Programming with OmpSs

Release

BSC Programming Models

Jun 28, 2019

CONTENTS

1 OmpSs Specification **3**
- 1.1 Introduction to OmpSs . . . 3
  - 1.1.1 Reference implementation . . . 3
  - 1.1.2 A bit of history . . . 4
  - 1.1.3 Influence in OpenMP . . . 4
  - 1.1.4 Glossary of terms . . . 5
- 1.2 Programming model . . . 7
  - 1.2.1 Execution model . . . 7
  - 1.2.2 Memory model . . . 7
  - 1.2.3 Data sharing rules . . . 10
  - 1.2.4 Asynchronous execution . . . 10
  - 1.2.5 Dependence flow . . . 11
  - 1.2.6 Task scheduling . . . 15
  - 1.2.7 Task reductions . . . 16
  - 1.2.8 Runtime Library Routines . . . 17
  - 1.2.9 Environment Variables . . . 17
- 1.3 Language description . . . 17
  - 1.3.1 Task construct . . . 18
  - 1.3.2 Target construct . . . 20
  - 1.3.3 Loop construct . . . 20
  - 1.3.4 Taskwait construct . . . 21
  - 1.3.5 Taskyield directive . . . 22
  - 1.3.6 Atomic construct . . . 22
  - 1.3.7 Critical construct . . . 23
  - 1.3.8 Declare reduction construct . . . 23

2 OmpSs User Guide **25**
- 2.1 Installation of OmpSs . . . 25
  - 2.1.1 Preparation . . . 25
  - 2.1.2 Installation of Extrae (optional) . . . 25
  - 2.1.3 Installation of Nanos++ . . . 26
  - 2.1.4 Installation of Mercurium C/C++/Fortran source-to-source compiler . . . 28
- 2.2 Compile OmpSs programs . . . 28
  - 2.2.1 Drivers . . . 28
  - 2.2.2 Compiling an OmpSs program for shared-memory . . . 29
  - 2.2.3 Compiling an OmpSs program with CUDA tasks . . . 30
  - 2.2.4 Compiling an OmpSs program with OpenCL tasks . . . 32
  - 2.2.5 Problems during compilation . . . 34
- 2.3 Running OmpSs Programs . . . 34
  - 2.3.1 Runtime Options . . . 35

2.3.2 Running on Specific Architectures . . . 36
2.3.3 Runtime Modules (plug-ins) . . . 46
2.3.4 Extra Modules (plug-ins) . . . 65
2.4 Installation of OmpSs from git . . . 65
2.4.1 Additional requirements when building from git . . . 65
2.4.2 Nanos++ from git . . . 65
2.4.3 Mercurium from git . . . 66
2.5 FAQ: Frequently Asked Questions . . . 66
2.5.1 What is the difference between OpenMP and OmpSs? . . . 66
2.5.2 How to create burst events in OmpSs programs . . . 67
2.5.3 How to execute hybrid (MPI+OmpSs) programs . . . 70
2.5.4 How to exploit NUMA (socket) aware scheduling policy using Nanos++ . . . 71
2.5.5 My application crashes. What can I do? . . . 75
2.5.6 I am trying to use regions, but tasks are serialised and I think they should not . . . 76
2.5.7 My application does not run as fast as I think it could . . . 77
2.5.8 How to run OmpSs on Blue Gene/Q? . . . 78
2.5.9 Why macros do not work in a #pragma? . . . 78
2.5.10 How to track dependences for a given task using paraver . . . 80

3 OmpSs Examples and Exercises **81**
3.1 Introduction . . . 81
3.1.1 System configuration . . . 81
3.1.2 Building the examples . . . 82
3.1.3 Job Scheduler: Minotauro . . . 83
3.1.4 Job Scheduler: Marenostrum . . . 84
3.1.5 Document's contributions . . . 84
3.2 Writing OmpSs programs . . . 85
3.2.1 Data Management . . . 85
3.2.2 Application's kernels . . . 89
3.3 Examples Using OmpSs . . . 91
3.3.1 Introduction . . . 91
3.3.2 Cholesky kernel . . . 92
3.3.3 Stream Benchmark . . . 93
3.3.4 Array Sum Benchmark (Fortran version) . . . 93
3.4 Beginners Exercises . . . 94
3.4.1 Matrix Multiplication . . . 94
3.4.2 Dot Product . . . 95
3.4.3 Multisort application . . . 96
3.5 GPU Device Exercises . . . 97
3.5.1 Introduction . . . 97
3.5.2 Saxpy kernel . . . 97
3.5.3 Krist kernel . . . 98
3.5.4 Matrix Multiply . . . 98
3.5.5 NBody kernel . . . 99
3.5.6 Cholesky kernel . . . 100
3.6 MPI+OmpSs Exercises . . . 100
3.6.1 Matrix multiply . . . 100
3.6.2 Heat diffusion (Jacobi solver) . . . 101
3.7 OmpSs+DLB Exercises . . . 102
3.7.1 PILS (Parallel ImbaLance Simulator) . . . 102
3.7.2 Lulesh . . . 102
3.7.3 LUB . . . 102
3.7.4 PILS - multiapp example . . . 103

Bibliography **105**

Index **107**

The information included in this document is provided “as is”, with no warranties whatsoever, including any warranty of merchantability, fitness for any particular purpose, or any warranty otherwise arising out of any proposal, specification, or sample. The document is not guaranteed to be complete and/or error-free at this stage and it is subject to changes without furthernotice. Barcelona Supercomputing Center will not assume any responsibility for errors or omissions in this document. Please send comments, corrections and/or suggestions to pm-tools at bsc.es. This document is provided for informational purposes only.

Note: There is a PDF version of this document at http://pm.bsc.es/ompss-docs/book/ProgrammingWithOmpSs.pdf

CHAPTER

ONE

OMPSS SPECIFICATION

The information included in this document is provided "as is", with no warranties whatsoever, including any warranty of merchantability, fitness for any particular purpose, or any warranty otherwise arising out of any proposal, specification, or sample. The document is not guaranteed to be complete and/or error-free at this stage and it is subject to changes without furthernotice. Barcelona Supercomputing Center will not assume any responsibility for errors or omissions in this document. Please send comments, corrections and/or suggestions to pm-tools at bsc.es. This document is provided for informational purposes only.

Note: There is a PDF version of this document at http://pm.bsc.es/ompss-docs/spec/OmpSsSpecification.pdf

1.1 Introduction to OmpSs

OmpSs 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 in order to develop concurrent applications. This programming model is an effort to integrate features from the StarSs programming model family, developed by the Programming Models group of the Computer Sciences department at Barcelona Supercomputing Center (BSC), into a single programming model.

OmpSs is based on tasks and dependences. Tasks are the elementary unit of work which represents a specific instance of an executable code. Dependences let the user annotate the data flow of the program, this way at runtime this information can be used to determine if the parallel execution of two tasks may cause data races.

The goal of OmpSs is to provide a productive environment to develop applications for modern High-Performance Computing (HPC) systems. Two concepts add to make OmpSs a productive programming model: performance and ease of use. Programs developed in OmpSs must be able to deliver a reasonable performance when compared to other programming models that target the same architectures. Ease of use is a concept difficult to quantify but OmpSs has been designed using principles that have been praised by their effectiveness in that area.

In particular, one of our most ambitious objectives is to extend the OpenMP programming model with new directives, clauses and API services or general features to better support asynchronous data-flow parallelism and heterogeneity (as in GPU-like devices).

This document, except when noted, makes use of the terminology defined in the OpenMP Application Program Interface version 3.0 *[OPENMP30]*

1.1.1 Reference implementation

The reference implementation of OmpSs is based on the Mercurium source-to-source compiler and the Nanos++ Runtime Library:

- The Mercurium source-to-source compiler provides the necessary support for transforming the high-level directives into a parallelized version of the application.

- The Nanos++ runtime library provides the services to manage all the parallelism in the user-application, including task creation, synchronization and data movement, and provide support for resource heterogeneity.

1.1.2 A bit of history

The name OmpSs comes from the name of two other programming models, OpenMP and StarSs. The design principles of these two programming models form the basic ideas used to conceive OmpSs.

OmpSs takes from OpenMP its philosophy of providing a way to, starting from a sequential program, produce a parallel version of the same by introducing annotations in the source code. This annotations do not have an explicit effect in the semantics of the program, instead, they allow the compiler to produce a parallel version of it. This characteristic feature allows the users to parallelize applications incrementally. Starting from the sequential version, new directives can be added to specify the parallelism of different parts of the application. This has an important impact on the productivity that can be achieved by this philosophy. Generally when using more explicit programming models the applications need to be redesigned in order to implement a parallel version of the application, the user is responsible of how the parallelism is implemented. A direct consequence of this is the fact that the maintenance effort of the source code increases when using a explicit programming model, tasks like debugging or testing become more complex.

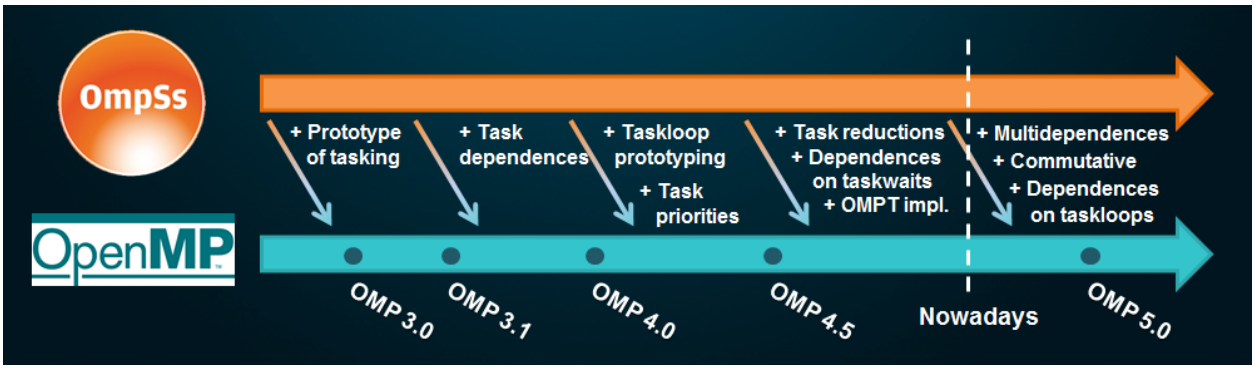
StarSs, or Star SuperScalar, is a family of programming models that also offer implicit parallelism through a set of compiler annotations. It differs from OpenMP in some important areas. StarSs uses a different execution model, thread-pool where OpenMP implements fork-join parallelism. StarSs also includes features to target heterogeneous architectures while OpenMP only targets shared memory systems. Finally StarSs offers asynchronous parallelism as the main mechanism of expressing parallelism whereas OpenMP only started to implement it since its version 3.0.

