5 Space weather with DLMOS, xPic and GMM

Jorge Amaya
Katholieke Universiteit Leuven, KU Leuven, Belgium
jorge.amaya@kuleuven.be

5.1 Introduction

The applications implemented by KU Leuven during the DEEP-EST project are used to study the Space Weather system connecting the Sun to the magnetosphere of our planet. There are three main applications in the system:

- **DLMOS** (Deep Learning Modelling of the Solar wind): used to forecast the solar wind conditions in front of the Earth from images of the Sun.
- **xPic** (extended Particle-in-cell): a first-principles plasma physics code used to study the plasma environment in the solar wind and the magnetosphere.
- **GMM** (Gaussian Mixture Model): a machine learning algorithm that analyses velocity distributions functions extracted from particle information generated in the xPic code.

5.2 Application structure

5.2.1 DLMOS

The DLMOS model is coded in Python. Multiple frameworks are available to code Machine Learning (ML) models. The efforts will concentrate on using PyTorch. As with another ML algorithm, DLMOS needs to be deployed in two modes: training and scoring (also called inference). The first mode takes large amounts of input data and trains the model. The second mode uses the trained model to predict a singular input sequence. While scoring a Deep Learning (DL) model can be generally performed quickly in regular CPU architectures, training the model requires important resources in disk access, memory size and computing power.

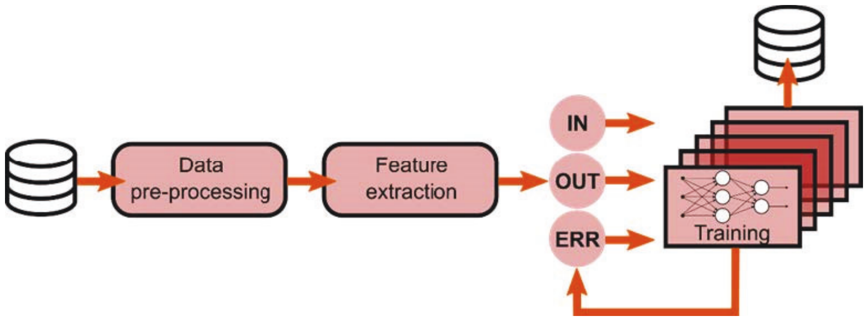

Figure 5.1: Training mode of the DLMOS model

Figure 5.1 shows the general structure of a ML algorithm. It consists of two main parts: 1) a data pre-processing pipeline, and 2) a Neural Network (NN). For high performance of the ML model, the input data of the NN has to be as clean and homogeneous as possible. So during the development of a ML project, most of the time is spent testing different methods to clean the raw input data. The pre-processing pipeline is the final result of this process.

5.2.2 xPic

This Particle-in-Cell code has been partitioned into two solvers:

- Field solver: a numerical algorithm that solves Maxwell's equations for electromagnetism in a 3D Cartesian grid. The solver uses an iterative Krylov subspace method to solve a large system of linear equations. This method requires the computation of a residual every Krylov iteration. Global communications are required to keep track of such residual, and neighbor communications are required to compute the derivatives of the source terms of the linear system. Parallelism is obtained by domain decomposition.
- Particle solver: uses Newton's equations of motion to compute the movement of billions of charged particles, which integrate the system. Collisions and other particle-particle interactions are not computed (their interaction is mediated through the electromagnetic fields). All particles are independent of each other. The particle solver is also parallelised by domain decomposition, so communications are required to move particles from one domain to the neighbouring domain when they cross boundaries. In addition to the movement of particles, the particle solver also performs the calculation of particle statistics called moment gathering. In this last step particle properties are projected on the 3D grid of the code and transmitted to the field solver.

The field solver and the particle solver are inter-dependent and require constant exchange of information. However, they show different numerical strategies that can be mapped to different hardware architectures.

5.2.3 GMM

The GMM analysis runs “on the fly”. The execution frequency depends on the particular simulation tested, but it is not performed at every xPic iteration. In a normal simulation, field and particle I/O is performed every few hundred iterations. GMM analysis will be executed at every few I/O calls (once every few thousand iterations). The Space Weather application allows to test the execution of consecutive jobs in the DEEP-EST system, the execution of concurrent jobs in different modules, and the new deferred job launch from SLURM.

5.3 Application mapping

5.3.1 DLMOS

The DLMOS application from KU Leuven is characterized by a heavy and continuous movement of data from hard disk to processor memory. Therefore, it requires good data movement management between the different levels of computer memory.

This application also uses large amounts of data to train a deep neural network. This process is based on the constant use of tensor operations (matrix and vector multiplications) that can benefit from relatively weak computing units with large number of threads and vectorisation. These operations are also relatively fast, so the memory bandwidth needs to be high. A more quantitative description of the requirements is not yet available.

These are the reasons why the DLMOS application would benefit from the large number of cores and higher memory bandwidth proposed for the DAM.

5.3.1.1 DLMOS-DPP

The data pre-processing procedure is not based on highly parallel and multi-threaded, vectorisable, tensor operations. It requires high performance per core and high memory capacity. It also requires the full I/O infrastructure to move the data from disk to processor. Data pre-processing can be implemented on accelerator cards and can also take advantage of the host processors of accelerator nodes (like the DAM nodes).

It would be possible to deploy the DLMOS-DPP sub-package to the CM where both memory and sequential performance is higher. The final decision on the correct mapping of the DLMOS-DPP (DAM or CM) will need to be done on the prototype modules.

As it is today, it is more convenient to maintain the multiple elements of the application as close as possible in hardware, using the same module for the DLMOS-DPP and the DLMOS-Training sub-packages.

DLMOS-DPP will take advantage of the different levels of data storage of the DEEP-EST system, moving data from the Internet to the SSSM, local disk, memory, and the processor. This package will generate new enhanced data that need to move in the opposite direction, up to the SSSM.

DLMOS-DPP can run continuously, processing new data, for multiple applications of multiple data-sets. It does not depend on the results of any other components of the DLMOS package. Parallelism can be achieved by processing multiple inputs at the same time. No data communications are required in this package.

The pre-processing also involves downloading of data to a local storage space, detecting anomalies, normalizing the data, and selecting and building the training data-sets for the DLMOS-Training sub-package.

5.3.1.2 DLMOS-Training

Training is based on highly parallel multi-threaded vectorisable tensor operations that can be deployed on accelerator cards. These operations also require a constant stream of data, thus taking advantage of high bandwidth memory.

For these reasons the DLMOS-Training sub-package will be mapped to the DAM (although it could be potentially translated to the GPUs in the ESB).

This python code takes the processed data from the DLMOS-DPP, stored in the SSSM, and performs the training process of the DLMOS Neural Network Models (DLMOS-NNMs). The architecture of the DLMOS-NNMs is not yet defined and will be tested during this project.

Parallelism of the DLMOS-Training will be achieved in three different ways:

- Model parallelism: Different accelerators will train different NNMs for the same training data, selecting and cross-breeding the best performing models and achieving good convergence to a satisfactory result.
- Data parallelism: A single data-set can be divided in multiple smaller data-sets that can be used to train a model in independent accelerators.
- Graph parallelism: A single ML model is divided in multiple sub-tasks that can be mapped to different accelerators, requiring continuous data exchanges.

The DLMOS-Training sub-package implements python algorithms for methods 1 and 2, but method 3 is implemented in an ML framework (e.g. TensorFlow, Keras, PyTorch). For methods 1 and 2, network latency and bandwidth are not a constraint. The amount of data transferred between nodes is very low in all methods. While method 1 does not require frequent MPI communication, method 2 can demand frequent ALL_REDUCE operations.

Method 3 is very restrictive and requires constant data movement of small amounts of data. Parallel efficiency is achieved by the framework's particular communication algorithm, which is not based on MPI. TensorFlow has shown in recent tests low parallel efficiency using multiple GPU node systems. We studied the use of other frameworks such as PyTorch and MXNet as potential replacements of TensorFlow and chose a combination of PyTorch and mpi4py.

5.3.1.3 DLMOS-Inference

This procedure is computationally similar to the DLMOS-Training sub-package but requires only one input (instead of hundreds of thousands) and produces only one output. The full procedure is extremely fast and does not require special hardware components. The sub-package is mapped to the ESB. The code takes inputs from two different sources: the SSSM, where recent input data and NNMs have been stored by the DLMOS-DPP.

The DLMOS-Inference code is mapped to the ESB, because it is also the current module used for I/O in the xPic code. The inference process is very fast and requires minimum resources, so it is not necessary to use the DAM for this procedure.

DLMOS-Inference will generate the output that will be stored in local disk and used by the xPic initialization tool to create the initial and boundary conditions of the code.

5.3.2 xPic

The code xPic will be run in Cluster-Booster configuration, i.e. using the CM and the ESB. All I/O is performed from the ESB.

5.3.2.1 xPic initialisation

Data from the DLMOS-Inference is read and interpreted by a python script that belongs to the xPic code. The script creates the initialization files for the code. It writes one field file (up to 1 GB of data for the largest cases) and, if it is possible, it creates a particle file (up to half a TB for the largest cases). The files are written on the SSSM and later read in parallel from all the allocated ESB nodes at the beginning of the xPic run.

5.3.2.2 xPic particle solver

The particle solver of xPic performs very fast calculations on a very large number of independent particles. The data of each particle is stored in aligned vectors in memory. Such vectors contain information about the particle location, velocity and charge.

Memory aligned temporal vectors are used to store the projected values of the electric and magnetic fields on the particles.

All vectors are fitted in the accelerator memory, aligned and ready for SIMD vector operations. The main operations performed are multiplications and additions. All these conditions demand a system which is highly parallel and independent, and benefit from a large number of cores with access to very high memory bandwidth. Following our past developments in the DEEP and DEEP-ER projects we decided to map the particle solver to the ESB.

The SLURM batch script calls the executable of xPic, pinning it to the allocated processors of the ESB and the CM. The executable splits the main communicator into two parts, each corresponding to the field (CM) and particle (ESB) solvers. Following the ESB architecture, each node runs a single MPI process connected to the accelerator. Each MPI process in the particle solver is connected to an MPI process of the field solver in the CM.

5.3.2.3 xPic field solver

The field solver requires the resolution of an iterative linear system. It also performs finite-element differential operations on a Cartesian grid. The differential operations communicate data between neighbouring processors and the iterative method does the global gathering of a residual value (the difference of the result between two iterations). These procedures are complex and require high performance in a single thread. Memory access is not necessarily cache optimised. The two communication patterns stress the system in different ways, but the amount of data transferred between processors in relatively low. The field solver of xPic is mapped to the CM to take advantage of the higher per-thread performance.

