OmpSs-2 User Guide

BSC Programming Models

Dec 21, 2020

CONTENTS

Note: Find a PDF version of this document at https://pm.bsc.es/ftp/ompss-2/doc/user-guide/OmpSs-2-User-Guide.pdf

CHAPTER

ONE

INSTALLATION OF OMPSS-2

The first step is choosing a directory where you will install OmpSs-2. In this document this directory will be referred to as the `INSTALLATION_PREFIX` directory. We recommend you to set an environment variable `INSTALLATION_PREFIX` with the desired installation directory. For instance:

```
$ export INSTALLATION_PREFIX=$HOME/installation/ompss-2
```

1.1 Installation of Extrae (optional)

This is just a quick summary of the installation of Extrae. For a more detailed information check Extrae Homepage

1. Get Extrae from https://tools.bsc.es/downloads (choose *Source tarball* of Extrae tool).
2. Unpack the tarball and enter the just created directory:

   ```
   $ tar xfz extrae-xxx.tar.gz
   $ cd extrae-xxx
   ```

3. Export a target installation directory for Extrae:

   ```
   $ export EXTRAE_PREFIX=$HOME/installation/extrae
   ```

4. Configure it:

   ```
   $ ./configure --prefix=$EXTRAE_PREFIX
   ```

5. Build and install:

   ```
   $ make
   $ make install
   ```

Note: Extrae may have other packages's dependences and may use several options with the configure script. Do not hesitate to check the Extrae User's Manual

1.2 Installation of Nanos6

Nanos6 is a runtime that implements the OmpSs-2 parallel programming model, developed by the Programming Models group at the Barcelona Supercomputing Center.

Nanos6 can be obtained from the github public repository or by contacting us at pm-tools@bsc.es.

1.2.1 Build requirements

To install Nanos6 the following tools and libraries must be installed:

1. automake, autoconf, libtool, pkg-config, make and a C and C++ compiler
2. boost >= 1.59
3. hwloc
4. numactl
5. Finally, it's highly recommended to have a installation of Mercurium with OmpSs-2 support enabled. When installing OmpSs-2 for the first time, you can break the chicken and egg dependence between Nanos6 and Mercurium in both sides: on one hand, you can install Nanos6 without specifying a valid installation of Mercurium. On the other hand, you can install Mercurium without a valid installation of Nanos6 using the `--enable-nanos6-bootstrap` configuration flag.

1.2.2 Optional libraries and tools

In addition to the build requirements, the following libraries and tools enable additional features:

1. Extrae to generate execution traces for offline performance analysis with Paraver
2. elfutils and libunwind to generate sample-based profiling
3. graphviz and pdfjam or pdfjoin from TeX to generate graphical representations of the dependency graph
4. parallel to generate the graph representation in parallel
5. PAPI to generate statistics that include hardware counters
6. CUDA to enable CUDA tasks
7. PGI to enable OpenACC tasks
8. PAPI to generate real-time statistics of hardware counters
9. PQOS to generate real-time statistics of hardware counters
10. DLB to enable dynamic management and sharing of computing resources
11. jemalloc to use jemalloc as the default memory allocator, providing better performance than the default glibc implementation. Jemalloc software must be compiled with `--enable-stats` and `--with-jemalloc-prefix=nanos6_je_` to link with the runtime

1.2.3 Build procedure

Nanos6 uses the standard GNU automake and libtool toolchain. When cloning from a repository, the building environment must be prepared through the following command:

```
$ autoreconf -f -i -v
```

When the code is distributed through a tarball, it usually does not need that command.

Then execute the following commands:

```
$ ./configure --prefix=$INSTALLATION_PREFIX ...other options...
$ make
$ make install
```

where `$INSTALLATION_PREFIX` is the directory into which to install Nanos6.

The configure script accepts the following options:

1. `--with-nanos6-mercurium=prefix` to specify the prefix of the Mercurium installation
2. `--with-boost=prefix` to specify the prefix of the Boost installation
3. `--with-libunwind=prefix` to specify the prefix of the libunwind installation
4. `--with-papi=prefix` to specify the prefix of the PAPI installation
5. `--with-libnuma=prefix` to specify the prefix of the numactl installation
6. `--with-extrae=prefix` to specify the prefix of the extrae installation
7. `--with-dlb=prefix` to specify the prefix of the DLB installation
8. `--with-jemalloc` to specify the prefix of the jemalloc installation
9. `--with-papi=prefix` to specify the prefix of the PAPI installation
10. `--with-pqos=prefix` to specify the prefix of the PQoS installation
11. `--with-cuda[=prefix]` to enable support for CUDA tasks
12. `--enable-openacc` to enable support for OpenACC tasks; requires PGI compilers
13. `--with-pgi=prefix` to specify the prefix of the PGI compilers installation, in case they are not in `$PATH`
14. `--enable-monitoring` to enable monitoring and predictions of task/CPU/thread statistics
15. `--enable-chrono-arch` to enable an architecture-based timer for the monitoring infrastructure

The location of elfutils and hwloc is always retrieved through pkg-config. If they are installed in non-standard locations, pkg-config can be told where to find them through the `PKG_CONFIG_PATH` environment variable. For instance:

```
$ export PKG_CONFIG_PATH=$HOME/installations-mn4/elfutils-0.169/lib/pkgconfig:/apps/
↪HWLOC/2.0.0/INTEL/lib/pkgconfig:$PKG_CONFIG_PATH
```

To enable CUDA the `--with-cuda` flag is needed. The location of CUDA can be retrieved automatically, if it is in standard system locations (`/usr/lib`, `/usr/include`, etc), or through pkg-config. Alternatively, for non-standard installation paths, it can be specified using the optional `=prefix` of the parameter. The location of PGI compilers can be retrieved from the `$PATH` variable, if it is not specified through the `--with-pgi` parameter.

Optionally, if you passed a valid Mercurium installation, you can execute the Nanos6 tests by running:

```
$ make check
```

1.2.4 Configuring GIT (contributing to github)

Please set up the following git configuration variables:

- user.name
- user.email

In addition we strongly suggest you to also set up the follwing pairs of variables and values:

- rebase.stat=true
- pull.rebase=true
- branch.autosetuprebase=always

- diff.submodule=log
- fetch.recursesubmodules=true
- status.submodulesummary=true
- rerere.enabled=true

1.3 Installation of Mercurium compiler

You can find the build requirements, the configuration flags and the instructions to build Mercurium in the following link: https://github.com/bsc-pm/mcxx

You should be able to compile and install Mercurium with the following commands:

```
$ autoreconf -fiv
$ ./configure --prefix=$INSTALLATION_PREFIX --enable-ompss-2 --with-nanos6=
↪$INSTALLATION_PREFIX
$ make
$ make install
```

1.4 Installation of LLVM-based compiler

The LLVM website describes a list of build requirements of LLVM.

