- **Node-level** [`PPMT_report_nodes.txt`]:
  For CPU, the sum of energy across all sockets on the node; for GPU, the sum of energy across all GPUs on the node.
- **Global (worldComm)** [`PPMT_report_world.txt`]:
  Final consolidated report, aggregated across all nodes.

This hierarchical, device-aware reporting strategy ensures that `PPMT` delivers both *architecturally accurate* and *computationally meaningful* energy profiles — enabling users to distinguish between socket-level CPU bottlenecks and GPU-wide energy consumption patterns in large-scale MPI applications.

4. **Key Design Goals & Use Cases**
   - **Scalability**: minimizes synchronization overhead via asynchronous sampling (`PMT`'s background thread per rank);
   - **Accuracy**: leverages vendor APIs (NVML, RAPL, rocm-smi) directly via `PMT`;
   - **Portability**: works in serial (non-MPI) mode for prototyping and scales to 10^4+ cores in MPI mode;
   - **Transparency**: instrumentation is non-invasive—only adds **`start*()/stop*()`** tags around application loops or kernels.

In summary, `PPMT` combines `PMT`'s hardware-agnostic sensor access with a purpose-built MPI orchestration layer, enabling system architects and HPC developers to quantitatively evaluate energy–performance trade-offs across large-scale, distributed computations.

4. PPMT USAGE EXAMPLES

We show a comprehensive set of test cases demonstrating its usage across serial, CPU-only, CPU/GPU, and MPI-distributed scenarios. We used three systems: a Small-GPU node (1× NVIDIA RTX 500 Ada), a Large-CPU cluster (2× 96 Intel-core nodes) and a Dual-GPU node (2× NVIDIA RTX 2080). The Large-CPU system was used for CPU-bound tasks, while Small-GPU and Dual-GPU were used to study scaling with GPU count.

Table 1. Hardware topology of three compute machines: nodes, cores per node, and GPUs per node

Platform	Nodes	Cores per Node	GPUs per Node
`Small-GPU`	1	16	1
`Large-CPU`	2	96	0
`Dual-GPU`	1	16	2

All tests (available in the repository `ppmt/examples`) use the same minimal instrumentation pattern: wrap target code regions with **`startCPU()/stopCPU()`** and **`startGPU()/stopGPU()`** calls, then invoke **`show()`** for aggregated output.

4.1. Serial CPU Test (`examples/app.cpp`, `examples/app.c`)

This example demonstrates how to use `PPMT` in a serial (single-process) environment, with optional `OpenMP` parallelism inside the instrumented region, but not across `PPMT` itself. `Small-GPU` platform is exploited.

- **Code Structure and Instrumentation**

  1. **Header inclusion**
     Only one header needs to be included to use `PPMT`:

```
=== "C++"
    // Import the native C++ wrapper
    #include <ppmt/ppmt.hpp>

=== "C"
    // Import the core C API
    #include <ppmt/ppmt_c.h>
```

  2. **Initialization**
     The profiler is initialized as:

```
// ================================================
// C++ Application (Object on the Stack)
// ================================================
// Uses constructor with default arguments:
```

```
// CPUs = true, GPUs = false,
// no specific GPU IDs
PPMT ppmt(true);

// ================================================
// C Application (Pointer to an Opaque Structure)
// ================================================
// Explicit factory function requiring all
// configuration arguments
ppmt_c *profiler = ppmt_create(1, 0, NULL,
                               0, 0, 0);

// Note: Don't forget to free
// the memory when done in C!
ppmt_destroy(profiler);
```

Since the library compilation option `PPMT_WITH_MPI` is not required for this test (serial mode), `PPMT` automatically falls back to the `SerialImpl`, where all MPI-related methods (`isWorldMaster()`, `getWorldSize()`, etc.) return serial-safe defaults (i.e., rank = 0, size = 1)

3. **Instrumenting regions**
   CPU regions are delimited using:

```
// ================================================
// C++ Application
// ================================================
ppmt.startCPU("tag_name");
// ... heavy computation here ...
ppmt.stopCPU("tag_name");

// ================================================
// C Application
// ================================================
ppmt_startCPU(profiler, "tag_name");
// ... heavy computation here ...
ppmt_stopCPU(profiler, "tag_name");
```

The test instruments two sections:

- `"sleep"`: a 10-second idle period (to capture baseline power);
- `"kernel"`: an OpenMP-parallelized vector kernel executed in a loop.

Importantly, **`PPMT` itself remains single-threaded**—the background sampling thread is managed by the underlying `PMT` library, and instrumentation calls occur on the main thread only. This aligns with current `PPMT` v0.1.0 limitations (no per-thread or OpenMP-integrated profiling yet).

4. **GPU placeholder**
   A GPU region is also started/stopped (e.g., `"phantom"` tag with devID=0) for completeness, though no GPU is active. This verifies `PPMT`'s graceful handling of unused GPU indices.
5. **Output**
   After instrumentation, `ppmt.show()` produces:
   - a per-rank energy report (in this case, rank 0 only);
   - aggregated metrics: energy (J), average power (W), execution time (s).

- **Generated Output**
  At the end of the run, `PPMT` writes the final report to `PPMT_report.txt`. Its structure is consistent across test variants and reflects hierarchical aggregation, even in serial mode, providing a uniform interface for MPI and non-MPI runs.
  The serial runs in `app.cpp, app.c` appear as follows:

```
============ PPMT MULTI-METRIC ENERGY REPORT ============
MPI processes : 1

> TASK: 0

>>> DEVICE: CPU
Tag: [sleep] (1 iterations)
Metric      Avg     StdDev      Min       Max     Total
-----------------------------------------------------------
Watts     8.691      0.000     8.691     8.691     8.691
Joules   86.910      0.000    86.910    86.910    86.910
Seconds  10.000      0.000    10.000    10.000    10.000

Tag: [kernel] (34 iterations)
Metric      Avg     StdDev      Min       Max     Total
-----------------------------------------------------------
Watts    75.897      1.529    71.241    78.416  2580.494
Joules   22.626      0.426    22.130    24.209   769.301
Seconds   0.298      0.010     0.282     0.329    10.142

===========================================================
```

Interpretation:

- **MPI processes**: confirms serial execution (1 process);
- **TASK**: The rank number (0 in serial);
- **DEVICE**: Section per device type (here only CPU);
- **Tag**: Each profiled region is grouped by name;
- **Iterations**: Number of times `start/stop` was called for that tag (e.g., `"kernel"` ran 34 times);
- **Metrics**:
  - **Avg / StdDev / Min / Max / Total**: per-statistic over the iterations;
  - **Watts**: average power draw during the region (W);
  - **Joules**: total energy consumed (W × s);
  - **Seconds**: total wall-clock time spent in the region.