StarSs raises the bar on how much implicitness is offered by the programming model. When programming using OpenMP, the developer first has to define which regions of the program will be executed on parallel, then he or she has to express how the inner code has to be executed by the threads forming the parallel region, and finally it may be required to add directives to synchronize the different parts of the parallel execution. StarSs simplifies part of this process by providing an environment where parallelism is implicitly created from the beginning of the execution, thus the developer can omit the declaration of parallel regions. The definition of parallel code is used using the concept of tasks, which are pieces of code which can be executed asynchronously in parallel. When it comes to synchronizing the different parallel regions of a StarSs applications, the programming model also offers a dependency mechanism which allows to express the correct order in which individual tasks must be executed to guarantee a proper execution. This mechanism enables a much richer expression of parallelism by StarSs than the one achieved by OpenMP, this makes StarSs applications to exploit the parallel resources more efficiently.

OmpSs tries to be the evolution that OpenMP needs in order to be able to target newer architectures. For this, OmpSs takes key features from OpenMP but also new ideas that have been developed in the StarSs family of programming models.

1.1.3 Influence in OpenMP

Many OmpSs and StarSs ideas have been introduced into the OpenMP programming model. The next figure summarizes our contributions:



Starting from the version 3.0, released on May 2008, OpenMP included the support for asynchronous tasks. The reference implementation, which was used to measure the benefits that tasks provided to the programming model, was developed at BSC and consisted on the Nanos4 run-time library and the Mercurium source-to-source compiler.

Our next contribution, which was included in OpenMP 4.0 (released on July 2013), was the extension of the tasking model to support data dependences, one of the strongest points of OmpSs that allows to define fine-grain synchronization among tasks.

In OpenMP 4.5, which is the newest version, the tasking model was extended with the `taskloop` construct. The reference implementation of this construct was developed at BSC. Apart from that, we also contributed to this version adding the `priority` clause to the `task` and `taskloop` constructs.

For the upcoming OpenMP versions, we plan to propose more tasking ideas that we already have in OmpSs like task reductions or new extensions to the tasking dependence model.

1.1.4 Glossary of terms

ancestor tasks The set of tasks formed by your *parent* task and all of its *ancestor tasks*.

base language The *base language* is the programming language in which the program is written.

child task A task is a child of the task which encounters its *task generating code*.

construct A *construct* is an executable directive and its associated statement. Unlike the OpenMP terminology, we will explicitly refer to the *lexical scope* of a constructor or the *dynamic extent* of a construct when needed.

data environment The *data environment* is formed by the set of variables associated with a given *task*.

declarative directive A directive that annotates a declarative statement.

dependence Is the relationship existing between a *predecessor task* and one of its *successor tasks*.

descendant tasks The descendant tasks of a given task is the set of all its child tasks and the descendant tasks of them.

device A device is an abstract component, including hardware and/or software elements, allowing to execute tasks. Devices may be accessed by means of the offload technique. That means that there are tasks generated in one device that may execute in a different device. All OmpSs programs have at least one device (i.e. the *host device*) with one or more processors.

directive In C/C++ a *#pragma* preprocessor entity.

In Fortran a comment which follows a given syntax.

dynamic extent The *dynamic extent* is the interval between establishment of the execution entity and its explicit disestablishment. Dynamic extent always obey to a stack-like discipline while running the code and it includes any code in called routines as well as any implicit code introduced by the OmpSs implementation.

executable directive A directive that annotates an executable statement.

expression Is a combination of one or more data components and operators that the base program language may understand.

function task In C, an task declared by a `task` directive at *file-scope* that comes before a *declaration* that declares a single function or comes before a *function-definition*. In both cases the *declarator* should include a parameter type list.

In C++, a task declared by a `task` directive at *namespace-scope* or *class-scope* that comes before a *function-definition* or comes before a *declaration* or *member-declaration* that declares a single function.

In Fortran, a task declared by a `task` directive that comes before a the `SUBROUTINE` statement of an *external-subprogram*, *internal-subprogram* or an *interface-body*.

host device The *host device* is the device in which the program begins its execution.

inline task In C/C++ an explicit task created by a `task` directive in a statement inside a *function-definition*.

In Fortran, an explicit task created by a `task` directive in the executable part of a *program unit*.

lexical scope The *lexical scope* is the portion of code which is lexically (i.e. textually) contained within the establishing construct including any implicit code lexically introduced by the OmpSs implementation. The lexical scope does not include any code in called routines.

outline tasks An outlined task is also know as a *function tasks*.

predecessor task A task becomes *predecessor* of another task(s) when there are dependence(s) between this task and the other ones (i.e. its *successor tasks*. That is, there is a restriction in the order the runtime must execute them: all *predecessor tasks* must complete before a *successor task* can be executed.

parent task The task that encountered a *task generating code* is the parent task of the new created task(s).

ready task pool Is the set of tasks ready to be executed (i.e. they are not blocked by any condition).

sliceable task A task that may generate other tasks in order to compute the whole computational unit.

slicer policy The slicer policy determines the way a sliceable task must be segmented.

structured block An executable statement with a single entry point (at the top) and a single exit point (at the bottom).

successor task A task becomes *successor* of another task(s) when there are dependence(s) between these tasks (i.e. its *predecessors tasks*) and itself. That is, there is a restriction in the order the runtime must execute them: all the *predecessor task* must complete before a *successor task* can be executed.

target device A device onto which tasks may be offloaded from the *host device* or other *target devices*. The ability of offloading tasks from a *target device* onto another *target device* is implementation defined.

task A task is the minimum execution entity that can be managed independently by the runtime scheduler (although a single task may be executed at different phases according with its *task switching points*). Tasks in OmpSs can be created by any *task generating code*.

task dependency graph The set of tasks and its relationships (*successor* / *predecessor*) with respect the correspondant scheduling restrictions.

task generating code The code which execution create a new task. In OmpSs it can occurs when encountering a `task` construct, a `loop` construct or when calling a routine annotated with a `task` declarative directive.

thread A thread is an execution entity that may execute concurrently with other threads within the same process. These threads are managed by the OmpSs runtime system. In OmpSs a thread executes tasks.

1.2 Programming model

1.2.1 Execution model

The OmpSs runtime system creates a team of threads when starting the user program execution. This team of threads is called the initial team, and it is composed by a single master thread and several additional workers threads. The number of threads that forms the team depend on the number of workers the user have asked for. So, if the user have required 4 workers then one single master thread is created and three additional workers threads will be attached to that initial team.

The master thread, also called the initial thread, executes sequentially the user program in the context of an implicit task region called the initial task (surrounding the whole program). Meanwhile, all the other additional worker threads will wait until concurrent tasks were available to be executed.

Multiple threads execute tasks defined implicitly or explicitly by OmpSs directives. The OmpSs programming model is intended to support programs that will execute correctly both as parallel and as sequential programs (if the OmpSs language is ignored).

When any thread encounters a loop construct, the iteration space which belongs to the loop inside the construct is divided in different chunks according with the given schedule policy. Each chunk becomes a separate task. The members of the team will cooperate, as far as they can, in order to execute these tasks. There is a default `taskwait` at the end of each loop construct unless the `nowait` clause is present.

When any thread encounters a task construct, a new explicit task is generated. Execution of explicitly generated tasks is assigned to one of the threads in the initial team, subject to the thread's availability to execute work. Thus, execution of the new task could be immediate, or deferred until later according to task scheduling constraints and thread availability. Threads are allowed to suspend the current task region at a task scheduling point in order to execute a different task. If the suspended task region is for a tied task, the initially assigned thread later resumes execution of the suspended task region. If the suspended task region is for an untied task, then any thread may resume its execution.

The OmpSs specification makes no guarantee that input or output to the same file is synchronous when executed in parallel. In this case, the programmer is responsible for synchronizing input and output statements (or routines) using the provided synchronization constructs or library routines. For the case where each thread accesses a different file, no synchronization by the programmer is necessary

1.2.2 Memory model

One of the most relevant features of OmpSs is to handle architectures with disjoint address spaces. By disjoint address spaces we refer to those architectures where the memory of the system is not contained in a single address space. Examples of these architectures would be distributed environments like clusters of SMPs or heterogeneous systems built around accelerators with private memory.

Single address space view

OmpSs hides the existence of other address spaces present on the system. Offering the single address space view fits the general OmpSs philosophy of freeing the user from having to explicitly expose the underlying system resources. Consequently, a single OmpSs program can be run on different system configurations without any modification.

In order to correctly support these systems, the programmer has to specify the data that each task will access. Usually this information is the same as the one provided by the data dependencies, but there are cases where there can be extra data needed by the task that is not declared in the dependencies specification (for example because the programmer knows it is a redundant dependence), so OmpSs differentiates between the two mechanisms.

Specifying task data

A set of directives allows to specify the data that a task will use. The OmpSs run-time is responsible for guaranteeing that this data will be available to the task code when its execution starts. Each directive also specifies the directionality of the data. The data specification directives are the following:

- `copy_in(memory-reference-list)` The data specified must be available in the address space where the task is finally executed, and this data will be read only.
- `copy_out(memory-reference-list)` The data specified will be generated by the task in the address space where the task will be executed.
- `copy_inout(memory-reference-list)` The data specified must be available in the address space where the task runs, in addition, this data will be be updated with new values.
- `copy_deps` Use the data dependencies clauses (`in/out/inout`) also as if they were `copy_[in/out/inout]` clauses.

The syntax accepted on each clause is the same as the one used to declare data dependencies (see Dependence flow section). Each data reference appearing on a task code must either be a local variable or a reference that has been specified inside one of the copy directives. Also, similar to the specification of data dependencies, the data referenced is also limited by the data specified by the parent task, and the access type must respect the access type of the data specified by the parent. The programmer can assume that the implicit task that represents the sequential part of the code has a read and write access to the whole memory, thus, any access specified in a user defined top level task is legal. Failure to respect these restrictions will cause an execution error. Some of these restrictions limit the usage of complex data structures with this mechanism.

The following code shows an OmpSs program that defines two tasks:

```
float x[128];
float y[128];

int main () {
  for (int i = 0; i < N; i++) {
    #pragma omp target device(smp)
    #pragma omp task inout(x) copy_deps // implicit copy_inout(x)
    do_computation_CPU(x);

    #pragma omp target device(cuda)
    #pragma omp task inout(y) copy_deps // implicit copy_inout(x)
    do_computation_GPU(y);
  }
#pragma omp taskwait
  return 0;
}
```

One that must be run on a regular CPU, which is marked as `target device(smp)`, and the second which must be run on a CUDA GPU, marked with `target device(cuda)`. This OmpSs program can run on different system configurations, with the only restriction of having at least one GPU available. For example, it can run on a SMP machine with one or more GPUs, or a cluster of SMPs with several GPUs on each node. OmpSs will internally do the required data transfers between any GPU or node of the system to ensure that each tasks receives the required data. Also, there are no references to these disjoint address spaces, data is always referenced using a single address space. This address space is usually referred to as the host address space.