You should be able to compile and install the LLVM-based compiler with the following commands:

```
$ cmake -DCMAKE_BUILD_TYPE=Release \
        -DCMAKE_INSTALL_PREFIX=$INSTALLATION_PREFIX \
        -DLLVM_ENABLE_PROJECTS=clang \
        -DLLVM_INSTALL_TOOLCHAIN_ONLY=ON \
        -DCLANG_DEFAULT_NANOS6_HOME=$INSTALLATION_PREFIX
$ make
$ make install
```

More details about customizing the LLVM build can be found in the LLVM website

CHAPTER

TWO

COMPILER OPTIONS

This section describes how to use Mercurium to compile OmpSs-2 programs.

2.1 Mercurium drivers

The list of available drivers can be found here: https://github.com/bsc-pm/mcxx/blob/master/doc/md_pages/profiles.md

2.2 Common compilation flags

Usual flags like `-O`, `-O1`, `-O2`, `-O3`, `-D`, `-c`, `-o`, ... are recognized by Mercurium.

Almost every Mercurium-specific flag is of the form `--xxx`.

Mercurium drivers are deliberately compatible with `gcc`. This means that flags of the form `-fXXX`, `-mXXX` and `-Wxxx` are accepted and passed onto the backend compiler without interpretation by Mercurium drivers.

> **Warning:** In GCC a flag of the form `-fXXX` is equivalent to a flag of the form `--XXX`. This is **not** the case in Mercurium.

2.2.1 Getting command line help

You can get a summary of all the flags accepted by Mercurium using `--help` with any of the drivers:

```
$ mcc --help
Usage: mcc options file [file..]
Options:
  -h, --help               Shows this help and quits
  --version                Shows version and quits
  --v, --verbose           Runs verbosely, displaying the programs
                           invoked by the compiler
...
```

2.2.2 Passing vendor-specific flags

While almost every `gcc` of the form `-fXXX` or `-mXXX` can be passed directly to a Mercurium driver, some other vendor-specific flags may not be well known or be misunderstood by Mercurium. When this happens, Mercurium has a generic way to pass parameters to the backend compiler and linker.

`--Wn,<comma-separated-list-of-flags>` Passes comma-separated flags to the native compiler. These flags are used when Mercurium invokes the backend compiler to generate the object file (`.o`)

`--Wl,<comma-separated-list-of-flags>` Passes comma-separated-flags to the linker. These flags are used when Mercurium invokes the linker

`--Wp,<comma-separated-list-of-flags>` Passes comma-separated flags to the C/Fortran preprocessor. These flags are used when Mercurium invokes the preprocessor on a C or Fortran file.

These flags can be combined. Flags `--Wp,a --Wp,b` are equivalent to `--Wp,a,b`. Flag `--Wnp,a` is equivalent to `--Wn,a --Wp,a`

Important: Do not confuse `--Wl` and `--Wp` with the `gcc` similar flags `-Wl` and `-Wp` (note that `gcc` ones have a single `-`). The latter can be used with the former, as in `--Wl,-Wl,muldefs`. That said, Mercurium supports `-Wl` and `-Wp` directly, so `-Wl,muldefs` should be enough.

2.3 Compile OmpSs-2 programs

For OmpSs-2 programs that run in SMP or NUMA systems, you do not have to do anything. Just pick one of the drivers above.

Following is a very simple OmpSs-2 program in C:

```
/* test.c */
#include <stdio.h>

int main(int argc, char *argv[])
{
  int x = argc;
  #pragma oss task inout(x)
  {
    x++;
  }
  #pragma oss task in(x)
  {
    printf("argc + 1 == %d\n", x);
  }
  #pragma oss taskwait
  return 0;
}
```

Compile it using *mcc*:

```
$ mcc -o test --ompss-2 test.c
```

Important: Do not forget the flag `--ompss-2` otherwise your program will be compiled without parallel support.

And run it:

```
$ ./test
argc + 1 == 2
```

2.3.1 Compile OpenMP programs with OmpSs-2 support

By passing the `--openmp-compatibility` flag, Mercurium will support some constructs of OpenMP (e.g. taskloop, parallel for, etc.).

Warning: Experimental flag. There are no guarantees the generated code behaves as the OpenMP program.

2.4 Problems during compilation

While we put big efforts to make a reasonably robust compiler, you may encounter a bug or problem with Mercurium.

There are several errors of different nature that you may run into:

- Mercurium ends abnormally with an internal error telling you to open a ticket.
- Mercurium does not crash but gives an error on your input code and compilation stops, as if your code were not valid.
- Mercurium does not crash, but gives an error involving an `internal-source`.
- Mercurium generates wrong code and native compilation fails on an intermediate file.
- Mercurium forgets something in the generated code and linking fails.

In order for us to fix your problem we need the *preprocessed* file. If your program is C/C++ we need you to do:

1. Figure out the compilation command of the file that fails to compile. Make sure you can replicate the problem using that compilation command alone.
2. If your compilation command includes `-c`, replace it by `-E`. If it does not include `-c` simply add `-E`.
3. If your compilation command includes `-o file` (or `-o file.o`) replace it by `-o file.ii`. If it does not include `-o`, simply add `-o file.ii`.
4. Now run the compiler with this modified compilation command. It should have generated a `file.ii`.
5. These files are usually very large. Please compress them with `gzip` (or `bzip2` or any similar tool).

Send us an email to pm-tools at bsc.es with the error message you are experiencing and the (compressed) preprocessed file attached. If your program is Fortran just the input file may be enough, but you may have to add all the `INCLUDE`d files and modules.

CHAPTER

THREE

RUNTIME OPTIONS

This section describes how to run OmpSs-2 applications and which runtime options are available.

3.1 Executing and determining the number of CPUs

Nanos6 applications can be executed as is:

```
# Compile OmpSs-2 program with Mercurium
$ mcc --ompss-2 app.c -o app

# Execute on all available cores of the current session
$ ./app
```

The number of cores that are used is controlled by running the application through the `taskset` command. For instance:

```
# Execute on cores 0, 1, 2 and 4
$ taskset -c 0-2,4 ./app
```

3.2 Runtime configuration options (NEW)

The behaviour of the Nanos6 runtime can be tuned after compilation by means of a configuration file. All former Nanos6 environment variables are obsolete and will be ignored by the runtime system. Currently, the supported configuration file format is TOML v1.0.0-rc1. The default configuration file is named `nanos6.toml` and can be found in the documentation directory of the Nanos6 installation:

```
$INSTALLATION_PREFIX/share/doc/nanos6/scripts
```

To override the default configuration, it is recommended to copy the default file and change the relevant options. The first configuration file found will be interpreted, according to the following order:

1. The file pointed by the `NANOS6_CONFIG` environment variable.
2. The file `nanos6.toml` found in the current working directory.
3. The file `nanos6.toml` found in the installation path (default file).

