8 Best Practices Guide

Anke Kreuzer, Jochen Kreutz, Benedikt Steinbusch, Zia Ul Huda
Jülich Supercomputing Centre, Forschungszentrum Jülich GmbH, Leo Brandt Strasse, 52428 Jülich, Germany

a.kreuzer@fz-juelich.de

Germán Llort, Julita Corbalan, Lau Mercadal Melià, Pedro Martinez
Barcelona Supercomputing Center, BSC, Spain

Jorge Amaya
Katholieke Universiteit Leuven, KU Leuven, Belgium

Hans-Christian Hoppe[89]
Intel, Germany

John Romein
Netherlands Institute for Radio Astronomy, ASTRON, Netherlands

Carsten Clauss
ParTec AG, Germany

8.1 Introduction

The DEEP-EST system implements the Modular Supercomputing Architecture (MSA) which has evolved within the DEEP project family over several years[90]. One of the most important advantages of the MSA is its flexibility: MSA systems can target a wide range of applications with widely different characteristics and system requirements. This guide shows how to port applications to the DEEP-EST system (described in Chapter 1 of this volume) and gives advice on how to get good performance out of it. Each kind of application (as with the different co-design applications within the DEEP-EST project) may have different ways to use the DEEP-EST system.

In this document, several use cases will be explained, and advice will be given about how an application can benefit most from the system architecture. Examples of the improvements that could be achieved for demonstrator applications will be shown. This guide is structured in the following way:

[89] Now at scapos AG, Germany.

[90] https://deeptrac.zam.kfa-juelich.de:8443/trac/wiki/Public/User_Guide/Tutorial1/MSA_Idea

- First in Section 8.2 we describe how to analyse an application and figure out which modules to use in Sections 8.2 and 8.3.
- Once this decision is taken, Section 8.4 focus on the real porting work (mostly porting to GPU, plus a short introduction on the FPGA porting).
- The next topic is how to partition the application code to enable it to run across multiple modules. This is covered by Section 8.5.
- Section 8.6 describes several different file systems which are provided in the DEEP-EST system.
- Section 8.7 covers certain additional features provided on the DEEP-EST system.

Last, but not least, Section 8.8 summarizes the most important lessons learned by the application developers in the DEEP-EST project, which refer to their experience adapting the codes to MSA, but also more in general when preparing them to exploit heterogeneous computing at the Exascale era.

8.2 Analysis

The three DEEP-EST prototype modules were designed to fit the needs of different kinds of applications. The ESB has the highest node count and is equipped with GPGPU accelerators coupled to relatively weak CPUs in the interest of energy efficiency. Highly scalable applications or codes with data and control structures suited to GPGPU computation can run perfectly on the ESB, yet it is essential that the computation happens exclusively on the GPGPU, and that all data structures do fit within the 32 GB of GPGPU high-bandwidth memory. Codes or code parts that require high amounts of memory, for example, should run on the DAM with 384 GB DRAM and 3 TB Persistent Memory attached to each node. There are also different ways to distribute the code parts depending on the individual application. There might even be applications using only one module, with the choice of the module depending, among others, on the problem size. Applications which combine parts best suited for different modules have the option of running simultaneously across multiple modules, while other codes with a workflow structure will run different steps on different modules, for instance as a job chain.

A detailed analysis of the code is essential to get to know which parts of the code can benefit from which parts of the architecture. Without this it is not possible to get all the benefits out of the DEEP-EST system. Here are some recommended profiling tools (all available on the DEEP-EST system):

- Intel VTune Amplifier [91]
- Intel Vector Advisor [92]
- JSC Scalasca [93]
- BSC Extrae/Paraver (for basic instruction please see Section 8.2.1) [94] [95]

These tools will help determining which are the most time-consuming parts, whether the application is compute, memory or bandwidth bound, and how well balanced the application is. With this insight the developer can decide how to map the application to the MSA: for example, time consuming parallel code parts, should exploit the scalable GPU nodes on the ESB, whereas code parts that need a large amount of (fast) memory should use the DAM with Intel Persistent Memory.

8.2.1 Performance analysis tools, Extrae, Paraver & Dimemas (BSC)

Extrae is a dynamic instrumentation package to trace programs. It generates trace files that can be later visualized with **Paraver**. To use Extrae on the DEEP-EST system load first a compiler and the MPI distribution that you want to use, e.g. GCC and ParaStationMPI, and then load the Extrae module:

```
ml GCC

ml ParaStationMPI

ml Extrae
```

8.2.1.1 Using Extrae in 3 steps

8.2.1.1.1 Adapt the job script to use Extrae

The job script needs to be adapted in three aspects (Figure 8.1):

- Load the above mentioned modules
- Specify the name for the output traces (optionally)
- Run with Extrae

[91] https://software.intel.com/content/www/us/en/develop/download/intel-vtune-amplifier-2019-help.html

[92] https://software.intel.com/content/www/us/en/develop/articles/quick-analysis-of-vectorization-using-intel-advisor-2019.html

[93] https://apps.fz-juelich.de/scalasca/releases/scalasca/2.5/docs/UserGuide.pdf

[94] https://tools.bsc.es/doc/html/extrae

[95] https://deeptrac.zam.kfa-juelich.de:8443/trac/wiki/Public/User_Guide/Tutorial2

```
#!/bin/bash
#SBATCH --job-name=lulesh2.0_27p
#SBATCH --output=\%x_\%j.out
#SBATCH --error=\%x_\%j.err
#SBATCH --ntasks=27
#SBATCH --nodes=2
#SBATCH --cpus-per-task=1
#SBATCH --exclusive
#SBATCH --time=00:10:00
#SBATCH --partition=dp-cn

ml GCC
ml ParaStationMPI
ml Extrae

export TRACE_NAME=lulesh2.0_27p.prv

srun ./trace.sh ./lulesh2.0 -i10-s65
```

Figure 8.1: Job script with Extrae

The `trace.sh` wrapper loads Extrae. The user needs to select the proper tracing library depending on their application type (MPI, OpenMP, CUDA, hybrid, etc.) and language (C, Fortran). The available libraries can be found under $EBROOTEXTRAE/lib.

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

export EXTRAE_ENFORCE_FS_SYNC=1

# Configure Extrae
export EXTRAE_CONFIG_FILE=./extrae.xml

# Load the tracing library (choose C/Fortran)
export LD_PRELOAD=$EBROOTEXTRAE/lib/libmpitrace.so
#export LD_PRELOAD=$EBROOTEXTRAE/lib/libmpitracef.so

# Run the program
$*
```

Figure 8.2: `trace.sh` to extract traces with Extrae

8.2.1.1.2 Extrae XML configuration

Within the `trace.sh` file you have to specify the XML file containing your Extrae configuration. In Figure 8.2 it is called *extrae.xml*. Here you can configure what will be traced, e.g. if you want to trace the MPI calls and the call-stack the file should look like this:

```
<mpi enabled="yes">
    <counters enabled="yes" />
</mpi>
...
<callers enabled="yes">
    <mpi enabled="yes">1-3</mpi>
    <sampling enabled="no">1-5</sampling>
</callers>
```

There are several other options which can all be found in the Extrae documentation[96].

8.2.1.1.3 Run it

Now you can submit your job as usual:

```
sbatch job.slurm
```

Please note: Always run your job from the /work directory not from $HOME!

Once the job finishes you will have the trace (3 files):

- lulesh2.0_27p.pcf
- lulesh2.0_27p.prv
- lulesh2.0_27p.row

8.2.1.2 First steps of analysis

To analyse the traces first copy them to your local computer and then load them with Paraver. Several guided demos are included with Paraver, which walk the users through the first steps of analysis with real applications examples. These are available for download clicking on *Help → Tutorials → Download* and *install tutorials*. Following the tutorials is as easy as clicking on the hyperlinks which open pre-generated example traces and different analysis views.

For new users it is recommended to start with Tutorial 1 which explains basic control and navigation with the tool; and Tutorials 4 & 5 which show two examples of complete analyses with pre-generated traces from real applications. More advanced users will find Tutorial 3 interesting as it describes an analysis methodology that focuses on detecting work and performance imbalances. If the users already have a trace of their own application and load it on Paraver, the tutorials can be likewise applied on their traces, and the analysis views will be computed on the users' application. For more information on how to analyse the traces and using Paraver, we refer to the Tutorial[95].

[96] https://tools.bsc.es/doc/html/extrae

8.2.1.3 Simulations with Dimemas

Dimemas is a simulator that reconstructs the time behaviour of a parallel application on a machine modelled by a set of performance parameters. Performance experiments can be done easily changing the target architecture by modifying network and CPU parameters. For communications, a linear performance model is used, but some non-linear effects such as network conflicts are taken into account. The simulator allows specifying different task-to-node mappings.

This simulator is useful to predict the behaviour of applications on non-existent machines, perform parametric sweeps (e.g., mass-evaluate different BW and latencies), and conduct 'what if' analyses to answer questions like: *"Does the application have load balanced and dependence problems?"*, *"Would we benefit from grouping messages?"*, *"Is bandwidth the problem?"*, *"Is network contention the problem?"*.

Dimemas generates Paraver trace files enabling the user to conveniently examine any performance problems indicated by a simulator run. The Paraver Tutorial 2 contains an introduction to the use of Dimemas with an example and guidelines to get started with this tool. For more information on the architecture and use of the simulator, the user may refer to the tool website[97].

[97] https://tools.bsc.es/Dimemas

8.3 MSA Usage Models

Within the DEEP-EST project we identified 6 different usage models for our applications, which can be sorted into two different categories: Single and multi-module usage. In each category we have three different usage models (see Figure 8.3).

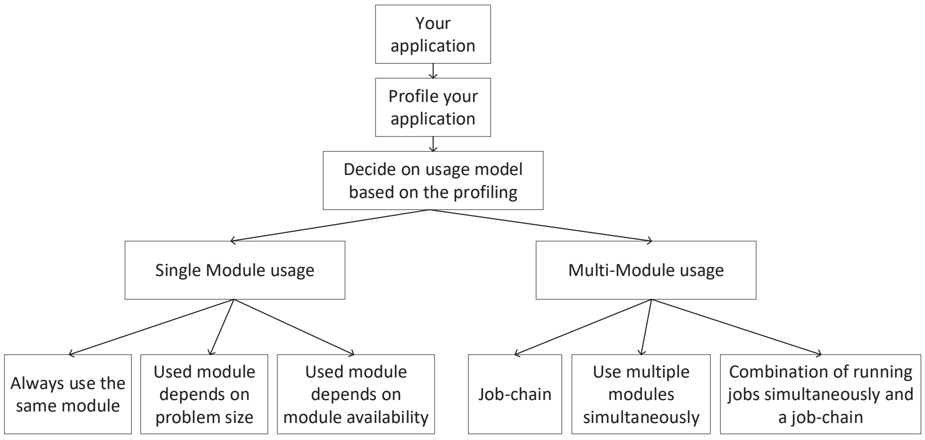

Your application
Profile your application
Decide on usage model based on the profiling
Single Module usage
Multi-Module usage
Always use the same module
Used module depends on problem size
Used module depends on module availability
Job-chain
Use multiple modules simultaneously
Combination of running jobs simultaneously and a job-chain

Figure 8.3: Different usage models on the MSA

8.3.1 Single module usage

Always use the same module: Examples for this usage model are NEST, or the GPU/FPGA Imager.

- NEST needs strong CPUs and cannot take advantage of GPGPU accelerators so it can either use the CM or DAM. Since NEST does not makes use of GPUs, FGPAs, or a huge amount of memory, the DAM nodes are somewhat over dimensioned. The CM is therefore the best suited for executing NEST.
- The GPGPU and FGPA Imagers used in radio astronomy need to run on the DAM. For the FPGA imager this is obvious since only the DAM nodes are equipped with FPGA accelerator. The GPU imager needs a huge amount of memory, thus running on the GPUs in the ESB nodes is not an option.

Used module depends on use case: This is the case in the single module version of GROMACS. The CM is used for small size problems, whereas the GPUs on the ESB are needed for the larger use cases. GROMACS could also use the GPUs on the DAM but the ESB is a better choice because the code does not need much memory and is scalable over many nodes.

Used module depends on module availability: Finally, there are applications, such as the CMS Reconstruction, which can run on all the modules. The CMS Reconstruction has a CPU version for the CM and a GPU version for ESB and DAM. Since the execution runs in all nodes independently it can just utilize any kind of nodes that are available at any given time.

8.3.2 Multi module usage

Job chain: An example for the “Job chain” model are the coupled versions of NEST plus Arbor (Figure 8.4) and NEST plus Elephant (Figure 8.5). NEST first runs on the CM (as explained in Section 8.3.1) and after that Arbor starts to work on the output from NEST using the GPUs of the ESB. Similarly, Elephant starts the data analysis on the output from NEST on the DAM.

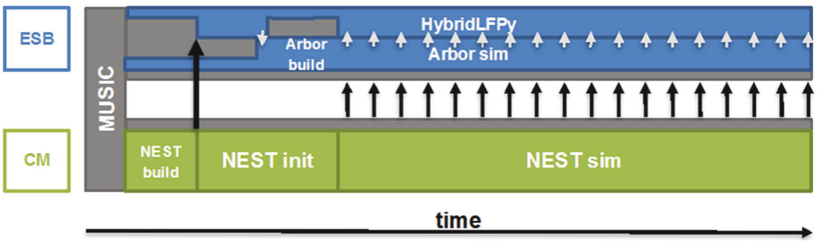


Figure 8.4: NEST plus Arbor workflow

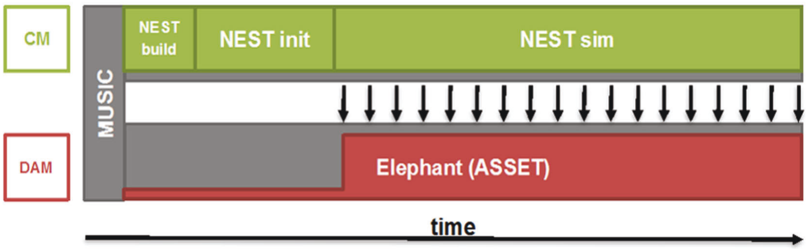


Figure 8.5: NEST plus Elephant workflow

Use multiple modules simultaneously: A good example for this fifth usage model is the xPic and GMM combination: xPic's particle solver runs on the GPUs of the ESB and its field solver runs on the CPUs of the CM, while GMM runs on the DAM. Particle and field solvers from xPic run simultaneously and exchange data during runtime. The data produced by the particle solver is analysed by GMM on the DAM nodes (Figure 8.6).

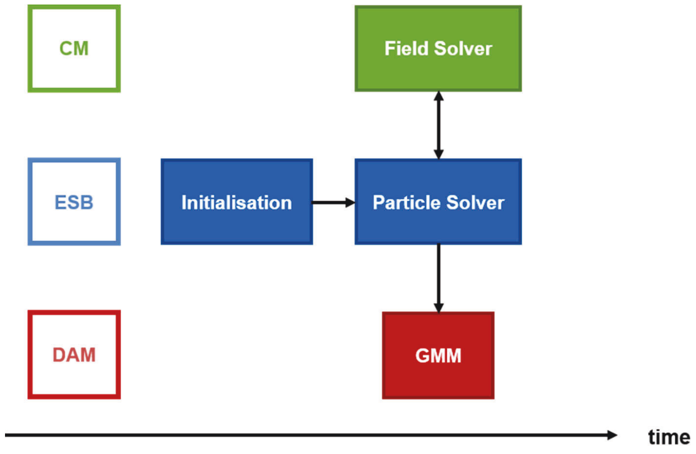


Figure 8.6: xPic plus GMM workflow

Another example is GROMACS in the offload version. It simultaneously uses the CM and the ESB within one job (Figure 8.7).

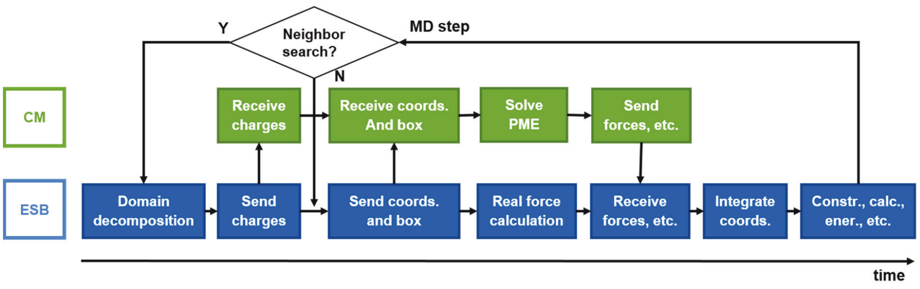


Figure 8.7: GROMACS workflow for the offload version

Combination of job chain and jobs running simultaneously: An example of this last usage model is the workflow of DLMOS (DAM) → xPic (CM+ESB) → GMM (DAM). The ML codes DLMOS and GMM will run on the DAM and between these two jobs xPic will run on CM+ESB (Figure 8.8).

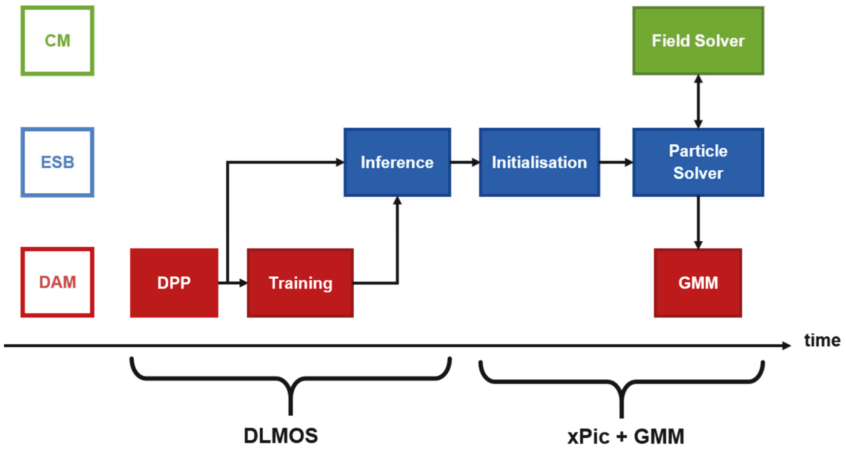


Figure 8.8: Workflow of DLMOS plus xPic plus GMM

8.4 Porting

The DEEP-EST system provides the GCC (8.3.0, 9.3.0, 10.2.0), Intel (2019.5.281) and NVHPC (20.9, experimental) compilers for C, C++ and Fortran. There are also different MPI versions available (ParaStationMPI, Intel MPI and OpenMPI), but it is recommended to use ParaStationMPI because it enables all the MSA features on the system. If the code needs specific software packages, it should be checked if they are provided on the DEEP-EST system. Detailed information on all available packages and the module environment used on the system can be found in the DEEP-EST Wiki[98]. Jobs can be submitted to the job queue for all compute modules (CM, ESB, DAM) via the Slurm resource manager.

[98] https://deeptrac.zam.kfa-juelich.de:8443/trac/wiki/Public/User_Guide/Information_on_software

8.4.1 The resource manager

Slurm supports both interactive and batch jobs (scripts submitted into the system). This is an example on how to allocate an interactive session on the CM (`-p dp-cn)` with 4 nodes (`-N 4`) and 2 tasks per node (`-n 8`) for 30 minutes (`-t 00:30:00`):

srun -p dp-cn -N 4 -n 8 -t 00:30:00 --pty /bin/bash –i

The following example shows a job script for submitting a batch job using the same parameters (number of nodes, runtime etc.) as before:

```
#!/bin/bash
#SBATCH --partition=dp-cn
#SBATCH -A deep
#SBATCH -N 4
#SBATCH -n 8
#SBATCH -o /Path/to/output
#SBATCH -e /Path/to/erroutput
#SBATCH --time=00:30:00
```

Figure 8.9: Job script example

For more details (all available partitions, `srun` and `sbatch` options and useful Slurm commands) refer to the batch system section in the DEEP-EST Wiki [99].

8.4.2 Code porting and optimisation on the CM

Porting the codes to the CM should be straightforward since the CM is equipped with standard, general purpose CPUs and every code targeting multi-core CPUs should work.

8.4.3 Code porting and optimisation on the ESB

The ESB is equipped with NVIDIA Tesla V100 GPUs. The code parts that were identified to be compute intensive and can be parallelized should be ported to the GPUs. If serial code parts are just included to manage the GPU computation, they do not need a high computing capacity, and can fit all application data into the 32 GB high-bandwidth GPGPU memory, using only ESB nodes with its comparatively weak CPUs is sufficient. If the serial code parts need stronger CPUs, the developer should strongly consider dividing the code onto CM (strong CPU) and ESB (GPU). The DAM would also be an option for strong CPU plus GPGPU runs, but because there are only 16 nodes, running on the CM plus ESB (see Section 8.5) makes more sense to scale the application.

[99] https://deeptrac.zam.kfa-juelich.de:8443/trac/wiki/Public/User_Guide/Batch_system

Porting code to the GPU can be done with different programming models. On the DEEP-EST system we support the following: CUDA, OpenACC, OmpSs, and OpenMP5.0. Below we give some introductory information on how to use these programming models. For more details we refer to the specific user guides and documentation, since detailed explanations of the programming models and their usage would go far beyond the scope of this document.

8.4.3.1 Using CUDA

Since the GPGPUs in the DEEP-EST system are NVIDIA GPGPUs, using CUDA is likely the way to get the maximum performance out of the code. However, it should be kept in mind that CUDA code is not the best option for non-NVIDIA GPUs. In addition, a lot of effort may be required to port an application to CUDA if one has to start from scratch. As an example, we will use a simple vector addition (see Figure 8.10).

First, the computations that should run on the GPGPU have to be turned into CUDA kernels. For this the `__global__` keyword has to be added to the affected functions. If the Host device needs the results from the GPU, it must be ensured that the host waits for the GPGPU to finish the calculations. For this the function `cudaDeviceSynchronize()` can be used.

In addition, one has to manage memory and potentially data placement. With the available Unified Memory, a memory space can be allocated and then be used by the CPU as well as by the GPU, so that the data does not need to be transferred manually anymore. To allocate and later free Unified Memory, two functions need to be called (as replacement for ‘`malloc`’ and ‘`free`’): `cudaMallocManaged(…)` and `cudaFree(…)`

Now the code is ready to run on the GPU, so finally the kernel can be launched with `vectoradd<<<x, y>>>(n, a, b)`.

```
#include <iostream>
#include <math.h>

// function to add the elements of two arrays
void add(int n, float *x, float *y)
{
  for (int i = 0; i < n; i++)
      y[i] = x[i] + y[i];
}

int main(void)
{
  int N = 1<<20; // 1M elements

  float *x = new float[N];
  float *y = new float[N];

  // initialize x and y arrays on the host
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  // Run kernel on 1M elements on the CPU
  add(N, x, y);

  // Check for errors (all values should be 3.0f)
  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, fabs(y[i]-3.0f));
  std::cout << "Max error: " << maxError << std::endl;

