Offloading Computation Tasks of Hybrid Applications to GPUs

With **MPI** + **OpenMP** / **OmpSs-2**

Table of contents:

Quick Overview

Current and near-future High Performance Computing (HPC) systems consist of thousands of parallel computing nodes, connected by high-bandwidth network interconnections, and in most of the cases, each node leveraging one or more **GPU devices**.

Moreover, some of the most modern GPU devices, such as NVIDIA Tesla V100, support the **Unified Memory** which facilitates the task of the users. With those devices, users do **not have to move or copy the data to/from GPUs**, and also, **pointers at the host are the same at the device**.

For these reasons, parallel applications should take benefit from these GPU resources, and they should try to **offload** the most compute-intensive parts of to the available GPUs. In this page, we briefly explain the approaches proposed by the **OpenMP** and the **OmpSs-2** programming models to facilitate the offloading of computation tasks to Unified Memory GPUs. Then, we show an hybrid application with **MPI+OpenMP** and **MPI+OmpSs-2** variants that offloads some of the computation tasks, and that can be executed on the DEEP system.

On the one hand, **OpenMP** provides the `target` directive which is the one used for offloading computational parts of OpenMP programs to the GPUs. It provides multiple clauses to specify the **copy directionality** of the data, how the computational workload is **distributed**, the **data dependencies**, etc. The user can annotate a part of the program inside using the `target` directive and **without having to program it with a special programming language**. For instance, when offloading a part of the program to NVIDIA GPUs, the user is not required to provide any implementation in CUDA. That part of the program is handled transparently by the compiler.

On the other hand, **OmpSs-2** proposes another approach targeting NVIDIA Unified Memory devices. CUDA kernels can be annotated as **regular tasks**, and they can declare the corresponding **data dependencies** on the data buffers. When all the dependencies of a CUDA task are satisfied, the CUDA kernel associated to the task is **automatically** and **asynchronously offloaded** to one of the available GPUs. To use that functionality, the user only has to allocate the buffers that CUDA kernels will access as Unified Memory buffers (i.e., using the `cudaMallocManaged()` function).

N-Body Benchmark

An N-Body simulation 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 the bodies attract each other through the gravitational force.

N-Body simulation arises in many other computational science problems as well. For example, protein folding is studied using N-body simulation to calculate electrostatic and *Van der Waals* forces. Turbulent fluid flow simulation and global illumination computation in computer graphics are other examples of problems that use N-Body simulation.

Users can clone or download this example from the ?https://pm.bsc.es/gitlab/DEEP-EST/apps/NBody repository and transfer it to a DEEP working directory.

Requirements

The requirements of this application are shown in the following lists. The main requirements are:

- The **GNU** or **Intel®** Compiler Collection.
- A **Message Passing Interface (MPI)** implementation supporting the **multi-threading** level of thread support.
- The **Task-Aware MPI (TAMPI)** library which defines a clean **interoperability** mechanism for MPI and OpenMP/OmpSs-2 tasks. It supports both blocking and non-blocking MPI operations by providing two different interoperability mechanisms. Downloads and more information at ?https://github.com/bsc-pm/tampi.
- The **OmpSs-2** model which is the second generation of the **OmpSs** programming model. It is a **task-based** programming model originated from the ideas of the OpenMP and StarSs programming models. The specification and user-guide are available at ?https://pm.bsc.es/ompss-2-docs/spec/ and ?https://pm.bsc.es/ompss-2-docs/user-guide/, respectively. OmpSs-2 requires both **Mercurium** and **Nanos6** tools. Mercurium is a source-to-source compiler which provides the necessary support for transforming the high-level directives into a parallelized version of the application. The Nanos6 runtime system provides the services to manage all the parallelism in the application (e.g., task creation, synchronization, scheduling, etc).

Downloads at ?https://github.com/bsc-pm.