  For `"sleep"`, power is stable (8.7 W) with zero variance, as expected for a passive wait. For `"kernel"`, higher average power (75.9 W) and dynamic variation (±1.5 W) reflect active computation, while the total energy (∼769 J) quantifies the energy cost of the 10-second vector kernel loop.

4.2. Serial GPU Test (`examples/app_omp_gpu.cpp`)

This test validates `PPMT`'s ability to profile GPU-only workloads on a single rank, using shared-memory OpenMP GPU-offloading. It does not involve MPI, so the `SerialImpl` backend is used automatically. The test demonstrates how to initialize `PPMT` for GPU profiling, map GPU regions, and interpret per-device energy metrics. `Small-GPU` platform is exploited.

- **Code Structure and Instrumentation**

1. **Header inclusion**
   `#include <ppmt/ppmt.hpp>`
2. **Initialization**
   The constructor is invoked with:

```
const int NumDev = omp_get_num_devices();
std::vector<std::size_t> gpuIDs(NumDev);
std::iota(gpuIDs.begin(), gpuIDs.end(), 0);
PPMT ppmt(true, true, gpuIDs); // CPUs = true
                               // GPUs = true
                               // GPU IDs vector
```

This enables both CPU and GPU profiling and registers all detected GPUs in the compute node (e.g., IDs 0–N) for monitoring. In this case only one GPU is recognised because a single-GPU platform is actually used (i.e. `NumDev = 1`); however, `PPMT` has been designed to exploit any number of GPU per rank.

3. **Region instrumentation**
   Three GPU-related tags are used:

- `"HostToDevice"` wraps the host to device data transfer `#pragma omp target enter data`;
- `"kernel_gpu"` wraps the compute kernel lunched by `#pragma omp target`;
- `"DeviceToHost"` wraps the device to host data transfer `#pragma omp target exit data`.

Each is paired with `ppmt.startGPU("tag_name", devID)` and `ppmt.stopGPU("tag_name", devID)`, where `devID` iterates over `ppmt.gpuIDs`.

4. **CPU baseline**
A short `"sleep"` region (2 seconds) is profiled with `startCPU/stopCPU` to capture idle CPU power.
5. **Output**
`ppmt.show()` prints a structured report grouped by device (CPU and GPU) and tag, which appears as follows:

```
========== PPMT MULTI-METRIC ENERGY REPORT ============
 MPI processes : 1

> TASK: 0

>>> DEVICE: CPU
Tag: [sleep] (1 iterations)
Metric      Avg    StdDev     Min       Max     Total
-----------------------------------------------------------
Watts     6.723     0.000     6.723     6.723     6.723
Joules   13.452     0.000    13.452    13.452    13.452
Seconds   2.001     0.000     2.001     2.001     2.001

>>> DEVICE: GPU [ID: 0]
Tag: [HostToDevice] (1 iterations)
Metric    Avg      StdDev     Min       Max     Total
-----------------------------------------------------------
Watts     3.657     0.000     3.657     3.657     3.657
Joules    0.845     0.000     0.845     0.845     0.845
Seconds   0.231     0.000     0.231     0.231     0.231

Tag: [kernel_gpu] (80 iterations)
Metric    Avg      StdDev     Min       Max     Total
-----------------------------------------------------------
Watts    24.329     2.562     8.812    25.259  1946.315
Joules    3.059     0.302     1.401     3.168   244.733
Seconds   0.126     0.004     0.125     0.159    10.080

Tag: [DeviceToHost] (1 iterations)
Metric    Avg      StdDev     Min       Max     Total
-----------------------------------------------------------
Watts    25.124     0.000    25.124    25.124    25.124
Joules    3.170     0.000     3.170     3.170     3.170
Seconds   0.126     0.000     0.126     0.126     0.126

===========================================================
```

Interpretation:

- **CPU baseline** (`"sleep"`): very low and stable power (∼6.7 W), matching a passive wait period;
- **GPU memory transfers** (`"HostToDevice"`, `"DeviceToHost"`): higher power (∼25 W), but very short duration (∼0.13–0.23 s), yielding modest total energy (0.85–3.17 J);
- **GPU compute** (`"kernel_gpu"`):
  - High average power (∼24.3 W), but fluctuates (±2.6 W) due to short compute time;
  - Many iterations (80 loops), totaling ∼10 s, resulting in the largest energy contribution (∼245 J).

  This test validates PPMT's ability to:
  - Distinguish energy across CPU, GPU, and data movement phases;
  - Track per-device and per-tag metrics independently;
  - Provide fine-grained insight into energy distribution in GPU-accelerated applications—even in a serial (single-process) context.

4.3. MPI Test multiple-nodes (examples/app_mpi.cpp)

This test validates PPMT's ability to profile CPU energy consumption in a distributed-memory environment with a hierarchical topology: 2 nodes, each with 4 sockets, and 16 MPI ranks mapped such that 2 ranks per socket (–map-by socket). This configuration ensures that PPMT's internal MPI topology detection (`MPI_Comm_split_type(MPI_COMM_TYPE_SHARED, ...)`) correctly identifies socket-level affinity, enabling precise hierarchical energy aggregation. `Large-CPU` platform is used.

Test Setup and Topology

- **Hardware**: 2 nodes × 4 sockets/node × 12 cores/socket × 2 threads/core = 96 logical CPUs per node.
- **MPI Mapping**: `mpirun -n 16 --map-by socket` ensures
  - 8 MPI ranks per node (16 total),
  - 2 MPI ranks per socket (4 sockets × 2 ranks = 8 ranks per node).
- **PPMT Initialization**: `PPMT ppmt(true);` — CPU profiling enabled, GPU disabled. No OpenMP used.
- **Instrumented Regions**:
  - "sleep": 10 iterations of a 1-second `std::this_thread::sleep_for()` — simulating idle or I/O-bound phases.

Report Interpretation

1. **Per-Rank Report** [`PPMT_report_all_processes.txt`]:

   The report lists metrics for all 16 MPI ranks, grouped by their global rank, node ID, and socket ID. For example for rank 0:

```
> TASK-> Global Rank        : 0
Local  Rank        : 0
Socket Rank        : 0
Node ID            : 0
Socket IntraNode ID: 0
Socket InterNode ID: 0
>>> DEVICE: CPU
Tag: [sleep] (10 iterations)
Watts: 20.412 +/- 1.092, Joules: 20.431 +/- 1.088, Seconds:
     ↪ 1.001 +/- 0.001
Tag: [pausa] (1 iteration)
Watts: 20.095, Joules: 10.055, Seconds: 0.501

.... other Ranks
```

   Key Insight: Each rank reports independent CPU power measurements. Variance in Watts (e.g., 18.465–27.031 W) reflects:
   - Socket-level thermal and frequency scaling;
   - Resource contention (e.g., memory bandwidth, cache pressure) between the two co-located ranks per socket.