The field solver runs with its own global communicator on the CM. The field solver does not perform any type of I/O. Communication between the field and the particle solver is performed using a point-to-point MPI intercommunication. This intercommunication is less frequent than the communication inside each one of the solvers. The message size is about ten times the size of the Cartesian grid in each MPI process. For a typical run of 10×10×10 cells per MPI process, the message size between the CM and ESB modules is around 80 KB.

5.3.3 GMM

A second ML model, the GMM, is used to discover new information hidden in hundreds of GB of particle data. It allows to discover the real velocity distribution of particles in the plasma, as a combination of a few Gaussian distributions, instead of considering a single Maxwellian distribution. The difference between these two representations allow to uncover critical zones in the plasma and to correct real energy miscalculations. The analysis of particle data from the xPic particle solver with GMM runs on the DAM.

The use of GMM is a gold mine of information for a plasma scientist. Each particle output from the xPic code can carry up to a few Terabytes of information, so storing a few time steps for later analysis is impossible. On-the-fly analysis of particle information using GMM is a major advantage in the discovery of new plasma physics.

5.3.4 Space Weather workflow

First, in the DLMOS workflow, solar images and solar wind information is downloaded and pre-processed using a Data Pre-Processing (DPP) tool. These large datasets are used to train a Neural Network that is capable of predicting the solar wind properties from images of the Sun. The *training phase of DLMOS* is performed in the DAM. A second independent phase of the application, the Inference, uses the trained model to infer a single solar wind condition from a single solar image. This *inference* can be performed in the ESB before the initialization of an xPic job. Following this inference the initial conditions required by the xPic code are generated. The code xPic is then executed concurrently in the ESB and the CM modules. Every few hundred iterations, detailed particle data is transferred to the GMM for analysis. The GMM is executed in the DAM while the xPic code runs. Figure 5.2 shows this workflow.

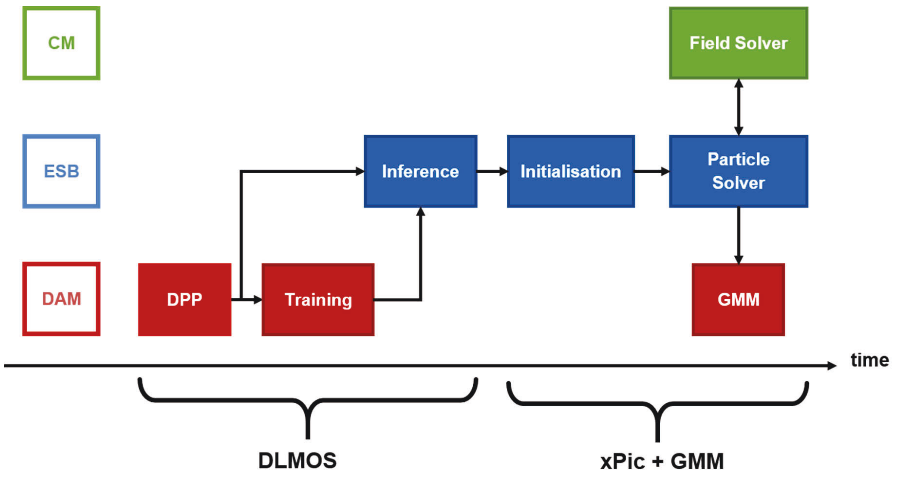

Figure 5.2: Schematic workflow of DLMOS + xPic + GMM in the MSA

5.4 Porting experience

5.4.1 Porting of DLMOS

The code DLMOS has been developed during the DEEP-EST project. It has been created and tested on the local laptops and workstations of KU Leuven, and porting them to the DEEP-EST system only required the use of the appropriate python software packages. We rely on the competence of the Jülich Supercomputing Centre (JSC) to compile the compute critical packages of the system, including PyTorch and mpi4py, but we require additional packages that cannot be fully compiled from source in each cluster. The DLMOS code uses packages like SunPy, AstroPy, scikit-learn, Pandas, among others. These, and other dependent packages, are installed using “conda install” or “pip install”. It would be extremely cumbersome if each one of them had to be installed by the system administrators.

An alternative is to use containers. With Singularity we can package and deploy our software without worrying about the installation and compilation of all our required packages. We keep outside the container the python packages that are critical for the execution in the MSA, including PyTorch, mpi4py and TensorFlow.

This approach limits the complexity of porting the code to the MSA. The DLMOS application is independent from the other two applications and does not require special data transfers.

Porting the code to the DEEP-EST system only required the intervention of the JSC team in the installation of the critical Python packages. Secondary packages were installed in the user home directory as part of their virtual environment. We estimate that the porting effort took about one working day.

5.4.2 Porting of GMM

The second ML application of KU Leuven is GMM. This software has already been used to study the complexity of plasma flows in the magnetosphere of our planet. The results have already been published in an international peer-reviewed journal[75]. The model uses mainly a customized version of the scikit-learn package. This customized version includes new functions that add features to scikit-learn. These python scripts do not need to be compiled, and run on top of the existing installed packages.

[75] Dupuis et al (2020): https://doi.org/10.3847/1538-4357/ab5524

The GMM application performs the characterization of particle velocity distributions in different regions of space. The code is embarrassingly parallel. It subdivides the physical space in multiple sub-domains and each one of them is processed in parallel by a different process running on the DAM.

We have ported the GMM algorithm to use PyTorch for the computations, following the developments of external authors. Our plan was to use the GPU offloading capabilities of PyTorch to maximally utilize the DAM. However, the advanced features that we included in the scikit-learn version could not be added in time to the PyTorch version. These functionalities are critical for our application and include a full correlation matrix and a point weighting for each plasma particle. For this report we will be using the GMM version developed with scikit-learn.

Porting the code required the intervention of the JSC team. To allow a connection with the code xPic, the mpi4py python script must be installed using the same software stack used by xPic, including the same version of ParaStation MPI. The estimated porting time was one working day.

5.4.3 Porting of xPic

5.4.3.1 Initial XeonPhi version

The code xPic was already ported to the Cluster-Booster system in the DEEP-ER project. Three different levels of parallelization were used: 1) MPI parallelism for inter-node communications, 2) OpenMP multithreading for intra-node computation and memory sharing, and 3) SIMD vectorization to take advantage of Intel vector registers in the Xeon processor line.

In addition, to make a good use of the cache hierarchy we implemented tiling of the particle solver deployed in the Booster module. The size of the tiles has an important effect on the cache accesses, in particular for the moment gathering in the particle solver. This three levels of parallelism and four levels of memory management were specially designed to work on Intel Xeon Phi processors.

Memory management is a critical component of Intel processors. In particular the allocation of aligned vectors and registers for the vectorization of operations. These fine-grained implementation details are less relevant today as compilers have implemented more and more optimization procedures. However, the code sections dedicated to memory handling become irrelevant when new architectures are used to run the codes. In our case the use of GPUs required KU Leuven to restructure large parts of the code.

5.4.3.2 GPU version

For the ESB of the DEEP-EST MSA we needed to re-orient the parallel structure and memory strategy in order to take advantage of the GPU accelerators. At that point, we made an strategic decision: we cannot depend any longer on a single technology that can disappear in the future. The discontinuation of the Xeon Phi processor line was a strong reminder that other proprietary technologies create dependencies that can have a very strong effect on our productivity. There is no perfect scaling that will recover the months of lost simulations due to a forced change in the basic structure of the code provoked by a change in the hardware architecture. This is why at KU Leuven we strongly feel that the path towards Exascale must include decoupling the hardware and the software development. A clear example of the need for vendor-agnostic programming approaches is the recently announced European LUMI supercomputer. This EuroHPC JU project is based on AMD technology, including GPU and Data Analytics partitions based on AMD GPUs.

For these reasons we decided not to use CUDA when porting the particle solver to GPUs. We opted for the most recent possible versions of OpenMP to offload computations to the GPU of the ESB (or the DAM if needed). This required a close collaboration with the JSC support team to compile a full stack based on GCC 10.2.0, compatible with parts of the OpenMP 5.0 standard. This was a delicate procedure as the full stack had to be updated, including the compilation of cross compilers with support to NVPTX, the use of updated glibc and binutils, and the recompilation of all the libraries in the stack. The full procedure was not straightforward and might require updates when new versions of the GCC compiler are available.

Among all the compilers GCC presented the closest compatibility with the OpenMP 5.0 standard and with the stack libraries. However, we are aware that the performances of offloaded code to the GPUs with OpenMP 5.0 and GCC is not the best in published tests. Right now we are focusing our efforts on the deployment of a code that can be easily transported to a different system without major headaches. Performance optimizations will be carried on in future projects (e.g. the DEEP-SEA project).

The method used to port the code to the GPU architecture is detailed in the Best Practice Guide, section 8.4.3.4 of this volume (OpenMP 5.0). The main changes performed are the following:

- Replace the existing OpenMP SIMD pragmas with a generic macro-defined pragma. This macro definition changes depending on the type of offloading device used. For GPUs the pre-compiler imposes a target section, while for CPUs it imposes a SIMD for loop section.

- Memory transfers from/to host and device are performed using the OpenMP map-directives of the Cartesian fields used by the particle solver. These include the electric and magnetic fields (to), and the particle moments (from).
- Particle initialization and transport is performed only in the device. Information about the position, charge, and velocity of the particles is maintained only in the device memory.
- Particle memory allocation is performed by a wrapper routine that selects the memory allocator between `malloc` (CPU) and `omp_target_alloc` (GPU).
- All pointers to particle information are carefully identified as `is_device_ptr` in the target directive sections.
- The sorting algorithm used to arrange and select particles for communications had to be re-written from scratch to compensate the limitations of the OpenMP offloading system. The sorting algorithm has been decomposed and reduced to basic for loops with auxiliary vectors. It currently still requires the use of a serialized section that hinders its scalability. This serial section will need to be re-worked in a future version of the code. We believe that sorting algorithms that use the OpenMP offloading to the GPU will be a major requirement in future developments for KU Leuven and for other application developers.
- The moment gathering algorithm of the code also needed a major restructure. The previous version of the code included sections that where tailored for the vector registers of the Intel processors. These algorithms could not work under the GPU architecture. Major changes, simplifications and testing under CPU and GPU architectures implied a large number of bugs and memory leaks that required repairs. This is the single largest change in the code.
- The previously existing parallelism layers are maintained, but in the case of GPU offloading we limit the number of OpenMP threads to 1 per each MPI process in the particle solver, in order to use only 1 GPU per MPI process. We also limit the number of blocks to 1. This means that each MPI process will count only 1 OpenMP thread, with only one block. The CPU version of the code can make use of more complex combinations to take advantage of memory cache in Xeon processors. In the future we will work on the use of multiple threads per MPI process to deploy on more than one GPU per MPI.
- GPUDirect communications are used to move particles between subdomains located in different GPUs. We have added a new communication buffer in the particle solver to define specific memory locations in the GPU devices, required by the CUDA-aware MPI layer.