- A derivative **Clang + LLVM OpenMP** that supports the non-blocking mode of TAMPI. Not released yet.
- The **CUDA** tools and NVIDIA **Unified Memory** devices for enabling the CUDA variants, in which some of the N-body kernels are executed on the available GPU devices.

Versions

The N-Body application has several versions which are built in different binaries. All of them divide the particle space into smaller blocks. MPI processes are divided into two groups: GPU processes and CPU processes. GPU processes are responsible for computing the forces between each pair of particles blocks, and then, these forces are sent to the CPU processes, where each process updates its particles blocks using the received forces. The particles and forces blocks are equally distributed amongst each MPI process in each group. Thus, each MPI process is in charge of computing the forces or updating the particles of a consecutive chunk of blocks.

The available versions are:

- `nbody.mpi.bin`: Simple MPI parallel version using **blocking MPI** primitives for sending and receiving each block of particles/forces.
- `nbody.mpi.ompss2.bin`: Parallel version using **MPI + OmpSs-2 tasks**. Both **computation** and **communication** phases are **taskified**, however, communication tasks (each one sending or receiving a block) are serialized by an artificial dependency on a sentinel variable. This is to prevent deadlocks between processes, since communication tasks perform **blocking MPI** calls.
- `nbody.mpi.ompss2.cuda.bin`: The same as the previous version but **offloading** the tasks that compute the forces between particles blocks to the available GPUs. Those computation tasks are offloaded by the **GPU processes** and they are the most compute-intensive parts of the program. The `calculate_forces_block_cuda` task is annotated as a regular task (e.g., with their dependencies) but implemented in **CUDA**. However, since it is Unified Memory, the user **does not need to move the data to/from the GPU** device.
- `nbody.tampi.ompss2.bin`: Parallel version using **MPI + OmpSs-2 tasks + TAMPI** library. This version disables the artificial dependencies on the sentinel variable, so communication tasks can run in parallel and overlap computations. The TAMPI library is in charge of managing the **blocking MPI** calls to avoid the blocking of the underlying execution resources.
- `nbody.tampi.ompss2.cuda.bin`: A mix of the previous two variants where **TAMPI** is leveraged for allowing the concurrent execution of communication tasks, and GPU processes **offload** the compute-intensive tasks to the GPUs.
- `nbody.mpi.omp.bin`: Parallel version using **MPI + OpenMP tasks**. Both **computation** and **communication** phases are **taskified**, however, communication tasks (each one sending or receiving a block) are serialized by an artificial dependency on a sentinel variable. This is to prevent deadlocks between processes, since communication tasks perform **blocking MPI** calls.
- `nbody.mpi.omptarget.bin`: The same as the previous version but **offloading** the tasks that compute the forces between particles blocks to the available GPUs. Those computation tasks are offloaded by the **GPU processes** and they are the most compute-intensive parts of the program. This is done through the `omp target` directive, declaring the corresponding dependencies, and specifying the `target` as `nowait` (i.e., asynchronous offload). Additionally, the target directive does not require the user to provide a CUDA implementation of the offloaded task. Finally, since we are using the Unified Memory feature, we do not need to specify any data movement clause. We only have to specify that the memory buffers are already device pointers (i.e., with `is_device_ptr` clause). **Note:** This version is not compiled by default since it is still in a *Work in Progress* state.
- `nbody.tampi.omp.bin`: Parallel version using **MPI + OpenMP tasks + TAMPI** library. This version disables the artificial dependencies on the sentinel variable, so communication tasks can run in parallel and overlap computations. Since OpenMP only supports the non-blocking mechanism of TAMPI, this version leverages non-blocking primitive calls. In this way, TAMPI library is in charge of managing the **non-blocking MPI** operations to efficiently overlap communication and computation tasks.
- `nbody.tampi.omptarget.bin`: A mix of the previous two variants where **TAMPI** is leveraged for allowing the concurrent execution of communication tasks, and GPU processes **offload** the compute-intensive tasks to the GPUs. **Note:** This version is not compiled by default since it is still in a *Work in Progress* state.