   This confirms that PPMT captures per-rank, per-socket behavior, essential for identifying energy hotspots caused by core density or memory access patterns.
2. **Socket-Level Report** [`PPMT_report_sockets.txt`]:

   PPMT aggregates data at the socket level using `MAX` for power, energy and time (as explained in Section 3 item 3):

```
> SOCKET STATISTICS
Socket IntraNode ID: 0
Socket InterNode ID: 0
Node ID: 0
MPI processes per socket: 2
>>> DEVICE: CPU
Tag: [sleep]
Watts: 22.475 +/- 0.002, Joules: 204.237, Seconds: 10.005
```

```
Tag: [pausa]
Watts: 20.077 +/- 0.018, Joules: 10.052, Seconds: 0.501

... other Sockets
```

- **Watts (Avg):** The average of the two ranks' power values, useful for comparing socket load;
- **Watts (StdDev):** Low (e.g., 0.002 W) indicates consistent behavior between the two ranks on the same socket;
- **Joules and Seconds:** Max across both ranks (total energy and time consumed by the socket during the region in the assumption of balanced-workload across the ranks).

3. **Node-Level Report** [PPMT_report_nodes.txt]:

   Aggregation across all 4 sockets per node:

```
> NODE STATISTICS
Node ID: 0
MPI processes per node: 8
Active Sockets within the node: 4
>>> DEVICE: CPU
Tag: [sleep]
Watts: 98.244 +/- 1.666, Joules: 890.573, Seconds: 40.058

... other Nodes
```

4. **World-Level Report** [PPMT_report_world.txt]:

   Global aggregation across both nodes:

```
> WORLD STATISTICS
MPI processes in WORLD     : 16
Active Nodes within WORLD  : 2
Active Sockets within WORLD: 8
>>> DEVICE: CPU
Tag: [sleep]
Watts: 210.971 +/- 7.241, Joules: 1846.392, Seconds: 80.111
```

   - **Watts (Avg):** Mean of node-level averages.
   - **Watts (StdDev):** 7.241 W reveals inter-node imbalance (e.g., one node running hotter or with higher baseline power).
   - **Joules (Sum):** 1846 J total energy consumed by the entire job across 2 nodes.

   This is the final metric for total job energy cost, critical for energy-footprint analysis.

Summary: PPMT's MPI Topology-Aware Design
This test demonstrates that PPMT:

- Automatically detects MPI topology (node, socket) via MPI_Comm_split_type and MPI_Scan (ppmt/include/internal/pinterface/mpi_impl.hpp);
- Correctly maps ranks to sockets and nodes using getSocketIdIntraNode() and getSocketIdInterNode() (ppmt/include/internal/pinterface/mpi_impl.hpp);
- Applies appropriate reductions: SUM for energy/time, MAX for peak power, AVG for reporting (ppmt/include/ppmt/ppmt.inl);
- Generates hierarchical reports (all_processes, sockets, nodes, world) without user intervention;
- Scales seamlessly from single-node to multi-node, multi-socket HPC systems.

The results confirm that PPMT is not merely a distributed version of PMT, but it is a **topology-aware energy profiler** that enables fine-grained, accurate, and actionable energy profiling in real-world HPC deployments.

4.4. MPI+GPU Test single node (examples/app_mpi_omp_gpu.cpp)

This test validates PPMT's ability to profile distributed GPU workloads across multiple MPI ranks and multiple GPUs. It exercises the full PPMT MPI infrastructure—including communicator splitting, socket/node/world hierarchies—and GPU device assignment logic. Dual-GPU platform is exploited.

- **Hardware Context**
  - **CPU**: Intel Xeon Silver 4108 (1 socket, 8 cores, 16 threads), single NUMA node.
  - **GPU**: Two NVIDIA GeForce RTX 2080 GPUs (gpuIDs 0 and 1), each with 8 GiB memory.
  - **MPI Configuration**: 2 MPI ranks on the same node (1 socket), sharing both GPUs.
- **Code Structure and Instrumentation**
  1. **MPI Initialization**
     The test explicitly calls MPI_Init, creates a node-local communicator using MPI_Comm_split_type(MPI_COMM_WORLD, ↪ MPI_COMM_TYPE_SHARED, ...), and determines local rank;
  2. **GPU Assignment**

```
// = 2 (RTX 2080s)
const int NumDev = omp_get_num_devices();
std::vector<std::size_t> gpuIDs{local_rank %
    ↪ NumDev};
```

     - MPI-Rank 0 → uses GPU 0
     - MPI-Rank 1 → uses GPU 1

     This ensures balanced GPU usage per rank.
  3. **PPMT Initialization**

```
PPMT ppmt(true, true, gpuIDs);
// CPUs and GPUs enabled, per-rank GPU list
```

     The library internally builds socketComm, nodeComm, and worldComm
  4. **Instrumented Regions**
     Each rank runs three GPU-specific sections for its assigned GPU only:
     - "HostToDevice" wraps data copy via #pragma omp target enter data map(to: ...)
     - "kernel_gpu" wraps compute loop with #pragma omp target
     - "DeviceToHost" wraps data copy via #pragma omp target exit data map(from: ...)

     Each region is wrapped with startGPU("tag_name", ↪ devID) and stopGPU("tag_name", devID).
  5. **CPU Baseline** A short "sleep" region (2 seconds) captures idle power on CPU.
- **Example Output Interpretation** The test produces several hierarchical reports; we summarize the most comprehensive one:
  1. **Per-Rank Report** [PPMT_report_all_processes.txt]:

     shows per-rank raw data.

```
> TASK: 0
>>> DEVICE: GPU [ID: 0]
Tag: [HostToDevice] (1 iter): 58.062 W, 51.304 J
Tag: [kernel_gpu] (133 iter): 78.568 W +/- 1.407,
    ↪ 772.234 J
> TASK: 1
>>> DEVICE: GPU [ID: 1]
Tag: [HostToDevice] (1 iter): 54.526 W, 47.087 J
```

```
Tag: [kernel_gpu] (164 iter): 110.573 W +/- 5.628,
    ↪ 1078.116 J
```

Rank 1 shows higher compute power (110.6 W vs 78.6 W) and more iterations (164 vs 133), reflecting actual workload imbalance. The larger GPU joules for rank 1 (1078.1 vs 772.2 J) confirm this.

2. **Node-Level Aggregation** [PPMT_report_nodes.txt]:

   aggregates across both ranks on the node using the following policy:
   - **GPU metrics**: SUM for joules/time, MAX for watts (peak power);
   - **CPU metrics**: SUM for joules/time, MAX for watts (peak power).

```
>>> DEVICE: GPU
Tag: [kernel_gpu]
  Avg: 194.005 W
  Total Joules: 1850.350 J (sum of 772.234 + 1078.116)
  Total Seconds: 19.588 s
```

This shows that the combined GPU energy consumption for the compute kernel is ∼1.85 kJ, dominated by rank 1.

5. BUILDING AND DEPLOYMENT FOR PPMT

This section describes how to build, install and use the PPMT library.