  // Free memory
  delete [] x;
  delete [] y;

  return 0;
}
```

Figure 8.10: C++ version of the vector addition

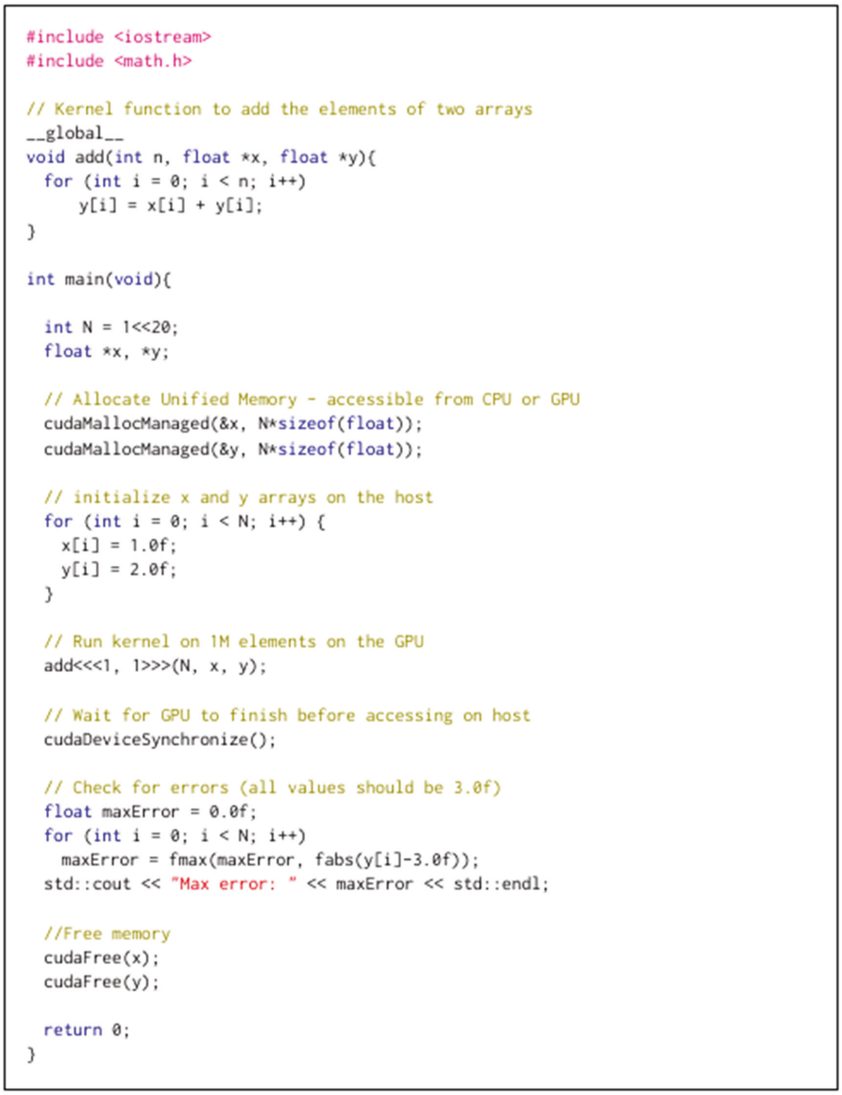
Figure 8.11 shows a code including the above changes. Although these changes make the code run on the GPU, there is plenty of room for optimization, so the basic code should then be analysed with a profiler, e.g. `nvprof`, to get an idea of what to optimize. There are plenty of tutorials, guides and courses on how to write and/or optimize CUDA codes: here are just a few examples[100] [101] [102].

[100] https://developer.nvidia.com/blog/even-easier-introduction-cuda/

[101] https://fz-juelich.de/SharedDocs/Termine/IAS/JSC/EN/courses/2020/ptc-gpu-cuda-2020.html

[102] https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html

It should be noted that for certain codes, manually managing the location of data objects (on host or GPU memory) can extract more performance than relying on the Unified Memory mechanisms; this is akin to cache optimizations on traditional CPU systems. For the ESB, it is critical to ensure that all application data objects are located in GPU memory.



```
#include <iostream>
#include <math.h>

// Kernel function to add the elements of two arrays
__global__
void add(int n, float *x, float *y){
  for (int i = 0; i < n; i++)
      y[i] = x[i] + y[i];
}

int main(void){

  int N = 1<<20;
  float *x, *y;

  // Allocate Unified Memory - accessible from CPU or GPU
  cudaMallocManaged(&x, N*sizeof(float));
  cudaMallocManaged(&y, N*sizeof(float));

  // initialize x and y arrays on the host
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  // Run kernel on 1M elements on the GPU
  add<<<1, 1>>>(N, x, y);

  // Wait for GPU to finish before accessing on host
  cudaDeviceSynchronize();

  // Check for errors (all values should be 3.0f)
  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = fmax(maxError, fabs(y[i]-3.0f));
  std::cout << "Max error: " << maxError << std::endl;

  //Free memory
  cudaFree(x);
  cudaFree(y);

  return 0;
}
```

Figure 8.11: Vector addition example in the CUDA version

8.4.3.1.1 The new Magma library

Within the DEEP-EST project, UoI faced the concern of having to significantly refactor their codebase, so that they could develop multiple application versions supporting both CPUs and GPUs. To mitigate this problem, they developed a library (called "Magma") that mimics the C++ standard library blueprint. The library takes care of linking the source code to either the C++ STL (in case of CPU) or CUDA Thrust (in case of GPGPU) libraries.

The Magma library is available to all as free-open-source-software via a public GitHub repository[103] and its functionality is detailed in Section 6.4 of this book.

```
magma::for_each(n_offset_size, v_coord_cell_size.size() - 1, [=]
#ifdef CUDA_ON
    __device__
#endif
    (auto const &i) -> void {
        it_coord_cell_size[i] = it_coord_cell_offset[i + 1] - it_coord_cell_offset[i];
    });
```

Figure 8.12: Example of a for-each loop with Magma

Figure 8.12 shows an example on how the Magma library is used in the NextDBSCAN application of UoI. The source code is identical for both the CPU and GPGPU platform with the exception of the necessary host and/or device annotations which CUDA requires to specify the execution target.

8.4.3.2 Using OpenACC

Another option to offload code to the GPUs is using OpenACC via the NVIDIA NVHPC compiler. OpenACC is a directive-based performance-portable parallel programming model. With OpenACC applications can be ported to a wide variety of heterogeneous HPC hardware platforms and architectures with significantly less programming effort than required for a low level model such as CUDA. Programming with OpenACC should happen in 4 steps:

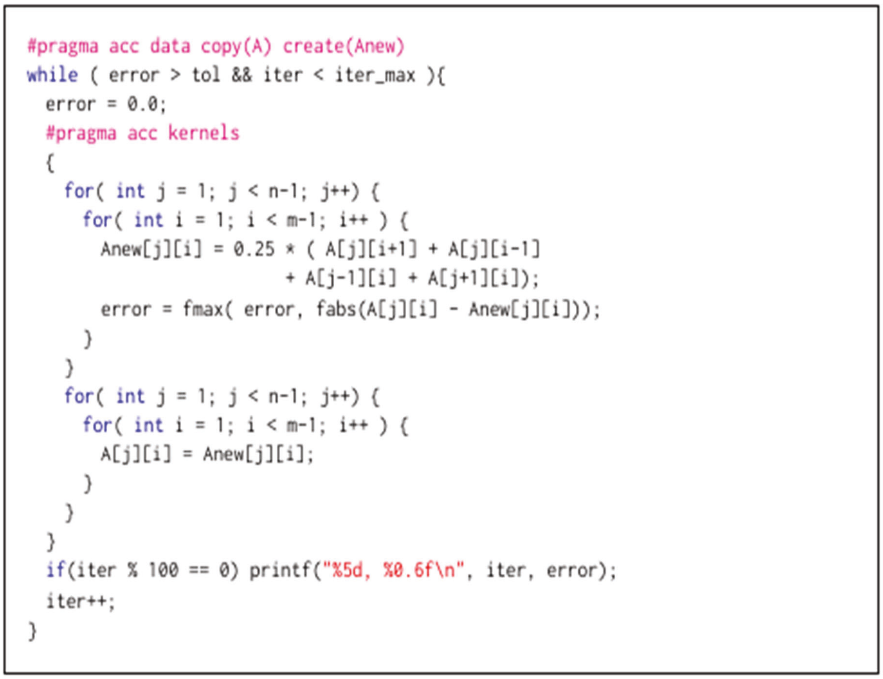
1. Identify parallelism (already done in Section 8.2)
2. Parallelize code parts with OpenACC
3. Express data locality
4. Optimize performance

After the analysis phase described in Section 8.2, it is known which code parts should be parallelized on the GPU. These code parts will be put within a pragma region as shown in Figure 8.13 for a small Jacobi iteration. The two nested inner loops (over `i`

[103] https://github.com/ernire/magma/tree/master

and `j`) can be parallelized. The `kernels` directive tells the compiler to analyse the code and look for parallel loops in the specified region. In this case, the compiler identifies two regions of code to generate an accelerator kernel. The compiler also analyses which arrays are used in the calculation and generates code to move `A` and `Anew` into GPU memory. The compiler even detects that it needs to perform a *max reduction* on the `error` variable.

The next step is to express the data locality. Sometimes not everything needs to be copied on and from the device. With the `data` pragma the relevant data locations can be specified. The `copy` clause in the `data` pragma tells the compiler that it should copy the `A` array to and from the device as it enters and exits the region, respectively. Since the `Anew` array is only used within the convergence loop, the `create` clause is used to request the compiler to create temporary space on the device, since we do not care about the initial or final values of that array.



```
#pragma acc data copy(A) create(Anew)
while ( error > tol && iter < iter_max ){
  error = 0.0;
  #pragma acc kernels
  {
    for( int j = 1; j < n-1; j++) {
      for( int i = 1; i < m-1; i++ ) {
        Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                            + A[j-1][i] + A[j+1][i]);
        error = fmax( error, fabs(A[j][i] - Anew[j][i]));
      }
    }
    for( int j = 1; j < n-1; j++) {
      for( int i = 1; i < m-1; i++ ) {
        A[j][i] = Anew[j][i];
      }
    }
  }
  if(iter % 100 == 0) printf("%5d, %0.6f\n", iter, error);
  iter++;
}
```

Figure 8.13: Jacobi example in the OpenACC version

In following references the reader can find more detailed guides and courses to get started with OpenACC[104],[105],[106].

For the ESB, data objects have to stay in GPGPU memory as long as possible, so the programmer should radically limit copying between CPU and GPGPU as long as it is not strictly necessary for code correctness.

8.4.3.3 Using OmpSs-2

The OmpSs-2 task-based programming model supports message-passing libraries (MPI) and improved GPU programming of the MSA.

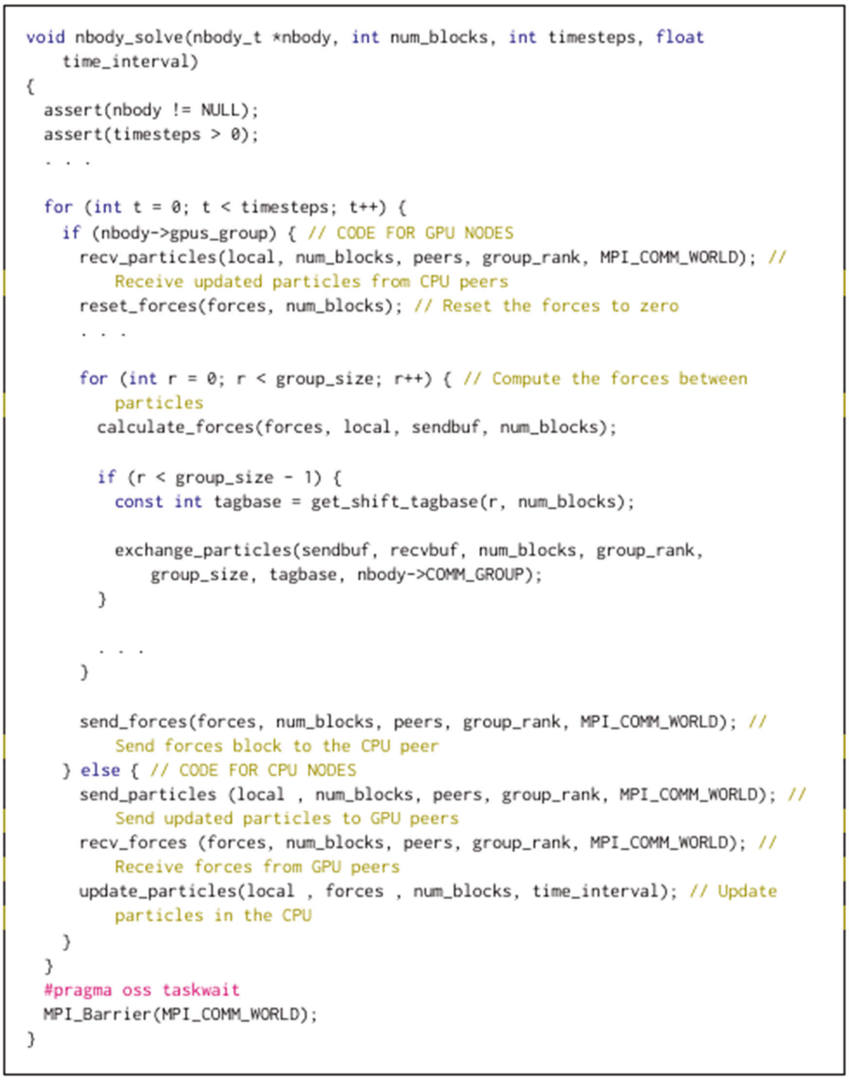
Herein we will cover the well-known N-Body benchmark, which numerically approximates the evolution of a system of bodies in which each body continuously interacts with every other body. A familiar example is an astrophysical simulation in which each body represents a galaxy or an individual star and all bodies attract each other through gravitational force.

In the benchmark presented here the particle space is divided into smaller blocks. Similarly, MPI processes are also divided into two groups: CPU processes and GPU processes. Firstly, GPU processes are responsible for computing the forces between each pair of blocks of particles; secondly, these forces are sent to CPU processes, where each process updates its blocks of particles using the received forces. The blocks of particles and forces are equally distributed amongst each MPI process within each group. Thus, each MPI process is in charge of computing the forces or updating the particles of a consecutive chunk of blocks.

[104] https://developer.nvidia.com/blog/getting-started-openacc/

[105]https://www.fz-juelich.de/SharedDocs/Termine/IAS/JSC/DE/Kurse/2020/ptc-gpu-openacc-2020.html?nn=2320772

[106] https://www.openacc.org/sites/default/files/inline-files/OpenACC_Programming_Guide_0.pdf

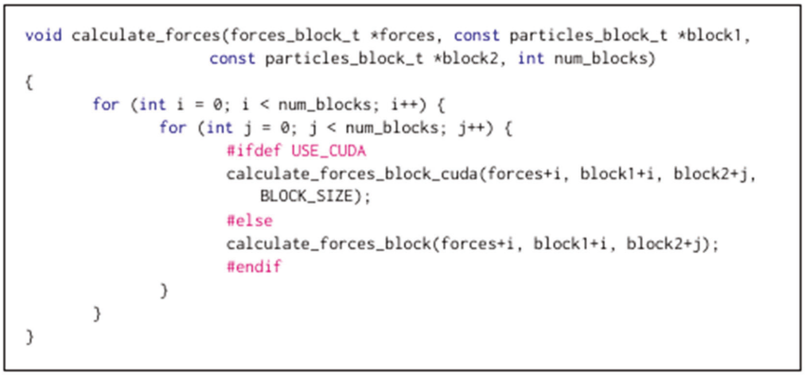