Building & Executing on DEEP

The simplest way to compile this application is:

```
# Clone the benchmark's repository
$ git clone https://pm.bsc.es/gitlab/DEEP-EST/apps/NBody.git
$ cd NBody

# Load the required environment (MPI, CUDA, OmpSs-2, OpenMP, etc.)
# Needed only once per session
$ source ./setenv_deep.sh

# Compile all N-Body variants
$ make
```

The benchmark versions are built with a specific block size, which is decided at compilation time (i.e., the binary names contain the block size). The default block size is 2048. Optionally, we can indicate a different block size when compiling by doing:

```
$ make BS=1024
```

The next step is the execution of the benchmark on the DEEP system. Since this benchmarks targets the offloading of computational tasks to the Unified Memory GPU devices, we must execute it in a DEEP partition that features this kind of devices. A good example is the dp-dam partition, where each nodes features:

- 2x Intel® Xeon® Platinum 8260M CPU @ 2.40GHz (24 cores/socket, 2 threads/core), **96 CPUs/node**
- 1x **NVIDIA Tesla V100** (Volta)
- **Extoll** network interconnection

In this case, we are going to request an interactive job in a `dp-dam` node. All we need to is:

```
$ srun -p dp-dam -N 1 -n 8 -c 12 -t 01:00:00 --pty /bin/bash -i
```

With that command, we will be prompted to an interactive session in an exclusive `dp-dam` node. We indicated that we want to launch 8 processes with 12 CPUs per process at the moment of launching a binary in the allocated node through `srun`. However, we should be able to change the configuration (without overtaking the initial number of resources) when executing the binaries passing a different configuration to the `srun` command.

At this point, we are ready to execute the benchmark with multiple MPI processes. The benchmark accepts several options. The most relevant options are the total number of **particles** with `-p`, the number of **timesteps** with `-t`, and the maximum number of **GPU processes** with `-g`. More options can be seen passing the `-h` option. An example of an execution is:

```
$ srun -n 8 -c 12 ./nbody.tampi.ompss2.cuda.2048bs.bin -t 100 -p 16384 -g 4
```

in which the application will perform 100 timesteps in 8 MPI processes with 12 cores per process (used by the OmpSs-2's runtime system). The maximum number of GPU processes is 4, so there will be 4 CPU processes and 4 GPU processes (all processes have access to GPU devices). Since the total number of particles is 16384 and the blocks size is 2048, each process will be in charge of computing / updating 4096 forces / particles, which are 2 blocks.

In the CUDA variants, a process can belong to the GPU processes group if it has access to at least one GPU device. However, in the case of the non-CUDA versions, all processes can belong to the GPU processes group (i.e., the GPU processes are simulated). For this reason, the application provides `-g` option in order to control the maximum number of GPU processes. By default, the number of GPU processes will be half of the total number of processes. Also note that the non-CUDA variants cannot compute kernels on the GPU. In these cases, the structure of the application is kept but the CUDA tasks are replaced by regular CPU tasks.

Similarly, the OpenMP variants can be executed following the same steps but setting the `OMP_NUM_THREADS` to the corresponding number of CPUs per process. As an example, we could execute the following command:

```
$ OMP_NUM_THREADS=24 srun -n 4 -c 24 ./nbody.tampi.omp.2048bs.bin -t 100 -p 8912 -g 2
```

Finally, the `submit.job` script can be used to submit a non-interactive job into the job scheduler system. Feel free to modify the script with other parameters or job configurations. The script can be submitted by:

```
$ sbatch submit.job
```

References

- ?https://pm.bsc.es/ompss-2
- ?https://github.com/bsc-pm
- ?https://github.com/bsc-pm/tampi
- ?https://en.wikipedia.org/wiki/N-body_simulation
- ?https://pm.bsc.es/gitlab/DEEP-EST/apps/NBody