5.1. PMT Dependency Management

PPMT uses CMake's FetchContent mechanism to automatically acquire and build the required PMT library during its own build process (see the implementation within CMakeLists.txt).

- **Strategy**: PPMT treats PMT as an internal dependency and avoids requiring a separate system installation. It either fetches PMT from the Git repository at build time or uses a pre-downloaded local source, supporting offline builds on compute nodes.
- **Offline Build Support**: If the user defines FETCHCONTENT_SOURCE_DIR_PMT (e.g., after cloning on the login node), PPMT skips the network fetch and builds from the local source. A status message explicitly indicates whether the local or remote source is used.
- **Manual Cloning Instructions**: On a login node with internet access, clone PMT as follows:

```
export PMT_TAG="1.3.1"                          && \
git clone --recursive https://git.astron.nl/ \
RD/pmt.git pmt-${PMT_TAG}                       && \
cd pmt-${PMT_TAG}                               && \
git switch --detach ${PMT_TAG}                  && \
[git log -1 --oneline # optional check]
```

  Then set

```
FETCHCONTENT_SOURCE_DIR_PMT=/path/to/ \
pmt-${PMT_TAG}
```

  during the CMake configuration on the compute node.
- **Implementation**: The project uses include(FetchContent) and pins PMT to release 1.3.1, with submodules cloned recursively.
- **Compiler Configuration**: PPMT forwards its own C++ compiler (CMAKE_CXX_COMPILER) to the PMT build via PMT_CXX_COMPILER to ensure consistent compilation across environments.
- **Build Type & Feature Flags**: To guarantee reproducibility and compatibility, PPMT forces PMT to be built in Release mode (ignoring user-provided CMAKE_BUILD_TYPE) and explicitly enables/disables optional features (e.g., PMT_BUILD_RAPL, PMT_BUILD_NVML, Python bindings) via CMake cache variables.

This approach ensures reproducibility, eliminates external installation dependencies, and keeps the PMT source pinned to a tested version, while remaining robust in HPC environments with limited compute-node connectivity.

5.2. PPMT Core Build Rules

- **Explicit Compilers Required**: The configuration requires the user to explicitly pass the C++ compiler via command-line arguments (-DCMAKE_CXX_COMPILER);
- **C++20 Standard**: The library strictly target-compiles with C++20 features enabled;
- **Compiler-Isolated Deployment**: Binaries and dynamically generated compilation reports are sandboxed into folders named after the compiler's executable short name (e.g., g++ or mpic++).

5.3. Step-by-step build instructions

To generate the combined deployment structure, the user needs to run the CMake configuration, compilation, and installation steps twice—once for the serial (e.g. using g++) variant and once for the MPI parallel (e.g. using mpic++) variant.

Choose a central installation folder (for example, /workspace/ppmt or any target destination). Replace /path/to/ppmt/ in the commands below with the chosen directory.

- **Variant A: Serial build with GNU g++**
  This builds the standard serial library variant (libppmt.so) using the GNU C++ compiler.

```
cd ppmt_source_directory && \
cmake -S . -B build_gxx     \
-DCMAKE_CXX_COMPILER=g++    \
-DCMAKE_INSTALL_PREFIX=path/to/ppmt/
[-DFETCHCONTENT_SOURCE_DIR_PMT=/path/to/ \
pmt-${PMT_TAG}]
```

Code 1. Configure the build directory and set the C++ compiler. CPU-only energy profiler. The ppmt_source_directory is where CMakeLists.txt resides. If the user defines FETCHCONTENT_SOURCE_DIR_PMT (e.g., after cloning on the login node), PPMT skips the network fetch of PMT and builds from the local source, as explained in 5.1

```
cmake --build build_gxx [--target doc] && \
cmake --install build_gxx
```

Code 2. Compile the binaries and install to the target layout. The command --target doc optionally creates the Doxygen documentation

- **Variant B: Serial build with NVIDIA nvc++**
  This builds the standard serial library variant with NVIDIA support (libppmt_nvidia.so) using the NVIDIA nvc++ compiler.

```
cd ppmt_source_directory &&          \
cmake -S . -B build_nvcxx            \
-DCMAKE_CXX_COMPILER=nvc++           \
-DCMAKE_INSTALL_PREFIX=path/to/ppmt/ \
-DPPMT_WITH_NVIDIA=ON
[-DFETCHCONTENT_SOURCE_DIR_PMT=/path/to/ \
pmt-${PMT_TAG}]
```

Code 3. Configure the build directory and set the C++ compiler. CPU and GPU energy profiler. The `ppmt_source_directory` is where `CMakeLists.txt` resides. If the user defines `FETCHCONTENT_SOURCE_DIR_PMT` (e.g., after cloning on the login node), `PPMT` skips the network fetch of `PMT` and builds from the local source, as explained in 5.1

```
cmake --build build_nvcxx [--target doc] && \
cmake --install build_nvcxx
```

Code 4. Compile the binaries and install to the target layout. The command `--target doc` optionally creates the Doxygen documentation

- **Variant C: Parallel build with mpic++ (GNU/MPI)**
  This builds the parallel library variant (`libppmt_mpi.so`) using the environment's MPI wrapper compiler.

```
cd ppmt_source_directory &&                \
cmake -S . -B build_mpi_gnu                \
-DCMAKE_CXX_COMPILER=path/to/openmpi/mpic++ \
-DCMAKE_INSTALL_PREFIX=path/to/ppmt/       \
-DPPMT_WITH_MPI=ON
[-DFETCHCONTENT_SOURCE_DIR_PMT=/path/to/ \
pmt-${PMT_TAG}]
```

Code 5. Configure the build directory and set the C++ compiler. CPU-only energy profiler. The `ppmt_source_directory` is where `CMakeLists.txt` resides. If the user defines `FETCHCONTENT_SOURCE_DIR_PMT` (e.g., after cloning on the login node), `PPMT` skips the network fetch of `PMT` and builds from the local source, as explained in 5.1

```
cmake --build build_mpi_gnu \
[--target doc] &&          \
cmake --install build_mpi_gnu
```

Code 6. Compile the binaries and install to the target layout. The command `--target doc` optionally creates the Doxygen documentation

- **Variant D: Parallel build with mpic++ (NVIDIA)**
  This builds the parallel library variant (`libppmt_mpi_nvidia.so`) using the environment's MPI wrapper compiler.

```
cd ppmt_source_directory &&                \
cmake -S . -B build_mpi_nvidia             \
-DCMAKE_CXX_COMPILER=path/to/nvidia/mpic++ \
-DCMAKE_INSTALL_PREFIX=path/to/ppmt/       \
-DPPMT_WITH_MPI=ON                         \
-DPPMT_WITH_NVIDIA=ON
[-DFETCHCONTENT_SOURCE_DIR_PMT=/path/to/ \
pmt-${PMT_TAG}]
```