Accessing children task data from the parent task

Data accessed by children tasks may not be accessible by the parent task code until a synchronization point is reached. This is so because the status of the data is undefined since the children tasks accessing the data may not have completed

the execution and the corresponding internal data transfers. The following code shows an example of this situation:

```
float y[128];

int main () {
  #pragma omp target device(cuda)
  #pragma omp task copy_inout(y)
  do_computation_GPU(y);

  float value0 = y[64]; // illegal access

  #pragma omp taskwait

  float value1 = y[64]; // legal access

  return 0;
}
```

The parent task is the implicitly created task and the child task is the single user defined task declared using the task construct. The first assignment (`value0 = y[64]`) is illegal since the data may be still in use by the child task, the `taskwait` in the code guarantees that following access to array `y` is legal.

Synchronization points, besides ensuring that the tasks have completed, also serve to synchronize the data generated by tasks in other address spaces, so modifications will be made visible to parent tasks. However, there may be situations when this behavior is not desired, since the amount of data can be large and updating the host address space with the values computed in other address spaces may be a very expensive operation. To avoid this update, the clause `noflush` can be used in conjunction with the synchronization `taskwait` construct. This will instruct OmpSs to create a synchronization point but will not synchronize the data in separate address spaces. The following code illustrates this situation:

```
float y[128];

int main() {
  #pragma omp target device(cuda)
  #pragma omp task copy_inout(y)
  do_computation_GPU(y);

  #pragma omp taskwait noflush
  float value0 = y[64]; // task has finished, but 'y' may not contain updated data

  #pragma omp taskwait
  float value1 = y[64]; // contains the computed values

  return 0;
}
```

The assignment `value0 = y[64]` may not use the results computed by the previous task since the `noflush` clause instructs the underlying run-time to not trigger data transfers from separate address spaces. The following access (`value1 = y[64]`) after the taskwait (withouth the `noflush` clause) will access the updated values.

The taskwait's dependence clauses (`in, out, inout and on`) can also be used to synchronize specific pieces of data instead of synchronizing the whole set of currently tracked memory. The following code shows an example of this scenario:

```
float x[128];
float y[128];

int main () {
```

```
  #pragma omp target device(cuda)
  #pragma omp task inout(x) copy_deps // implicit copy_inout(x)
  do_computation_GPU(x);

  #pragma omp target device(cuda)
  #pragma omp task inout(y) copy_deps // implicit copy_inout(y)
  do_computation_GPU(y);

  #pragma omp taskwait on(y)
  float value0 = x[64]; // may be a not updated value
  float value1 = y[64]; // this value has been updated

  ...
}
```

The value read by the definition of `value0` corresponds to the value already computed by one of the previous generated tasks (i.e. the one annotated with `inout(y) copy_deps`). However, the value read by the definition of `value1` may not corresponds with the updated value of `x`. The `taskwait` in this code only synchronizes data referenced in the clause `on` (i.e. the array `y`).

1.2.3 Data sharing rules

TBD .. ticket #17

1.2.4 Asynchronous execution

The most notable difference from OmpSs to OpenMP is the absence of the `parallel` clause in order to specify where a parallel region starts and ends. This clause is required in OpenMP because it uses a fork-join execution model where the user must specify when parallelism starts and ends. OmpSs uses the model implemented by StarSs where parallelism is implicitly created when the application starts. Parallel resources can be seen as a pool of threads–hence the name, thread-pool execution model–that the underlying run-time will use during the execution. The user has no control over this pool of threads, so the standard OpenMP methods `omp_get_num_threads()` or its variants are not available to use.

OmpSs allows the expression of parallelism through tasks. Tasks are independent pieces of code that can be executed by the parallel resources at run-time. Whenever the program flow reaches a section of code that has been declared as task, instead of executing the task code, the program will create an instance of the task and will delegate the execution of it to the OmpSs run-time environment. The OmpSs run-time will eventually execute the task on a parallel resource.

Another way to express parallelism in OmpSs is using the clause `for`. This clause has a direct counterpart in OpenMP and in OmpSs has an almost identical behavior. It must be used in conjunction with a for loop (in C or C++) and it encapsulates the iterations of the given for loop into tasks, the number of tasks created is determined by the OmpSs run-time, however the user can specify the desired scheduling with the clause `schedule`.

Any directive that defines a task or a series of tasks can also appear within a task definition. This allows the definition of multiple levels of parallelism. Defining multiple levels of parallelism can lead to a better performance of applications, since the underlying OmpSs run-time environment can exploit factors like data or temporal locality between tasks. Supporting multi-level parallelism is also required to allow the implementation of recursive algorithms.

Synchronizing the parallel tasks of the application is required in order to produce a correct execution, since usually some tasks depend on data computed by other tasks. The OmpSs programming model offers two ways of expressing this: data dependencies, and explicit directives to set synchronization points.

1.2.5 Dependence flow

Asynchronous parallelism is enabled in OmpSs by the use data-dependencies between the different tasks of the program. OmpSs tasks commonly require data in order to do meaningful computation. Usually a task will use some input data to perform some operations and produce new results that can be later on be used by other tasks or parts of the program.

When an OmpSs programs is being executed, the underlying runtime environment uses the data dependence information and the creation order of each task to perform dependence analysis. This analysis produces execution-order constraints between the different tasks which results in a correct order of execution for the application. We call these constraints task dependences.

Each time a new task is created its dependencies are matched against of those of existing tasks. If a dependency, either Read-after-Write (RaW), Write-after-Write (WaW) or Write-after-Read(WaR), is found the task becomes a successor of the corresponding tasks. This process creates a task dependency graph at runtime. Tasks are scheduled for execution as soon as all their predecessor in the graph have finished (which does not mean they are executed immediately) or at creation if they have no predecessors.

The OpenMP task construct is extended with the `in` (standing for input), `out` (standing for output), `inout` (standing for input/output), `concurrent` and `commutative` clauses to this end. They allow to specify for each task in the program what data a task is waiting for and signaling is readiness. Note that whether the task really uses that data in the specified way its the programmer responsibility. The meaning of each clause is explained below:

- `in(memory-reference-list)`: If a task has an `in` clause that evaluates to a given lvalue, then the task will not be eligible to run as long as a previously created sibling task with an `out`, `inout`, `concurrent` or `commutative` clause applying to the same lvalue has not finished its execution.
- `out(memory-reference-list)`: If a task has an `out` clause that evaluates to a given lvalue, then the task will not be eligible to run as long as a previously created sibling task with an `in`, `out`, `inout concurrent` or `commutative` clause applying to the same lvalue has not finished its execution.
- `inout(memory-reference-list)`: If a task has an `inout` clause that evaluates to a given lvalue, then it is considered as if it had appeared in an `in` clause and in an `out` clause. Thus, the semantics for the `in` and `out` clauses apply.
- `concurrent(memory-reference-list)`: the `concurrent` clause is a special version of the `inout` clause where the dependencies are computed with respect to `in`, `out`, `inout` and `commutative` but not with respect to other concurrent clauses. As it relaxes the synchronization between tasks users must ensure that either tasks can be executed concurrently either additional synchronization is used.
- `commutative(memory-reference-list)`: If a task has a `commutative` clause that evaluates to a given lvalue, then the task will not become a member of the commutative task set for that lvalue as long as a previously created sibling task with an `in`, `out`, `inout` or `concurrent` clause applying to the same lvalue has not finished its execution. Given a non-empty commutative task set for a certain lvalue, any task of that set may be eligible to run, but just one at the same time. Thus, tasks that are members of the same commutative task set are executed keeping mutual exclusion among them.

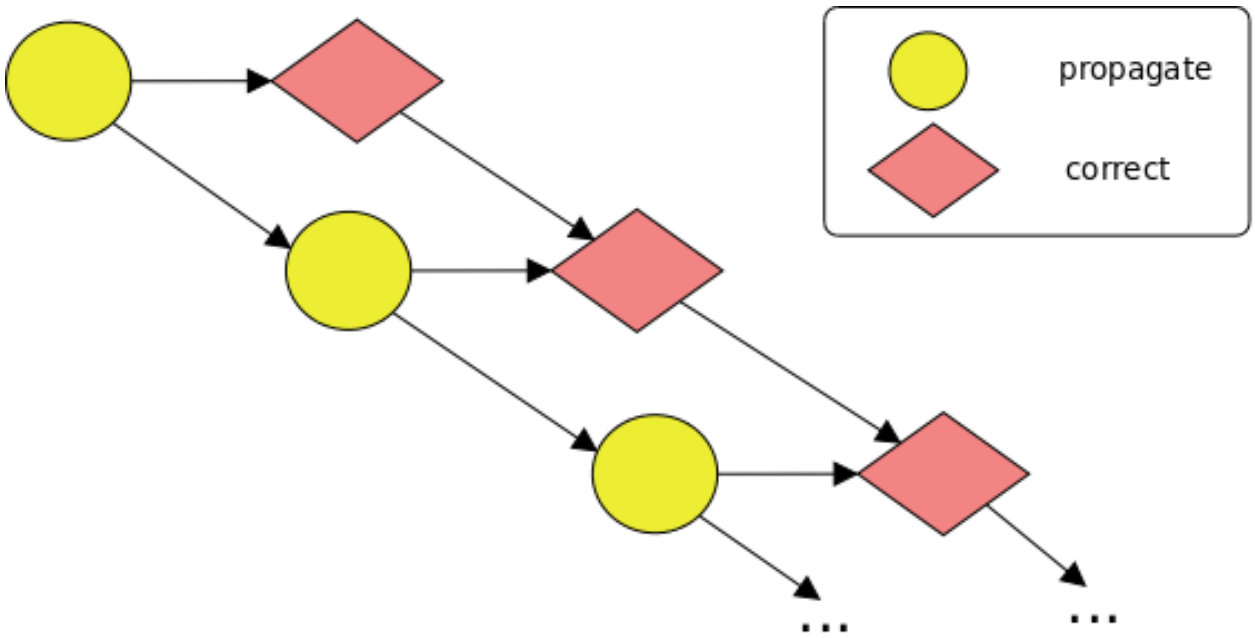
All of these clauses receive a list of memory references as argument. The syntax permitted to specify memory references is described in Language section. The data references provided by these clauses are commonly named the data dependencies of the task.

Note: For compatibility with earlier versions of OmpSs, you can use clauses `input` and `output` with exactly the same semantics of clauses `in` and `out` respectively.