The configuration file is organized into different sections and subsections. Option names have the format `<section>[.<subsection>].<option_name>`. For instance, the dependency system that the runtime should load is specified by the `version.dependencies` option. Inside the TOML configuration file, that option is placed inside the `version` section as follows:

```
[version]
    dependencies = "discrete"
```

Alternatively, if the configuration has to be changed programatically and creating new files is not practical, the configuration options can be overriden using the `NANOS6_CONFIG_OVERRIDE` environment variable. The content of this variable has to be in the format `option1=value1,option2=value2,option3=value3,...`, providing a comma-separated list of assignations.

For example, you can run the following command to change the dependency implementation and use CTF instrumentation:

```
NANOS6_CONFIG_OVERRIDE="version.dependencies=discrete,version.instrument=ctf" ./ompss-
↪program`
```

By default, the runtime system emits a warning during initialization when it detects the definition of irrelevant environment variables that start with the `NANOS6` prefix. The exceptions are the previous two variables `NANOS6_CONFIG` and `NANOS6_CONFIG_OVERRIDE`, but also the `NANOS6_HOME` variable. This latter is not relevant for the Nanos6 runtime system but it is often defined by users and it is relevant for the OmpSs-2 LLVM-based compiler. The warning can be disabled by setting the `loader.warn_envars` configuration option to `false`.

3.3 Runtime variants

There are several Nanos6 runtime variants, each one focusing on different aspects of parallel executions: performance, debugging, instrumentation, etc. OmpSs-2 applications do not require recompiling their code to run with instrumentation, e.g., to extract Extrae traces or to generate additional information. This is instead controlled through configration options, at run-time. Users can select a specific Nanos6 variant when running an application by setting the `version.instrument`, `version.debug` and `version.dependencies` configuration variables. This section explains the different values for these variables.

The instrumentation is specified by the `version.instrument` configuration variable and can take the following values:

`version.instrument = none` This is the **default** value and does not enable any kind of instrumentation. This is the variant that should be used when executing peformance experiments since it is the one that adds no instrumentation overhead. Continue reading this section for more information about performance runs with Nanos6.

`version.instrument = extrae` Instrumented to produce Paraver traces. See: *Generating Extrae traces*.

`version.instrument = ctf` Instrumented to produce CTF traces and convert them to the Paraver format. See: *Generating CTF traces*.

`version.instrument = graph` Instrumented to produce a graph of the execution. Only practical for small graphs. See: *Generating a graphical representation of the dependency graph*.

`version.instrument = verbose` Instrumented to emit a log of the execution. See: *Verbose instrumentation*.

`version.instrument = stats` Instrumented to produce a summary of metrics of the execution. See: *Obtaining statistics*.

`version.instrument = lint` Instrumented to support the OmpSs-2@Linter tool.

By default, Nanos6 loads runtime variants that are compiled using high optimization flags and most of the internal assertions turned off. This is the configuration that should be used along with the default `none` instrumentation for benchmarking experiments. However, these are not the only options that provide the best performance. See *Benchmarking* for more information.

Sometimes it is useful to run an OmpSs-2 program with debug information for debugging purposes (e.g., when a program crashes). The runtime system provides the option `version.debug` to load a runtime variant that has been compiled without optimizations and with all internal assertions turned on. The default value for this option is `false` (no debug), but can be changed to `true` to enable the debug information. Please note that the runtime system will significantly decrease its performance when enabling this option. Additionally, all instrumentation variants have their optimized and debug variants.

Finally, the last configuration variable used to specify a runtime variant is the `version.dependencies`, which is explained in the next section.

3.4 Task data dependencies

The Nanos6 runtime has support for different dependency implementations. The `discrete` dependencies are the default dependency implementation. This is the most optimized implementation but it does not fully support the OmpSs-2 dependency model since it does not support region dependencies. In the case the user program requires region dependencies (e.g., to detect dependencies among partial overlapping dependency regions), Nanos6 privides the `regions` implementation, which is completely spec-compliant. This latter is also the only implementation that supports OmpSs-2@Cluster.

The dependency implementation can be selected at run-time through the `version.dependencies` configuration variable. The available implementations are:

`version.dependencies = "discrete"` Default and optimized implementation not supporting region dependencies. Region syntax is supported but will behave as a discrete dependency to the first address. Scales better than the default implementation thanks to its simpler logic and is functionally similar to traditional OpenMP model.

`version.dependencies = "regions"` Supporting all dependency features. Default implementation in OmpSs-2@Cluster installations.

In cases where an OmpSs-2 program requires region dependency support, we recommended to add the declarative directive `assert` in any of the program source files, as shown below. Then, before the program is started, the runtime system will check whether the loaded dependency implementation is `regions` and will abort the execution if it is not true.

```
#pragma oss assert("version.dependencies==regions")

int main() {
    // ...
}
```

Notice that the `assert` directive could also check whether the runtime is using `discrete` dependencies.

3.5 Task scheduler

The scheduling infrastructure provides the following configuration options to modify the behavior of the task scheduler:

`scheduler.policy` (default: `fifo`) Specifies whether ready tasks are added to the ready queue using a FIFO (`fifo`) or a LIFO (`lifo`) policy.

`scheduler.immediate_successor` (default: `true`) Boolean indicating whether the immediate successor policy is enabled. If enabled, once a CPU finishes a task, the same CPU starts executing its successor task (computed through the data dependencies) such that it can reuse the data on the cache.

scheduler.priority (default: true) Boolean indicating whether the scheduler should consider the task priorities defined by the user in the task's priority clause.

3.6 Task worksharings

Worksharing tasks are a special type of tasks that can only be applied to for-loops. The key point of worksharing tasks is their ability to run concurrently on different threads, similarly to OpenMP parallel fors. In contrast, worksharing tasks do not force all the threads to collaborate neither introduce any kind of barrier.

An example is shown below:

```
#pragma oss task for chunksize(1024) inout(array[0;N]) in(a)
for (int i = 0; i < N; ++i) {
    array[i] += a;
}
```

In our implementation, worksharing tasks are executed by taskfor groups. Taskfor groups are composed by a set of available CPUs. Each available CPU on the system is assigned to a specific taskfor group. Then, a worksharing task is assigned to a particular taskfor group, so it can be run by at most as many CPUs (also known as collaborators) as that taskfor group has. Users can set the number of groups (and so, implicitly, the number of collaborators) by setting the `taskfor.groups` configuration option. By default, there are as many groups as NUMA nodes in the system.

Finally, taskfors that do not define any chunksize leverage a chunksize value computed as their total number of iterations divided by the number of collaborators per taskfor group.

3.7 Stack size

By default, Nanos6 allocates stacks of 8 MB for its worker threads. In some codes this may not be enough. For instance, when converting Fortran codes, some global variables may need to be converted into local variables. This may increase substantially the amount of stack required to run the code and may surpass the space that is available.