Random number generators and sorting could be wrapped in functions that call standard C or CUDA routines. This shows interoperability between OpenMP and CUDA runtime libraries. However, for the current version of the code we have used the standard C library implementations of all mathematical functions. For scientific codes the correct generation of random numbers is a major issue that requires more careful developments in the future. In particular for MSA systems, where CPUs, GPUs, multiple nodes, and multiple modules are involved, the generation of random numbers will require much more attention.

We estimate the total personnel effort of the changes and adaptation of the code to roughly 4 PM, where the typical ratio of development time to bug correction time is in the order of 3:1, i.e. one day of developments lead to 3 days of corrections, testing and optimizations.

5.5 Scalability

5.5.1 Scalability of GMM

The code has been executed in the DAM of the DEEP-EST system, using the results of a previously executed simulation. A list of 384 particle files were available for processing, each one corresponding to a subdomain of the simulated 2D box. The files were written independently by each process of a parallel execution that ran on a NASA supercomputer using 16 nodes with 24 cores each, which by coincidence has the same core distribution per node as the CM of the DEEP-EST system.

We performed a weak scaling test, where a single processor reads and processes a single file. We use 48 threads per node and scale the execution from 1 to 8 nodes, for a total of 384 threads in the largest execution, which processes the full dataset.

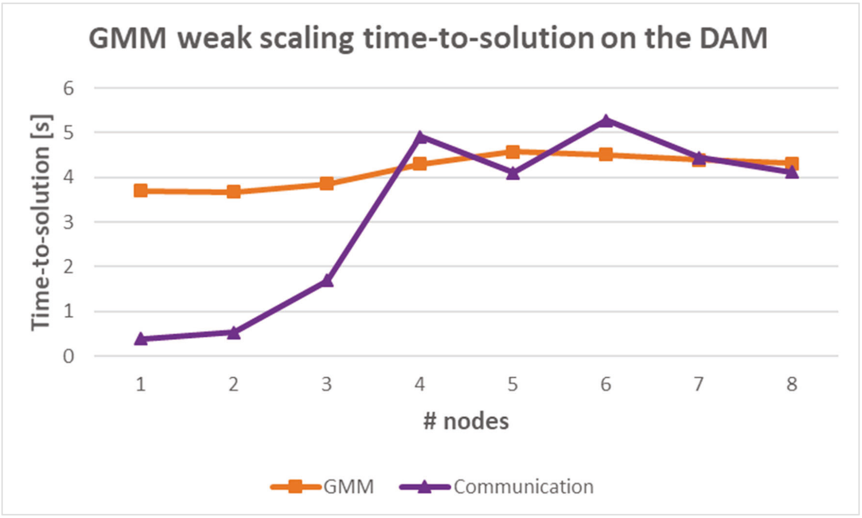

Figure 5.3: GMM weak scaling performance on the DAM

Figure 5.3 shows the runtime of the GMM in the DAM. We have extracted three timers from the script: 1) I/O (not shown in the figure), 2) GMM execution, and 3) MPI communications. The code uses MPI at the end of the execution to collect in a single node all the results and produce the final output figures. The I/O time was negligible for all the runs, in the order of milliseconds. The points in the figure correspond to the average value extracted from all the runs.

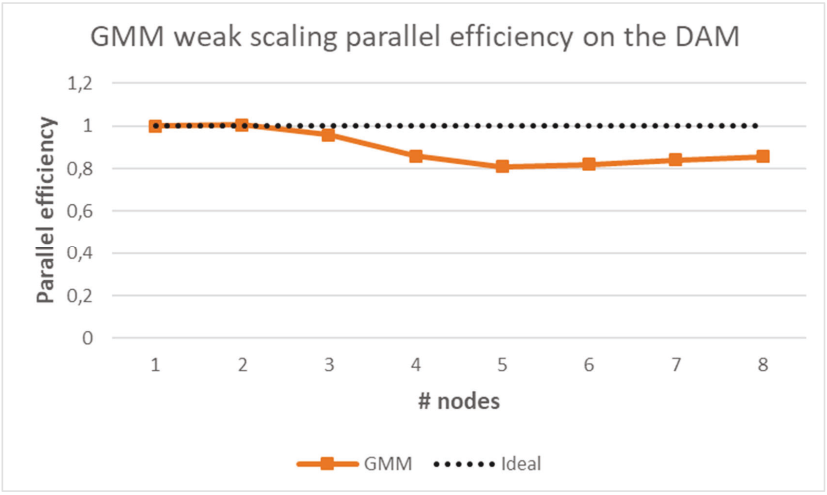

Figure 5.4: GMM weak scaling parallel efficiency on the DAM

It is clear from Figure 5.3 and Figure 5.4 that the GMM execution presents an almost perfect weak scaling efficiency with very small variability (i.e. good load-balancing). However, the very rudimentary MPI communications implemented present a major bottleneck in the code. When the number of MPI processes exceeds 144, the communication time becomes equal to the processing time.

We are satisfied with the performances of the machine learning algorithm, but we will work on the parallelization of the final I/O to avoid this critical bottleneck. Such work will take place after the end of DEEP-EST in the frame of the DEEP-SEA project.

5.5.2 Scalability of the particle mover of xPic

In this section we will focus specifically on the scalability of the particle mover on the ESB. To isolate the particle mover and test the basic functionalities of the newly improved code, we turn off the moment gathering in the particle solver and the field solver entirely. We execute the code on the ESB exclusively. With this setup the particles will still move following the Newton equations of motion in a constant electromagnetic field that does not change.

The tests show the scalability of the particle mover and the performances of the GPU under low and high memory loads. Figure 5.5 and Figure 5.6 present the parallel efficiency of the xPic code for a strong scaling case. The memory occupancy of the strong scaling tests changes from 31 GB per GPU (1 node) to 0.5 GB per GPU (32 ESB nodes). Such a decrease in the use of GPU resources (memory and computing) has an impact on the efficiency of the code. Moving from 8 to 16 ESB nodes we can see a strong drop in Figure 5.6. This is a trend that continues when using 32 nodes.

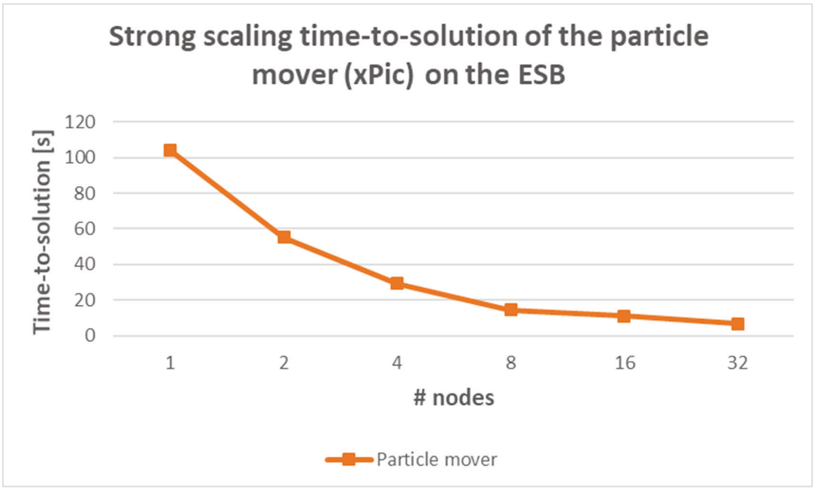

Figure 5.5: Strong scaling time-to-solution of the xPic particle mover on the ESB

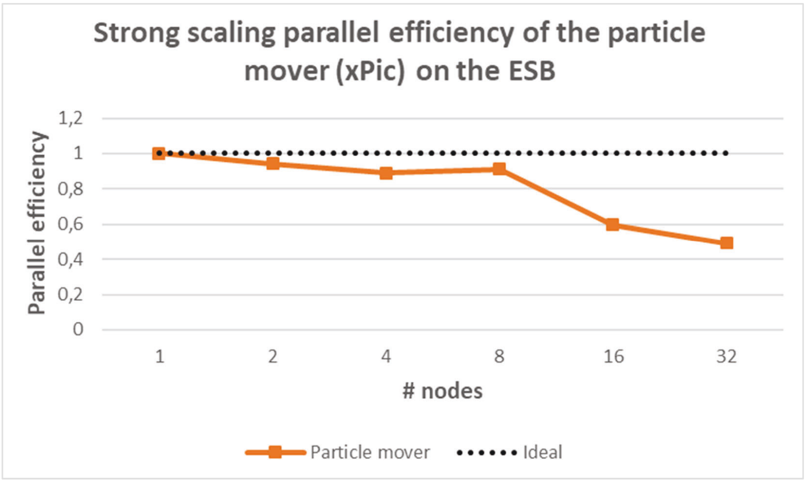

Figure 5.6: Strong scaling parallel efficiency of the xPic particle mover on the ESB

This shows that it is extremely important to occupy as much memory as possible in the GPU to take advantage of the fast calculations on the simple operations used in xPic, and to minimize memory management overheads. The NVIDIA V100 GPUs used in the ESB have a capacity of 32 GB that we can fill with half a billion particles.

One of the priorities for KU Leuven is to demonstrate a good weak scaling of the code. Figure 5.8 and Figure 5.8 present the results of a weak scaling test performed from 1 to 32 nodes on the ESB. The figures show that the particle mover keeps a constant, nearly perfect scalability, but has a small performance-drop moving from one to multiple nodes. We think that GPUDirect communications between GPU nodes can account for this initial loss in performance.