Code 7. Configure the build directory and set the C++ compiler. CPU and GPU energy profiler. The `ppmt_source_directory` is where `CMakeLists.txt` resides. If the user defines `FETCHCONTENT_SOURCE_DIR_PMT` (e.g., after cloning on the login node), `PPMT` skips the network fetch of `PMT` and builds from the local source, as explained in 5.1

```
cmake --build build_mpi_nvidia
[--target doc] && \
cmake --install build_mpi_nvidia
```

Code 8. Compile the binaries and install to the target layout. The command `--target doc` optionally creates the Doxygen documentation

5.4. Resulting Installation Structure

After successfully running both installation passes, the target directory (set using `CMAKE_INSTALL_PREFIX`) is populated with the unified layout below:

```
ppmt/
├── include/
│   ├── internal/
│   │   ├── pinterface/
│   │   │   ├── mpi_impl.hpp
│   │   │   ├── mpi_utils.hpp
│   │   │   ├── parallel_c_api.h
│   │   │   ├── parallel_interface.hpp
│   │   │   └── serial_impl.hpp
│   │   └── utility/
│   │       ├── memory_manager.hpp
│   │       └── underlying.hpp
│   ├── pmt/
│   │   ├── common/
│   │   │   └── PMT.h
│   │   ├── Dummy.h
│   │   ├── NVIDIA.h
│   │   ├── NVML.h
│   │   └── Rapl.h
│   ├── pmt.h
│   └── ppmt/
│       ├── ppmt_c.h
│       ├── ppmt.hpp
│       └── ppmt.inl
└── lib/
    ├── g++/
    │   ├── libpmt.so
    │   ├── libppmt.so
    │   └── doc/
    │       ├── libppmt_compiler_report.txt
    │       └── doxygen/
    │           └── libppmt/
    │               ├── Doxyfile
    │               ├── html/
    │               └── latex/
    ├── mpic++/
    │   ├── libpmt.so
    │   ├── libppmt_mpi.so
    │   ├── libppmt_mpi_nvidia.so
    │   └── doc/
    │       ├── libppmt_mpi_compiler_report.txt
    │       ├── libppmt_mpi_nvidia_compiler_report.txt
    │       └── doxygen/
    │           ├── libppmt_mpi/
    │           │   ├── Doxyfile
    │           │   ├── html/
    │           │   └── latex/
    │           └── libppmt_mpi_nvidia/
    │               ├── Doxyfile
    │               ├── html/
    │               └── latex/
    └── nvc++/
        ├── libpmt.so
        ├── libppmt_nvidia.so
        └── doc/
            └── libppmt_nvidia_compiler_report.txt
```

```
└── doxygen/
    └── libppmt_nvidia/
        ├── Doxyfile
        ├── html/
        └── latex/
```

5.4.1. Directory Layout Breakdown

- `include/`
  Contains the public API headers and internal template implementations.
  - `ppmt/`: main developer facing public headers
  - `internal/`: subsystem utilities, memory managers, and concrete implementation headers for serial/MPI targets
- `lib/g++`
  Houses the assets generated during the serial `g++` compilation sequence
  - `libppmt.so`: the standard shared object library
  - `doc/libppmt_compiler_report.txt`: a dynamically evaluated file tracking the system's exact compiler versions, architecture optimization flags and build modifiers used during generation.

```
====================================
COMPILATION FLAGS & SWITCHES REPORT
====================================
Target Name:         ppmt
Compiler ID:         GNU
Compiler Version:    14.2.0
Compiler Path:       /usr/bin/g++
Build Mode:          Release
MPI Enabled:         No
NVIDIA Enabled:      No
------------------------------------
Global CXX Flags (CMake Default):


Target Compile Options:
  -Wall
  -Wextra
  -march=native
  -mtune=native
  -std=c++20
  -O3

Target Compile Definitions:

====================================
```

Code 9. lib/g++/doc/ libppmt_compiler_report.txt

- `lib/nvc++` Houses the assets generated during the parallel `nvc++` compilation sequence.
  - `libppmt_nvidia.so`: the shared object library compiled with NVIDIA support.
  - `doc/libppmt_nvidia_compiler_report.txt`:

```
====================================
COMPILATION FLAGS & SWITCHES REPORT
====================================
Target Name:         ppmt_nvidia
Compiler ID:         NVHPC
Compiler Version:    26.3.0
Compiler Path:       /opt/nvidia/hpc_sdk/ \
                     Linux_x86_64/26.3/   \
                     compilers/bin/nvc++
Build Mode:          Release
MPI Enabled:         No
NVIDIA Enabled:      Yes
------------------------------------
Global CXX Flags (CMake Default):


Target Compile Options:
  -Wall
  -Wextra
  -march=native
  -mtune=native
  -std=c++20
  -O3

Target Compile Definitions:
  PPMT_WITH_NVIDIA
====================================
```

Code 10. lib/nvc++/doc/ libppmt_nvidia_compiler_report.txt

- `lib/mpic++/`
  Houses the assets generated during the parallel `mpic++` compilation sequence.
  - `libppmt_mpi.so`: the shared object library compiled with internal MPI dependencies linked.
  - `doc/libppmt_mpi_compiler_report.txt`:

```
====================================
COMPILATION FLAGS & SWITCHES REPORT
====================================
Target Name:         ppmt_mpi
Compiler ID:         GNU
Compiler Version:    14.2.0
Compiler Path:       /usr/local/openmpi/ \
                     5.0.9/bin/mpic++
Build Mode:          Release
MPI Enabled:         Yes
NVIDIA Enabled:      No
------------------------------------
Global CXX Flags (CMake Default):


Target Compile Options:
  -Wall
  -Wextra
  -march=native
  -mtune=native
  -std=c++20
  -O3

Target Compile Definitions:
  PPMT_WITH_MPI
====================================
```

Code 11. lib/mpic++/doc/ libppmt_mpi_compiler_report.txt

  - `libppmt_mpi_nvidia.so`: the shared object library compiled with NVIDIA support and internal MPI dependencies linked.
  - `doc/libppmt_mpi_nvidia_compiler_report.txt`:

```
====================================
COMPILATION FLAGS & SWITCHES REPORT
====================================
Target Name:         ppmt_mpi_nvidia
Compiler ID:         NVHPC
Compiler Version:    26.3.0
Compiler Path:       /opt/nvidia/hpc_sdk/ \
                     Linux_x86_64/26.3/   \
                     comm_libs/mpi/bin/mpic++
Build Mode:          Release
MPI Enabled:         Yes
NVIDIA Enabled:      Yes
------------------------------------
Global CXX Flags (CMake Default):


Target Compile Options:
  -Wall
  -Wextra
  -march=native
  -mtune=native
  -std=c++20
  -O3

Target Compile Definitions:
  PPMT_WITH_MPI
  PPMT_WITH_NVIDIA
====================================
```

Code 12. lib/mpic++/doc/libppmt_mpi_nvidia_compiler_report.txt

5.4.2. *Doxygen documentation [optional]*

Based on the final installation structure, the Doxygen documentation is organized as follows:

- Each compiler variant (`g++`, `mpic++`, `nvc++`) has its own subdirectory under `lib/`.
- Within each compiler directory, documentation is stored under `doc/doxygen/<target_name>/`, where `<target_name>` encodes the configuration (e.g., `libppmt_mpi`, `libppmt_mpi_nvidia`, `libppmt_nvidia`).
- Each `<target_name>` subdirectory contains:
  - `Doxyfile`: the configuration used to generate that variant's documentation.
  - `html/`: HTML documentation
  - `latex/`: LaTeX sources for PDF generation.