To solve that problem, the stack size can be set through the `misc.stack_size` configuration variable. Its value is expressed in bytes but it also accepts the `K`, `M`, `G`, `T` and `E` suffixes, that are interpreted as power of 2 multipliers. For instance:

```
[misc]
    stack_size = "16M"
```

3.8 Dynamic Load Balancing (DLB)

DLB is a library devoted to speed up hybrid parallel applications and maximize the utilization of computational resources. More information about this library can be found at the following link: https://pm.bsc.es/dlb.

To enable DLB support for Nanos6, a working DLB installation must be present in your environment. Configuring Nanos6 with DLB support is done through the `--with-dlb` flag, specifying the root directory of the DLB installation.

After configuring DLB support for Nanos6, its enabling can be controlled at run-time through the `dlb.enabled` configuration variable. To run Nanos6 with DLB support then, this variable must be set to true (`dlb.enabled=true`), since by default DLB is disabled.

Once DLB is enabled for Nanos6, OmpSs-2 applications will benefit from dynamic resource sharing automatically. Assuming that DLB has been explicitly enabled at the configuration file, the following example showcases the executions of two applications that share the available CPUs between them:

```
# Run the first application using 10 CPUs (0, 1, ..., 9)
$ taskset -c 0-9   ./merge-sort.test &

# Run the second application using 10 CPUs (10, 11, ..., 19)
$ taskset -c 10-19 ./cholesky-fact.test &

# Now those applications should be running while sharing resources
# ...
```

3.9 CPU manager policies

Nanos6 offers different policies when handling CPUs through the `cpumanager.policy` configuration variable:

`cpumanager.policy = idle` Activates the idle policy, in which idle threads halt on a blocking condition, while not consuming CPU cycles.

`cpumanager.policy = busy` Activates the busy policy, in which idle threads continue spinning and never halt, consuming CPU cycles.

`cpumanager.policy = lewi` If DLB is enabled, activates the LEnd When Idle policy, which is similar to the idle policy but in DLB mode. In this policy, idle threads lend their CPU to other runtimes or processes.

`cpumanager.policy = greedy` If DLB is enabled, activates the greedy policy, which disables lending CPUs from the process' mask, but allows acquiring and lending external CPUs.

`cpumanager.policy = default` Fallback to the **default** implementation. If DLB is disabled, this policy falls back to the idle policy, while if DLB is enabled it falls back to the lewi policy.

3.10 Benchmarking, instrumenting and debugging

As previously stated, there are several Nanos6 runtime variants, each one focusing on different aspects of parallel executions: performance, debugging, instrumentation, etc. OmpSs-2 applications do not require recompiling their code to run with instrumentation, e.g., to extract Extrae traces or to generate additional information. This is instead controlled through configration options, at run-time. Users can select a specific Nanos6 variant when running an application by setting the `version.instrument`, `version.debug` and `version.dependencies` configuration variables. The next subsections explain the details of the different variants and options of Nanos6.

3.10.1 Benchmarking

Nanos6 loads an optimized runtime variant if the `version.debug` option is disabled (by default is disabled). The optimized runtime variants are compiled with high optimization flags, they do not perform validity checks and they do not provide debug information. The benchmarking of OmpSs-2 applications should be performed with an optimized version (`version.debug=false`) and no instrumentation (`version.instrument=none`) to get optimal perfomance.

Additionally, Nanos6 offers an extra performance option called Turbo, which enables even more aggressive optimizations. This option is disabled by default but it can be enabled by setting `turbo.enabled=true`. Currently, this option enables two Intel® floating-point (FP) unit optimizations in all tasks: flush-to-zero (FZ) and denormals are zero (DAZ). However, please note these FP optimizations could alter the precision of floating-point computations.

We recommend to combine a optimized version without instrumentation (`version.debug=false` and `version.instrument=none`) with the Turbo feature (`turbo.enabled=true`), the discrete dependency implementation (`version.dependencies=discrete`) and the Jemalloc memory allocator to obtain the best performance.

Jemalloc must be enabled at configuration time by passing `--with-jemalloc` to the Nanos6 configure script. See the Nanos6 building requirements to check which are the requirements for supporting jemalloc.

3.10.2 Generating Extrae traces

To generate a Paraver trace using Extrae, the `version.instrument` configuration variable must be set to `extrae` before running the application.

By default, the runtime will generate a trace as if `EXTRAE_ON` was set to 1. In addition, the `EXTRAE_CONFIG_FILE` environment variable can be set to an Extrae configuration file for fine tuning, for instance, to enable recording hardware counters. See: the Extrae documentation.

The amount of information generated in the Extrae traces can be controlled through the `instrument.extrae.detail_level` configuration option. Its value determines a level of detail that goes from 0, which is the least detailed, up to 8. The default level is 1. Lower levels incur in less overhead and produce smaller traces. Higher levels have more overhead, produce bigger traces, but are more precise and comtain more information. The information generated at each level is incremental and is the following:

Level 0 Basic level

Includes basic information about the runtime state and the execution of tasks.

Counters about the number of tasks in the system are approximate and are sampled at periodic intervals.

Level 1 Default level

Counters about the number of tasks in the system are precise in terms of time and value.

Adds communication records that link the point of instantiation of a task to the point where they start their execution.

Adds communication records that link the point where a task get blocked to the point where a task unblocks it (makes it ready), and from that point to the point where the task actually resumes its execution.

Adds communication records that show task dependency relations. The set of predecessors a task that is shown is limited to the set of tasks that have not finished their execution once said task has been instantiated. The links go from the end of the execution of a predecessor to the start of the execution of the successor.

Levels 2 to 7 Unused

Currently they do not add further information.

Level 8 Very detailed level

Adds communication records that link the end of the execution of a task to the point where its parent returns from a taskwait. Similarly to the graph information, the trace only contains links for the tasks that have not finished once their parent enters the taskwait. This information is only available at level 8, since it may make the trace significantly bigger.

The runtime installation contains a set of already made Paraver configuration files at the following subdirectory:

```
$INSTALLATION_PREFIX/share/doc/nanos6/paraver-cfg/nanos6
```

Support for hardware counters is enabled through file specified in the `EXTRAE_CONFIG_FILE` environment variable. The procedure is explained in the Extrae documentation. However, the `instrument.extrae.as_threads` configuration variable must also be set to `true`. This is a temporary measure that is needed to produce correct

hardware counter information. The resulting trace will expose the actual runtime threads, as opposed to the CPU view that is generated by default.

3.10.3 Generating CTF traces

The CTF instrumentation backend is a performance evaluation and debugging tool aimed for Nanos6 users and developers alike. Its purpose is to record the most relevant information of a Nanos6 execution efficiently and to provide means for visually inspecting it offline. The Nanos6 CTF backend is simple (no installation prerequisites) and mostly lock-free (see the section "Implementation details" for more information).

Nanos6 stores traces in the Common Trace Format (CTF). CTF traces can be directly visualized with tools such as babeltrace1 or babeltrace2 (raw command line inspector). But it is recommended first to convert them to Paraver traces using the provided `ctf2prv` command. Although Nanos6 requires no special packages to write CTF traces, the `ctf2prv` converter needs python3 and the babeltrace2 python bindings.