The following example shows how to define tasks with task dependences:

```
void foo (int *a, int *b) {
  for (int i = 1; i < N; i++) {
#pragma omp task in(a[i-1]) inout(a[i]) out(b[i])
      propagate(&a[i-1], &a[i], &b[i]);

#pragma omp task in(b[i-1]) inout(b[i])
       correct(&b[i-1], &b[i]);
  }
}
```

This code generates at runtime the following task graph:



The following example shows how we can use the `concurrent` clause to parallelize a reduction over an array:

```
#include<stdio.h>
#define N 100

void reduce(int n, int *a, int *sum) {
  for (int i = 0; i < n; i++) {
#pragma omp task concurrent(*sum)
    {
#pragma omp atomic
      *sum += a[i];
    }
  }
}

int main() {
  int i, a[N];
  for (i = 0; i < N; ++i)
      a[i] = i + 1;

  int sum = 0;
  reduce(N, a, &sum);

#pragma omp task in(sum)
```

```
printf("The sum of all elements of 'a' is: %d\n", sum);

#pragma omp taskwait
  return 0;
}
```

Note that all tasks that compute the sum of the values of 'a' may be executed concurrently. For this reason we have to protect with an atomic construct the access to the 'sum' variable. As soon as all of these tasks finish their execution, the task that prints the value of 'sum' may be executed.

Note that the previous example can also be implemented using the `commutative` clause (we only show the reduce function, the rest of the code is exactly the same):

```
void reduce(int n, int *a, int *sum) {
  for (int i = 0; i < n; i++) {
#pragma omp task commutative(*sum)
      *sum += a[i];
  }
}
```

Note that using the `commutative` clause only one task of the commutative task set may be executed at the same time. Thus, we don't need to add any synchronization when we access to the 'sum' variable.

Extended lvalues

All dependence clauses allow extended lvalues from those of C/C++. Two different extensions are allowed:

- Array sections allow to refer to multiple elements of an array (or pointed data) in single expression. There are two forms of array sections:
  - `a[lower : upper]`. In this case all elements of 'a' in the range of lower to upper (both included) are referenced. If no lower is specified it is assumed to be 0. If the array section is applied to an array and upper is omitted then it is assumed to be the last element of that dimension of the array.
  - `a[lower; size]`. In this case all elements of 'a' in the range of lower to lower+(size-1) (both included) are referenced.
- Shaping expressions allow to recast pointers into array types to recover the size of dimensions that could have been lost across function calls. A shaping expression is one or more `[size]` expressions before a pointer.

The following example shows examples of these extended expressions:

```
void sort(int n, int *a) {
  if (n < small) seq_sort(n, a);

#pragma omp task inout(a[0:(n/2)-1]) // equivalent to inout(a[0;n/2])
       sort(n/2, a);
#pragma omp task inout(a[n/2:n-1]) // equivalent to inout(a[n/2;n-(n/2)])
       sort(n/2, &a[n/2]);
#pragma omp task inout(a[0:(n/2)-1], a[n/2:n-1])
       merge (n/2, a, a, &a[n/2]);
#pragma omp taskwait
}
```

Note: Our current implementation only supports array sections that completely overlap. Implementation support for partially overlapped sections is under development.

Note that these extensions are only for C/C++, since Fortran supports, natively, array sections. Fortran array sections are supported on any dependence clause as well.

Dependences on the taskwait construct

In addition to the dependencies mechanism, there is a way to set synchronization points in an OmpSs application. These points are defined using the `taskwait` directive. When the control flow reaches a synchronization point, it waits until all previously created sibling tasks complete their execution.

OmpSs also offers synchronization point that can wait until certain tasks are completed. These synchronization points are defined adding any kind of task dependence clause to the taskwait construct.

In the following example we have two tasks whose execution is serialized via dependences: the first one produces the value of `x` whereas the second one consumes `x` and produces the value of `y`. In addition to this, we have a taskwait construct annotated with an input dependence over `x`, which enforces that the execution of the region is suspended until the first task complete execution. Note that this construct does not introduce any restriction on the execution of the second task:

```
int main() {
  int x = 0, y = 2;

  #pragma omp task inout(x)
  x++;

  #pragma omp task in(x) inout(y)
  y -= x;

  #pragma omp taskwait in(x)
  assert(x == 1);

  // Note that the second task may not be executed at this point

  #pragma omp taskwait
  assert(x == y);
}
```

Note: For compatibility purposes we also support the `on` clause on the taskwait construct, which is an alias of `inout`.

Multidependences

Multidependences is a powerful feature that allow us to define a dynamic number of any kind of dependences. From a theoretical point of view, a multidependence consists on two different parts: first, an lvalue expression that contains some references to an identifier (the iterator) that doesn't exist in the program and second, the definition of that identifier and its range of values. Depending on the base language the syntax is a bit different:

- `dependence-type({memory-reference-list, iterator-name = lower; size})` for C/C++.
- `dependence-type([memory-reference-list, iterator-name = lower, size])` for Fortran.

Despite having different syntax for C/C++ and Fortran, the semantics of this feature is the same for the 3 languages: the lvalue expression will be duplicated as many times as values the iterator have and all the references to the iterator will be replaced by its values.

The folllowing code shows how to define a multidependence in C/C++:

```
void foo(int n, int *v)
{
// This dependence is equivalent to inout(v[0], v[1], ..., v[n-1])
#pragma omp task inout({v[i], i=0;n})
    {
        int j;
        for (int j = 0; j < n; ++j)
            v[j]++;
    }
}
```

And a similar code in Fortran:

```
subroutine foo(n, v)
    implicit none
    integer :: n
    integer :: v(n)

    ! This dependence is equivalent to inout(v[1], v[2], ..., v[n])
    !$omp task inout([v(i), i=1, n])
        v = v + 1
    !$omp end task
end subroutine foo
```

Warning: Multidependences syntax may change in a future

1.2.6 Task scheduling

When the current executed task reaches a *task scheduling point*, the implementation may decide to switch from this task to another one from the set of eligible tasks. Task scheduling points may occur at the following locations:

- in a task generating code
- in a taskyield directive
- in a taskwait directive
- just after the completion of a task

The fact of switching from a task to a different one is known as *task switching* and it may imply to begin the execution of a non-previously executed task or resumes the execution of a partially executed task. Task switching is restricted in the following situations:

- the set of eligible tasks (at a given time) is initially formed by the set of tasks included in the ready task pool (at this time).
- once a tied task has been executed by a given thread, it can be only resumed by the very same thread (i.e. the set of eligible tasks for a thread does not include tied tasks that has been previously executed by a different thread).
- when creating a task with the if clause for which expression evaluated to false, the runtime must offer a mechanism to immediately execute this task (usually by the same thread that creates it).
- when executing in a final context all the encountered *task generating codes* will execute the task immediately after creating it as if it was a simple routine call (i.e. the set of eligible tasks in this situation is restricted to include only the newly generated task).

Note: Remember that the *ready task pool* does not include tasks with dependences still not fulfilled (i.e. not all its predecessors have finished yet) or blocked tasks in any other condition (e.g. tasks executing a taskwait with non-finished child tasks).

1.2.7 Task reductions

The reduction clause allow us to define an asynchronous task reduction over a list of items. For each item, a private copy is created for each thread that participates in the reduction. At task synchronization (dependence or `taskwait`), the original list item is updated with the values of the private copies by applying the combiner associated with the `reduction-identifier`. Consequently, the scope of a reduction begins when the first reduction task is created and ends at a task synchronization point. This region is called the reduction domain.

The `taskwait` construct specifies a wait on the completion of child tasks in the context of the current task and combines all privately allocated list items of all child tasks associated with the current reduction domain. A `taskwait` therefore represents the end of a domain scope.

The following example computes the sum of all values of the nodes of a linked-list. The final result of the reduction is computed at the `taskwait`:

```
struct node_t {
  int val;
  struct node_t* next;
};

int sum_values(struct node_t* node) {
  int red=0;
  struct node_t* current = node;

  while (current != 0) {
#pragma omp task reduction(+: red)
    {
      red += current->val;
    }
    node = node->next;
  }
#pragma omp taskwait

  return red;
}
```

Data-flow based task execution allows a streamline work scheduling that in certain cases results in higher hardware utilization with relatively small development effort. Task-parallel reductions can be easily integrated into this execution model but require the following assumption. A list item declared in the task reduction directive is considered as if declared `concurrent` by the depend clause.

In the following code we can see an example where a reduction domain begins with the first occurrence of a participating task and is implicitly ended by a dependency of a successor task.:

```
#include<assert.h>

int main(int argc, char *argv[]) {

  const int SIZE  = 1024;
  const int BLOCK =   32;
  int array[SIZE];
```

```
  int i;

  for (i = 0; i < SIZE; ++i)
    array[i] = i+1;

  int red = 0;
  for (int i = 0; i < SIZE; i += BLOCK) {
#pragma omp task shared(array) reduction(+:red)
    {
      for (int j = i; j < i+BLOCK; ++j)
        red += array[j];
    }
  }
#pragma omp task in(red)
    assert(red == ((SIZE * (SIZE+1))/2));

#pragma omp taskwait
}
```

Nested task constructs typically occur in two cases. In the first, each task at each nesting level declares a reduction over the same variable. This is called multilevel reduction. It is important to point out that only task synchronization that occurs at the same nesting level at which a reduction scope was created (that is the nesting level that first encounter a reduction task for a list item), ends the scope and reduces private copies. Within the reduction domain, the value of the reduction variable is unspecified.

In the second occurrence each nesting level reduces over a different reduction variable. This happens for example if a nested task performs a reduction on task local data. In this case a `taskwait` at the end of each nesting level is required. We call this occurrence a nested-domain reduction.

1.2.8 Runtime Library Routines

OmpSs uses runtime library routines to set and/or check the current execution environment, locks and timing services. These routines are implementation defined and you can find a list of them in the correspondant runtime library user guide.

1.2.9 Environment Variables

OmpSs uses environment variables to configure certain aspects of its execution. The set of these variables is implementation defined and you can find a list of them in the correspondant runtime library user guide.

1.3 Language description

This section describes the OmpSs language, this is, all the necessary elements to understand how an application programmed in OmpSs executes and/or behaves in a parallel architecture. OmpSs provides a simple path for users already familiarized with the OpenMP programming model to easily write (or port) their programs to OmpSs.

This description is completely guided by the list of OmpSs directive constructs. In each of the following sections we will find a short description of the directive, its specific syntax, the list of its clauses (including the list of valid parameters for each clause and a short description for them). In addition, each section finalizes with a simple example showing how this directive can be used in a valid OmpSs program.

As is the case of OpenMP in C and C++, OmpSs directives are specified using the *#pragma* mechanism (provided by the base language) and in Fortran they are specified using special comments that are identified by a unique sentinel. The sentinel used in OmpSs (as is the case of OpenMP) is *omp*. Compilers will typically ignore OmpSs directives if support is disabled or not provided.