This structure ensures that:

- Documentation is versioned by build configuration (MPI, CUDA/NVIDIA, etc.).
- Users can inspect the exact `Doxyfile` used for reproducibility.
- Generated HTML and other assets remain accessible and isolated per target.

This organization aligns with the current directory layout and supports multiple parallel builds (e.g., `build_mpi`, `build_nvidia`, `build_mpi_nvidia`) without conflicts.

`Cmake` handles `Doxygen` (through `CMakeLists.txt`) using the command `find_package(Doxygen QUIET)`, which allows the project to configure cleanly even if `Doxygen` is missing, leaving it up to the script to check `if(DOXYGEN_FOUND)` before building documentation.

5.5. Usage by Users

This paragraph contains example programs demonstrating `PPMT` usage patterns (directory `examples/` in the repository).

5.5.1. *Serial C++*

```cpp
#include <ppmt/ppmt.hpp>
int main()
{
    // use PPMT
}
```

Code 13. app.cpp

Compiled with (shell commands):

```
export PPMT_PATH=path/to/ppmt        && \
export PPMT_INC=${PPMT_PATH}/include && \
export PPMT_LIB=${PPMT_PATH}/lib/g++ && \
g++ -std=c++20 app.cpp              \
    -O3 -march=native -mtune=native \
    -I${PPMT_INC}                   \
    -L${PPMT_LIB}                   \
    -lpmt -lppmt -o app_cpp
```

Code 14. Set the (absolute) path to the ppmt shared library (i.e. library prefix build). Compilation command for user application. The compiler must validate and optimize the source code according to the official ISO C++20 standard specifications. The user should compile against both PMT and PPMT libraries

And then run the user application with (shell commands):

```
export LD_LIBRARY_PATH=${PPMT_LIB}:${LD_LIBRARY_PATH} && \
./app_cpp
```

Code 15. Add PPMT directory to the dynamic linker on Linux and run the application

5.5.2. *Serial C*

```c
#include <ppmt/ppmt_c.h>
int main()
{
    // use PPMT
}
```

Code 16. app.c

Compiled with (shell commands):

```
export PPMT_PATH=path/to/ppmt        && \
export PPMT_INC=${PPMT_PATH}/include && \
export PPMT_LIB=${PPMT_PATH}/lib/g++ && \
gcc app.c                           \
    -O3 -march=native -mtune=native \
    -I${PPMT_INC}                   \
    -L${PPMT_LIB}                   \
    -lstdc++ -lpmt -lppmt -lm -o app_c
```

Code 17. Set the (absolute) path to the ppmt shared library (i.e. library prefix build). Compilation command for user application. The C compiler must link against the GNU C++ Standard Library, which provides the runtime support needed by C++ programs (-lstdc++). The user should compile against both PMT and PPMT libraries

And then run the user application with (shell commands):

```
export LD_LIBRARY_PATH=${PPMT_LIB}:${LD_LIBRARY_PATH} && \
./app_c
```

Code 18. Add PPMT directory to the dynamic linker on Linux and run the application

5.5.3. MPI C++

```
#include <ppmt/ppmt.hpp>
int main()
{
    MPI_Init(nullptr, nullptr);
    ...
    // use PPMT
    ...
    MPI_Finalize();
}
```

Code 19. app_mpi.cpp

Compiled with (shell commands):

```
export PPMT_PATH=path/to/ppmt           && \
export PPMT_INC=${PPMT_PATH}/include    && \
export PPMT_LIB=${PPMT_PATH}/lib/mpic++ && \
mpic++ -std=c++20 app_mpi.cpp       \
    -O3 -march=native -mtune=native \
    -DPPMT_WITH_MPI                 \
    -I${PPMT_INC}                   \
    -L${PPMT_LIB}                   \
    -lpmt -lppmt_mpi -o app_mpi
```

Code 20. Set the (absolute) path to the ppmt shared library (i.e. library prefix build). Compilation command for user application. The compiler must validate and optimize the source code according to the official ISO C++20 standard specifications. The user should compile with PPMT_WITH_MPI macro and compile against both PMT and PPMT libraries

And then run the user application with (shell commands):

```
export LD_LIBRARY_PATH=${PPMT_LIB}:${LD_LIBRARY_PATH} && \
mpirun -n <ntasks> ./app_mpi
```

Code 21. Add PPMT directory to the dynamic linker on Linux and run the application

5.5.4. NVIDIA GPU C++

```
#include <ppmt/ppmt.hpp>
void kernel_gpu(args...)
{
    #pragma omp target ...
}
int main()
{
    // use PPMT
    kernel_gpu(...);
}
```

Code 22. app_omp_gpu.cpp. NVIDIA-GPU is exploited through OpenMP

Compiled with (shell commands):

```
export PPMT_PATH=path/to/ppmt                       && \
export PPMT_INC=${PPMT_PATH}/include                && \
export PPMT_LIB=${PPMT_PATH}/lib/nvc++              && \
export GPU_ARCH=$(nvidia-smi --query-gpu=compute_cap    \
--format=csv,noheader | head -n 1 | tr -d '.')      && \
nvc++ app_omp_gpu.cpp                           \
    -O3 -std=c++20 -march=native -mtune=native \
    -mp=gpu -gpu=cc${GPU_ARCH} -Minfo=accel    \
    -DPPMT_WITH_NVIDIA                         \
    -I${PPMT_INC}                              \
    -L${PPMT_LIB}                              \
    -lpmt -lppmt_nvidia -o app_omp_gpu
```

Code 23. Set the (absolute) path to the ppmt shared library (i.e. library prefix build). Compilation command for user application. The compiler must validate and optimize the source code according to the official ISO C++20 standard specifications, and to support OpenMP target offload targeting NVIDIA GPUs. The user should compile with PPMT_WITH_NVIDIA macro and compile against both PMT and PPMT libraries

And then run the user application with (shell commands):

```
export LD_LIBRARY_PATH=${PPMT_LIB}:${LD_LIBRARY_PATH} && \
./app_omp_gpu
```

Code 24. Add PPMT directory to the dynamic linker on Linux and run the application