The Nanos6 CTF backend will transparently collect hardware counters information if the Nanos6 Hardware counters infrastructure has been enabled. See the User Guide for more information on how to enable hardware counters.

This backend objective is not to replace Extrae, but to provide a minimum set of features for easy Nanos6 introspection. More sophisticated analysis (such as task dependency graphs) will have to be performed with Extrae as usual.

Implementation details

Nanos6 keeps a per-core buffer to store CTF events before flushing them to the storage device. Therefore, events generated by Nanos6 Worker threads do not require to take a look. However, events triggered by a thread not directly controlled by Nanos6 share a buffer, and it is necessary to take a lock.

When a buffer is full, it is needed to flush it to the storage device. Flushing is done synchronously (i.e. a worker thread that generates a tracepoint will have to flush the buffer if there no free space left) and it might affect the execution workflow. Flushing operations are also recorded and can be inspected with the "CTF flush buffers to disk" view.

Traces are written by default under the `TMPDIR` environment variable or under `/tmp` if not set. Traces written to `/tmp` are kept in RAM memory (see tmpfs for more information) and flushing translates to a memory copy operation. When an application execution finishes, Nanos6 copies the trace to the current directory. The configuration file provides an option `instrument.ctf.tmpdir` to override the default location.

Usage

To generate a CTF trace, run the application with the `version.instrument` configuration variable set to `ctf`.

This will create a `trace-<app_name>-<app_pid>` folder in the current directory, hereinafter refered to as `$TRACE` for convinience. The subdirectory `$TRACE/ctf` contains the ctf trace as recorded by Nanos6.

By default, Nanos6 will convert the trace automatically at the end of the execution unless the user explicitly sets the configuration variable `instrument.ctf.conversor.enabled = false`. The converted Paraver trace will be stored under the `$TRACE/prv` subdirectory. The environment variable `CTF2PRV_TIMEOUT=<minutes>` can be set to stop the conversion after the specified elapsed time in minutes. Please note that the conversion tool requires python3 and the babeltrace2 package.

Additionally, there is command to manually convert a trace:

```
$ ctf2prv $TRACE
```

which will generate the directory `$TRACE/prv` with the Paraver trace.

Linux Kernel tracing

The requirements for tracing kernel events are: - Linux Kernel >= 4.1.0 - Set /proc/sys/kernel/perf_event_paranoid to -1 - Grant the current user read permissions for /sys/kernel/debug/tracing or /sys/kernel/tracing (tracefs standard mountpoints)

The Linux Kernel provides a set of events (also named tracepoints). To enable kernel events tracing in Nanos6, the user needs to specify which events wants to collect through the nanos6 configuration file. The option `instrument.ctf.events.kernel.presets` allows to speciy a list of presets that ease selecting events for supported Paraver views. For instance, try to set `instrument.ctf.events.kernel.presets=[preemption]` to enable preemption-related events.

Please check the Paraver views description below for the list of presets that each view requries.

If the user does not want to enable some of the events defined in a preset, it can blacklist them in the `instrument.ctf.events.kernel.exclude` list.

Additionally, the user can provide a file path in `instrument.ctf.events.kernel.file` with a list of raw kernel events to enable. An example file named `kernel_events.conf` follows.

```
$ cat kernel_events.conf
sched_switch
sys_enter_open
sys_exit_open
# mm_page_alloc # comented lines are not enabled
```

A list of events that your system supports can be obtained with the command:

Or by inspecting the contents of tracefs, usually mounted by default at `/sys/kernel/tracing/events` or `/sys/kernel/debug/tracing/events`. Also, Nanos6 creates the file `$TRACE/nanos6_kerneldefs.json` which contains a human-readable format definition of all supported system events. Unfortunately, there is no official Linux description of each event at the moment; please check the Linux Kernel source code for a definition of the events you are interested into.

If too many events are enabled, tracing might fail due to reaching the limit of open files in the system (one file descriptor is opened per event and core). In that case, please, increase the open file limit, reduce the number of events and/or the number of cores.

The CTF kernel trace is stored under `$TRACE/ctf/kernel` whilst the Nanos6 internal events trace are stored in `$TRACE/ctf/ust`. Nanos6 will convert the generated CTF kernel trace (both user and kernel) into Paraver at the end of the execution, as usual.

Paraver views

A number of Paraver views are provided with each Nanos6 distribution under the directory:

```
$INSTALLATION_PREFIX/share/doc/nanos6/paraver-cfg/ctf2prv
```

Some views refer to “Nanos6 core code”. This encompasses all code that it is not strictly user-written code (task code). This includes, for instance, the worker thread main loop (where tasks are requested constantly) the Nanos6 initialization code or the shutdown code.

The Paraver minimum resolution is 1 pixel. Hence, if the current zoom level is not close enough, multiple events might fall into the same pixel. In this situation, Paraver must choose which of these events must be colored. Several visualization options to tackle this problem can be found under the “Drawmode” option in the context menu (right mouse click on a Paraver view). By default, most views have the “Random” mode, which means that drawn events are selected randomly. This is useful to avoid giving priority to an event or another, but the Paraver user must be aware that when zooming in (or out) events might seem to “move” due to another set of events being selected randomly after

zooming. In the particular case of graphs (such as the "Number of" views), the Drawmode "Maximum" option must be selected if searching for maximum peaks, otherwise the event with the maximum value might not be selected to be drawn and the peak might remain hidden. Similarly, "Minimum" must be selected if searching for minimum peaks.

Please, note that some views might complain about missing events when opened in Paraver. It is likely that the missing event is "Runtime: Busy Waiting" as it is only generated when running Nanos6 under the `busy` policy. You can safely ignore the message. If unsure, you can try disabling the "Runtime: Busy Waiting" in your view, save the cfg, and reload the cfg again.

Do not use the Extrae cfg's views as some event identifiers are not compatible.

Tasks Shows the name (label) of tasks executed in each core.

Tasks and runtime Shows the name (label) of tasks executed in each core. It also displays the time spent on Nanos6 Runtime (shown as "Runtime") and threads busy waiting (shown as "busy wait") while waiting for ready tasks to execute.

Task source code Shows the source code location of tasks executed in each core. It also displays the time spent on Nanos6 Runtime (shown as "Runtime") and threads busy waiting (shown as "busy wait") while waiting for ready tasks to execute.

Hardware counters Shows the collected hardware counters (HWC) information. Please note that you must enable HWC collection in Nanos6 first. See the User Guide for more details.

The HWC information is collected per each task burst (a task executing without interruption such as blocking) and per also inside Nanos6 core code. By default, it displays `PAPI_TOT_INS`. Each HWC is displayed as a different Extrae Event. If you want to inspect another counter, please, modify the "Value to display" subview of the "Hardware counters" view to display the appropriate Extrae Event (HWC event) under Paraver's "Filter -> Events -> Event Type -> Types" menu.