1.3.1 Task construct

The programmer can specify a task using the `task` construct. This construct can appear inside any code block of the program, which will mark the following statement as a task.

The syntax of the `task` construct is the following:

```
#pragma omp task [clauses]
structured-block
```

The valid clauses for the `task` construct are:

- private(<list>)
- firstprivate(<list>)
- shared(<list>)
- depend(<type>: <memory-reference-list>)
- <depend-type>(<memory-reference-list>)
- reduction(<operator>: <memory-reference-list>)
- priority(<value>)
- if(<scalar-expression>)
- final(<scalar-expresion>)
- label(<string>)
- tied

The `private` and `firstprivate` clauses declare one or more list items to be private to a task (i.e. the task receives a new list item). All internal references to the original list item are replaced by references to this new list item.

List items privatized using the `private` clause are uninitialized when the execution of the task begins. List items privatized using the `firstprivate` clause are initialized with the value of the corresponding original item at task creation.

The `shared` clause declare one or more list items to be shared to a task (i.e. the task receives a reference to the original list item). The programmer must ensure that shared storage does not reach the end of its lifetime before tasks referencing this storage have finished.

The `depend` clause allows to infer additional task scheduling restrictions from the parameters it defines. These restrictions are known as dependences. The syntax of the `depend` clause include a dependence type, followed by colon and its associated list items. The list of valid type of dependences are defined in section "Dependence flow" in the previous chapter. In addition to this syntax, OmpSs allows to specify this information using as the name of the clause the type of dependence. Then, the following code:

```
#pragma omp task depend(in: a,b,c) depend(out: d)
```

Is equivalent to this one:

```
#pragma omp task in(a,b,c) out(d)
```

The `reduction` clause allows to define the task as a participant of a reduction operation. The first occurrence of a participating task defines the begin of the scope for the reduction. The end of the scope is implicitly ended by a `taskwait` or a dependence over the `memory-reference-item`.

If the expression of the `if` clause evaluates to *true*, the execution of the new created task can be deferred, otherwise the current task must suspend its execution until the new created task has complete its execution.

If the expression of the `final` clause evaluates to *true*, the new created task will be a final tasks and all the *task generating code* encountered when executing its *dynamic extent* will also generate final tasks. In addition, when executing within a final task, all the encountered *task generating codes* will execute these tasks immediately after its creation as if they were simple routine calls. And finally, tasks created within a final task can use the data environment of its parent task.

The `tied` clause defines a new task scheduling restriction for the newly created tasks. Once a thread begins the execution of this task, the task becomes tied to this thread. In the case this task has suspended its execution by any *task scheduling point* only the same thread (i.e. the thread to which the task is tied to) may resume its execution.

The `label` clause defines a string literal that can be used by any performance or debugger tool to identify the task with a more *human-readable* format.

The following C code shows an example of creating tasks using the `task` construct:

```
float x = 0.0;
float y = 0.0;
float z = 0.0;

int main() {

  #pragma omp task
  do_computation(x);

  #pragma omp task
  {
    do_computation(y);
    do_computation(z);
  }

  return 0;
}
```

When the control flow reaches `#pragma omp task` construct, a new task instance is created, however when the program reaches `return 0` the previously created tasks may not have been executed yet by the OmpSs run-time.

The task construct is extended to allow the annotation of function declarations or definitions in addition to structured-blocks. When a function is annotated with the task construct each invocation of that function becomes a task creation point. Following C code is an example of how task functions are used:

```
extern void do_computation(float a);
#pragma omp task
extern void do_computation_task(float a);

float x = 0.0;
int main() {
   do_computation(x); //regular function call
   do_computation_task(x); //this will create a task
   return 0;
}
```

Invocation of `do_computation_task` inside `main` function create an instance of a task. As in the example above, we cannot guarantee that the task has been executed before the execution of the `main` finishes.

Note that only the execution of the function itself is part of the task not the evaluation of the task arguments. Another restriction is that the task is not allowed to have any return value, that is, the return must be void.

1.3.2 Target construct

To support heterogeneity a new construct is introduced: the target construct. The intent of the target construct is to specify that a given element can be run in a set of devices. The target construct can be applied to either a task construct or a function definition. In the future we will allow to allow it to work on worksharing constructs.

The syntax of the `target` construct is the following:

```
#pragma omp target [clauses]
task-construct  | function-definition | function-header
```

The valid clauses for the `target` construct are the following:

- device(target-device) - It allows to specify on which devices should be targeting the construct. If no device clause is specified then the SMP device is assumed. Currently we also support the CUDA device that allows the execution of native CUDA kernels in GPGPUs.
- copy_in(list-of-variables) - It specifies that a set of shared data may be needed to be transferred to the device before the associated code can be executed.
- copy_out(list-of-variables) - It specifies that a set of shared data may be needed to be transferred from the device after the associated code is executed.
- copy_inout(list-of-variables) - This clause is a combination of copy_in and copy_out.
- copy_deps - It specifies that if the attached construct has any dependence clauses then they will also have copy semantics (i.e., in will also be considered copy_in, output will also be considered copy_out and inout as copy_inout).
- implements(function-name) - It specifies that the code is an alternate implementation for the target devices of the function name specified in the clause. This alternate can be used instead of the original if the implementation considers it appropriately.

Additionally, both SMP and CUDA tasks annotated with the target construct are eligible for execution a cluster environment in an experimental implementation. Please, contact us if you are interested in using it.

1.3.3 Loop construct

When a task encounters a loop construct it starts creating tasks for each of the chunks in which the loop's iteration space is divided. Programmers can choice among different schedule policies in order to divide the iteration space.

The syntax of the `loop` construct is the following:

```
#pragma omp for [clauses]
loop-block
```

The valid clauses for the `loop` construct are the following:

- schedule(schedule-policy[, chunk-size]) - It specifies one of the valid partition policies and, optionally, the chunk-size used to divide the iteration space. Valid schedule policies are one the following options:
  - dynamic - loop is divided to team's threads in tasks of chunk-size granularity. Tasks are assigned as threads request them. Once a thread finishes the execution of one of these tasks it will request another task. Default chunk-size is 1.

  - guided - loop is divided as the executing threads request them. The chunk-size is proportional to the number of unassigned iterations, so it starts to be bigger at the beginning, but it becomes smaller as the loop execution progresses. Chunk-size will never be smaller than chunk-size parameter (except for the last iteration chunk).
  - static - loop is divided into chunks of size chunk-size. Each task is divided among team's threads following a round-robin fashion. If no chunk-size is provided all the iteration space is divided by number-of-threads chunks of the same size (or proximately the same size if number-of-threads does not divide number-of-iterations).

- nowait - When this option is specified the encountering task can immediately proceed with the following code without wait for all the tasks created to execute each of the loop's chunks.

Following C code shows an example on loop distribution:

```
float x[10];
int main() {
#pragma omp for schedule(static)
   for (int i = 0; i < 10; i++) {
      do_computation(x[i]);
   }
   return 0;
}
```

1.3.4 Taskwait construct

Apart from implicit synchronization (task dependences) OmpSs also offers mechanism which allow users to synchronize task execution. `taskwait` construct is an stand-alone directive (with no code block associated) and specifies a wait on the completion of all direct descendant tasks.

The syntax of the `taskwait` construct is the following:

```
#pragma omp taskwait [clauses]
```

The valid clauses for the `taskwait` construct are the following:

- on(list-of-variables) - It specifies to wait only for the subset (not all of them) of direct descendant tasks. `taskwait` with an `on` clause only waits for those tasks referring any of the variables appearing on the list of variables.

The `on` clause allows to wait only on the tasks that produces some data in the same way as in clause. It suspends the current task until all previous tasks with an out over the expression are completed. The following example illustrates its use:

```
int compute1 (void);

int compute2 (void);

int main()
{
  int result1, result2;

#pragma omp task out(result1)
  result1 = compute1();

#pragma omp task out(result2)
  result2 = compute2();
```

```
#pragma omp taskwait on(result1)
  printf("result1 = %d\n",result1);

#pragma omp taskwait on(result2)
  printf("result2 = %d\n",result2);

  return 0;
}
```

1.3.5 Taskyield directive

The `taskyield` directive specifies that the current task can be suspended and the scheduler runtime is allowed to scheduler a different task. The `taskyield` explicitly includes a *task schedule point*.

The syntax of the `taskyield` directive is the following:

```
#pragma omp taskyield
```

The `taskyield` directive has no related clauses.

In the following example we can see how to use the `taskyield` directive:

```
void compute ( void ) {
   int a=0,b=0;

   #pragma omp task shared(a)
   { a++;}

   #pragma omp taskyield

   #pragma omp task shared(b)
   { b++; }

   #pragma omp taskwait
}

int main() {
   #pragma omp task
   compute();

   #pragma omp taskwait
   return 0;
}
```

When encountering the `taskyield` directive the runtime system may decide among continue execute the task compute (i.e. the current task) or begins the execution of the a++ task (if not yet executed).

1.3.6 Atomic construct

The atomic construct ensures that following expression is executed atomically. Runtime systems will use native machine mechanism to guarantee atomic execution. If there is no native mechanism to guarantee atomicity (e.g. function call) it will use a regular critical section to implement the atomic construct.

The syntax of the `atomic` construct is the following:

```
#pragma omp atomic
structured-block
```

Atomic construct has no related clauses.

1.3.7 Critical construct

The `critical` construct allows programmers to specify regions of code that will be executed in mutual exclusion. The associated region will be executed by a single thread at a time, other threads will wait at the beginning of the critical section until no thread in the team was executing it.

The syntax of the `critical` construct is the following:

```
#pragma omp critical
structured-block
```

Critical construct has no related clauses.

1.3.8 Declare reduction construct

The user can define its own reduction-identifier using the `declare reduction` directive. After declaring the UDR, the reduction-identifier can be used in a reduction clause. The syntax of this directive is the following one:

```
#pragma omp declare reduction(reduction-identifier : type-list : combiner-expr)␣
↪[initializer(init-expr)]
```

where:

- *reduction-identifier* is the identifier of the reduction which is being declared
- *type-list* is a list of types
- *combiner-expr* is the expression that specifies how we have to combine the partial results. We can use two predefined identifiers in this expression: *omp_out* and *omp_in*. The *omp_out* identifier is used to represent the result of the combination whereas the *omp_in* identifier is used to represent the input data.
- *init-expr* is the expression that specifies how we have to initialize the private copies. We can use also two predefined identifiers in this expression: *omp_priv* and *omp_orig*. The *omp_priv* identifier is used to represent a private copy whereas the *omp_orig* identifier is used to represent the original variable that is being involved in a reduction.

