Wikiprint Book

Title: Offloading computational tasks of hybrid MPI + OpenMP/OmpSs-2 …

Subject: DEEP - Public/User_Guide/Offloading_hybrid_apps

Version: 24

Date: 20.04.2025 00:07:46

Table of Contents

Offloading computational tasks of hybrid MPI + OpenMP/OmpSs-2 applications to GPUs

Table of contents:

Quick Overview

N-Body Benchmark

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

Description

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.

Requirements

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

- **GNU Compiler Collection**.
- **OmpSs-2**: OmpSs-2 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 library 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.
- **Clang + LLVM OpenMP** (derived):
- **MPI**: This application requires an MPI library supporting the multi-threading level of thread support.

In addition, there are some optional tools which enable the building of other application versions:

- **CUDA** and NVIDIA **Unified Memory** devices: This application has CUDA variants in which some of the N-body kernels are executed on the available GPU devices.
- **Task-Aware MPI (TAMPI)**: The Task-Aware MPI library provides the interoperability mechanism for MPI and OpenMP/OmpSs-2. Downloads and more information at ?https://github.com/bsc-pm/tampi.

Versions

The N-Body application has several versions which are compiled in different binaries, by executing the `make` command. 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`: Parallel version using MPI.
- `nbody.mpi.ompss2.bin`: Parallel version using MPI + OmpSs-2 tasks. Both computation and communication phases are taskified, however, communication tasks are serialized by declaring 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 using CUDA tasks to execute the most compute-instensive parts of the application at the available GPUs.
- `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. 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`: The same as the previous version but using CUDA tasks to execute the most compute-instensive parts of the application at the available GPUs.
- `nbody.mpi.omp.bin`:
- `nbody.mpi.omptarget.bin`:
- `nbody.tampi.omp.bin`:
- `nbody.tampi.omptarget.bin`:

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 the code
$ 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 of the benchmark is `2048`. Optionally, you 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 have indicated that we are going to create 8 processes with 12 CPUs per process when executing binaries with the `srun` from within the node. However, you 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 number of total 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, 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.

Finally, the OpenMP variants can be executed similarly, 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
```

References