```
void nbody_solve(nbody_t *nbody, int num_blocks, int timesteps, float
    time_interval)
{
  assert(nbody != NULL);
  assert(timesteps > 0);
  . . .

  for (int t = 0; t < timesteps; t++) {
    if (nbody->gpus_group) { // CODE FOR GPU NODES
      recv_particles(local, num_blocks, peers, group_rank, MPI_COMM_WORLD); //
          Receive updated particles from CPU peers
      reset_forces(forces, num_blocks); // Reset the forces to zero
      . . .

      for (int r = 0; r < group_size; r++) { // Compute the forces between
          particles
        calculate_forces(forces, local, sendbuf, num_blocks);

        if (r < group_size - 1) {
          const int tagbase = get_shift_tagbase(r, num_blocks);

          exchange_particles(sendbuf, recvbuf, num_blocks, group_rank,
              group_size, tagbase, nbody->COMM_GROUP);
        }

        . . .
      }

      send_forces(forces, num_blocks, peers, group_rank, MPI_COMM_WORLD); //
          Send forces block to the CPU peer
    } else { // CODE FOR CPU NODES
      send_particles (local , num_blocks, peers, group_rank, MPI_COMM_WORLD); //
          Send updated particles to GPU peers
      recv_forces (forces, num_blocks, peers, group_rank, MPI_COMM_WORLD); //
          Receive forces from GPU peers
      update_particles(local , forces , num_blocks, time_interval); // Update
          particles in the CPU
    }
  }
  #pragma oss taskwait
  MPI_Barrier(MPI_COMM_WORLD);
}
```

Figure 8.14: NBody solver

The computation pattern in the code (Figure 8.14) is repeated during multiple time steps. The communication pattern during each time step consists of GPGPU processes, which exchange their particles with each other in a circular manner in order to compute the forces between their own particles against those from other GPGPUs. For the purpose of simplifying this pattern, this benchmark uses a different MPI communicator for the circular exchange. Once a GPU process finishes the computation of its forces, it sends the forces to the corresponding CPU process(es) and then it receives the updated particles. MPI sends/receives are performed separately on each block.

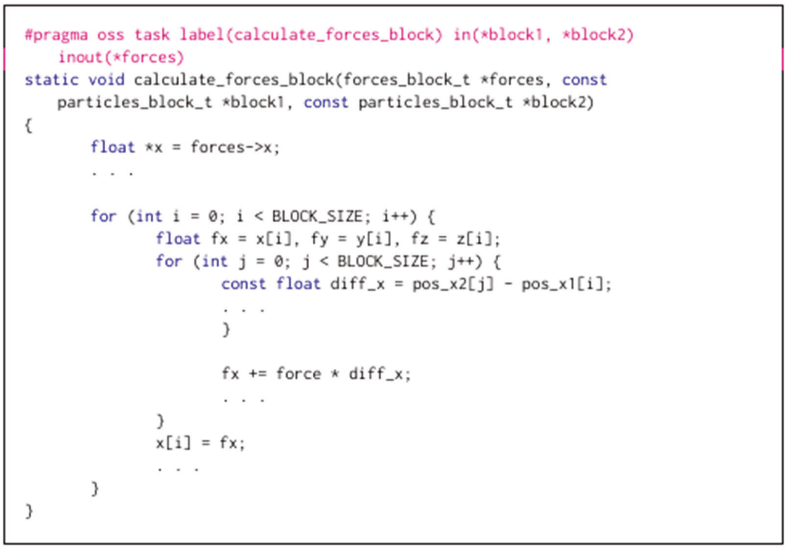
The actual GPGPU computation takes place within the function `calculate_forces`. A closer look to this function in Figure 8.15 reveals that the programmer has indeed the choice of using either the CPU or the GPU version of this function.



```
void calculate_forces(forces_block_t *forces, const particles_block_t *block1,
                      const particles_block_t *block2, int num_blocks)
{
        for (int i = 0; i < num_blocks; i++) {
                for (int j = 0; j < num_blocks; j++) {
                        #ifdef USE_CUDA
                        calculate_forces_block_cuda(forces+i, block1+i, block2+j,
                            BLOCK_SIZE);
                        #else
                        calculate_forces_block(forces+i, block1+i, block2+j);
                        #endif
                }
        }
}
```

Figure 8.15: Calculate_forces function

The code part in Figure 8.16 shows the CPU version of the kernel associated to the computation of forces inside a block. It is worth noting that the programmer is responsible for annotating this function with OmpSs-2 pragmas in order to convert this kernel into a regular task, to be later executed in parallel by the CPU.



```
#pragma oss task label(calculate_forces_block) in(*block1, *block2)
    inout(*forces)
static void calculate_forces_block(forces_block_t *forces, const
    particles_block_t *block1, const particles_block_t *block2)
{
        float *x = forces->x;
        . . .

        for (int i = 0; i < BLOCK_SIZE; i++) {
                float fx = x[i], fy = y[i], fz = z[i];
                for (int j = 0; j < BLOCK_SIZE; j++) {
                        const float diff_x = pos_x2[j] - pos_x1[i];
                        . . .
                        }

                        fx += force * diff_x;
                        . . .
                }
                x[i] = fx;
                . . .
        }
}
```

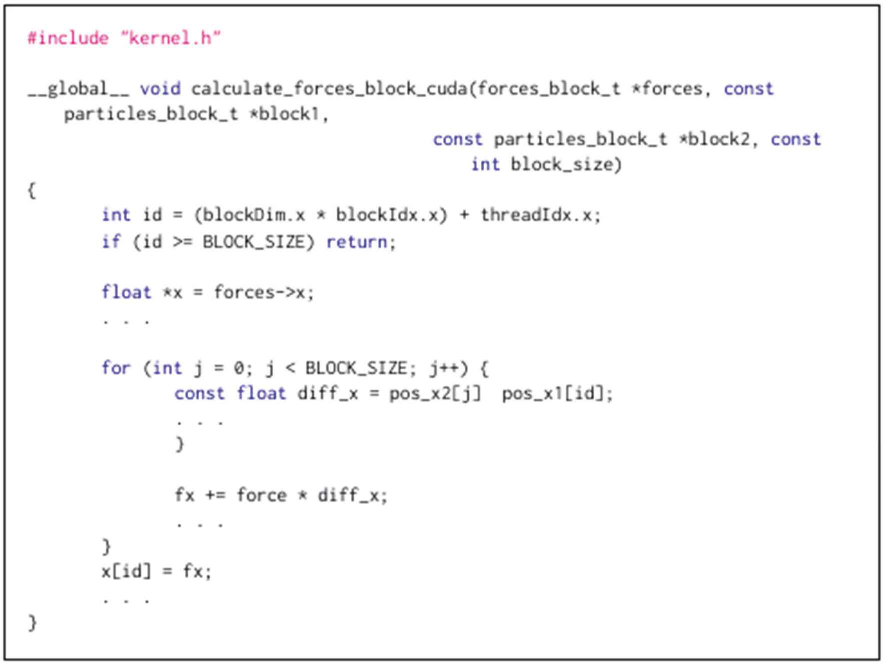
Figure 8.16: CPU kernel

More interesting is to see which modifications are now necessary to convert the previous, original CPU code into its equivalent GPU code and, at the same time, render it compatible with OmpSs-2. In Figure 8.17 we add the CUDA kernel declaration in the header file `kernel.h`. It is important to highlight that the programmer is responsible for annotating this CUDA kernel as if it were a regular (i.e., CPU) function that can later be invoked by the OmpSs-2 runtime. Note, for instance, that now it is necessary to indicate the clauses `device` and `ndrange`. It can be readily seen that this procedure eases the development of GPU programming and is rendered possible thanks to the OmpSs-2 runtime, which takes care of data movements and correct synchronization between the host (CPU) and device (GPU) tasks and kernels following a true data-flow execution model.

```
#include "nbody.h"

#pragma oss task label(calculate_forces_block) in(*block1, *block2)
    inout(*forces) \
                device(cuda) ndrange(1, block_size, 128)
__global__ void calculate_forces_block_cuda(forces_block_t *forces, const
    particles_block_t *block1,
                                        const particles_block_t *block2, const
                                            int block_size);
```

Figure 8.17: CUDA header file defining OmpSs-2 tasks for GPU

Finally, the code in Figure 8.18 shows the `calculate_force_block_cuda` CUDA C kernel from the N-Body application. This kernel is almost identical to the CPU kernel illustrated in Figure 8.16. It is important to point out that the CUDA kernel code is located in a different file that is separately compiled by the CUDA C compiler. For completeness, the definition of the `forces_block_t` has been added to highlight that it is a `struct` of static arrays, thus suitable for host–device data movement. Data movement makes use of the CUDA unified memory. The OmpSs-2 runtime has been extended to explicitly manage data transfers, so that unified memory is no longer a hard requirement.



```
#include "kernel.h"

__global__ void calculate_forces_block_cuda(forces_block_t *forces, const
   particles_block_t *block1,
                                    const particles_block_t *block2, const
                                       int block_size)
{
       int id = (blockDim.x * blockIdx.x) + threadIdx.x;
       if (id >= BLOCK_SIZE) return;

       float *x = forces->x;
       . . .

       for (int j = 0; j < BLOCK_SIZE; j++) {
              const float diff_x = pos_x2[j]  pos_x1[id];
              . . .
              }

              fx += force * diff_x;
              . . .
       }
       x[id] = fx;
       . . .
}
```

Figure 8.18: CUDA kernel

8.4.3.4 Using OpenMP 5.0

GPGPU (or more generally, accelerator) offloading was introduced into the OpenMP standard with version 4.5 and enhanced functionality is provided with OpenMP 5.0. Like OpenACC and OmpSs, OpenMP relies on using directives and supports Fortran, C and C++. The fork-join model used by OpenMP 5.0 is similar to OpenACC, but OpenACC is more descriptive and OpenMP 5.0 is more prescriptive.

This section shows multiple steps to transform a basic "*SAXPY*" code into a fully functional OpenMP 5.0 application. It covers the most common and useful approaches to offload a computationally intensive loop to the GPU accelerators in the ESB and DAM modules of the DEEP-EST system.

The basic code contains two `for` loops, one for the initialization of the values and a second for the main operations. In this example we have wrapped the main `for` loop in a function in order to show how such externally defined methods can be called from within OpenMP sections. All vectors are allocated dynamically.

Porting the SAXPY code to the GPU using OpenMP 4.5/5.0 requires the addition of only one line of code (see Figure 8.19). This new line performs the memory transfers between Host and Device and divides the computation of the for loop among the different threads in the GPU.