In the following example we can see how we declare a UDR and how we use it:

```
struct C {
    int x;
};

void reducer(struct C* out, struct C* in) {
    (*out).x += (*in).x;
}

#pragma omp declare reduction(my_UDR : struct C : reducer(&omp_out,&omp_in))␣
↪initializer(omp_priv = {0})

int main() {
    struct C res = { 0 };
    struct C v[N];
```

```
    init(&v);

    for (int i = 0; i < N; ++i) {
#pragma omp task reduction(my_UDR : res) in(v) firstprivate(i)
        {
           res.x += v[i].x;
        }
    }
#pragma omp taskwait
}
```

CHAPTER

TWO

OMPSS USER GUIDE

The information included in this document is provided "as is", with no warranties whatsoever, including any warranty of merchantability, fitness for any particular purpose, or any warranty otherwise arising out of any proposal, specification, or sample. The document is not guaranteed to be complete and/or error-free at this stage and it is subject to changes without furthernotice. Barcelona Supercomputing Center will not assume any responsibility for errors or omissions in this document. Please send comments, corrections and/or suggestions to pm-tools at bsc.es. This document is provided for informational purposes only.

Note: There is a PDF version of this document at http://pm.bsc.es/ompss-docs/user-guide/OmpSsUserGuide.pdf

2.1 Installation of OmpSs

2.1.1 Preparation

You must first choose a directory where you will install OmpSs. In this document this directory will be referred to as `TARGET`. We recommend you to set an environment variable `TARGET` with the desired installation directory. For instance:

```
$ export TARGET=$HOME/ompss
```

2.1.2 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. Configure it:

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

   Note: Extrae may have a number of dependences. Do not hesitate to check the Extrae User's Manual

4. Build and install:

```
$ make
$ make install
```

Now you can proceed to the *Installation of Nanos++*. Do not forget to pass `--with-extrae=$TARGET` to Nanos++ `configure`.

2.1.3 Installation of Nanos++

1. First, make sure that you fulfill all the *Nanos++ build requirements*.
2. Get the latest Nanos++ *tarball* (`nanox-version-yyyy-mm-dd.tar.gz`) from https://pm.bsc.es/ompss-downloads Unpack the file and enter the just created directory:

```
$ tar xfz nanox-version-yyyy-mm-dd.tar.gz
$ cd nanox-version
```

3. Run `configure`. There are a number of flags that enable or disable different features of Nanos++. Make sure you pass `--prefix` with the destination directory of your OmpSs installation (in our case `TARGET`).

   Important: If you want instrumentation support, you must add `--with-extrae=dir` to your *configure-flags* (see below), where `dir` is the directory where you installed Extrae (usually `TARGET`).

   Check *Nanos++ configure flags* for more information about *configure-flags*.

```
$ ./configure --prefix=$TARGET configure-flags
```

   Note: You can pass an empty set of *configure-flags*.

   Hint: `configure` prints lots of output, but a small summary of enabled features is printed at the end. You may want to check it to ensure you are correctly passing all the flags

4. Build

```
$ make
```

   This may take some time. You may build in parallel using `make -jN` where `N` is the maximum number of threads you want to use for parallel compilation.

5. Install

```
$ make install
```

   This will install Nanos++ in `TARGET`.

Now you can proceed to *Installation of Mercurium C/C++/Fortran source-to-source compiler*.

Nanos++ build requirements

There are a number of software requirements to successfully build Nanos++ from the source code.

Important: Additional software is needed if you compile from the git repository. This section details the requirements when building from a *tarball* (a *tar.gz* file).

- A supported platform running Linux (i386, x86-64, ARM, PowerPC or IA64).
- GNU C/C++ compiler versions 4.4 or better.

If you want CUDA support:

- CUDA 5.0 or better.

If you want to enable the cluster support in Nanos++ you will need:

- GASNet 1.14.2 or better

Nanos++ configure flags

By default Nanos++ compiles three versions: *performance*, *debug* and *instrumentation*. Which one is used is usually governed by flags to the Mercurium compiler. You can speedup the build of the library by selectively disabling these versions.

You can also enable a fourth version: *instrumentation-debug*. That one is probably of little interest to regular users of Nanos++ as it enables debug and instrumentation at the same time.

--disable-instrumentation Disables generation of instrumentation version

--disable-debug Disables generation of debug version

--disable-performance Disables generation of performance version

--enable-instrumentation-debug Enables generation of instrumentation-debug version

Besides the usual shared memory environment (which is called the *SMP* device), Nanos++ supports several devices. Such devices are automatically enabled if enough support is detected at the host. You can disable them with the following flags.

--disable-gpu-arch Disables CUDA support

--disable-opencl-arch Disables OpenCL support

Nanos++ includes several plugins that are able to use other software packages. You can enable them using the following flags.

--with-cuda=dir Directory of CUDA installation. By default it checks `/usr/local/cuda`. If a suitable installation of CUDA is found, CUDA support is enabled in Nanos++ (unless you disable it)

--with-opencl=dir Directory of OpenCL installation. By default it checks `/usr`. If a suitable installation of OpenCL is found, OpenCL support is enabled in Nanos++ (unless you disable it)

--with-opencl-include=dir If you use –with-opencl=dir, `configure` assumes that dir/`include` contains the headers. Use this flag to override this assumption.

--with-opencl-lib=dir If you use –with-opencl=dir, `configure` assumes that dir/`lib` contains the libraries. Use this flag to override this assumption.

--with-extrae=dir Directory of Extrae installation. **This is mandatory if you want instrumentation**. Make sure you have already installed Extrae first. See *Installation of Extrae (optional)*

--with-mpitrace=dir This is a deprecated name for `--with-extrae`

--with-nextsim=dir	Directory of NextSim installation
--with-ayudame=dir	Directory of Ayudame installation
--with-hwloc=dir	Directory of Portable Hardware Locality (hwloc) installation. This is highly recommended for NUMA setups
--with-chapel=dir	Directory of Chapel installation
--with-mcc=dir	Directory of Mercurium compiler. *This is only for testing Nanos++ itself and only useful to Nanos++ developers*.

2.1.4 Installation of Mercurium C/C++/Fortran source-to-source 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/blob/master/README.md

Once you complete all the steps listed in the link above you should be ready to *Compile OmpSs programs*.

2.2 Compile OmpSs programs

After the *Installation of OmpSs* you can now start compiling OmpSs programs. For that end you have to use the Mercurium compiler (which you already installed as described in *Installation of Mercurium C/C++/Fortran source-to-source compiler*).

See sections *Compiling an OmpSs program with CUDA tasks* and *Compiling an OmpSs program with OpenCL tasks* for more information. .. index:

```
double: Mercurium; drivers
```

2.2.1 Drivers

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

Common flags accepted by Mercurium drivers

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.

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
...
```

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.

2.2.2 Compiling an OmpSs program for shared-memory

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

Your first OmpSs program

Following is a very simple OmpSs program in C:

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

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

```
    }
#pragma omp taskwait
    return 0;
}
```

Compile it using *mcc*:

```
$ mcc -o test --ompss test.c
Nanos++ prerun
Nanos++ phase
```

Important: Do not forget the flag `--ompss` otherwise the OmpSs directives will be ignored.

And run it:

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

2.2.3 Compiling an OmpSs program with CUDA tasks

To compile an OmpSs + CUDA program you must use one of the OmpSs/OmpSs-2 profiles (see *Drivers*) and the `--cuda` flag. Note that the Nanos++ installation that was provided to Mercurium must have been configured with CUDA (see *Nanos++ configure flags*).

The usual structure of an OmpSs program with CUDA tasks involves a set of C/C++ files (usually `.c` or `.cpp`) and a set of CUDA files (`.cu`). You can use one of our profiles to compile the CUDA files, but the driver will simply call the NVIDIA Cuda Compiler `nvcc`. We discourage mixing C/C++ code and CUDA code in the same file when using Mercurium. While this was supported in the past, such approach is regarded as deprecated. The recommended approach is to move the CUDA code to a `.cu` file.

Note: When calling kernels from C, make sure the `.cu` file contains an `extern "C"`, otherwise C++ mangling in the name of the kernels will cause the link step to fail.

Warning: Fortran support for CUDA is experimental. Currently you cannot pass a `.cu` file to Mercurium: compile them separatedly using `nvcc`.

Consider the following CUDA kernels:

```
/* cuda-kernels.cu */

extern "C" {  // We specify extern "C" because we will call them from a C code

  __global__ void init(int n, int *x)
  {
      int i = blockIdx.x * blockDim.x + threadIdx.x;
      if (i >= n)
          return;
      x[i] = i;
  }
```

```
  __global__ void increment(int n, int *x)
  {
      int i = blockIdx.x * blockDim.x + threadIdx.x;
      if (i >= n)
          return;
      x[i]++;
  }

} /* extern "C" */
```

You can use the following OmpSs program in C to call them:

```
/* cuda-test.c */

#include <stdio.h>
#include <assert.h>

#pragma omp target device(cuda) copy_deps ndrange(1, n, 1)
#pragma omp task out(x[0 : n-1])
__global__ void init(int n, int *x);

#pragma omp target device(cuda) copy_deps ndrange(1, n, 1)
#pragma omp task inout(x[0 : n-1])
__global__ void increment(int n, int *x);

int main(int argc, char *argv[])
{
    int x[10];

    init(10, x);
    increment(10, x);

#pragma omp taskwait

    int i;
    for (i = 0; i < 10; i++)
    {
        fprintf(stderr, "x[%d] == %d\n", i, x[i]);
        assert(x[i] == (i+1));
    }

    return 0;
}
```

Compile these files using `mcc`:

```
$ mcc -o cuda-test cuda-test.c  cuda-kernels.cu  --ompss --cuda
cuda-test.c:5: note: adding task function 'init'
cuda-test.c:9: note: adding task function 'increment'
cuda-test.c:16: note: call to task function 'init'
cuda-test.c:17: note: call to task function 'increment'
cudacc_cuda-test.cu: info: enabling experimental CUDA support
```

Note: You can compile this example separatedly, as usual: using `-c` and linking the intermediate `.o` files.

Now you can run it:

```
$ ./cuda-test
x[0] == 1
x[1] == 2
x[2] == 3
x[3] == 4
x[4] == 5
x[5] == 6
x[6] == 7
x[7] == 8
x[8] == 9
x[9] == 10
MSG: [12] GPU 0 TRANSFER STATISTICS
MSG: [12]     Total input transfers: 0 B
MSG: [12]     Total output transfers: 0 B
MSG: [12]     Total device transfers: 0 B
MSG: [13] GPU 1 TRANSFER STATISTICS
MSG: [13]     Total input transfers: 0 B
MSG: [13]     Total output transfers: 40 B
MSG: [13]     Total device transfers: 0 B
```

Note that due to the caching mechanism of Nanos++ only 40 bytes (10 integers) have been transferred from the GPU to the host. See *Running OmpSs Programs* for detailed information about options affecting the execution of OmpSs programs.

2.2.4 Compiling an OmpSs program with OpenCL tasks

To compile an OmpSs + OpenCL program you must use one of the OmpSs/OmpSs-2 profiles (see *Drivers*) and the `--opencl` flag. Note that the Nanos++ installation that was provided to Mercurium must have been configured with OpenCL (see *Nanos++ configure flags*).

Consider this OpenCL file with kernels similar to the CUDA example above:

```
/* ocl_kernels.cl */
__kernel void init(int n, int  __global * x)
{
    int i = get_global_id(0);
    if (i >= n)
        return;
    x[i] = i;
}

__kernel void increment(int n, int  __global * x)
{
    int i = get_global_id(0);
    if (i >= n)
        return;
    x[i]++;
}
```

We can call these kernels from the OmpSs program:

```
#include <stdio.h>
#include <assert.h>

#pragma omp target device(opencl) copy_deps ndrange(1, n, 8) file(ocl_kernels.cl)
#pragma omp task out(x[0 : n-1])
void init(int n, int *x);
```

```
#pragma omp target device(opencl) copy_deps ndrange(1, n, 8) file(ocl_kernels.cl)
#pragma omp task inout(x[0 : n-1])
void increment(int n, int *x);

int main(int argc, char *argv[])
{
    int x[10];

    init(10, x);
    increment(10, x);

#pragma omp taskwait

    int i;
    for (i = 0; i < 10; i++)
    {
        fprintf(stderr, "x[%d] == %d\n", i, x[i]);
        assert(x[i] == (i+1));
    }

    return 0;
}
```

Note: OpenCL files must be passed to the driver if they appear in `file` directives. This also applies to separate compilation using `-c`.

Compile the OmpSs program using `mcc`:

```
$ mcc  -o ocl_test ocl_test.c ocl_kernels.cl --ompss
ocl_test.c:5: note: adding task function 'init'
ocl_test.c:9: note: adding task function 'increment'
ocl_test.c:18: note: call to task function 'init'
ocl_test.c:19: note: call to task function 'increment'
```

Now you can execute it:

```
$ ./ocl_test
x[0] == 1
x[1] == 2
x[2] == 3
x[3] == 4
x[4] == 5
x[5] == 6
x[6] == 7
x[7] == 8
x[8] == 9
x[9] == 10
MSG: [12] OpenCL dev0 TRANSFER STATISTICS
MSG: [12]     Total input transfers: 0 B
MSG: [12]     Total output transfers: 0 B
MSG: [12]     Total dev2dev(in) transfers: 0 B
MSG: [13] OpenCL dev1 TRANSFER STATISTICS
MSG: [13]     Total input transfers: 0 B
MSG: [13]     Total output transfers: 40 B
MSG: [13]     Total dev2dev(in) transfers: 0 B
```

Again, due to the caching mechanism of Nanos++, only 40 bytes (10 integers) have been transferred from the OpenCL device (in the example above it is a GPU) to the host.

2.2.5 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.

How can you help us to solve the problem quicker?

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@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.

2.3 Running OmpSs Programs

Nanos++ is an extensible runtime library designed to serve as a runtime support in parallel environments. It is mainly used to support OmpSs (an extension to the OpenMP programming model) developed at BSC. Nanos++ also has modules to support OpenMP and Chapel.

With Nanox++ also comes a small application offering a quick help about the runtime common options. It lists the available configuration options and modules. For each module also lists all configuration flags available. This application is installed in the NANOX_INSTALL_DIR/bin directory and it is called 'nanox'. It accepts several command line options:

--help	Shows command line and environment variables available.
--list-modules	Shows the modules that are available with the current installation of Nanos++.

--version Shows the installed version of Nanos++ and the configure command line used to install it.

Some parts of the runtime come in the form of modules that are loaded on demand when the library is initialized. Each module can have its own particular options which can be set using also command-line parameters. In order to see the list of available modules and options use the nanox tool. As in the case of the runtime options you can also choose among all available plugins using a environment variable (or the equivalent command-line option like using NX_ARGS).

2.3.1 Runtime Options

Nanos++ has different configuration options which can be set through environment variables. These runtime parameters can be also passed as a command-line option like through the environment variable NX_ARGS. For example, these two invocations of a Nanos++ application are equivalent:

```
$ export NX_FANCY_OPTION=whatever
$./myProgram

$ export NX_ARGS='--fancy-option=whatever'
$ ./myProgram
```

General runtime options

Nanos++ have several common options which can be specified while running OmpSs's programs.

--smp-cpus=<n> Set number of requested CPUs.

--smp-workers=<n> Set number of SMP worker threads.

--cpus-per-socket=<n> Set number of CPUs per socket.

--num-sockets=<n> Set number of sockets available.

--hwloc-topology=<xml-file> Overrides hwloc's topology discovery and uses the one provided by *xml-file*.

--stack-size=<n> Defines default stack size for all devices.

--binding-start=<cpu-id> Set initial CPU for binding (binding required).

--binding-stride=<n> Set binding stride (binding required).

--disable-binding, --no-disable-binding Disables/enables thread binding (enabled by default).

--verbose, --no-verbose Activate verbose mode (requires Nanos++ debug version).

--disable-synchronized-start, --no-disable-synchronized-start Disables synchronized start (enabled by default).

--architecture=<arch> Defines default architecture. Where *arch* can be one of the following options: `smp`, `gpu`, `opencl` *or* `cluster`.

--disable-caches, --no-disable-caches Disables/enables the use of caches (enabled by default).

--cache-policy=<cache-policy> Set default cache policy. Where *cache-policy* can be one of the following options: `wt` (write through) *or* `wb` (write back).

--summary, --no-summary Enables/disables runtime summary mode, printing statistics at start/end phases (disabled by default)

CUDA specific options

This is the list of CUDA specific options:

NX_ARGS option	Environment variable	Description
–disable-cuda	NX_DISABLECUDA	Enable or disable the use of GPUs with CUDA
–gpus	NX_GPUS	Defines the maximum number of GPUs to use
–gpu-warmup	NX_GPUWARMUP	Enable or disable warming up the GPU before running user's code
–gpu-prefetch	NX_GPUPREFETCH	Set whether data prefetching must be activated or not
–gpu-overlap	NX_GPUOVERLAP	Set whether GPU computation should be overlapped with all data transfers, whenever possible, or not
–gpu-overlap-inputs	NX_GPUOVERLAP_INPUTS	Set whether GPU computation should be overlapped with host –> device data transfers, whenever possible, or not
–gpu-overlap-outputs	NX_GPUOVERLAP_OUTPUTS	Set whether GPU computation should be overlapped with device –> host data transfers, whenever possible, or not
–gpu-max-memory	NX_GPUMAXMEM	Defines the maximum amount of GPU memory (in bytes) to use for each GPU. If this number is below 100, the amount of memory is taken as a percentage of the total device memory
–gpu-cache-policy	NX_GPU_CACHE_POLICY	Defines the cache policy for GPU architectures: write-through / write-back / do not use cache
–gpu-cublas-init	NX_GPUCUBLASINIT	Enable or disable CUBLAS initialization

Following table summarizes valid and default values:

NX_ARGS option	Environment variable	Values	Default
–disable-cuda	NX_DISABLECUDA	yes / no	Enabled
–gpus	NX_GPUS	integer	All GPUs
–gpu-warmup	NX_GPUWARMUP	yes / no	Enabled
–gpu-prefetch	NX_GPUPREFETCH	yes / no	Disabled
–gpu-overlap	NX_GPUOVERLAP	yes / no	Disabled
–gpu-overlap-inputs	NX_GPUOVERLAP_INPUTS	yes / no	Disabled
–gpu-overlap-outputs	NX_GPUOVERLAP_OUTPUTS	yes / no	Disabled
–gpu-max-memory	NX_GPUMAXMEM	positive integer	No limit
–gpu-cache-policy	NX_GPU_CACHE_POLICY	wt/wb/nocache	wb
–gpu-cublas-init	NX_GPUCUBLASINIT	yes / no	Disabled

Cluster specific options

TBD

2.3.2 Running on Specific Architectures

The specific information related to the different architectures supported by OmpSs is explained here.

CUDA Architecture

Performance guidelines and troubleshooting advices related to OmpSs applications using CUDA are described here.

Tuning OmpSs Applications Using GPUs for Performance

In general, best performance results are achieved when prefetch and overlap options are enabled. Usually, a *write-back* cache policy also enhances performance, unless the application needs lots of data communication between host and GPU devices.

Other Nanox options for performance tuning can be found at *My application does not run as fast as I think it could*.

Running with cuBLAS (v2)

Since CUDA 4, the first parameter of any cuBLAS function is of type `cublasHandle_t`. In the case of OmpSs applications, this handle needs to be managed by Nanox, so `--gpu-cublas-init` runtime option must be enabled.

From application's source code, the handle can be obtained by calling `cublasHandle_t nanos_get_cublas_handle()` API function. The value returned by this function is the cuBLAS handle and it should be used in cuBLAS functions. Nevertheless, the cuBLAS handle is only valid inside the task context and should not be stored in a variable, as it may change over application's execution. The handle should be obtained through the API function inside all tasks calling cuBLAS.

Example:

```
#pragma omp target device (cuda) copy_deps
#pragma omp task inout([NB*NB]C) in([NB*NB]A, [NB*NB]B)
void matmul_tile (double* A, double* B, double* C, unsigned int NB)
{
    REAL alpha = 1.0;