5.5.5. MPI-NVIDIA GPU C++

```
#include <ppmt/ppmt.hpp>
void kernel_gpu(args...)
{
    #pragma omp target ...
}
int main()
{
    MPI_Init(nullptr, nullptr);
    // use PPMT
    kernel_gpu(...);
    MPI_Finalize();
}
```

Code 25. app_mpi_omp_gpu.cpp. NVIDIA-GPU is exploited through OpenMP

Compiled with (shell commands):

```
export PPMT_PATH=path/to/ppmt                       && \
export PPMT_INC=${PPMT_PATH}/include                && \
export PPMT_LIB=${PPMT_PATH}/lib/mpic++             && \
export GPU_ARCH=$(nvidia-smi --query-gpu=compute_cap    \
--format=csv,noheader | head -n 1 | tr -d '.')      && \
mpic++ app_mpi_omp_gpu.cpp                      \
    -O3 -std=c++20 -march=native -mtune=native \
    -mp=gpu -gpu=cc${GPU_ARCH} -Minfo=accel    \
    -DPPMT_WITH_MPI                            \
    -DPPMT_WITH_NVIDIA                         \
    -I${PPMT_INC}                              \
    -L${PPMT_LIB}                              \
    -lpmt -lppmt_mpi_nvidia -o app_mpi_omp_gpu
```

Code 26. Set the (absolute) path to the ppmt shared library (i.e. library prefix build). Compilation command for user application. The compiler must validate and optimize the source code according to the official ISO C++20 standard specifications, and to support OpenMP target offload targeting NVIDIA GPUs. The user should compile with PPMT_WITH_MPI and PPMT_WITH_NVIDIA macros and compile against both PMT and PPMT libraries

And then run the user application with (shell commands):

```
export LD_LIBRARY_PATH=${PPMT_LIB}:${LD_LIBRARY_PATH} && \
mpirun -n <ntasks> ./app_mpi_omp_gpu
```

Code 27. Add PPMT directory to the dynamic linker on Linux and run the application

6. PPMT GITLAB@INAF CI

Modern software development makes extensive use of CI/CD. Within the framework of the PPMT, an initial version of an optimized pipeline has been developed. This pipeline enables relatively fast execution of software tests in an environment whose specifications mirror those of the production HPC cluster where operations will ultimately take place. The advantage of using an optimized, fast-failing pipeline is that it allows developers to quickly identify critical issues, thereby enabling them to focus more on software development.

6.1. Pipeline definition: stages, jobs and runners

A CI/CD pipeline is a sequence of operations that comprises several key stages in the software development lifecycle, that are automatically triggered and executed on project files whenever changes are committed to the GitLab-hosted copy of a repository. A pipeline is organized in stages. Stages usually group tasks (jobs) that are functionally related. The most commonly stages are "build", "test" and "deploy". Within each stage, the jobs related to the stage are organized so that they can be processed in parallel (default behavior) or serially (when dependencies between jobs are specified). Job execution is handled by runners (detailed in the next section). Jobs are assigned to runners using tags, which match job requirements with specific runner capabilities [1]. A CI/CD pipeline can be defined through a file called `.gitlab-ci.yml` (default name) that must be placed in the root folder of the repository.

6.2. Runners

Runners are the execution engine of the CI/CD pipeline. They are typically hosted on different servers than the one hosting the GitLab instance. They are able to run on different infrastructures (e.g. physical machines, VMs, containers) and on different hardware architectures (e.g. x86_64, ARM). Each runner is associated to a well-defined executor. The executor determines the execution environment of the job and is specified during the runner registration phase. Runners constantly poll the GitLab server for new jobs through HTTP requests [4].

6.3. Runner configuration

Here below are described the necessary steps to setup a runner.

6.3.1. Install GitLab runner

This is a one-time operation performed on a host running a Debian-based Linux distro:

```
curl -L "https://packages.gitlab.com/install/    \
repositories/runner/gitlab-runner/script.deb.sh" \
-o script.deb.sh && \
sudo bash script.deb.sh && \
sudo apt-install gitlab-runner && \
sudo systemctl enable --now gitlab-runner
```

Code 28. Install the GitLab runner package and start the service

6.3.2. Create a new project runner

On the GitLab project go to Settings → CI/CD → Runners and click on "Create project runner". At this point add one or more tags in the respective field. Tags are used to execute one or more CI/CD jobs on runners that meet specific requirements. Although runners can run untagged, defining tags is a best practice. It ensures that jobs requiring special resources (like GPUs, high RAM, or specific OS architectures) run only on capable runners. Finally, click on "Create runner" and then copy the authentication token.

6.3.3. Register the runner

Return to the host where the GitLab Runner package is installed, and run the following shell command:

```
sudo gitlab-runner register          \
--url https://www.ict.inaf.it/gitlab \
--token YOUR_GITLAB_RUNNER_AUTH_TOKEN
```

Code 29. Runner registration command

An interactive setup will prompt you to choose an executor. Depending on your choice, you may need to provide additional details. For example, selecting Docker requires specifying a default image (e.g., `gcc:13`). At the end, the runner is registered and the configuration is stored in `/etc/gitlab-runner/config.toml`. The runner configuration file includes a global section for all runners and specific sections for individual runners. For the first pipeline version, we used two runners: one with a Shell executor for building and pushing Docker images to the GitLab registry, and the other with a Docker executor for running the tests. To support parallel jobs within a pipeline stage, we configured a global concurrency of 4 (`concurrent = 4`) and a local concurrency of 2 jobs per runner (`limit = 2`). Additionally, to read the RAPL counters, we made two changes to the Docker runner configuration: we enabled the `privileged = true` option and mounted the following paths as read-only volumes:

```
volumes = [
  ...
  "/sys/class/powercap:/sys/class/powercap:ro",
  "/sys/devices/virtual/powercap:/sys/devices/ \
   virtual/powercap:ro"
]
```

Code 30. GitLab Runner configuration for mounting host paths containing RAPL counters as read-only volumes

6.4. PPMT GitLab@INAF CI pipeline

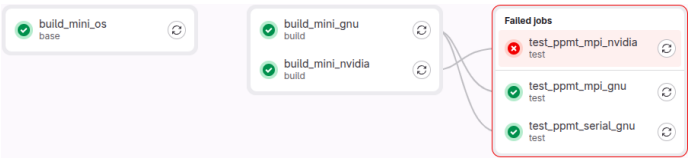

Figure 2. First version of the PPMT optimized CI pipeline: the curved lines show the dependencies between test jobs and their respective Docker build/push jobs. This enables tests to trigger as soon as the target image is ready, optimizing pipeline efficiency