Task id Shows the unique id of the tasks executed in each core.

The default colouring of this view is "Not Null Gradient Color", which eases identifying the execution progress of tasks as they were created. The id's 0, 1, 2 and 3 are reserved for Idle, Runtime, Busy Wait and Task, respectively. Hence, it might be interesting to manually set the Paraver's "Semantic Minimum" value to 4, which will not include these values for the gradient colouring. This is the value by default, but if you perform a semantic adjustment, you will need to change the "Semantic Minimum" manually again.

It might also be interesting to draw this view as "Code Color" which will make it easier to spot different consecutive tasks. Under this colouring scheme, it is easier to see the time spent running Nanos6 core code shown as "Runtime" and when threads perform a "busy wait" while waiting for more tasks to execute.

Thread id Shows the thread Id (TID) of the thread that was running in each core. Be aware that this view only shows the thread placement according to Nanos6 perspective, not the OS perspective. This means that even if this view shows a thread running uninterruptedly in a core for a long time, the system might have preempted the Nanos6 thread by another system thread a number of times without this being displayed in the view.

Runtime status simple Shows the a simplified view of the runtime states. It displays the time spent running Nanos6 core code shown as "Runtime", when threads perform a "busy wait" while waiting for more tasks to execute and when the runtime is running task's code shown as "Task".

Runtime subsystems Shows the activity of the most interesting Nanos6 subsystems. In essence, a coarse-grained classification of the time spent while running Nanos6 core code. The displayed subsystems include: Task creation/initialization, dependency registration/unregistration, scheduling add/get tasks and polling services.

Number of blocked tasks Shows a graph of tasks in the blocked state. Blocked tasks are tasks that started running at some point and then stopped running before completing. This might block, for instance, due to a taskwait directive.

You might want to change paraver's "Drawmode" option to "Maximum" or "Minimum" if searching for maximum or minimum peaks. By default, it is set to "Maximum", which means that minimum peaks might be hidden

if the view is too much zoomed out.

Number of running tasks Shows a graph of tasks being executed by some worker thread.

Number of created tasks Shows a graph with the count of total created tasks. A task is created when its Nanos6 data structures are allocated and registered within the Nanos6 core.

Number of created workers Shows a graph of created worker threads by Nanos6. Once a worker thread is created, it is not destroyed until the end of the application's execution.

Number of running workers Shows a graph of Nanos6 worker threads that are allowed to run on a core. Please note that even if Nanos6 allows a worker to run, another system thread might have temporarily preempted that worker. System preemptions are not displayed in this view.

Number of blocked workers Shows a graph of worker threads that are blocked (no longer running), either because they are idle or because they are waiting for some event. This view only counts workers that blocked due to Nanos6 will. If a worker that is running a task blocks because of a blocking operation performed by the task code (hence, outside the scope of Nanos6) it will not be shown in this view.

CTF flush buffers to disk Eventually, the Nanos6 buffers that hold the events are filled and need to be flushed to disk. When this happens, the application's execution might be altered because of the introduced overhead. This flushing can occur in a number of places within the Nanos6 core, either because the buffers were completely full or because Nanos6 decided to flush them before reaching that limit.

This view shows exactly when the flushing happened. If attempting to write the event A into the Nanos6 event buffer triggers a flush, the produced Nanos6 trace will first show the event that triggered the flush followed by the flushing events.

Paraver views for kernel events

All the views listed below require Linux Kernel events. Please, check the section "Linux Kernel tracing" for more details.

The Paraver views that depend on kernel events can be found under:

```
$INSTALLATION_PREFIX/share/doc/nanos6/paraver-cfg/nanos6/ctf2prv/kernel
```

Kernel preemptions Uses Linux Kernel events to show preemptions affecting the cores where the traced application runs. Preemptions include interruptions and both user and kernel threads. Interrupts will be shown as "IRQ" and "Soft IRQ". IRQs are run in interrupt context. Soft IRQs are deferred interrupt work that run out of interrupt context.

This view requires the Linux Kernel `preemption` preset (interrupts and threads) or the `context_switch` preset (only threads). If using the `context_switch` preset, Paraver might warn about not finding interrupt events, but it is safe to inspect the view.

Kernel thread ids Uses Linux Kernel events to show the Thread id (TID) of the running thread in each core used by Nanos6. This view requires the Linux Kernel `context_switch` preset.

Kernel system calls Uses Linux Kernel events to show the system calls performed by each thread on cores used by Nanos6. In this view, all threads but the traced application will be displayed as "Other threads" to enhance the readability of system calls.

This view requires the Linux Kernel `syscall` preset to trace all system calls. If the user only wants to trace a set of syscalls, it can instead specify the `context_switch` preset and provide a list of system call events manually using the `instrument.ctf.events.kernel.file` option in the Nanos6 configuration file.

3.10.4 Verbose instrumentation

To enable verbose logging, run the application with the `version.instrument` configuration variable set to `verbose`.

By default it generates a lot of information. This is controlled by the `instrument.verbose.areas` configuration variable, which can contain a list of areas. The areas are the following:

AddTask Task creation.

Blocking Blocking and unblocking within a task through calls to the blocking API.

ComputePlaceManagement Starting and stopping compute places (CPUs, GPUs, etc).

DependenciesByAccess Dependencies by their accesses.

DependenciesByAccessLinks Dependencies by the links between the accesses to the same data.

DependenciesByGroup Dependencies by groups of tasks that determine common predecessors and common successors.

LeaderThread Execution of the leader thread.

LoggingMessages Additional logging messages.

TaskExecution Task execution.

TaskStatus Task status transitions.

TaskWait Entering and exiting taskwaits.

ThreadManagement Thread creation, activation and suspension.

UserMutex User-side mutexes (critical).

The case is ignored, and the `all` keyword enables all of them. Additionally, an area can have the `!` prepended to it to disable it. For instance, `areas = [ "AddTask", "TaskExecution" , "TaskWait" ]` is a good starting point.

The default value is:

```
[instrument]
    [instrument.verbose]
        areas = ["all", "!ComputePlaceManagement", "!DependenciesByAccess", "!
↪DependenciesByAccessLinks",
            "!DependenciesByGroup", "!LeaderThread", "!TaskStatus", "!ThreadManagement
↪"]
```

By default, events are recorded with their timestamp and ordered accordingly. This can be disabled by setting the `instrument.verbose.timestamps` configuration variable to `false`.

The output is emitted by default to standard error, but it can be sent to a file by specifying it through the `instrument.verbose.output_file` option. Also the `instrument.verbose.dump_only_on_exit` can be set to `true` to delay the output to the end of the program to avoid getting it mixed with the output of the program.

3.10.5 Generating a graphical representation of the dependency graph

To generate the graph, run the application with the `version.instrument` configuration variable set to `graph`.