```
void saxpy(long long int n, float s, float* x, float* y, float* z){
  #pragma omp target teams distribute parallel for map(to:x[:n],y[:n])
      map(from:z[:n])
  for (auto i=0; i<n; i++)
    z[i] = s*x[i] + y[i];
}
int main(){

  long long int n = 1000000000;
  int s = 2.0;
  float* x = new float[n];
  float* y = new float[n];
  float* z = new float[n];
  for (auto i=0; i<n; i++){
    x[i] = i;
    y[i] = i%3;
  }
  saxpy(n, s, x, y, z);

  free(x);
  free(y);
  free(z);
}
```

Figure 8.19: Example for OpenMP 5.0 offload

Here we explain each one of the terms in the pragma call:

- `#pragma omp`: signals to the compiler that the following code section will be processed by OpenMP.
- `target`: tells the compiler that the following section of code will be executed on the GPU. This is equivalent to the definition of a kernel function around the for loop as shown in the CUDA code in Figure 8.11.
- `teams`: instructs the main thread in the Device to spawn multiple, isolated, threads associated with the different processor blocks (SMs) in the GPU.
- `distribute`: instructs the GPU to decompose the loop iterations and assign different chunks to the different teams requested.
- `parallel`: instructs each team master thread to spawn a group of threads for each team.
- `for`: distribute the loop iterations in each teams' chunk across the threads in the team.
- `map(to:…)`: perform a data transfer of the listed vectors from the Host to the Device on entering the OpenMP section.

- `map(from:…)`: perform a data transfer of the listed vectors from the Device to the Hoist on exiting the OpenMP section.

There are several metadirectives, declare target, macro defined directives, and device allocations, which will be explained below.

8.4.3.4.1 Declare target

In the previous section one single `for` loop was offloaded to the accelerator, but in most useful cases the programmer wants to offload more complex code, usually encapsulated in functions (or kernels). Functions that can be called from within an accelerated `target` region must be defined by opening and closing `declare target` pragmas, as shown in Figure 8.20.

```
#pragma omp declare target
void saxpy(long long int n, float s, float* x, float* y, float* z){
  #pragma omp distribute parallel for
  for (auto i=0; i<n; i++)
    z[i] = s*x[i] + y[i];
}
#pragma omp end declare target
int main(){

  long long int n = 1000000000;
  int s = 2.0;
  float* x = new float[n];
  float* y = new float[n];
  float* z = new float[n];
  for (auto i=0; i<n; i++){
    x[i] = i;
    y[i] = i%3;
  }
  #pragma omp target teams map(to:x[:n],y[:n]) map(from:z[:n])
  saxpy(n, s, x, y, z);
  free(x);
  free(y);
  free(z);
}
```

Figure 8.20: OpenMP 5.0 offload: Declare target example

With this change, it is possible to offload the `saxpy` function to the accelerator in any location of the code. The function must only be called from within a `target` region (in the snippet above the scope of the target region contains only one line). The offloading line used in Figure 8.19 has been divided here in two parts: 1) the `target teams` pragma that spawns a set of master teams in the accelerator and performs all memory transfers to the device, and 2) the `distribute parallel for` pragma, called from within the accelerator in the `saxpy` function, that segments the `for` loop and distributes it among the teams and the corresponding threads.

8.4.3.4.2 Declare target + declare variant

```
void gpu_saxpy(long long int, float, float*, float*, float*);

#pragma omp declare variant (gpu_saxpy) match(construct={target})
void saxpy(long long int n, float s, float* x, float* y, float* z){
  #pragma omp for simd
  for (auto i=0; i<n; i++)
    z[i] = s*x[i] + y[i];
}
#pragma omp declare target
void gpu_saxpy(long long int n, float s, float* x, float* y, float* z){
  #pragma omp distribute parallel for
  for (auto i=0; i<n; i++)
    z[i] = s*x[i] + y[i];
}
#pragma omp end declare target
int main(){

  long long int n = 1000000000;
  int s = 2.0;
  float* x = new float[n];
  float* y = new float[n];
  float* z = new float[n];
  for (auto i=0; i<n; i++){
    x[i] = i;
    y[i] = i%3;
  }
  #pragma omp parallel
  saxpy(n, s, x, y, z);
  #pragma omp target teams map(to:x[:n],y[:n]) map(from:z[:n])
  saxpy(n, s, x, y, z);
  free(x); free(x);
  free(y);
  free(z);
}
```

Figure 8.21: OpenMP 5.0 offload: Declare target + declare variant example

If the programmer wants to use the `saxpy` function both in the host (CPU) and the device (accelerator), it is possible to create alternative kernels of the function with their respective OpenMP pragmas. Figure 8.21 shows that in the main code the function is called, first in the Host (without `target` pragma) and once in the Device (inside a `target` pragma). Two different versions of the routine are activated for each case. The `declare variant` call instructs the compiler to look for an alternative version of the code following the `match` conditions. In this case, if the function is called within a target region, the variant `gpu_saxpy` function is called.

This division of work is interesting for applications that want to perform the same procedure both on the Host and on the Device. This could allow workload balancing between CPU and Accelerator, maximizing the use of the available computational resources.

8.4.3.4.3 Macros

```
void saxpy(long long int n, float s, float* x, float* y, float* z){
  #ifdef __GPU__
  #pragma omp target map(to:x[:n],y[:n]) map(from:z[:n])
  #endif
  {
    #ifdef __GPU__
    #pragma omp teams distribute parallel for
    #else
    #pragma omp parallel for simd
    #endif
    for (auto i=0; i<n; i++){
      z[i] = s*x[i] + y[i];
    }
  }
}
```

Figure 8.22: OpenMP 5.0 offload: Example for Macro usage

The problem with the use of the `declare variant` clause is that important parts of the code need to be duplicated. This is a potential source of bugs and can complicate its maintenance. To avoid code duplication the OpenMP pragmas can be surrounded by macros defined by the user. In Figure 8.22 the Host and Device versions of the `saxpy` function have been separated by the use of the compile-time variable `__GPU__`. At compile-time it is possible to generate one version with offloading or a different version without offloading. This approach also allows the inclusion of details of the architecture. For example, the programmer can define the flags `__INTELCPU__`, `__AMDCPU__`, `__AMDGPU__`, `__NVIDIAGPU__`, corresponding to the four most common hardware architectures today. Each one will encompass a different OpenMP pragma line before the for loop in the `saxpy` code.

8.4.3.4.4 Metadirectives

```
void saxpy(long long int n, float s, float* x, float* y, float* z){
  #pragma omp target map(to:x[:n],y[:n]) map(from:z[:n]) device(dvc)
  {
    #pragma omp metadirective when (device={arch("nvptx")}: teams distribute
        parallel for) default (parallel for simd)
    for (auto i=0; i<n; i++){
      z[i] = s*x[i] + y[i];
    }
  }
}
```

Figure 8.23: OpenMP 5.0 offload: Metadirectives example

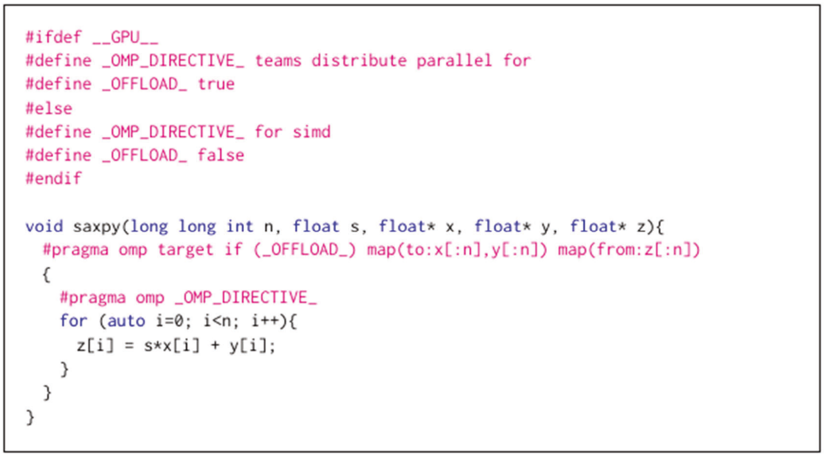
The previous approach is very useful but can become cumbersome and it almost feels like OpenMP should support such a use case scenario. The OpenMP 5.0 standard does provide an alternative called *metadirectives*. Instead of using compilation macros the `metadirective` is built using the following schema:

```
#pragma omp metadirective \
        when (<condition> : teams distribute parallel for) \
        default (parallel for simd)
```

This structure allows the programmer to get rid of macro definitions and uses `<conditions>` to choose one OpenMP line instead of the `default` OpenMP line. In Figure 8.23 the condition selects the outcome of the `metadirective` based on the type of hardware architecture in which the loop is running.

This is a very handy option but presents two drawbacks: 1) it currently allows only two options, the one selected by the `<condition>` and the `default`, and 2) it is not currently supported by most compilers, including LLVM (March 2021).

8.4.3.4.5 Macro defined directives



```
#ifdef __GPU__
#define _OMP_DIRECTIVE_ teams distribute parallel for
#define _OFFLOAD_ true
#else
#define _OMP_DIRECTIVE_ for simd
#define _OFFLOAD_ false
#endif

void saxpy(long long int n, float s, float* x, float* y, float* z){
  #pragma omp target if (_OFFLOAD_) map(to:x[:n],y[:n]) map(from:z[:n])
  {
    #pragma omp _OMP_DIRECTIVE_
    for (auto i=0; i<n; i++){
      z[i] = s*x[i] + y[i];
    }
  }
}
```

Figure 8.24: OpenMP 5.0 offload: macro defined directives example

A workaround to avoid code duplication and simplify its main structure, without using metadirectives, is to define the OpenMP lines with a global macro that can then be referenced inside the code as shown in Figure 8.24. This approach makes the code much cleaner but requires the programmer to specify all the possible OpenMP calls at the beginning of the code. This could lead to a large number of macros that can be included in a separate file. Although this approach can complicate the maintenance of the code, the final result is much cleaner and easier to follow for non-experts in OpenMP.

8.4.3.4.6 Device allocation

```
#include <omp.h>

#ifdef __GPU__
#define _OMP_DIRECTIVE_ teams distribute parallel for
#else
#define _OMP_DIRECTIVE_ parallel for simd
#endif
void init(long long int n, float* x, float* y, int ndvc, int dvc){
  #pragma omp target if (ndvc>0) is_device_ptr(x,y) device(dvc)
  {
    #pragma omp _OMP_DIRECTIVE_
    for (auto i=0; i<n; i++){
      x[i] = i;
      y[i] = i%3;
    }
  }
}
void saxpy(long long int n, float s, float* x, float* y, float* z, int ndvc,
    int dvc){
  #pragma omp target if (ndvc>0) map(from:z[:n]) is_device_ptr(x,y) device(dvc)
  {
    #pragma omp _OMP_DIRECTIVE_
    for (auto i=0; i<n; i++){
      z[i] = s*x[i] + y[i];
    }
  }
}
int main(){
  long long int n = 1000000000;
  int s = 2.0;
  float* x;
  float* y;
  float* z;
  int ndvc = omp_get_num_devices();
  int dvc = omp_get_default_device();
  int hst = omp_get_initial_device();
  x = (float*) omp_target_alloc(sizeof(float) * n, dvc);
  y = (float*) omp_target_alloc(sizeof(float) * n, dvc);
  z = (float*) malloc(sizeof(float) * n);
  init(n, x, y, ndvc, dvc);
  saxpy(n, s, x, y, z, ndv, dvc);
  omp_target_free(x, dvc);
  omp_target_free(y, dvc);
  free(z);
}
```

Figure 8.25: OpenMP 5.0 offload: Device allocation example

One of the most important optimizations in OpenMP, and in general for any code that uses offloading, is to minimize the transfers of information between the Host and the Device. Up until now we have used the `map(…)` clause. This performs a memory transfer between the CPU and the Accelerator. To avoid such transfer, the programmer can allocate memory directly on the Accelerator with the API functions `omp_target_alloc()` and `omp_target_free()`. These two functions work in almost the same way as C/C++ `malloc()` and `free()` functions, but require also the number of the target accelerator device. The memory allocation function returns a pointer that is associated with memory in the accelerator. Any access to this pointer from code outside a target region will produce a memory access error.

In the initialization and in the `saxpy` functions shown in Figure 8.25, the pointer corresponding to the dynamically allocated accelerator memory is identified by the clause `is_device_ptr(…)`. The allocation functions must be called at any point outside the target region, but the pointers must only be referenced inside them.

In this snippet we show how the `saxpy` function receives the addresses of the two dynamically allocated accelerator vectors and returns the result by memory transfer to the Host device using the `map(from:…)` clause. This version of the `saxpy` test results in the best performance. It is also the cleanest version and the easiest to maintain. We recommend other programmers to understand the sections above, but to use the pattern presented in this section as a starting point of their code porting procedure.

8.4.4 Code porting and optimisation on the DAM

The DAM nodes are equipped with two different kinds of accelerators: NVIDIA Tesla V100 GPUs and Intel Stratix10 FPGAs. Section 8.4.3 already covers porting to GPUs. This section will have a look at the FPGAs.

8.4.4.1 oneAPI

Intel oneAPI[107] is an open, unified programming model. It is used to simplify programming across CPUs, GPUs, FPGAs and other accelerators. On the DEEP-EST system oneAPI is interesting for either working on FPGAs or CPUs. Information on how to work with it on FPGAs can be found here[108]. Section 7.4 of this book explains how to use it for GPU portable programming.

[107] https://software.intel.com/content/www/us/en/develop/tools/oneapi.html

[108] https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/fpga.html

8.4.4.2 OpenCL

OpenCL is an industry standard for programming systems that contain several heterogeneous devices and memory spaces. Like CUDA, the standard uses a kernel language to specify optimized code parts that run on accelerators like GPGPUs or FPGAs, an API to define and direct code parts to be run on a specific device, and an API to manage the (usually disjoint) memory spaces of devices and transfer data between them. OpenCL is used in non-HPC applications, such as heterogeneous embedded or mobile systems, and it has emerged as the method of choice to program FPGAs if the significant additional effort to develop RTL or VHDL code is seen as not worth the potential performance gain.

OpenCL for Intel CPUs and the FPGAs of the DAM module is provided by the Intel® FPGA SDK for OpenCL™ [109], which is currently in version 20.4, complemented by a BSP (board support package) matching the installed Stratix 10 devices. Figure 8.26 shows an example of an OpenCL kernel to compute and print out the Fibonacci numbers on the FPGA. A very detailed programming guide with information on how to build and optimize your OpenCL kernels, how to adapt your host program, and how to compile the code, can be found here[110].