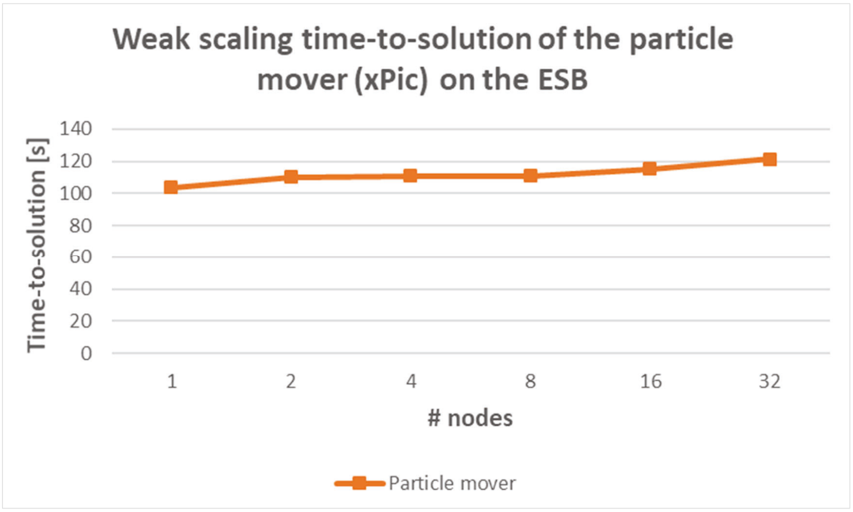

Figure 5.7: Weak scaling performance of the xPic particle mover on the ESB

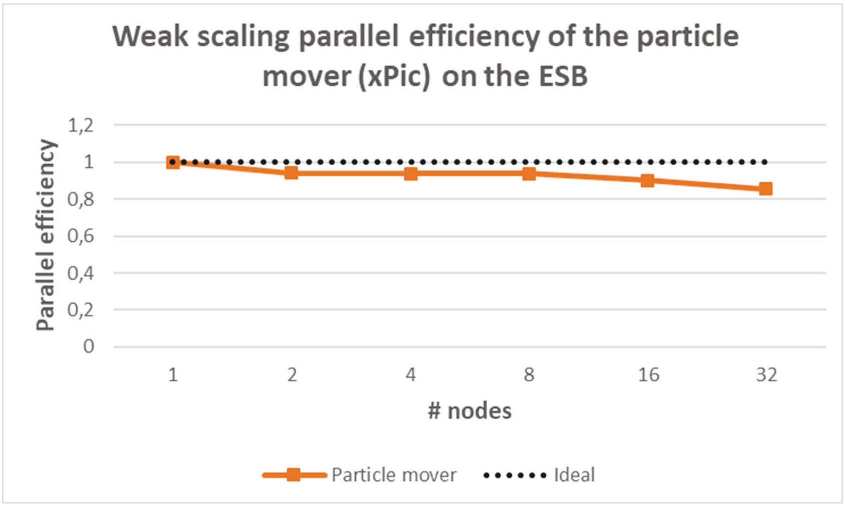

Figure 5.8: Weak scaling parallel efficiency of the xPic particle mover on the ESB

5.5.3 Testing of the number of tasks per node in the ESB

For the results in this section we have activated all the phases of the xPic code: the field solver, the particle mover, and the moment gathering. We have executed the code in what we call the “MONO” mode (from the word monolithic). In this mode the code is executed using a classical monolithic architecture where all the phases of the code run in the same node.

The MONO mode can be run in any of the modules of the DEEP-EST system. When executed in the CM, the code runs each one of the compute phases in the CPU, including the two phases of the particle solver. When the code is run with the MONO mode in the ESB or the DAM, the particle solver offloads its compute intensive parts to the GPU in the corresponding node. A single ESB or DAM node contains a single GPU. However, when multiple MPI processes are launched in the same node, each one makes use of the same GPU. This action is possible thanks to the Multi-Process Service (MPS) of the NVIDIA V100 cards: the GPU can hold multiple concurrent tasks. It is then possible to launch an 8 MPI process job in a single ESB node with one MPI process per core, while these 8 processes share the single GPU available in the node. This means that we either: a) use a single core per node attached to a single GPU to maximize its performance, or b) use all the CPU cores of the node, each one sharing the GPU and using 1/8th of its memory and computing capacity.

To test the MPS of the NVIDIA V100 cards we tested the number of concurrent tasks that can be executed in a GPU. We performed a weak scaling test with a fixed number of 16 ESB nodes, this time changing the number of MPI processes per node, from 1 to 16. For this weak scaling test we used a number of particles that could be fitted in the GPU memory.

In this test we perform a basic neutral plasma simulation that solves the transport of particles immersed in an electromagnetic field. The simulation is 2D and the total number of cells used for each case is shown in Table 5.1. The total number of particles per task in each node is equal to 1,572,864. The largest of the simulations listed in the table contains a total of 402,653,184 particles, corresponding to 64 particles per cell for each one of the two particle species used, multiplied by the total number of cells listed in the last row of the table.

# nodes	# task/node	# cellx	# celly	# total cells	Total (sec)
16	1	384	512	196,608	137,442
16	2	768	512	393,216	143,21
16	4	768	1024	786,432	157,467
16	8	1536	1024	1,572,864	194,166
16	16	1536	2048	3,145,728	433,753

Table 5.1: Experiment setup for testing the number of tasks per node

Figure 5.9 and Figure 5.10 show the weak scaling efficiency of the field solver in the CPU when testing the number of tasks per node and Figure 5.11 and Figure 5.12 show the strong scale efficiency of the particle solver when testing the fraction of GPUs used per task. There is a reason for the appearance of these plots: a single ESB node is composed of an 8-core CPU and a single V100 GPU. Going from 1 task to 2 tasks per node means that the field solver will be using double amount of CPU cores, while the particle solver will be using only 0.5 GPUs per task. For this particular reason the efficiency of the two solvers has to be computed using a different metric.

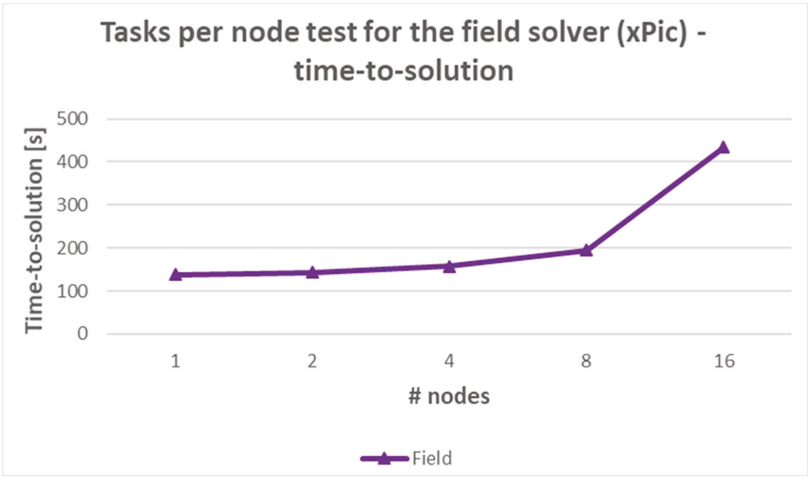

Figure 5.9: Tasks per node test for the xPic field solver – time-to-solution

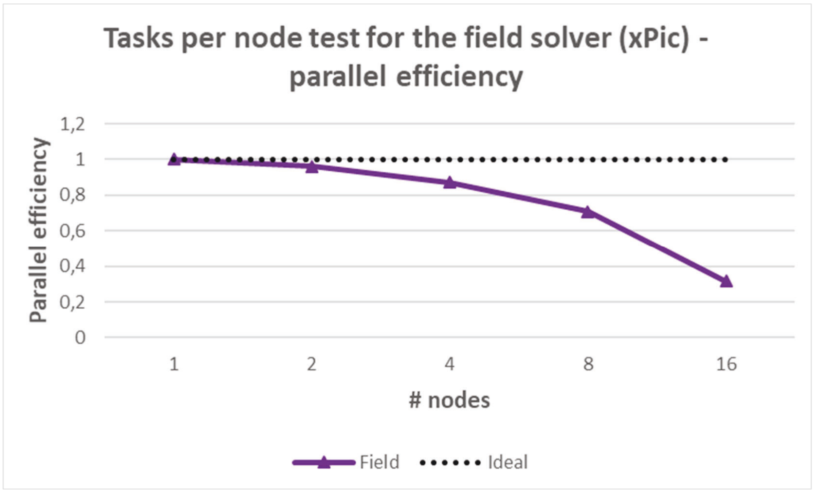

Figure 5.10: Tasks per node test for the xPic field solver - parallel efficiency

In this case the field solver shows a clear degradation of its scalability. We are still investigating the reason for such poor performances in the CPU side. This behaviour has been observed in other runs. The field solver relies on the parallel algorithms of the library PETSc. It is possible that the amount of cells used per task (12,000 cells) is too large for the solver to handle it efficiently. In different runs with a smaller number of cells per task we have noticed that PETSc switches to a different numerical solver that converges faster.

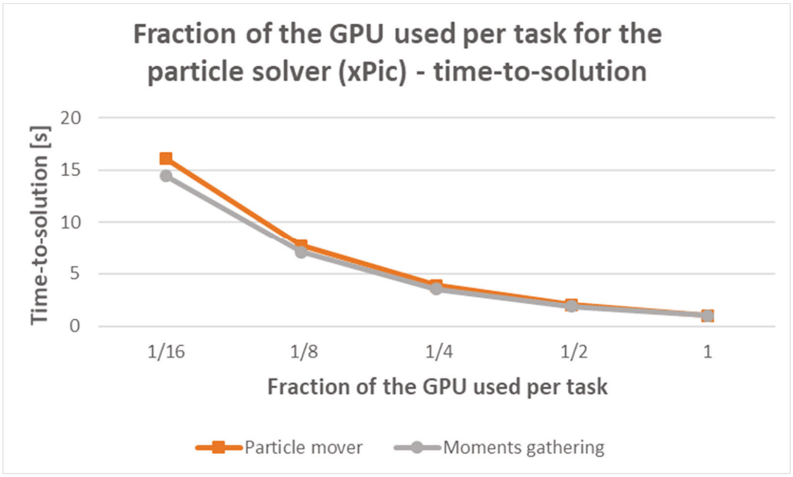

Figure 5.11: Fraction of the GPU used per task for the xPic particle solver - time-to-solution

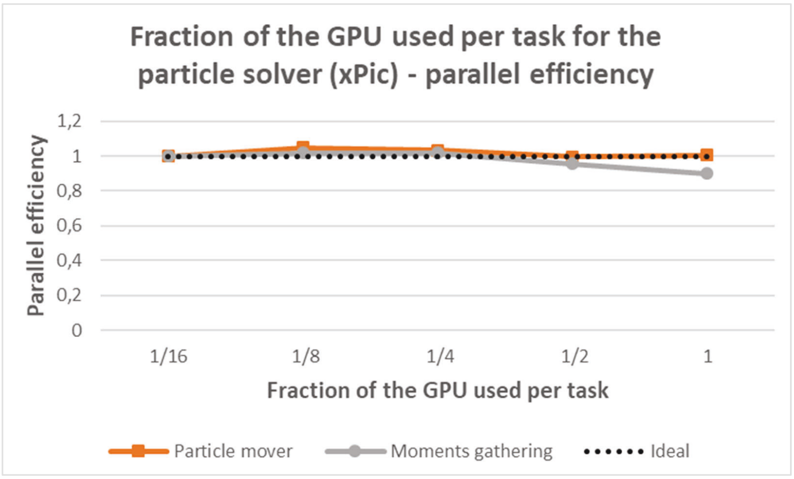

Figure 5.12: Fraction of the GPU used per task for the xPic particle solver - parallel efficiency

On the other hand, the particle solver presents a nearly ideal parallel efficiency. This shows that the MPS of the Nvidia GPU has no problem handling multiple simultaneous tasks. Every time the number of tasks doubled, the runtime of the particle solver halved.