The graph instrumentation creates a subdirectory with the graph in several stages and a script that can be executed to generate a PDF that combines each step in a different page. The progress of the execution can be visualized by advancing the pages. This PDF is intended to be viewed in whole page mode, instead of continuous mode.

The most common graph instrumentation options are the following:

`instrument.graph.shorten_filenames` (default: `false`) When generating nodes, do not emit the directory together with the source code file name.

`instrument.graph.show_all_steps` (default: `false`) Instead of trying to collapse in one step as many related changes as possible, show one at a time.

`instrument.graph.display` (default: `false`) Automatically process the graph through graphviz and display it.

`instrument.graph.display_command` (default: `xdg-open` or `evince` or `okular` or `acroread`) Command to display the final PDF of the graph. Only effective if `instrument.graph.display` is `true`.

`instrument.graph.show_log` (default: `false`) Emit a table next to the graph with a description of the changes in each frame.

In addition, the following advanced configuration variables can be used to debug the runtime:

`instrument.graph.show_dependency_structures` (default: `false`) Show the internal data structures that determine when tasks are ready.

`instrument.graph.show_spurious_dependency_structures` (default: `false`) Do not hide internal data structures that do not determine dependencies or that are redundant by transitivity.

`instrument.graph.show_dead_dependency_structures` (default: `false`) Do not hide the internal data structures after they are no longer relevant.

`instrument.graph.show_regions` (default: `false`) When showing internal data structures, include the information about the range of data or region that is covered.

3.10.6 Obtaining statistics

To enable collecting statistics, run the application with the `version.instrument` configuration variable set to `stats`. This variant collects timing statistics.

By default, the statistics are emitted to the standard error when the program ends. The output can be sent to a file through the `instrument.stats.output_file` configuration option.

The contents of the output contains the average for each task type and the total task average of the following metrics:

- Number of instances
- Mean instantiation time
- Mean pending time (not ready due to dependencies)
- Mean ready time
- Mean execution time
- Mean blocked time (due to a critical or a taskwait)
- Mean zombie time (finished but not yet destroyed)
- Mean lifetime (time between creation and destruction)

The output also contains information about:

- Number of CPUs
- Total number of threads
- Mean threads per CPU
- Mean tasks per thread

- Mean thread lifetime
- Mean thread running time

Most codes consist of an initialization phase, a calculation phase and final phase for verification or writing the results. Usually these phases are separated by a taskwait. The runtime uses the taskwaits at the outermost level to identify phases and emit individual metrics for each phase.

3.10.7 Debugging

By default, the runtime is optimized for speed and will assume that the application code is correct. Hence, it will not perform most validity checks. To enable validity checks, run the application with the `version.debug` configuration variable set to `true`. This will enable many internal validity checks that may be violated when the application code is incorrect.

To debug an application with a regular debugger, please compile its code with the regular debugging flags and also the `-keep` flag (Mercurium). This flag forces Mercurium to dump the transformed code in the local file system, so that it is available for the debugger.

To debug dependencies, it is advised to reduce the problem size so that very few tasks trigger the problem, and then let the runtime make a graphical representation of the dependency graph. See *Generating a graphical representation of the dependency graph*.

The runtime system has a component named "loader" that is the responsible for selecting a runtime variant that meets the requirements set by the user though the configuration variables. The loader is also responsible for calling some functions that initialize the runtime variant and convert the `main` function into a regular task. To debug problems due to the installation, run the application with the `loader.verbose` configuration variable set to `true`.

3.10.8 Runtime loader verbosity

The `loader.verbose` configuration option controls the verbosity of the Nanos6 loader. By default (`false` value), it is quiet. If the configuration variable has value `true` it will emit to standard error the actions that it takes and their outcome.

By default, the loader will attempt to load the actual runtime library from the path determined by the operating system (taking into account the rpath and the `LD_LIBRARY_PATH` environment variable). If it fails to load the library, then it will attempt to locate the library at the same location as the Nanos6 loader.

The default search path can be overriden through the `loader.library_path` configuration variable. If it exists the first attempt at loading the runtime will be performed at the directory specified in that variable. The loader does not accept multiple directories in that variable.

The Nanos6 loader resolves the addresses of the API functions to the actual runtime implementation. In addition, it also checks for the implementation of some features, and if they are not found, it will either complain or emit a warning and fall back to a compatible but less powerful implementation. More specifically, the loader accepts running applications that make use of weak dependencies and will fall back to strong dependencies if the runtime does not have support for them.

3.10.9 Reporting runtime information

Information about the runtime may be obtained by running the application with the `loader.report_prefix` configuration variable, or by invoking the following command:

```
$ nanos6-info --runtime-details
Runtime path /opt/nanos6/lib/libnanos6-optimized.so.0.0.0
Runtime Version 2017-11-07 09:26:03 +0100 5cb1900
Runtime Branch master
Runtime Compiler Version g++ (Debian 7.2.0-12) 7.2.1 20171025
Runtime Compiler Flags -DNDEBUG -Wall -Wextra -Wdisabled-optimization -Wshadow -
↪fvisibility=hidden -O3 -flto
Initial CPU List 0-3
NUMA Node 0 CPU List 0-3
Scheduler priority
Dependency Implementation regions (linear-regions-fragmented)
Threading Model pthreads
```

The `loader.report_prefix` configuration variable may be defined as an empty string or it may contain a string that will be prepended to each line. For instance, it can contain a sequence that starts a comment in the output of the program. Example:

```
$ NANOS6_CONFIG_OVERRIDE="loader.report_prefix=#" ./app
Some application output ...
#   string  version 2017-11-07 09:26:03 +0100 5cb1900       Runtime Version
#   string  branch  master      Runtime Branch
#   string  compiler_version    g++ (Debian 7.2.0-12) 7.2.1 20171025        Runtime␣
↪Compiler Version
#   string  compiler_flags  -DNDEBUG -Wall -Wextra -Wdisabled-optimization -Wshadow -
↪fvisibility=hidden -O3 -flto       Runtime Compiler Flags
#   string  initial_cpu_list    0-3     Initial CPU List
#   string  numa_node_0_cpu_list    0-3     NUMA Node 0 CPU List
#   string  scheduler   priority        Scheduler
#   string  dependency_implementation   regions (linear-regions-fragmented)      ␣
↪Dependency Implementation
#   string  threading_model pthreads        Threading Model
```

3.11 Throttle

There are some cases where user programs are designed to run for a very long time, instantiating in the order of tens of millions of tasks or more. These programs can demand a huge amount of memory in small intervals when they rely only on data dependencies to achieve task synchronization. In these cases, the runtime system could run out of memory when allocating internal structures for task-related information if the number of instantiated tasks is not kept under control.

To prevent this issue, the runtime system offers a *throttle* mechanism that monitors memory usage and stops task creators while there is high memory pressure. This mechanism does not incur too much overhead because the stopped threads execute other ready tasks (already instantiated) until the memory pressure decreases. The main idea of this mechanism is to prevent the runtime system from exceeding the memory budget during execution. Furthermore, the execution time when enabling this feature should be similar to the time in a system with infinite memory.