```
__kernel void print_fibonacci_number(int n)
{
  int2 history = (int2) (1, 1);

  for (int i = 0; i < n; i ++)
    history = (int2) (history.y, history.x + history.y);

  printf("fib(%d) = %d\n", n, history.x);
}
```

Figure 8.26: Fibonacci OpenCL kernel

OpenCL is also supported on a range of GPUs, including the NVIDIA Tesla V100.

8.4.5 Data Analytics & Machine Learning frameworks

The DEEP-EST system also provides specific frameworks targeting Data Analytics and Machine Learning applications:

[109] https://www.intel.de/content/www/de/de/programmable/products/design-software/embedded-software-developers/opencl/support.html

[110] https://www.intel.com/content/www/us/en/programmable/documentation/mwh1391807965224.html

- TensorFlow: an end-to-end platform that makes it easy for developers to build and deploy ML models.[111] On the DEEP-EST system TensorFlow versions 2.2 and 1.13.1 based on Python 3.6.8 are deployed.
- PyTorch: a Python package that provides two high-level features: Tensor computation (like NumPy) with GPU acceleration and Deep Neural Networks built on a tape-based autograd system[112]. On the DEEP-EST system PyTorch versions 1.1.0 and 1.4.0 based on GCC are deployed.
- Horovod: a distributed deep learning training framework for TensorFlow, Keras, PyTorch, etc.[113]. On the DEEP-EST system Horovod version 0.16.2 based on GCC and ParaStationMPI is deployed.

These frameworks can be used on either CPUs or GPUs, so in theory they can run on all three compute modules. But since for data analytics (in most cases) a large amount of memory is necessary, the DAM would in general be the best suited module.

For trained networks there is the option of generating an interoperable ONNX[114] version which can be used for inference on many platforms including accelerators, which do not support the full-blown neural network development platforms listed above. This is a potential migration path to the FPGA accelerators of the DAM nodes, should users be interested in performing inference there.

8.5 Use of multiple modules

To run an application on multiple modules, it has to be partitioned: the code parts optimized for the different modules need to be separated and communication between the different parts has to be coded (preferably using MPI or files). As shown in Figure 8.3 there are three ways of using multiple modules: running on multiple modules at the same time (multi-module jobs), running consecutively on different modules (job chains and workflows), or a combination of both.

There might be jobs that need more than one module either at the same time or consecutively. In both cases one has to first divide the code in the parts for each module, and then make sure that both parts can communicate if necessary (either with MPI or through the file system).

[111] https://www.tensorflow.org/tutorials?hl=en

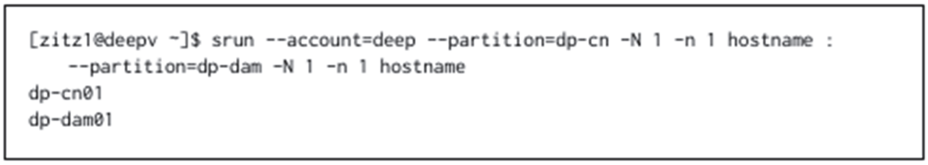
[112] https://github.com/pytorch/pytorch; the term refers to reverse-mode automatic differentiation.

[113] https://github.com/horovod/horovod#usage

[114] https://onnx.ai/, term refers to Open Neural Network Exchange

8.5.1 Running on Multiple Modules at the Same Time – Multi-Module Jobs

The Slurm resource manager supports allocating heterogeneous jobs (using more than one module). Figure 8.27 shows an example how to allocate one node on the CM and one node on the DAM and executing the `hostname` command on both.



```
[zitz1@deepv ~]$ srun --account=deep --partition=dp-cn -N 1 -n 1 hostname :
    --partition=dp-dam -N 1 -n 1 hostname
dp-cn01
dp-dam01
```

Figure 8.27: `Srun` command to allocate a heterogeneous job

Heterogeneous jobs can also be launched in a batch script using the `packjob` keyword. For information on functionalities regarding heterogeneous jobs in Slurm please see the DEEP-EST Wiki[115].

8.5.1.1 Using MPI

After an MPI application has started its processes as shown above, they can determine their module affiliation and thus coordinate their work accordingly. For this purpose, the processes can query on which module they are currently running by looking it up as a *Module ID* in the `MPI_INFO_ENV` object, which is provided by the MPI standard for environmental adaptations (see Figure 8.28).

This assignment between ID and modules is not fixed, but can be set by the user according to the needs of the application by using the environment variable `PSP_MSA_MODULE_ID`. However, if the user does not set such a module affiliation, the assignment of the IDs is performed automatically according to the order of the modules in the `srun` call: the first module gets ID 0, the second module ID 1, and so forth. Hence, it is the user's responsibility to match the respective `srun` call with an appropriate evaluation of the queried module IDs at application level.

[115] https://deeptrac.zam.kfa-juelich.de:8443/trac/wiki/Public/User_Guide/Batch_system#HeterogeneousjobswithMPIcommunicationacrossmodules

```
#include <mpi.h>
int main(int argc, char* argv[])
{
int world_rank, world_size, flag;
    char value[MPI_MAX_INFO_VAL];
    MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    MPI_Info_get(MPI_INFO_ENV, "msa_module_id", MPI_MAX_INFO_VAL, value, &flag);
    if (flag) { /* This MPI environment is modularity-aware! */
        printf("(%d) \"msa_module_id\" found. ID is %s\n", world_rank, value);
    } else {
        printf("(%d) Found no entry for \"msa_module_id\"\n", world_rank);
    }
    MPI_Finalize();
    return 0;
}
```

Figure 8.28: MPI standard

8.5.1.2 Topology-aware MPI Communicator Creation

In addition to querying explicitly for the module affiliation, it is possible to split the `MPI_COMM_WORLD` communicator into sub-communicators reflecting the module affinity of processes by using the new communicator spilt type `MPIX_COMM_TYPE_MODULE`. However, please note that this split type is an extension in ParaStationMPI and that it is hence *not* part of the official MPI standard! One may use the macro `MPIX_TOPOLOGY_AWARENESS` to test whether this feature is available or not:

```
#if defined(MPIX_TOPOLOGY_AWARENESS) && MPIX_TOPOLOGY_AWARENESS
/* MPIX_COMM_TYPE_MODULE available as split type: */
MPI_Comm_split_type(MPI_COMM_WORLD, MPIX_COMM_TYPE_MODULE, 0,
MPI_INFO_NULL, &split_comm);
#else
split_comm = MPI_COMM_WORLD;
/* MPIX_COMM_TYPE_MODULE _not_ available... */
#endif
```

Figure 8.29: MPI_Comm_split_type

Please also note that to use these extensions, the so-called *Topology Awareness* of ParaStationMPI must be enabled, which has to be done at compile time of the MPI library by using the configure switch `--with-topology-awareness`, plus explicitly setting the environment variable `PSP_MSA_AWARENSS=1` for the MPI sessions.

8.5.1.3 Using MSA-aware Patterns for Collectives

When topology awareness is enabled for ParaStationMPI, locality information as well as module affiliations can be taken into account by collective MPI operations for choosing optimized communication patterns, for example, for global reduction algorithms. In doing so, the locality awareness can be two-fold: (1) with respect to intra-node vs. inter-node communication (*SMP awareness*), and (2) with respect to inter-module vs. intra-module communication (*MSA awareness*). The following environment variables can be used for enabling these different degrees of topology awareness:

- `PSP_SMP_AWARENESS=1` – Generally, take locality information into account, e.g. for a meaningful use of `MPI_Win_allocate_shared`. This feature is enabled by default.
- `PSP_MSA_AWARENESS=1` – Generally activate the consideration of modular topologies. This feature is *not* enabled by default (see also Section 8.5.1.2).
- `PSP_SMP_AWARE_COLLOPS=1` – Enable the use of MPICH's SMP-aware collectives. This feature is disabled by default and requires SMP awareness in general (see above).
- `PSP_MSA_AWARE_COLLOPS=0|1|2` – Select the feature level for MSA-aware collectives:
    - `0` – Disable MSA awareness for collective MPI operations.
    - `1` – Enable MSA awareness for collective MPI operations. This feature is enabled by default if `PSP_MSA_AWARENESS=1` is set.
    - `2` – Apply MSA awareness *recursively* in multi-level topologies. For MSA plus SMP awareness, this requires that also `PSP_SMP_AWARENESS=1` is enabled.

The benefits of these different feature levels will depend on the patterns and settings of the applications. Therefore, at this point the user is advised to test the different options and check for which setting the application achieves the best performance. Moreover, it has to be emphasized that only a suitable subset of the MPI collectives actually do provide topology awareness. These are: `MPI_Barrier`, `MPI_Bcast`, `MPI_Reduce`, `MPI_Allreduce` and `MPI_Scan`, as well as their respective non-blocking counterparts.

8.5.1.4 Realizing Workflows on MPI Level

To pass data between workflow steps, the DEEP-EST project supports different approaches – for instance, using fast local storage, and/or using the global parallel file system. In this subsection, a further approach will briefly be introduced: the use of the standardized `MPI_Comm_connect/accept` API for passing data directly via MPI messages between workflow steps. According to this approach, the preceding step of a workflow application opens a so-called *port* and forwards this port information to the subsequent step, which then in turn can connect to it so that both MPI sessions can communicate directly via an inter-communicator. A good approach for passing the port information is the use of a small file, where the preceding workflow step puts the port name when the end of this phase is reached. The next workflow step can wait for this file to be created and then connect to receive the data directly via MPI communication, which avoids the considerable overhead of storing and retrieving volume data via a storage device. The two functions in Figure 8.30 show draft code for realizing this between two steps of a workflow.