5.5.4 Testing the number of particles per cell

One of the main features of the xPic application is using accelerator nodes to execute the particle solver. The advantage of our application is that we can fill the accelerator with operations by increasing the number of particles used. Figure 5.13 shows a run of the xPic code using a 2D setup with 1,536×1,024 = 1,572,864 cells and two particle species. The number of particles per cell and per species was increased from 64 to 128 and finally 256 (×2 due to the number of species). Inspired from the results of the previous section we use a total of 8 tasks per ESB node. This means that each task working on the particle solver has a maximum memory capacity of 4 GB (8 tasks × 4 GB = 32 GB of memory onboard the GPU).

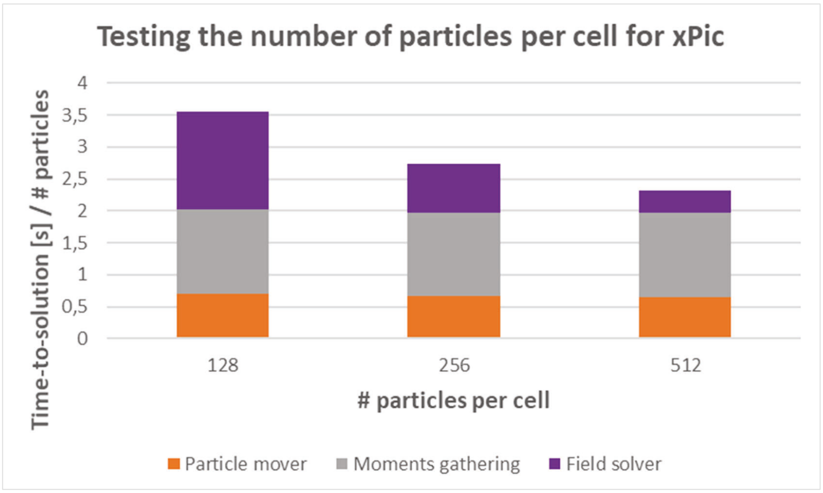

Figure 5.13: Increasing number of particles per cell

This figure shows that the particle solver scales almost perfectly with the number of particles. This also means that the GPU and its MPS makes very good use of the available resources when the GPU memory is full, as well as when only a fraction of the accelerator is used. Because the execution of the field solver does not depend on the number of particles, it shows a constant runtime for each one of the jobs above, i.e. the total runtime decreases when it is normalized by the number of particles.

In the next sections we will perform similar runs in the "MULTI" module mode, executing xPic across Cluster and Booster. We had to make a compromise between the performances of the code and the available number of resources. Each MPI process in the field solver can be executed in a single CPU core. This means that there is a total of 1,200 available cores for the particle solver, while the number of maximum GPUs available for the particle solver in a single MLA module is 50 (ESB nodes in the IB network). Using a distribution of 8 tasks per ESB node we can extend the available resources, dividing each GPU in eight tasks. We will be able to couple 400 CM cores with 400 ESB tasks, for a maximum run of 8 CM nodes coupled with 50 ESB nodes.

5.5.5 *xPic weak scaling on the CM and on the ESB*

We have performed a weak scaling test of the full code in the MONO mode using only the ESB nodes (Figure 5.14). Figure 5.15 shows the parallel efficiency of the code from 1 to 32 ESB nodes, where each one of the phases has been normalized to one. The setup of this test is the same as the one presented in the previous section. For these jobs each node has been divided in 8 tasks (MPI processes) and the number of particles per cell is set to 256.

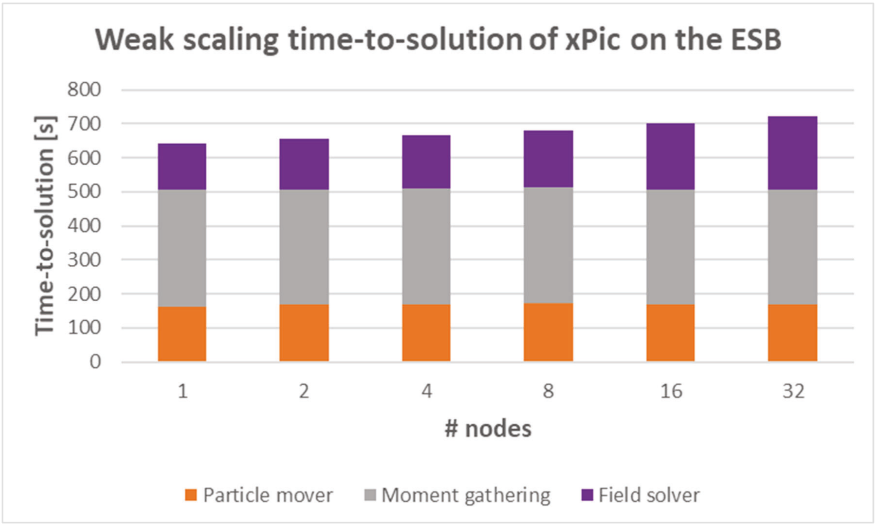

Figure 5.14: Weak scaling time-to-solution of xPic on the ESB

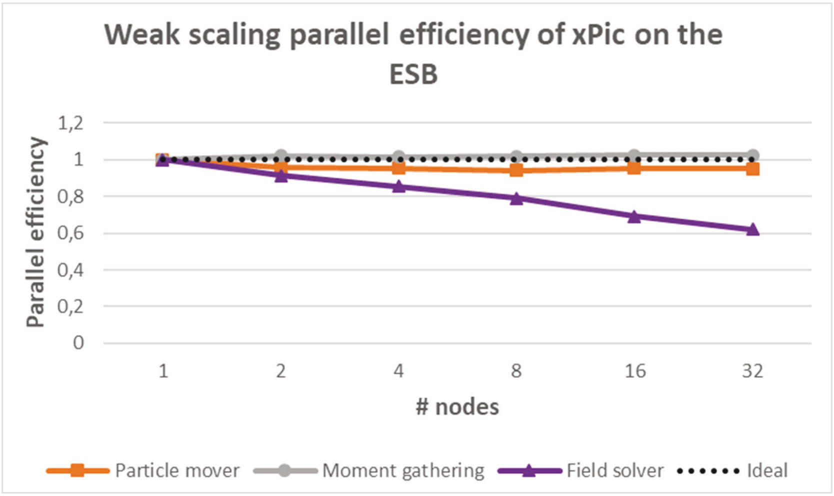

Figure 5.15: Weak scaling parallel efficiency of xPic on the ESB

These runs show again an almost perfect scalability of the particle solver. Both the particle mover and the moment gathering have an almost perfect parallel efficiency. Only the field solver struggles to maintain its scalability using the CPUs of the ESB nodes, as previously shown in the tests in Figure 5.9. The field solver efficiency drops to 62% at 32 ESB nodes.

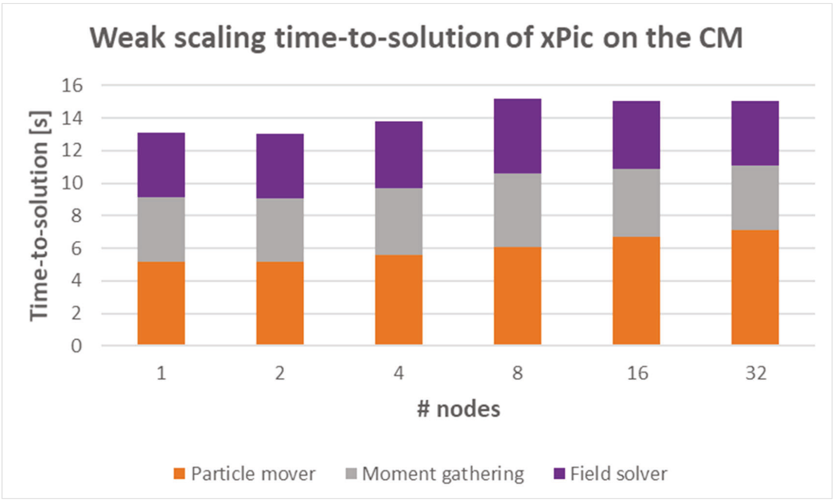

Figure 5.16: Weak scaling time-to-solution of xPic on the CM

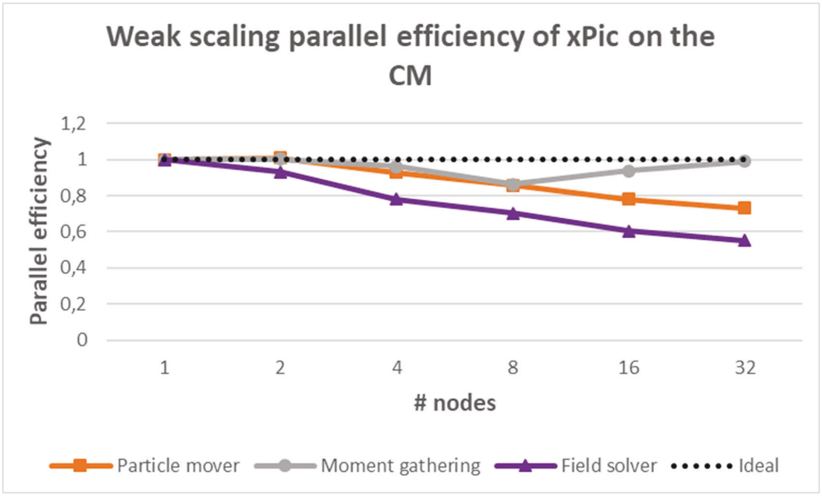

Figure 5.17: Weak scaling parallel efficiency of xPic on the CM

Figure 5.16 and Figure 5.17 show the same test, this time executed in the CM. In this test the field solver also shows a degradation in parallel efficiency, down to 55% at 32 nodes. At the same time the particle mover shows a reduction to 73% at 32 nodes, but the moment gathering execution fluctuates with a minimal efficiency of 86% at 8 nodes and a maximum of 99% at 32 nodes. These two figures show that an execution of the particle solver on the ESB accelerators allows to reach an almost perfect code-scalability, compared with the results obtained in the CPU runs. It also shows that the field solver requires a more careful optimization.

5.5.6 xPic weak scaling on CM+ESB

We have ported the code xPic to the Cluster-Booster (CN+ESB) mode, which we also call the „MODULAR“ mode. Both the MONO and the MODULAR versions of the code cohabitate in the same sources. They share the most important segments of the code, but have a critical difference: the transfer of information between the field solver and the particle solver is done using MPI communications, and each one of the two solvers is executed as an independent application.