The throttle mechanism requires a valid installation of Jemalloc, which is a scalable multi-threading memory allocator. Hence, the runtime system must be configured with the `--with-jemalloc` option. Although the throttle feature is disabled by default, it can be enabled and tunned at runtime through the following configuration variables:

`throttle.enabled` (default: `false`) Boolean variable that enables the throttle mechanism.

`throttle.tasks` (default: `5.000.000`) Maximum absolute number of alive childs that any task can have. It is divided by 10 at each nesting level.

`throttle.pressure` (default: `70`) Percentage of memory budget used at which point the number of tasks allowed to exist will be decreased linearly until reaching 1 at 100% memory pressure.

`throttle.max_memory` (default: available physical memory / 2) Maximum used memory or memory budget. Note that this variable can be set in terms of bytes or in memory units. For example: `throttle.max_memory=50GB`.

3.12 Hardware counters

Nanos6 offers an infrastructure to obtain hardware counter statistics of tasks with various backends. The usage of this API is controlled through the Nanos6 configuration file. Currently, Nanos6 supports the PAPI, RAPL and PQoS backends. All the available hardware counter backends are listed in the default configuration file, found in the scripts folder. To enable any of these, modify the `false` fields and change them to `true`. Specific counters can be enabled or disabled by adding or removing their name from the list of `counters` inside each backend subsection.

Next we showcase a simplified version of the hardware counter section of the configuration file, where the PAPI backend is enabled with counters that monitor the total number of instructions and cycles, and the PAPI backend is enabled as well:

```
[hardware_counters]
    [hardware_counters.papi]
        enabled = true
        counters = ["PAPI_TOT_INS", "PAPI_TOT_CYC"]
    [hardware_counters.rapl]
        enabled = true
```

3.13 OmpSs-2@Cluster

In order to enable OmpSs-2@Cluster support, you need a working MPI installation in your environment that supports multithreading, i.e. `MPI_THREAD_MULTIPLE`. Nanos6 needs to be configured with the `--enable-cluster` flag.

For more information on how to write and run cluster applications see Cluster.md.

CHAPTER

FOUR

LLVM-BASED COMPILER

This section describes how to use the LLVM-based compiler to compile OmpSs-2 programs. This compiler is based on the open-source LLVM infrastructure and is licensed under the LLVM license.

Important: The LLVM-based compiler is still a **beta** development and does not support all the features supported by Mercurium. Check *LLVM-based compiler OmpSs-2 support* for a description of the current support.

4.1 Compiling OmpSs-2 programs using the LLVM-based compiler

4.1.1 C/C++ programs

Following is a very simple OmpSs-2 program in C:

```
/* test.c */
#include <stdio.h>

int main(int argc, char *argv[])
{
  int x = argc;
  #pragma oss task inout(x)
  {
    x++;
  }
  #pragma oss task in(x)
  {
    printf("argc + 1 == %d\n", x);
  }
  #pragma oss taskwait
  return 0;
}
```

Compile it using `clang`:

```
$ clang -o test -fompss-2 test.c
```

Important: Do not forget the flag `-fompss-2` otherwise your program will be compiled without parallel support.

And run it:

```
$ ./test
argc + 1 == 2
```

Use the driver `clang++` to compile C++ applications:

```
/* test.cpp */
#include <iostream>

template <typename T>
void test(T x)
{
  T t = x;
  #pragma oss task inout(x)
  {
    x *= 2;
  }
  #pragma oss task in(x)
  {
    std::cout << t << " * 2 == " << x << "\n";
  }
  #pragma oss taskwait
}

int main(int argc, char *argv[])
{
  test(42);
  test(84.1);
  return 0;
}
```

Compile it using `clang++`:

```
$ clang++ -o test -fompss-2 test.cpp
```

And run it:

```
$ ./test
42 * 2 = 84
84.1 * 2 = 168.2
```

4.1.2 Nanos6 Libraries

By default *clang* will link the program with the Nanos6 libraries installed in the *CLANG_DEFAULT_NANOS6_HOME* directory (see InstallationLLVM). It is possible to override that default value using the environment variable *NANOS6_HOME*

You can set the environment variable before linking:

```
$ export NANOS6_HOME=directory-of-nanos6-installation
```

4.2 Problems with the LLVM-based compiler

While we put big efforts to make a reasonably robust compiler, you may encounter a bug or problem when using it. There are several errors of different nature that you may run into:

- The compiler ends abnormally with an internal error telling you to report a bug.
- The compiler does not crash but gives an error on your input code and compilation stops, as if your code were not valid.
- The compiler forgets something in the generated code and linking fails.
- Compilation succeeds but the program crashes at runtime.

4.2.1 How can you help us to solve the problem quicker in the LLVM compiler?

If clang crashes, it will generate a reproducer automatically with a message like this:

```
********************

PLEASE ATTACH THE FOLLOWING FILES TO THE BUG REPORT:
Preprocessed source(s) and associated run script(s) are located at:
clang: note: diagnostic msg: /tmp/t-56517d.c
clang: note: diagnostic msg: /tmp/t-56517d.sh
clang: note: diagnostic msg:

********************
```

Send us an email to `pm-tools at bsc.es` with those two files attached. Reproducer files are usually very large. Please compress them with `gzip` (or `bzip2` or any similar tool).

If clang does not crash, then follow the next steps to obtain a reproducer:

1. Figure out the compilation command of the file that fails to compile. Make sure you can replicate the problem using that compilation command alone.
2. Add the option `-gen-reproducer` to the compilation command. Clang will generate a reproducer. Send us the reproducer files.

If for some reason the option `-gen-reproducer` does not work for you, use the following steps:

1. Figure out the compilation command of the file that fails to compile. Make sure you can replicate the problem using that compilation command alone.
2. If your compilation command includes `-c`, replace it by `-E`. If it does not include `-c` simply add `-E`.
3. If your compilation command includes `-o file` (or `-o file.o`) replace it by `-o file.ii`. If it does not include `-o`, simply add `-o file.ii`.
4. Now run the compiler with this modified compilation command. It should have generated a `file.ii`.
5. Send us that file. Please compress it with `gzip` (or `bzip2` or any similar tool)

4.3 LLVM-based compiler OmpSs-2 support

The LLVM-based compiler for OmpSs-2 is still in beta and as such does not support all the features that are currently supported by Mercurium.

4.3.1 Unsupported features

Feature	Notes and workarounds	Planned as future work?
OpenMP compatibility mode	LLVM already supports OpenMP and it would conflict in the compiler itself	No
`atomic`	C11/C++11 atomics may be used instead	Yes
`critical`	C11/C++ locking facilities may be used instead	Yes
Device support		Yes
Fortran support	Planned once the LLVM's flang project reaches sufficient maturity	Yes

INDEX

O

P

R

S

T

V

X