```
int comm_accept(void){
/* Predecessor step in workflow: */
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) { /* = root */
    char port_name[MPI_MAX_PORT_NAME];
    std::ofstream portfile;
        MPI_Open_port(MPI_INFO_NULL, port_name);
portfile.open("portfile.txt");
portfile << port_name;
portfile.close();
}
MPI_Comm intercomm;
int remote_size;
MPI_Comm_accept(port_name, MPI_INFO_NULL, 0 /*= root */, MPI_COMM_WORLD,
          &intercomm);
    MPI_Comm_remote_size(intercomm, &remote_size);
/* Pass data to successor step in workflow: */
MPI_Send(..., intercomm);
/* ... */
}

int comm_connect(){
/* Successor step in workflow: */
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0) { /* = root */
        char port_name[MPI_MAX_PORT_NAME];
        std::ifstream portfile;

        do {
            portfile.open("portfile.txt");
            sleep(1);
} while (!portfile.is_open());
getline(portfile, line);
strncpy(port_name, line.c_str(), line.size()+1);
portfile.close();
}

MPI_Comm_connect(port_name, MPI_INFO_NULL, 0 /* = root */, MPI_COMM_WORLD,
          &intercomm);
    MPI_Comm_remote_size(intercomm, &remote_size);
    /* Receive data from predecessor step in workflow: */
    MPI_Recv(..., intercomm, MPI_STATUS_IGNORE);
/* ... */
}
```

Figure 8.30: MPI_comm_accept and MPI_comm_connect

How such steps of a workflow can be orchestrated at job level is described in the following sections.

8.5.2 *Running on Multiple Modules Consecutively – Workflows*

There are two ways of running jobs consecutively on the system: Using the `--delay` switch (where the jobs can have an overlap, e.g., for data exchange via MPI) or using Slurm job dependencies (where jobs start one after another).

8.5.2.1 `--delay` switch

The Slurm version running on the DEEP-EST system allows overlapping jobs inside a workflow: with the `--delay n` option the start of jobs in a job pack can be delayed by `n` minutes from the start of the first job of the job pack. Figure 8.31 shows a small example.

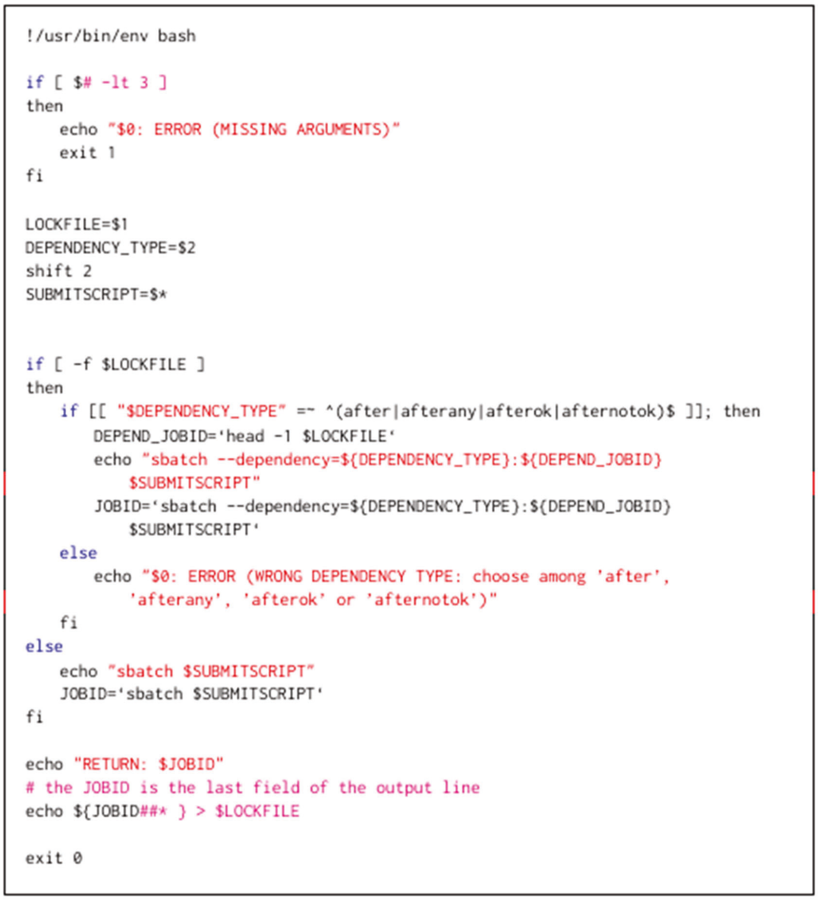
After submission of this job pack, Slurm divides it into separate jobs, and ensures that the delay is respected by using reservations, rather than the usual scheduler. Using this approach, the user has to estimate the duration of each sub-job to make a good choice of the interval that the jobs will be delayed. As the user-provided delay values tend to be not so accurate, we also provide API calls that a job can use to request Slurm to change the start times of the remaining jobs in the workflow it belongs to.

```
#!/bin/bash
#SBATCH -p sdv -N2 -t3
#SBATCH packjob
#SBATCH -p sdv -N1 -t3 --delay 2
srun hostname
```

Figure 8.31: Example for --delay switch

8.5.2.2 Slurm job dependencies

With this approach the jobs will not have a guaranteed overlap, yet will still run in a specified sequence. Using the Slurm dependencies, jobs can be chained with the following script:



```
!/usr/bin/env bash

if [ $# -lt 3 ]
then
   echo "$0: ERROR (MISSING ARGUMENTS)"
   exit 1
fi

LOCKFILE=$1
DEPENDENCY_TYPE=$2
shift 2
SUBMITSCRIPT=$*


if [ -f $LOCKFILE ]
then
   if [[ "$DEPENDENCY_TYPE" =~ ^(after|afterany|afterok|afternotok)$ ]]; then
      DEPEND_JOBID=‘head -1 $LOCKFILE‘
      echo "sbatch --dependency=${DEPENDENCY_TYPE}:${DEPEND_JOBID}
         $SUBMITSCRIPT"
      JOBID=‘sbatch --dependency=${DEPENDENCY_TYPE}:${DEPEND_JOBID}
         $SUBMITSCRIPT‘
   else
      echo "$0: ERROR (WRONG DEPENDENCY TYPE: choose among 'after',
         'afterany', 'afterok' or 'afternotok')"
   fi
else
   echo "sbatch $SUBMITSCRIPT"
   JOBID=‘sbatch $SUBMITSCRIPT‘
fi

echo "RETURN: $JOBID"
# the JOBID is the last field of the output line
echo ${JOBID##* } > $LOCKFILE

exit 0
```

Figure 8.32: Script for job chains

Job scripts can then be submitted in the following way:

```
./chain_jobs.sh lockfile afterok simple_job.sh
```

This creates a chain of jobs with the dependency type `afterok`. This halts the allocation of such jobs until the independent job finishes with success. The currently running independent job, when it deems fit, calls an API function to change the dependency type of all its dependent jobs to the type `after`. This enables Slurm to consider these jobs for allocation, provided that the resources are available.

For more details, see the DEEP-EST Wiki[116].

[116] https://deeptrac.zam.kfa-juelich.de:8443/trac/wiki/Public/User_Guide/Batch_system#Workflows

8.6 File system and Storage

A number of different storage locations and file systems are available on the DEEP-EST system:

- JSC GPFS file systems, provided via the JUST storage system and mounted on all JSC systems.
- Parallel BeeGFS `/work` file system, available on all the nodes of the DEEP-EST system and hosted on the SSSM module.
- Parallel BeeOND file systems, created for the lifetime of Slurm jobs on demand and using local node storage devices (SSDs or Persistent Memory).
- Local ext3/ext4 file systems hosted on the CM, DAM and ESB nodes.

The next subsections will briefly describe each file system. More details can be found in the DEEP-EST wiki[117].

8.6.1 Permanent Storage (GPFS)

In the usage model of JSC, each user has different home directories for each of the systems that they are using, so for the DEEP-EST system there will be a directory located in

```
/p/home/jusers/username/deep
```

These home folders have a low space quota and are meant to be used for configuration files, ssh keys, etc.

Data and computational resources are assigned to projects. As a consequence, each user can create folders within each of the projects that they are part of. For the DEEP project, the project folder is located under

```
/p/project/cdeep/username
```

Here is where the user should place data. Both `/p/home` and `/p/project` are provided by the shared GPFS file systems.

All data stored in the GPFS file system is regularly backed up by JSC.

8.6.2 Shared Fast Storage (BeeGFS) on SSSM

The SSSM module hosts a total of 304 TB of storage managed by the BeeGFS parallel file system[118]. The data is stored in two RAID arrays with 24 disks each, using a RAID6 storage scheme. Four file system data servers provide access to these data, which are

[117] https://deeptrac.zam.kfa-juelich.de:8443/trac/wiki/Public/User_Guide/Filesystems

[118] https://www.beegfs.io/docs/BeeGFS_Flyer.pdf

handled through BeeGFS clients on each of the CM, DAM and ESB nodes via standard POSIX I/O interfaces. The SSSM is connected to the rest of the system using 40 Gbit/s Ethernet technology, and data are passed on to the InfiniBand fabric via IP gateways. Metadata is handled by two additional servers and resides in two SSD RAID arrays.

Users just have to move data into the `/work` file system tree to use the SSSM BeeGFS – standard POSIX interfaces are supported in all the relevant programming frameworks.

As the name implies, the SSSM is considered a temporary storage device mainly to serve data required by applications, which run on the DEEP-EST system. Users are free to leave data on that system, but there is no backup and in case of resource shortage, data will be deleted.

8.6.3 Local Storage

The compute nodes of the different modules expose some local storage devices that can be used (via ext3/ext4 file systems) during job execution. On the CM, DAM and ESB, local SSDs on each node are available via `/scratch` directory. It is meant to be used instead of `/tmp` (which should be avoided). Please, consider that `/scratch` is local to each node, hence data in `/scratch` cannot be shared between nodes. Additionally, data in `/scratch` will be removed once the job is finished. The size of `/scratch` is:

- CM and ESB nodes: ~380 GB
- DAM nodes: ~128 GB

On the DAM nodes there is additional local storage available through NVMe devices in:

- /nvme/scratch: ~1.5 TB (formatted with xfs)
- /nvme/scratch2: ~1.5 TB (formatted with ext4)

As for the data in `/scratch`, the data in the `/nvme/scratchX` directories will be removed at job termination. The DAM nodes furthermore expose some very fast persistent memory which can (depending on the operation mode) directly be used by applications and is described in Section 8.7.1.

8.6.4 Local storage – BeeOND

The Slurm installation on the system provides a new switch `--beeond` for `sbatch` / `srun` / `salloc` commands. When this command is used, Slurm triggers the mechanism to start for this job the BeeOND server and clients on each assigned node at allocation time. The server and clients are then properly removed at the end of a job and all the data is deleted.

BeeOND provides the same POSIX interfaces as the standard BeeGFS, but the data is actually stored across node-local devices. Depending on the partition size and fabric used on the module(s), significantly higher I/O bandwidths are available compared to the BeeGFS system on the SSSM.

In contrast to the use of `/scratch` devices, the BeeOND data is available to all nodes in a job partition, regardless of its physical location.

8.7 Using DEEP-EST specific features

8.7.1 Persistent memory

The Data Analytics Module is composed of 16 nodes with 384 GB RAM plus 3 TB of Intel® Optane™ Persistent Memory. Compared to DRAM, Intel Optane Persistent Memory has higher latency and lower bandwidth yet offers much higher affordable capacities than DRAM and data persistence. It can be configured in two principal modes: *Memory Mode* and *App Direct* Mode.

In Memory Mode no changes to the application are required: the installed DRAM acts as a memory cache and the Intel Optane Persistent Memory transparently offers its full memory capacity to the OS and to applications. However, memory contents is volatile here. In DEEP-EST, the partner ASTRON has made use of this mode for applications running on the DAM nodes. No specific changes or adaptations were required to the base OS of the DAM or other SW packages -- Memory Mode is enabled via UEFI/BIOS settings and requires a node reboot. To get DAM nodes configured for Memory Mode, please contact the DEEP-EST support[119] to reconfigure some DAM nodes and create a reservation for you.

In App Direct Mode, DRAM and persistent memory are mapped onto separate memory address spaces (seen as memory nodes by Linux), and applications have to be modified in order to exploit the different characteristics of the two memory technologies. Access to the persistent memory occurs through regular load and store operations. Intel has released the Open Source Persistent Memory Development Kit (PMDK[120]) as Open Source, and recent Linux distributions do fully support it.

[119] sup@deep-est.eu

[120] Information about PMDK is available from https://pmem.io/pmdk/, including links to open repositories for source code and binaries

A special use case of App Direct mode is to map a file system onto a non-volatile memory partition; for this, the `fs-dax` layer provided by PMDK enables file system access while avoiding the need to go through a block device chain. For I/O-heavy applications, this usage mode can provide significant speed-ups, as for instance reported by the NextGenIO project[121]. The BeeOND parallel file system has been adapted to use the persistent memory as a storage target, enabling a job running on *n* DAM nodes to use a transient BeeGFS file system placed onto the n×3 TByte of persistent memory at a bandwidth significantly exceeding those achievable for the NVMe SSDs.

App direct mode and PMDK are in principle supported by the current base OS of the DAM (CentOS 7), which runs the 3.x Linux kernel. Newer OS versions (such as CentOS 8 with kernel 4.x) however provide significantly better performance, and experiments were run with a back-ported 4.19 kernel to establish whether the DAM nodes would be fully functional with a combination of CentOS 7 and such kernel. Therefore, access to BeeOND using the persistent memory will be available once the kernel version has been updated accordingly.

8.7.2 SIONlib (MSA features)

SIONlib[122] is an I/O concentrator library which can significantly speed up large-scale parallel I/O. It allows users to read and write binary data to/from several thousands of processors into one or a small number of physical files. SIONlib provides simplified file handling for parallel programs which logically read or write binary data in parallel into separate files (task-local files), yet want to avoid the significant management overhead caused by having thousands of these files.

For general information on SIONlib please see the SIONlib documentation[123]. During the DEEP-EST project, three new features were added to the library: MSA aware collectives, I/O forwarding, and a CUDA-aware interface. Within this document we will concentrate on those three new features. The basics will be explained in the following subsections, but there is a more detailed description in the DEEP-EST Wiki[124].

8.7.2.1 MSA aware collectives

Recent versions of SIONlib allow all parallel processes to take part in the I/O operation which enables an exchange of I/O data between the processes, allowing a subset of

[121] Information about the NEXTGenIO project can be found at http://www.nextgenio.eu/.

[122] https://www.fz-juelich.de/ias/jsc/EN/Expertise/Support/Software/SIONlib

[123] https://apps.fz-juelich.de/jsc/sionlib/docu/current/index.html

[124] https://deeptrac.zam.kfa-juelich.de:8443/trac/wiki/Public/User_Guide/SIONlib

all processes (the `collector` processes) to perform the transfer of data to the storage for all the processes. To achieve the maximum performance benefit, the `collector` processes should be the ones that are placed on parts of the MSA with a high bandwidth link to the parallel file system holding the large files created by SIONlib.

The new feature adds a MSA algorithm for the selection of collector processes. This algorithm is portable and relies on platform specific plug-ins which are specified during the installation of SIONlib (so this is nothing the user on the DEEP-EST system has to worry about). Through these plug-ins processes, which run on parts of the system that are well suited for the role of I/O, the collector processes are identified.

The MSA aware collective I/O has to be enabled when opening a file. This is done using in the open function the `file_mode` argument, which contains a string that consists of a comma-separated list of keys and key value pairs. The word `collmsa` must appear in that list to select MSA aware collective I/O, so the open-call should look like this:

```
sion_paropen_mpi("filename", "...,collmsa,...", ...);
```

The next step is to specify the nodes that should be used by setting an environment variable. For example, to select nodes from the DAM:

```
export SION_MSA_COLLECTOR_HOSTNAME_EREGEX="dp-dam.*"
```

8.7.2.2 I/O forwarding

The collective approach mentioned above has some constraints that make it inapplicable in certain scenarios:

- By design, collective I/O operations force application tasks to coordinate. This is at odds with SIONlib's world view of separate files per task that can be accessed independently.
- Collector tasks in general have to be application tasks, i.e. they have to run the user's application. This can generate conflicts on MSA systems, if the nodes that are capable of performing I/O operations efficiently are part of a module that the user application does not map well onto.

The new feature, called I/O forwarding, helps in both scenarios. It works by relaying calls to low-level I/O functions (e.g. `open`, `write`, `stat`, etc.) via a remote procedure call (RPC) mechanism from a client task (running the user's application) to a server task (running a dedicated server program), which then executes the functions on behalf of the client. Because the server tasks are dedicated to performing I/O, they can dynamically respond to individual requests from client tasks rather than imposing coordination constraints. Also, on MSA systems the server tasks can run on different modules than the user application.

```
#!/bin/bash
# Slurm's heterogeneous jobs can be used to partition resources
# for the user's application and the forwarding server, even
# when not running on an MSA system.
#SBATCH --nodes=32 --partition=dp-cn
#SBATCH packjob
#SBATCH --nodes=4 --cpus-per-task=1 --partition=dp-dam

module load "some compiler" ParaStationMPI SIONlib/1.7.6

# Defines a shell function sionfwd-spawn that is used to
# facilitate communication of MPI ports connection details
# between the server and the client.
eval $(sionfwd-server bash-defs)

# Work around an issue in ParaStationMPI
export PSP_TCP=0

# Spawns the server, captures the connection details and
# exports them to the environment to be picked up by the
# client library used from the user's application.
sionfwd-spawn srun --pack-group 1 sionfwd-server

# Spawn the user application.
srun --pack-group 0 user_application

# Shut down the server.
srun --pack-group 0 sionfwd-server shutdown

# Wait for all tasks to end.
wait
```

Figure 8.33: Sample job script to use I/O forwarding with SIONlib

To use the I/O forwarding within the application it has to be selected when opening the file. This is done by adding the word `sionfwd` to the `file_mode` argument of SIONlib's open functions:

sion_paropen_mpi("filename", "...,sionfwd,...", ...);

Be aware that the server processes are not spawned by MPI, so the server tasks have to be launched from the user's job script before the application tasks are launched. A typical job script is shown in Figure 8.33.

8.7.2.3 CUDA aware interface

To more closely match the programming interface offered by other libraries (such as ParaStationMPI), the SIONlib functions have been made CUDA aware. This means that applications are allowed to pass device pointers, which point to device-memory, to the various read and write functions of SIONlib without the need to manually copy their contents to the host memory. The user may pass the device pointers as the `data` argument to SIONlib's write and read functions.

8.8 Summary of lessons learned

The DEEP-EST project has demonstrated the potential of the MSA. The flexibility of the MSA concept allows very different usage models, so that a wide range of different applications can be addressed. This was shown and evaluated by a selection of large-scale, real-world applications from research fields relevant for the European research arena. Most of the DEEP-EST applications combine HPC computation with advanced data processing and analytics and therefore represent the HPC as well as the HPDA areas. Thus, they do consist of multiple parts with different resource requirements, which is suitable to assess the potential of the MSA and the DEEP-EST system.

8.8.1 Achievements of each application development team

During the project lifetime the application teams showed some very promising results which made the DEEP-EST project a part on their way towards Exascale:

- NMBU – Neuroscience: The focus on performance and scalability in the DEEP-EST project has allowed NMBU to enhance performance of their applications. This has driven the development in NEST of the optimised spike-delivery algorithm and the advanced dry-run mode. The work in the DEEP-EST project on the NEST-Arbor and NEST-Elephant couplings to combine on the one hand simulations at different levels of description and on the other hand simulations and analysis has shown the potential of distributing different parts of a workflow across different modules of a MSA.
- NCSA – Molecular Dynamics: During the DEEP-EST project NCSA came to some very important conclusions: in computer-aided drug design or life sciences on the MSA one can optimise the price/performance ratio by choosing the appropriate configuration of compute nodes for each particular task. The multi-GPU FMM was developed as part of this project because FMM starts to become beneficial for large volumes of the simulation box (more than the previously used PME algorithm). This new functionality allows the utilization of FMM on large number of GPUs and opens new possibilities for GROMACS to perform very

large simulations in fields like material science, polymer science, molecular biology, nanostructures and condensed systems. For biological systems in life science research, the existing PME method already provides excellent performance on the MSA.

- ASTRON – Radio Astronomy: During the DEEP-EST project, ASTRON has made significant improvements to both of their applications: the Correlator and the Imager. The use of tensor-core technology will have a disruptive impact on correlators, due to their order-of-magnitude increase in performance and significant reduction in energy consumption when compared to the use of regular GPU cores. It was also shown that for newer generation GPUs the benefit even increases. ASTRON explored a new technique, called W-tiling. This significantly reduces the amount of memory used to create (very) large sky images, at the expense of a minor increase in computations, so that the painstaking effort of stitching hundreds of facets together belongs to the past. All in all, the DEEP-EST project enabled ASTRON to improve the overall performance of the imager and brings them a big step closer to Exascale imaging. Even if the results for the FPGA imager were not as positive as originally expected, the experience that was obtained with the OpenCL/FPGA toolkit has been very useful. ASTRON now uses this experience for other applications where FPGAs are indispensable, such as in the upgrade of the LOFAR stations.
- KU Leuven – Space Weather: very valuable experience has been gained in the usage of OpenMP5.0 to offload code to the GPU. As a result the xPic code is now accelerated in a portable, vendor-independent manner. KU Leuven showed the nearly perfect scalability for the accelerated particle solver and the code was also identified as a good candidate for Exascale scalability. On the road towards Exascale, KU Leuven believes in the continuous development of the code xPic and coupling its execution with multiple on-the-fly machine learning analysis tools. KU Leuven has already applied for a pilot program with the LUMI supercomputer centre where the Cluster-Booster architecture will be deployed using AMD CPUs and GPUs. All developments during the DEEP-EST project led to a good energy balance of the code.
- UoI – Data Analytics in Earth Science: By completely rewriting two of their codes (NextDBSCAN and NextSVM) UoI made a huge step towards Exascale. Both applications are now much stronger than the previous applications. Research within the project indicates that NextDBSCAN is now a good candidate application for Exascale systems, using both CPUs and GPUs. The results with the Horovod framework for distributed deep learning show that more work must

be done in order for it to reach Exascale system potential. Another achievement during the DEEP-EST project was the development of the Magma library to ease the porting efforts.

- CERN – High Energy Physics: The DEEP-EST project was an important part for the High Energy Physics community on the way towards Exascale HPC systems for CMS reconstruction workloads as well as for CMS classification. Porting the most time critical parts of the reconstruction to NVIDIA GPUs resulted in a significant performance gain. The work done in DEEP-EST has already been included in the official CMSSW stack.

In addition to the evaluation of the MSA concept the DEEP-EST project allowed to gain many valuable experiences:

8.8.2 Portability nearly as important as performance

In the beginning of the project, and so also during the planning phase of the project, the idea was to equip the ESB with many-core CPUs of the Intel Xeon Phi series. After a few months within the project it became clear that this would not be possible, so the plan changed to using GPUs. This led to some difficulties for some of the applications, because they did not have GPU-code available. For example, KU Leuven had only a version of xPic optimised for Intel Xeon Phis. Also UoI and CERN had only CPU based code (with multi- and many-core versions). Each one of the three application partners used a different approach to implement a new GPU version, all of them striving towards a portable solution to become vendor independent.

KU Leuven used the pragma based OpenMP 5.0 offload (Section 5.4.3.2 and Section 8.4.3.4 of this book). UoI developed Magma, a C++ header library, that makes extensive use of C++ templates to offer compile-time polymorphism for increased usability at the expense of a small compile-time overhead (Section 6.4), and CERN made use of the oneAPI framework as a portability platform (Section 7.4).

8.8.3 Different code versions for different purposes

During the project we noticed that for some of the applications it makes sense to have different code versions for different purposes:

- If we take a look at GROMACS from NCSA we see that the non-offload version is very efficient for small and medium scale problems, whereas the offload version is very efficient for large scale problems, and the version using FFM is more efficient for extremely big problems than the PME version.

- For NEST from NMBU different optimisations are needed to achieve a good performance on runs on a small number of nodes, than when targeting a large amount of nodes.

8.8.4 FPGA challenges

It turned out that programming the FPGAs was more complicated than expected. In ASTRON's case, a complete code restructuring was needed to port the imager from one generation of FPGAs (Arria10) to the new one (Stratix10). Compiling FPGA code takes a very long time (in ASTRONs case sometimes up to 24 hours) which makes it a really time consuming work. A detailed explanation on the experience with the FPGA programming is given in Section 4.4.6. Nevertheless the experiences gained are very helpful for other applications where FPGAs are indispensable, such as in the upgrade of the LOFAR stations

8.8.5 Conclusion

This report on applications experience clearly shows that the DEEP-EST system is flexible enough to accommodate the requirements coming from different problem domains. Each co-design application has benefitted from the experience made by other applications, as well as from the support from the technical consortium members who developed the hardware and software in the project. DEEP-EST has also shown that an important investment in effort and time is required to enable highly complex HPC applications to run efficiently on the next generation supercomputers, but that these efforts definitely do pay off. After over three years of joint work, the DEEP-EST applications are better prepared to exploit heterogeneous supercomputers as those expected in the Exascale era.