The selection of the MODULAR version of xPic is performed during compile time. A CMake option turns ON/OFF the declaration of a compile-time variable that selects the pieces of code that need to be compiled. The particle solver can then be compiled with or without GPU offloading. This means that we can perform tests of the performances of the MODULAR version in a CN-CN arrangement, using the CN both for the field and the particle solver.

In this section we report the performances of the MODULAR xPic using the CN-ESB arrangement, with all the nodes using the same InfiniBand network.

For this test we have used a 2D plasma simulation with 256 particles per cell. Every time we double the number of nodes in our tests we try to increase the number of cells in the simulation by two. However, scaling-up the problem in this manner is not straightforward. First, to avoid doubling the size of communications in only one dimension, we enlarge the simulation domain every time in both dimensions. Second, the number of cells in each dimension must be divisible by the number of processors automatically assigned by the MPI cartesian communicators. Finally, we need to maintain a total number of cells per process as close as possible for each one of the runs.

Following these requirements we performed a first run, where the total number of cells per MPI process is approximatively 16,330. In this first run we tried to use the largest possible number of nodes in the CM. The runs use 1 to 16 CM nodes, with a total of 24 MPI process per node (1 per core). On the particle solver side, in order to reciprocate the very large number of MPI processes, we launch from 3 to 48 nodes, each one with 8 MPI processes per node. This run creates a very intense load in the ESB, and stresses the MPS of the NVIDIA V100 cards.

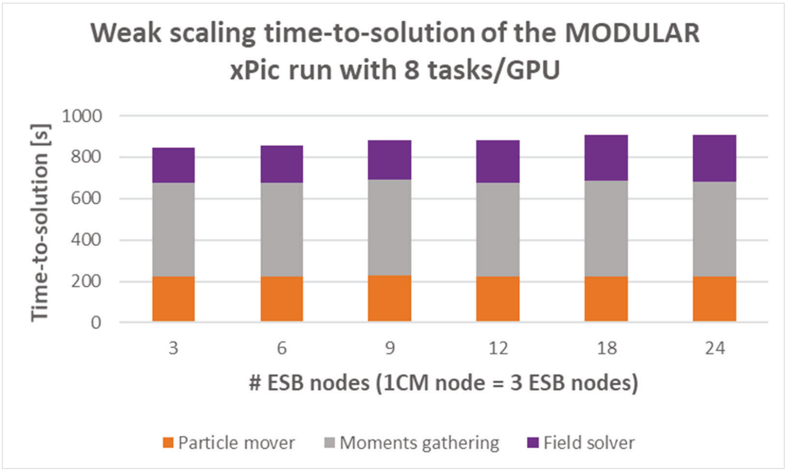

Figure 5.18: Weak scaling time-to-solution of the MODULAR xPic run with 8 tasks per GPU

Figure 5.18 shows the total runtime of the code. It shows that most of the execution time is spent in the moment gathering phase of the code. The particle solver, which includes the mover and the moment gathering, shows an almost ideal weak scale efficiency (Figure 5.19). The field solver on the other hand shows an increase of 34% in the execution time, moving from 1 to 8 CM nodes.

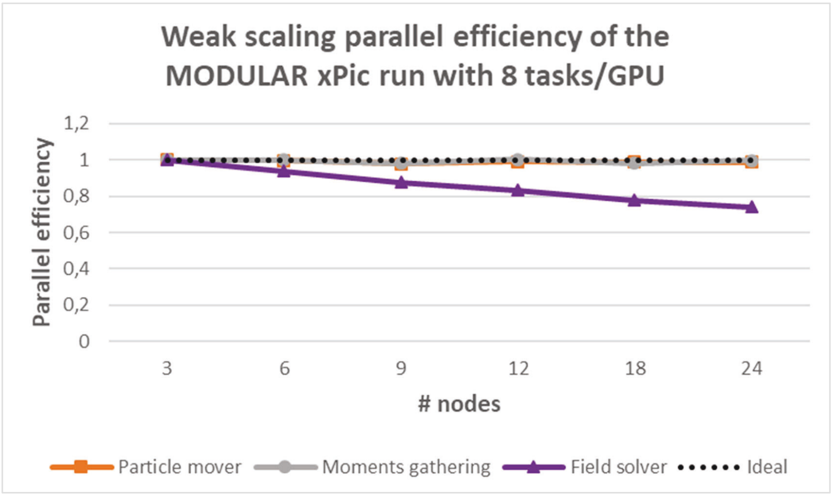

Figure 5.19: Weak scaling parallel efficiency of the MODULAR xPic run with 8 tasks per GPU

We performed a second test to verify the performances of the code when only one MPI process is assigned to each ESB node, i.e. we do not make use of the MPS of the GPU card. In this case the maximum number of MPI processes executed is 48 (the maximum allocation of ESB nodes was 48). We maintain the total number of cells per task in the order of 16,330, as in the previous test. Figure 5.20 shows the runtime of the code as a function of the number of ESB nodes.

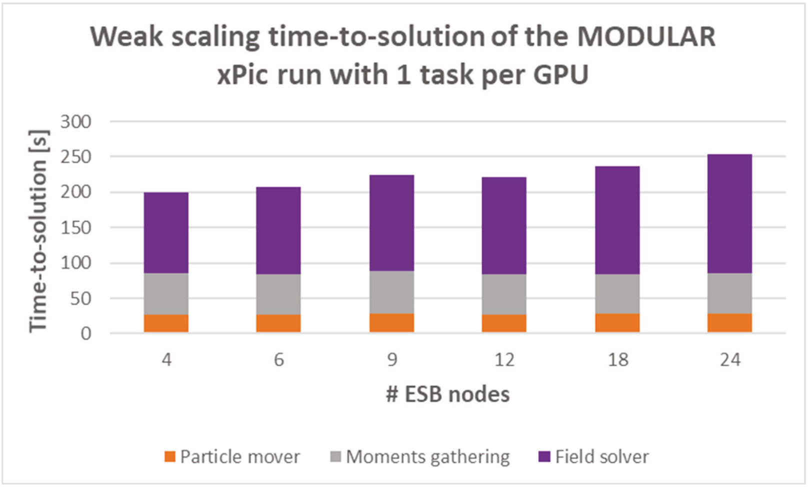

Figure 5.20: Weak scaling time-to-solution of the MODULAR xPic run with 1 task per GPU

As we expected, the particle solver is clearly accelerated by the use of the full GPU for each MPI task, instead of using only 1/8th. The particle solver presents again an almost ideal weak scaling efficiency, while the field solver once again shows difficulties to sustain an efficient scalability (Figure 5.21). The execution time of the field solver increases by 45% from the smaller to the largest simulation. Notice, however, that only one CM node was used for this test, employing a range of 1 to 24 cores of the CM node.

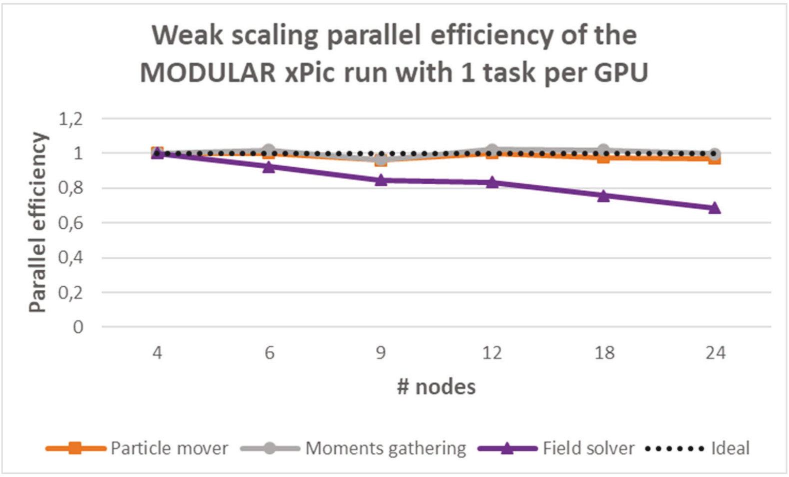

Figure 5.21: Weak scaling parallel efficiency of the MODULAR xPic run with 1 task per GPU

We decided to stress the system slightly further. We use exactly the same number of nodes and processes as in the previous test, but we dramatically increase the number of cells in each process. This allows us to fill the memory of the GPU cards and to increase the computation-to-communication ratio of the field solver. In this test we are still using only one CM node and up to 24 ESB nodes as in the previous case, but we charge each MPI process with a mean of 132,700 cells, a problem 8 times larger than before.

Figure 5.22 and Figure 5.23 show that the particle solver presents a clear parallel efficiency that is not matched by the field solver. The larger computing load also shows that the field solver does not scale as well as the particle solver and a larger portion of time is dedicated to this phase, in comparison to the previous test with lighter computing loads.

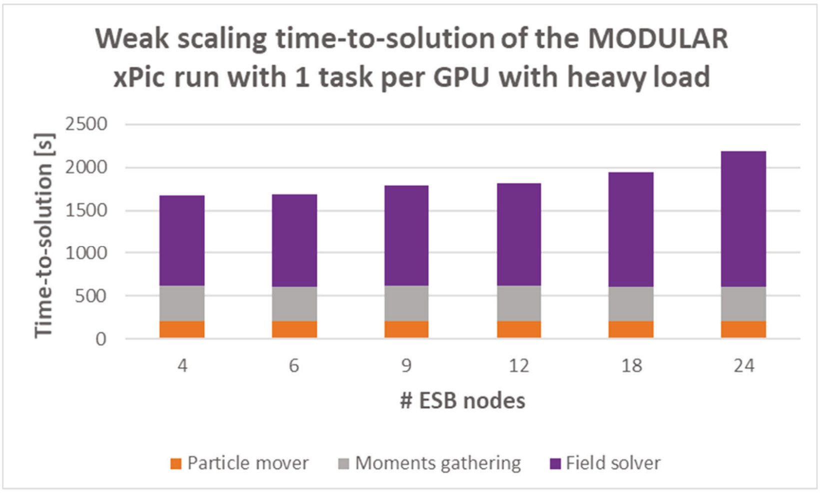

Figure 5.22: Weak scaling time-to-solution of the MODULAR xPic run (heavy load) with 1 task per GPU

We are convinced that the particle solver, the section of the code that required most changes during the DEEP-EST project, presents a parallelization strategy that is optimal in its current state. We have made last-minute corrections to some of the code-sections of the particle mover that included serial code, but we are aware that there are still some sections that can be further improved. The field solver on the other hand, requires a re-evaluation of our scaling strategy. We will evaluate the reasons why the PETSc algorithms do not scale as expected. Such evaluation is part of our future work.