The first version of the pipeline is structured into three stages: two dedicated to compilation (base and build) and one to verification (test). The first two stages handle image creation. Specifically, they generate a base image and two derived variants, and then push them to the GitLab container registry. This initial phase is executed via a runner

with a Shell executor. The three images contain the entire software stack required to compile and run various applications on computing clusters, including the PPMT. The main objective is to have specific environments configured with different compilers and optimized for diverse hardware architectures. Currently, the first integrated tests leverage the OpenMPI library, the GCC compiler, and the NVIDIA SDK, operating on a single-socket multicore hardware architecture equipped with an NVIDIA GPU. The third and final stage handles the actual execution of the software tests. For this task, the pipeline relies on a runner with a Docker executor, a choice that guarantees complete isolation and maximum reproducibility of the environment.We chose Docker Buildx to manage the image build and push processes. This extension speeds up the process with advanced caching and supports compilation for different hardware architectures.

```
docker buildx build \
-f Dockerfile.nvidia \
--platform linux/amd64 \
--cache-from type=registry,ref=$NVIDIA_IMG:cache-amd64 \
--cache-to
    ↪ type=registry,ref=$NVIDIA_IMG:cache-amd64,mode=max
    ↪ \
--build-arg BUILD_HOSTNAME=$(hostname) \
--build-arg GPU_ARCH=$(nvidia-smi --query-gpu=compute_cap
    ↪ --format=csv,noheader | head -n 1 | tr -d '.') \
--tag $NVIDIA_IMG:latest \
--push
```

Code 31. Building and pushing Docker images with Buildx: it is possible to build an image for a specific hardware architecture (–platform) and use caching functionalities (–cache-from and –cache-to). Notice the mode=max parameter that forces Buildx to save cache information for all Dockerfile layers, including intermediate ones and previous phases in multi-stage builds

7. PLANNED FEATURE DEVELOPMENT

Building upon the foundational capabilities of PPMT v0.1.0, future development will focus on enhancing accuracy, flexibility, and applicability to real-world, heterogeneous HPC workloads. Key planned enhancements include:

i) **Relaxing the balanced-workload assumption**: Current CPU energy aggregation relies on the assumption that MPI ranks on the same socket execute synchronized computational phases. Future versions will track precise `start` and `stop` timestamps per rank and label, enabling temporally-resolved, non-synchronized energy integration, allowing accurate attribution even under severe load imbalance or asynchronous execution.

ii) **MPI-rank-specific profiling labels**: The current implementation assumes uniform labeling across all ranks. We plan to introduce a *task-based labeling* system, where different MPI ranks can assign semantically distinct labels to the same profiling region (e.g., "`solve_Poisson`" on MPI-rank 0 vs. "`communicate_boundary`" on MPI-rank 1), enabling fine-grained, application-aware energy profiling that reflects actual computational roles within a distributed simulation.

iii) **GPU contention and multi-process profiling**: We will extend PPMT to detect and report energy impacts during GPU contention, such as when multiple MPI ranks share a single GPU. This includes tracking per-context energy usage and identifying performance-energy trade-offs induced by resource sharing.

iv) **Multi-threaded (OpenMP) support**: While PPMT currently profiles at the MPI-rank level, future versions will integrate per-thread energy sampling using thread-local storage and synchronized sampling, enabling profiling of hybrid MPI+OpenMP applications with full spatial resolution.

v) **Automated efficiency metrics**: Integration of PAPI-based [3] FLOP counting and memory bandwidth metrics will enable automatic computation of energy efficiency ratios (e.g., GFLOP/s/W, GB/s/W), providing direct insight into the scientific productivity per unit of energy consumed.

vi) **Python interface**: To lower the barrier to adoption and enable seamless integration with scientific Python workflows (e.g., Jupyter notebooks, PyTorch, NumPy-based simulations, and workflow managers like Snakemake or Prefect), we will develop a lightweight, stable Python wrapper (`PyPPMT`) built on top of the existing C/C++ API. This will allow users to instrument Python-controlled HPC kernels (e.g., via Cython, Numba, or external C++ libraries) with minimal code changes: simply wrapping sections with `ppmt.start_cpu("label")` and `ppmt.stop_gpu("label", gpuID)`, and retrieving hierarchical reports in native Python data structures (e.g., `pandas.DataFrame`).

These features will be developed iteratively, guided by user feedback and real-world use cases from the INAF HPC community.

8. AVAILABILITY AND COMMUNITY ENGAGEMENT

This first public release of PPMT v0.1.0 is now available in the INAF GitLab repository 8, having been rigorously tested on both CPU-only and GPU-accelerated HPC platforms as detailed in this technical report. PPMT is designed as a preliminary but *production-ready* tool, immediately applicable to real scientific workflows, from astrophysical simulations to data processing pipelines, enabling the INAF community to begin quantifying and optimizing the energy footprint of their HPC applications today. We warmly invite all INAF researchers, system administrators, and computational scientists to adopt PPMT in their workflows, report experiences, suggest enhancements, and contribute feedback. Together, we can build a culture of energy-aware HPC that supports scientific excellence while advancing the goals of the USC-C Green Computing strategy.

GITLAB REPOSITORY

The PPMT library is available at the link:

https://www.ict.inaf.it/gitlab/energy/ppmt/-/releases/v0.1.0
Software Version: v0.1.0
Publication Date: June 2026

AKNOWLEDGEMENT

The authors are sincerely grateful to Dr. Valentina Cesare for her rigorous evaluation of our work.

This work has been supported by National Centre for HPC, Big Data and Quantum Computing (ICSC) within the framework of INAF USC-C - Computing. We gratefully acknowledge the computational resources provided by the *Pleiadi Call 6* project at `Pleiadi@INAF`, which enabled this work. We gratefully acknowledge the collaboration and resources provided by the Italian Center for Astronomical Archives (IA2). We acknowledge fruitful discussions with Brigitte Gatto and Hendrik Heinl.

REFERENCES

[1] S. ZORBA and C. URBAN, *Devops at ia2 data centre*, 2021. [Online]. Available: http://hdl.handle.net/20.500.12386/30951.

[2] S. Corda, B. Veenboer, and E. Tolley, *Pmt: Power measurement toolkit*, 2022. arXiv: 2210.03724 [cs.PF]. [Online]. Available: https://arxiv.org/abs/2210.03724.

[3] H. Jagode, A. Danalis, G. Congiu, D. Barry, A. Castaldo, and J. Dongarra, "Advancements of papi for the exascale generation", *The International Journal of High Performance Computing Applications*, vol. 39, no. 2, pp. 251–268, 2025. DOI: 10.1177/10943420241303884. eprint: https://doi.org/10.1177/10943420241303884. [Online]. Available: https://doi.org/10.1177/10943420241303884.

[4] C. URBAN, D. TAVAGNACCO, M. SPONZA, C. KNAPIC, and M. LANDONI, *A small-scale infrastructure for gitlab ci/cd*, Jan. 2026. DOI: https://doi.org/10.2\0371/INAF/TechRep/358. [Online]. Available: http://hdl.handle.net/20.500.12386/46133.