Programming with OmpSs?-2

Table of contents:

Quick Overview

OmpSs?-2 is a programming model composed of a set of directives and library routines that can be used in conjunction with a high-level programming language (such as C, C++ or Fortran) in order to develop concurrent applications. Its name originally comes from two other programming models: **OpenMP** and **StarSs?**. The design principles of these two programming models constitute the fundamental ideas used to conceive the OmpSs? philosophy.

OmpSs?-2 **thread-pool** execution model differs from the **fork-join** parallelism implemented in OpenMP.

A **task** is the minimum execution entity that can be managed independently by the runtime scheduler. **Task dependences** let the user annotate the data flow of the program and are used to determine, at runtime, if the parallel execution of two tasks may cause data races.

The reference implementation of OmpSs?-2 is based on the **Mercurium** source-to-source compiler and the **Nanos6** runtime library:

- Mercurium source-to-source compiler provides the necessary support for transforming the high-level directives into a parallelized version of the application.
- Nanos6 runtime library provides services to manage all the parallelism in the user-application, including task creation, synchronization and data movement, as well as support for resource heterogeneity.

Additional information about the OmpSs?-2 programming model can be found at:

- OmpSs?-2 official website. ?https://pm.bsc.es/ompss-2
- OmpSs?-2 specification. ?https://pm.bsc.es/ftp/ompss-2/doc/spec
- OmpSs?-2 user guide. ?https://pm.bsc.es/ftp/ompss-2/doc/user-guide
- OmpSs?-2 examples repository. ?https://pm.bsc.es/gitlab/ompss-2/examples
- OmpSs?-2 manual with examples and exercises. ?https://pm.bsc.es/ftp/ompss-2/doc/examples/index.html
- Mercurium official website. ?Link 1, ?Link 2
- Nanos official website. ?Link 1, ?Link 2

Quick Setup on DEEP System

We highly recommend to log in a **cluster module (CM) node** to begin using OmpSs?-2. To request an entire CM node for an interactive session, please execute the following command:

```
srun --partition=dp-cn --nodes=1 --ntasks=48 --ntasks-per-socket=24 --ntasks-per-node=48 --pty /bin/bash -i
```

Note that the command above is consistent with the actual hardware configuration of the cluster module with **hyper-threading enabled**.

OmpSs?-2 has already been installed on DEEP and can be used by simply executing the following commands:

- `modulepath="/usr/local/software/skylake/Stages/2018b/modules/all/Core:$modulepath"`
- `modulepath="/usr/local/software/skylake/Stages/2018b/modules/all/Compiler/mpi/intel/2019.0.117-GCC-7.3.0:$modulepath"`
- `modulepath="/usr/local/software/skylake/Stages/2018b/modules/all/MPI/intel/2019.0.117-GCC-7.3.0/psmpi/5.2.1-1-mt:$module`
- `export MODULEPATH="$modulepath:$MODULEPATH"`
- `module load OmpSs-2`

Remember that OmpSs??-2 uses a **thread-pool** execution model which means that it permanently **uses all the threads** present on the system. Users are strongly encouraged to always check the **system affinity** by running the **NUMA command** `numactl --show`:

```
$ numactl --show
policy: bind
preferred node: 0
physcpubind: 0 1 2 3 4 5 6 7 8 9 10 11 24 25 26 27 28 29 30 31 32 33 34 35
cpubind: 0
nodebind: 0
membind: 0
```

as well as the **Nanos6 command** `nanos6-info --runtime-details | grep List`:

```
$ nanos6-info --runtime-details | grep List
Initial CPU List 0-11,24-35
NUMA Node 0 CPU List 0-35
NUMA Node 1 CPU List
```

Notice that both commands return consistent outputs and, even though an entire node with two sockets has been requested, only the first NUMA node (i.e. socket) has been correctly bind. As a result, only 48 threads of the first socket (0-11, 24-35), from which 24 are physical and 24 logical (hyper-threading enabled), are going to be utilised whilst the other 48 threads available on the second socket will remain idle. Therefore, **the system affinity showed above is not valid since it does not represent the resources requested via SLURM.**

System affinity can be used to specify, for example, the ratio of MPI and OmpSs?-2 processes for a hybrid application and can be modified by user request in different ways:

- Via SLURM. However, if the affinity does not correspond to the resources requested like in the previous example, system admins will need to fix it.
- Via the command `numactl`.
- Via the command `taskset`.

Repository with Examples

All the examples shown here are publicly available at ?https://pm.bsc.es/gitlab/ompss-2/examples. Users must clone/download each example's repository and then transfer it to a DEEP working directory.

System configuration

Please refer to section Quick Setup on DEEP System to get a functional version of OmpSs?-2 on DEEP. It is also recommended to run OmpSs?-2 on a cluster module (CM) node.

Building and running the examples

All the examples come with a Makefile already configured to build (e.g. `make`) and run (e.g. `make run`) them.

Controlling available threads

In order to limit or constraint the available threads for an application, the Unix *taskset* tool can be used to launch applications with a given thread affinity. In order to use taskset, simply precede the application's binary with *taskset* followed by a list of CPU IDs specifying the desired affinity:

```
taskset -c 0,2-4 ./application
```

The example above will run *application* with 4 cores: 0, 2, 3, 4.

Dependency graphs

Nanos6 allows for a graphical representation of data dependencies to be extracted. In order to generate said graph, run the application with the *NANOS6* environment variable set to *graph*:

```
NANOS6=graph ./application
```

By default graph nodes will include the full path of the source code. To remove these, set the following environment variable:

```
NANOS6_GRAPH_SHORTEN_FILENAMES=1
```

The result will be a PDF file with several pages, each representing the graph at a certain point in time. For best results, we suggest to display the PDF with *single page* view, showing a full page and to advance page by page.

Obtaining statistics

Another equally interesting feature of Nanos6 is obtaining statistics. To do so, simply run the application as:

`NANOS6=stats ./application` or `NANOS6=stats-papi ./application`

The first collects timing statistics while the second also records hardware counters (compilation with PAPI is needed for the second). By default, the statistics are emitted standard error when the program ends.

Tracing with Extrae

A *trace.sh* file can be used to include all the environment variables needed to get an instrumentation trace of the execution. The content of this file is as follows:

```
#!/bin/bash
export EXTRAE_CONFIG_FILE=extrae.xml
export NANOS6="extrae"
$*
```

Additionally, you will need to change your running script in order to invoke the program through this *trace.sh* script. Although you can also edit your running script adding all the environment variables related with the instrumentation, it is preferable to use this extra script to easily change between instrumented and non-instrumented executions. When in need to instrument your execution, simply include *trace.sh* before the program invocation. Note that the ??extrae.xml?? file, which is used to configure the Extrae library to get a Paraver trace, is also needed.

Example: Multisaxpy

The examples shown here are publicly available at ?https://pm.bsc.es/gitlab/ompss-2/examples.

Users must clone/download this example's repository from ?https://pm.bsc.es/gitlab/ompss-2/examples/multisaxpy and transfer it to a DEEP working directory.