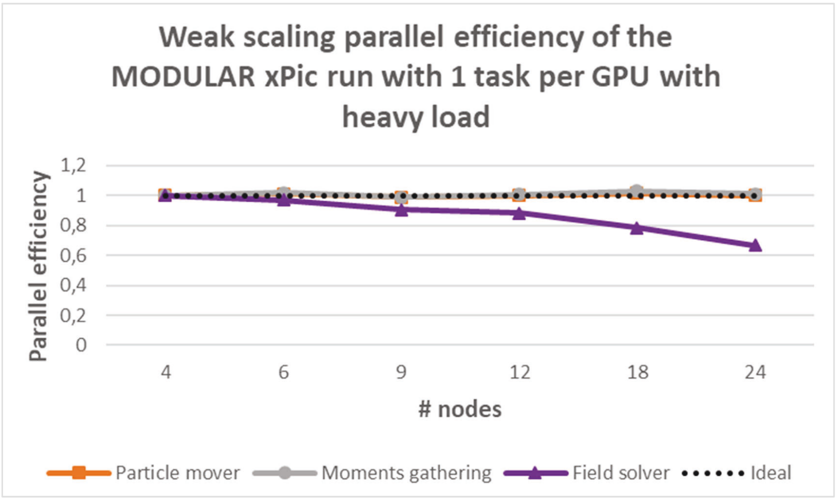

Figure 5.23: Weak scaling parallel efficiency of the MODULAR xPic run (heavy load) with 1 task per GPU

5.5.7 Our path to Exascale

5.5.7.1 What are the limitations? – Can they be fixed?

There are three factors that are currently limiting our ability to obtain better performances and reach scalability towards Exascale:

- First, the field solver presents deficiencies in its scalability. We are currently using the code at an inflection point of the library: smaller matrices could allow to switch the code into a faster and simpler model, at the expense of usability (large simulation runs require large matrices). On the other hand PETSc shows better scalability performances when the matrices solved are larger and the code spends more time performing computations and less time performing communications. This means that the subdomains have to be larger, containing many more cells, in order to gain in efficiency,
- This brings the second issue: we are currently coupling one MPI process in the CN with one MPI process in the ESB. The interface between the particle solver and the field solver uses MPI communications and a vector copy that maps 1-to-1 each side of the code. We believe that the next important step in the development of our code is to map one MPI process in the CN with multiple MPI processes (or a single process connected to multiple GPUs) in the ESB. With the good efficiency presented by the particle solver and a better ratio between CPUs and GPUs we expect to gain in scalability.
- Finally, the particle solver still contains segments that perform poorly when offloaded to the GPU. In particular the particle sorting and the moment gathering contain serialized segments that require more attention. We also need to explore the possibility of using in parallel the CPU and the GPU in the ESB nodes simultaneously. Some authors have described good performance by overlapping these two processors at the expense of data transfers between the Host (CPU in the ESB node) and the Device (GPU in the ESB node).

A more detailed analysis of all the different code phases is under investigation. We will continue the optimization and development of the application codes from KU Leuven in the future.

5.5.7.2 How to use future Exascale systems

There is a clear difference on the computational needs of the field solver of xPic and the particle solver. We believe that the Cluster-Booster division of work for this particular code is extremely important. Our goal is to perform simulations that minimize the noise. This is accomplished by increasing the number of particles in the simulation. Our goal is to reach up to 10,000 particles per cell in the future, while we currently use

256 particles in the tests performed above. This means that the stress of the system will be put in the accelerated section of the code.

We are planning to maintain the development of the Cluster-Booster model of xPic, but we think that our future code will make use of very large number of ESB nodes and a very small number of CN nodes. We will connect 1-to-many nodes between the CN and the ESB in future systems in order to increase the number of particles per cell.

5.5.7.3 Where did the DEEP-EST project help on the way to Exascale?

Until the beginning of the DEEP-EST project we were reluctant to use the GPU architectures with the CUDA language. We have been careful of not blocking our developments and become dependent on only one architecture. This is the reason behind our support for Intel Xeon Phi architecture that promised acceleration under the same software stack and hardware architecture as ordinary CPUs. It is obvious that such approach has changed since the beginning of this project, and the discontinuation of the Xeon Phi accelerators played a big role in our decision making.

However we were still reluctant to be forced to use a closed and proprietary language as CUDA. The advent of OpenMP 4.5/5.0 was a perfect opportunity for us. The knowledge we gathered in previous projects on the use of OpenMP has been very valuable here. Thanks to the help of the consortium we were able to port the code xPic to the GPUs of the ESB and the DAM, without making use of CUDA. This has been a very arduous process because compilers are still in the process of implementation of the newest OpenMP standards. We are persuaded that in the near future, compiler developers will be able to integrate OpenMP interfaces that can compete with native CUDA codes.

Without the help of the DEEP-EST consortium it would have been impossible to set up, deploy, and test this new software development approach. We are also looking forward to test our new code in systems that use AMD GPUs instead of NVIDIA GPUs. Computer centres financed by EuroHPC have invested in the purchase of large-scale AMD GPU clusters. OpenMP is the main method of porting proposed by these new computing systems, and the DEEP-EST project has enabled us to prepare port our code to this offload environment.

We have also relied on the competence of our partners to deploy libraries for parallel computing, I/O and machine learning. This has allowed us to port our ML codes and be ready for future large scale HPDA calls in European centres.

5.6 Energy consumption

We recovered the energy consumption for each one of the runs performed in the previous section from the DEEP-EST energy measurement database (DCDB[76]). We present in this section the energy consumption per node, calculating the total energy as a sum of multiple measurements of the instantaneous consumption every 10 seconds. Jobs that run during a short period produce data with larger error bars. All energies reported in the figures bellow are given in Megajoules (MJ).

Figure 5.24 shows the energy measurements obtained for the strong scaling tests in the ESB. As the work per node decreases, the time to solution is shorter and as a consequence the energy consumption per node is reduced. In this figure we plot the total energy consumed by all the nodes in each job. Under ideal conditions the energy consumed should remain equal throughout all the executions. We notice that for 8 nodes and above the energy consumption increases rapidly. As mentioned in the previous section, for more than 8 nodes the problem analysed here is too small and the GPU is launched with a very small load of work. This figure shows that the GPU is not used efficiently in this particular case for jobs with more than 8 nodes.

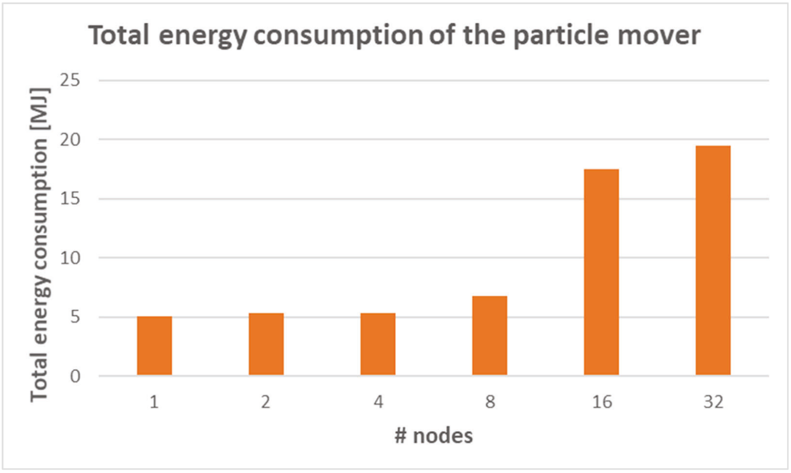

Figure 5.24: Total energy consumed for strong scaling test case

[76] https://deeptrac.zam.kfa-juelich.de:8443/trac/wiki/Public/Energy#UsingDCDB

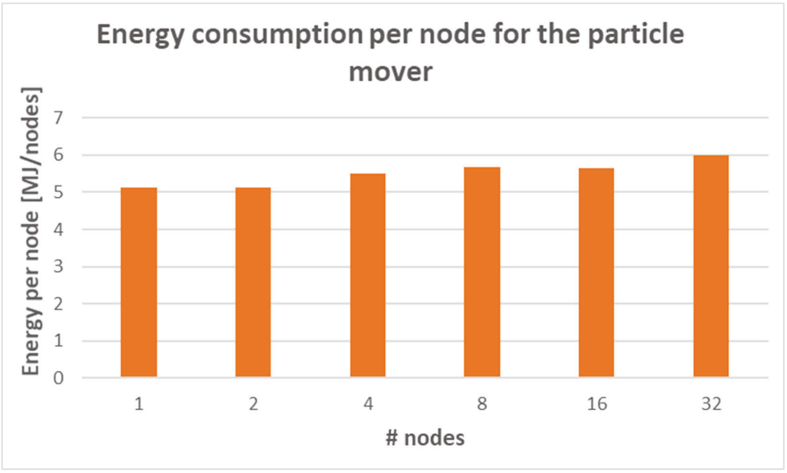

Figure 5.25: Energy consumption per node for the weak scaling test

Figure 5.25 shows the energy consumption per node for the weak scaling tests. In this case we would expect to have a constant energy consumption per node. However, this figure shows an increase when the problem is larger. We are recurrently investigating the causes and possible correctives for the strong increase of 20% from 1 to 32 nodes.

We have also gathered the energy consumption for a MODULAR run of the code xPic. The energy measurements presented here correspond to the simulation described in Figure 5.18 in Section 5.5.6. This test corresponds also to a weak scaling test. Figure 5.26 shows the total energy consumption in MJ for the CM and the ESB nodes. The measurements have not been scaled or normalized.

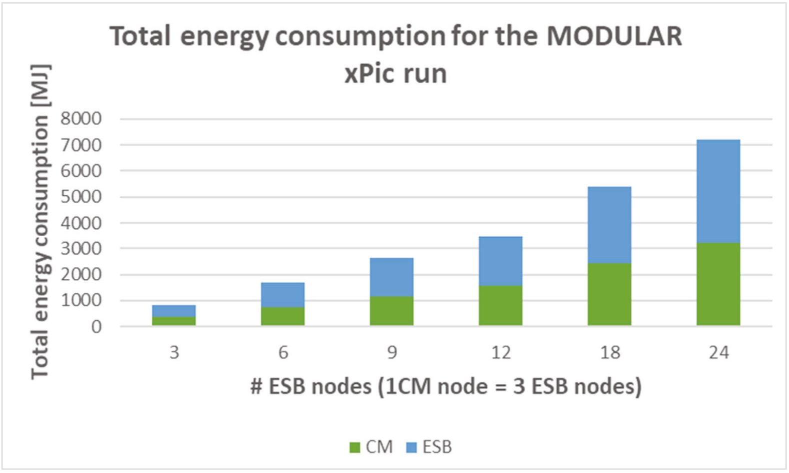

Figure 5.26: Total energy consumption for the MODULAR xPic run

Let us remember that this run was performed using one node in the CM for every 3 ESB nodes, and each ESB node was launched with 8 tasks per GPU. On average, the total consumption of energy was 24% higher in all the ESB nodes as it was in all the CN nodes. This is a very balanced use of resources.

When we perform a normalization by the number of nodes for each one of the runs (Figure 5.27), it is possible to observe the weak scaling efficiency of the energy consumption. It is also clear that each one of the ESB node consumes only 40% of the energy consumed by the CN nodes. The ratio of 1:3 nodes between the CN and the ESB is energetically balanced.

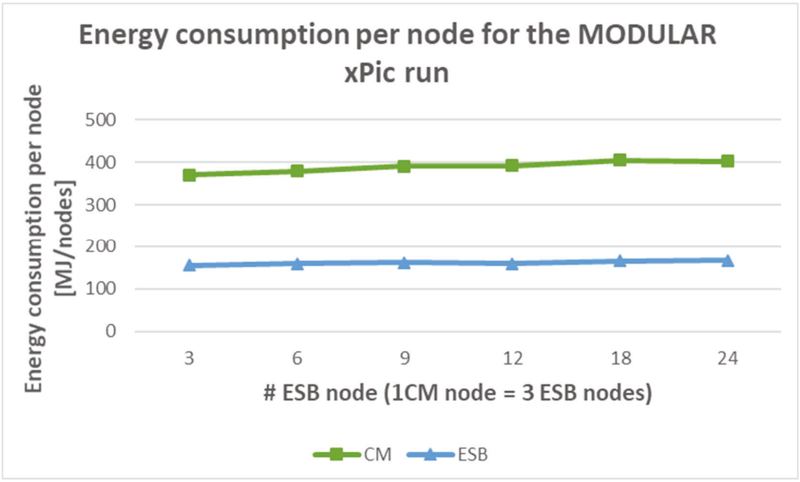

Figure 5.27: Energy consumption per node for the MODULAR xPic run

5.7 Performance comparison

5.7.1 Performance comparisons for the particle mover of xPic

Figure 5.28 shows a comparison between the runtimes for the particle mover deployed in the CPUs of the CM and on the GPUs of the ESB. The figure shows the strong scaling tests described in the previous sections. While there is a clear under-use of the ESB for runs with more than 16 nodes, the ESB performs almost at all times one order of magnitude better than the CPUs of the CM.

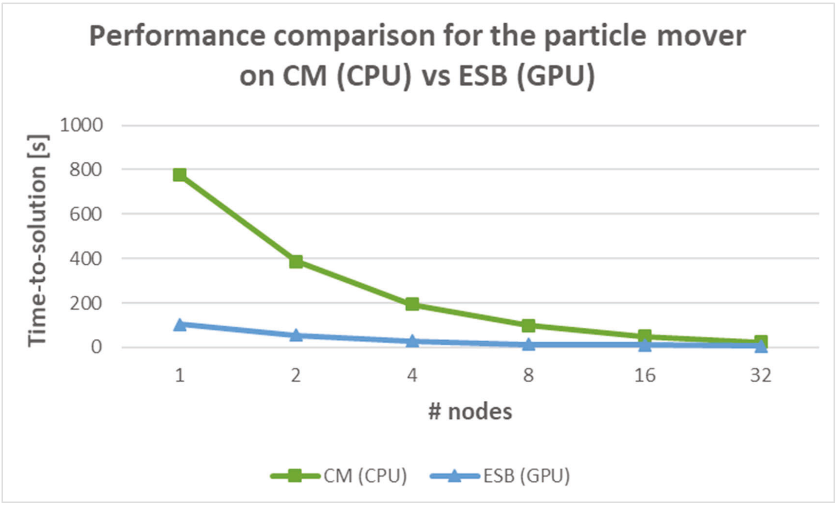

Figure 5.28: Runtime comparison (strong scaling) of the particle mover on CPUs (CM) and GPUs (ESB)

This performance comparison is more obvious in the weak scaling test (Section 5.7.2). The weak scaling bellow shows that the execution in the ESB and the CM presents good scalability, with runs on the GPU performing almost 10 times better than in the CPUs of the CM.

5.7.2 Weak scaling comparison between CM and ESB

It is difficult to perform a fair comparison between CPUs and GPUs. The main concern is the selection of what to compare: should we compare CPU core vs GPU SMD? Or is the unit of comparison the node? Here we compare the runtimes obtained during the weak scaling tests performed in Section 5.5.5. The code xPic was executed in the MONO mode, with all the phases of the code residing in the same module. In the figures bellow, the x axis represents the number of nodes used in the CN and in the ESB. The y axis represents the runtime per CPU core and the runtime per GPU. Our goal here is not to infer that one architecture is better that the other. We are simply collecting information on the current performances of the code under the basic computing units available to the users: the CPU core and the GPU card.

In Figure 5.29 we plot the total runtime in minutes for the jobs performed in the ESB (GPU) and we compare them to the execution time on the CM (CPUs). As discussed in previous sections each GPU is shared between 8 tasks, i.e. each task has only access to 1/8th of the GPU. In a similar way on CPU core has access to only 1/24th of the computing power of the full CN node. These runtimes take into account the execution of the field solver in the corresponding CPU of each node. The field solver has shown bad scalability, contrary to the particle solver that features almost a perfect scalability. The field solver accounts for around 80% of the execution time on the CM, and for 30% of the time on the ESB.

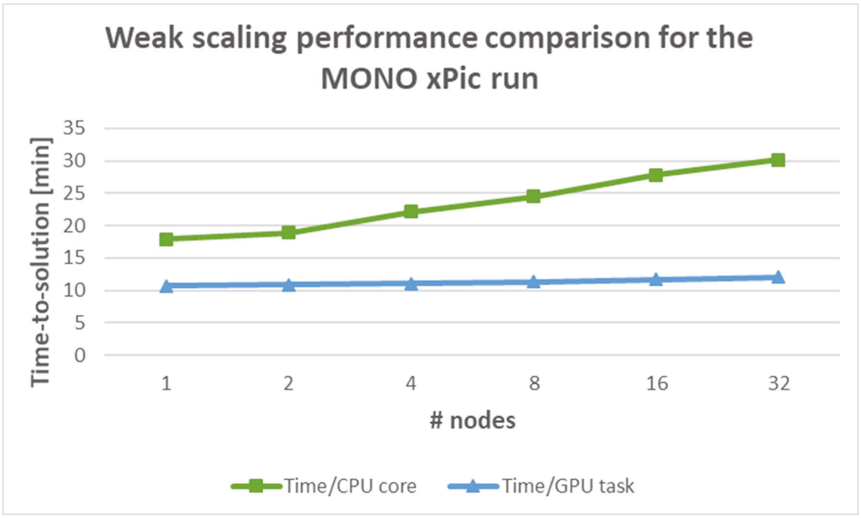

Figure 5.29: xPic runtime comparison on CM and ESB

In this figure each CPU core runs two to three times slower than the GPU. However, each CPU chip outperforms the execution time of a single GPU node. Notice that the CPU that manages the GPU in the ESB nodes is less powerful than the CPU processor in the CM nodes. The field solver is executed in these less powerful processors and accounts for at least 20% of the total execution time. While a single node of the CM advances the field solver in 35 seconds, the same procedure requires 130 seconds in the CPUs of a single ESB node.

5.8 Conclusion

As application developers for scientific software, we are constantly at the edge of new algorithm implementations. For the past few years this meant that we had to adapt our codes to different architectures: classical x86 processors, many-core architectures, and multi-core architectures. Each one of these hardware requires the developer to learn about the architecture itself and about the libraries and programming languages specific to each one of them. Now a new generation of processors (AMD, ARM, …) and a new generation of accelerators (AMD GPUs, NEC vector cards, NVIDIA GPUs, …) suggests that we need to adapt, once again all our codes to remain competitive.

We have been reluctant to use CUDA to offload computing to the GPUs. We do not want to become dependent on a single, non-European, company, which requires the use of proprietary compilers and libraries. The emergence of ROCm from AMD demonstrates that the community is eager to move forward with open source driven tools. For this reason, we favour the support and the use of OpenMP as an alternative for computing offloading to any accelerator. The EuroHPC JU program has also seen that companies like AMD are producing very competitive products, and it has invested a significant budget in the installation of pre-exascale centers based on AMD technology. This is an additional reason for us to stay away from pure CUDA implementations of our codes.

We showed in the previous sections the performances of the codes of our Space Weather application. For the past year and a half we focused our attention on the use of OpenMP to port the massively scalable particle solver of the xPic code. It consists of two phases: particle mover and moment gathering. We showed that the particle solver has an almost ideal performance when it is executed in the CM alone, in the ESB alone, and as part of the Cluster-Booster MODULAR mode, in which the field solver is executed in the CM and the particle solver is executed in the ESB.

We showed that the Multi-Process Service (MPS) of the NVIDIA cards can be safely used by application developers without compromising performances. In the MPS multiple tasks (kernels) can be executed concurrently in the same GPU accelerator. For our largest test we have used 24 ESB nodes with 8 MPI processes per GPU, while simultaneously launching 16 nodes with 24 MPI process per node in the CM. We are satisfied with the scalability efficiency of the particle solver, even as we know that some serial zones remain to be improved.

This good particle solver scalability was possible to detect and to improve thanks to the work done in the DEEP-EST project. KU Leuven has been able to track and improve segments of code which were previously identified as problematic. However, during our tests we have noticed that the performance issues have shifted towards the field solver running in the CM nodes. We will continue to perform improvements in the future.

We have used the resources of the DAM to test the scalability of the machine learning code GMM. We showed that the algorithm itself presents an almost ideal weak scaling, but the code uses MPI communications to gather the final results in a single process. Adding more and more nodes renders the MPI communications prevalent and hinders the scalability of the code. We are working on the parallelisation of the I/O in order to avoid such costly communications. Our current tests show that we can perform the machine learning analysis of the xPic particles in only a few seconds per subdomain. We are planning to launch very complex simulations with on-the-fly analysis of the particles with the GMM algorithm at a cadence equal to the I/O of the code xPic itself. This is an extremely useful development: we do not need to save the very large particle files at every output iteration, we just perform the analysis on-the-fly and retain only the high level data analysis results from our machine learning models.

In our road towards Exascale, we believe in the continuous development of the code xPic and coupling its execution with multiple on-the-fly machine learning analysis tools. We have already applied for a pilot program with the LUMI supercomputer centre where the Cluster-Booster architecture will be deployed using AMD CPUs and GPUs. We are also very happy that the energy consumption of our CPU and GPU computing loads is equal across the modules, i.e. our code presents a good energy balance.