    // handle is only valid inside this task context
    cublasHandle_t handle = nanos_get_cublas_handle();

    cublas_gemm(handle, CUBLAS_OP_T, CUBLAS_OP_T, NB, NB, NB, &alpha, A, NB, B, NB, &
↪alpha, C, NB);
}
```

GPU Troubleshooting

If the application has an unexpected behavior (either reports incorrect results or crashes), try using Nanox debug version (the application must be recompiled with flag `--debug`). Nanox debug version makes additional checks, so this version may trigger the cause of the error.

How to Understand GPU's Error Messages

When Nanox encounters an error, it aborts the execution throwing a **FATAL ERROR** message. In the case where the error is related to CUDA, the structure of this message is the following:

FATAL ERROR: [*#thread* **]** *what Nanox was trying to do when the error was detected* **:** *error reported by CUDA*

Note that the true error is the one reported by CUDA; not the one reported by the runtime. The runtime just gives information about the operation it was performing, but this does not mean that this operation caused the error. For

example, there can be an error launching a GPU kernel, but (unless this is checked by the user application) the runtime will detect the error at the next call to CUDA runtime that will probably be a memory copy.

Recommended Steps When an Error Is Found

1. Compile the application with `--debug` and `-ggdb3` flags, optionally use `-O0` flag, too.
2. Run normally and check if any **FATAL ERROR** is triggered. This will give a hint of what is causing the error. For getting more information, proceed to the following step.
3. Run the application inside gdb or cuda-gdb to find out the exact point where the application crashes and explore the backtrace, values of variables, introduce breakpoints for further analysis or any other tool you consider useful. You can also check *My application crashes. What can I do?* for further information.
4. If you think that the error is a bug in OmpSs, please report it.

Common Errors from GPUs

Here is a list of common errors found when using GPUs and how they can be solved.

- **No cuda capable device detected:** This means that the runtime is not able to find any GPU in the system. This probably means that GPUs are not detected by CUDA. You can check your CUDA installation or GPU drivers by trying to run any sample application from CUDA SDK, like *deviceQuery*, and see if GPUs are properly detected by the system or not.
- **All cuda capable devices are busy or unavailable:** Someone else is using the GPU devices, so Nanox cannot access them. You can check if there are other instances of Nanox running on the machine, or if there is any other application running that may be using GPUs. You will have to wait till this application finishes or frees some GPU devices. The *deviceQuery* example from CUDA SDK can be used to check GPU's memory state as it reports the available amount of memory for each GPU. If this number is low, it is most likely that another application is using that GPU device. This error is reported in CUDA 3 or lower. For CUDA 4 or higher an *Out of memory* error is reported.
- **Out of memory:** The application or the runtime are trying to allocate memory, but the operation failed because GPU's main memory is full. Nanox, by default allocates around 95% of each GPU device's memory (unless this value is modified using `--gpu-max-memory` runtime option). If the application needs additional device memory, try to tune the amount of memory that Nanox is allowed to use with `--gpu-max-memory` runtime option. This error is also displayed when there is someone else using the GPU device and Nanox cannot allocate device's memory. Refer to *All cuda capable devices are busy or unavailable* error to solve this problem.
- **Unspecified launch failure:** This usually means that a segmentation fault occurred on the device while running a kernel (an invalid or illegal memory position has been accessed by one or more GPU kernel threads). Please, check OmpSs source code directives related to dependencies and copies, compiler warnings and your kernel code.

Offload Architecture

Performance guidelines and troubleshooting advices related to OmpSs applications using offloaded codes are described here.

Offloading

In order to offload to remote nodes, the allocation of the nodes must be performed by the application by using provided API Calls.

From application's source code, the allocation can be done by calling `deep_booster_alloc(MPI_Comm spawners, int number_of_hosts, int process_per_host, MPI_Comm *intercomm)` API function. This function will create number_of_hosts*process_per_host remote workers and create a communicator with the offloaded processes in `intercomm`.

The masters who will spawn each process are the ones who are part of the communicator "spawners". This call is a collective operation and should be called by every process who is part of this communicator. Two values have been tested:

- MPI_COMM_WORLD: All the masters in the communicator will spawn number_of_hosts*process_per_host remote workers. All these workers can communicate using MPI (they are inside the same MPI_COMM_WORLD).
- MPI_COMM_SELF: Each master which calls the spawn will spawn number_of_hosts*process_per_host remote workers, only visible for himself. Workers spawned by the same node will be able to communicate using MPI, but they won't be able to communicate with the workers created by a different master.

This routine has different API calls (both work in C and also in Fortran (MPI_Comm/int/MPI_Comm* = INTEGER in Fortran, int* = INTEGER ARRAY in Fortran)):

- **deep_booster_alloc** *(MPI_Comm spawners, int number_of_hosts, int process_per_host, MPI_Comm *intercomm)*: Allocates process_per_host processes in each host. If there are not enough number of hosts, the call will fail and returned intercomm will be MPI_COMM_NULL.
- **deep_booster_alloc_list** *(MPI_Comm spawners, int pph_list_length, int* pph_list, MPI_Comm *intercomm)*: Provides a list with the number of processes which will be spawned in each node. For example, the following list {0,2,4,0,1,3} will spawn 10 processes split as indicated between hosts 1,2,4,5, skipping host 0 and 3. If there are not enough number of hosts, the call will fail and returned intercomm will be MPI_COMM_NULL.
- **deep_booster_alloc_nonstrict** *(MPI_Comm spawners, int number_of_hosts, int process_per_host, MPI_Comm *intercomm, int* provided)*: Allocates process_per_host processes in each host. If there are not enough number of hosts, the call will allocate as many as available and return the number of processes allocated (available_hosts*process_per_host) in "provided".
- **deep_booster_alloc_list_nonstrict** *(MPI_Comm spawners, int pph_list_length, int* pph_list, MPI_Comm *intercomm, int* provided)*: Provides a list with the number of processes which will be spawned in each node. For example, the following list {0,2,4,0,1,3} will spawn 10 processes split indicated between hosts 1,2,4,5, skipping host 0 and 3. If there are not enough number of hosts, the call will allocate as many as available and return the number of the number of processes allocated available in "provided".

Deallocation of the nodes will be performed automatically by the runtime at the end of the execution, however its strongly suggested to free them explicitly (it can be done at any time of the execution). This can be done by using the API function, `deep_booster_free(MPI_Comm *intercomm)`.

Communication between master and spawned processes (when executing offloaded tasks) in user code works, but it's not recommended.

Example in C:

```
int main (int argc, char** argv)
{
        int my_rank;
        int mpi_size;
        nanos_mpi_init(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD,&mpi_size);
```

```
        MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
        MPI_Comm boosters;
        int num_arr[2000];
        init_arr(num_arr,0,1000);
        //Spawn as one worker per master
        deep_booster_alloc(MPI_COMM_WORLD, mpi_size  , 1  , &boosters);
        if (boosters==MPI_COMM_NULL) exit(-1);

        //Each master will offload to the same rank than him, but in the workers
        #pragma omp task inout(num_arr[0;2000]) onto(boosters,my_rank)
        {
                        //Workers SUM all their num_arr[0,1000] into num_arr[1000,
↪2000]
                        MPI_Allreduce(&num_arr[0],&num_arr[1000],1000,MPI_INT,MPI_SUM,
↪MPI_COMM_WORLD);
        }
        #pragma omp taskwait

        print_arr(num_arr,1000,2000);

        deep_booster_free(&boosters);
        nanos_mpi_finalize();
        return 0;
}
```

Example in Fortran:

```
    PROGRAM MAIN
INCLUDE "mpif.h"
                    EXTERNAL :: SLEEP
                    IMPLICIT NONE
                    INTEGER :: BOOSTERS
                    INTEGER :: RANK, IERROR,MPISIZE, PROVIDED
                    INTEGER, DIMENSION(2000) :: inputs
                    INTEGER, DIMENSION(2000) :: outputs

                    !Initialize MPI with THREAD_MULTIPLE (required for offload)
                    call mpi_init_thread(MPI_THREAD_MULTIPLE,provided,ierror)
                    CALL MPI_COMM_RANK(MPI_COMM_WORLD, RANK, ierror)
                    CALL MPI_COMM_SIZE(MPI_COMM_WORLD, MPISIZE, ierror)

                    !!Spawn as one worker per master
                    CALL deep_booster_alloc(MPI_COMM_WORLD, MPISIZE, 1, BOOSTERS)
                    IF (boosters==MPI_COMM_NULL) THEN
                                    CALL exit(-1)
                    END IF
                    INPUTS=5
                    OUTPUTS=999

                    !!Each master will offload to the same rank than him, but in the␣
↪workers
                    !$OMP TASK IN(INPUTS) OUT(OUTPUTS) ONTO(BOOSTERS,RANK)
                                    CALL MPI_ALLREDUCE(INPUTS,OUTPUTS,2000,MPI_INT,
↪MPI_SUM,MPI_COMM_WORLD, IERROR);
                    !$OMP END TASK
                    !$OMP TASKWAIT
                    PRINT *, outputs(2)
```

```
                    CALL DEEP_BOOSTER_FREE(BOOSTERS)
                    CALL MPI_Finalize(ierror)
    END PROGRAM
```

Compiling OmpSs offload applications

OmpSs offload Applications can be compiled by using mercurium compilers, "mpimcc/mpimcxx" and "mpimfc" with –ompss flag (E.G.: mpimcc –ompss test.c), which are a wrapper for the current MPI implementation available in users PATH.

User MUST and provide use a multithread MPI implementation when offloading (otherwise nanox will give a explanatory warning and crash at start or hang) and link all the libraries of his program with the same one, -mt_mpi is automatically set for Intel MPI by our compiler.

If no OFFL_CC/OFFL_CXX/OFFL_FC environment variables are defined. OmpSs will use Intel (mpiicc, mpiicpc or mpiifort) compilers by default, and if they are not available, it fall back to OMPI/GNU ones (mpicc,mpicxx and mpicpc).

WARNING: When setting OFFL_CC/OFFL_FC, make sure that OFFL_CXX points to the same MPI implementation than OFFL_CC and OFFL_FC (as one C++ MPI-Dependant part of nanox is automatically compiled and linked with your application)

Naming Convention

MPI Offload executables have to follow a naming convention which identifies the executable and the architecture. Both executables have to be generated from the same source code. This convention is XX.YY, where XX is your executable name and must be the same for every architecture and YY is the architecture name (as given by 'uname -p' or 'uname -m' (both upper or lower case names are supported)). For x86_64 and k1om (MIC), two default alias are providen, intel64/INTEL64 and mic/MIC.

Example dual-architecture (MIC+Intel64) compilation:

```
mpimcc --ompss test.c -o test.intel64
mpimcc --ompss --mmic test.c -o test.mic
```

Offload hosts

In machines where there is no job manager integration with our offload, the hosts where to offload each process have to be specified manually in a host-file. The path of this file can be specified to Nanox in the environment variable `NX_OFFL_HOSTFILE`, like `NX_OFFL_HOSTFILE=./offload_hosts`

The syntax of this file is the following: hostA:n<env_var1,env_var2... hostB:n<env_var4,env_var1...

Example hostfile:

```
knights4-mic0:30<NX_SMP_WORKERS=1,NX_BINDING_STRIDE=4
knights3-mic0
knights2-mic0
knights1-mic0:3
knights0-mic0
```

Debugging your application

If you want to see where your offload workers are executed and more information about them, use the environment variable NX_OFFL_DEBUG=<level> level is a number in the range 1-5, higher numbers will show more debug information.

Sometimes offloading environment can be hard to debug and prints are not powerful enough, below you can find some techniques which will allow you to do so. You should be able to debug it with any MPI debugging tool if you point it to the right offload process ID. In order to do this you have two options.

1. Obtaining backtrace:

```
1- Compile your application with -k and -g and with --debug flag
2- Set "ulimit -c unlimited", after doing so, when the application crashes it␣
↪will dump the core.
3- Open it with "gdb ./exec corefile" and you will be able to see
some information about the crash (for example: see backtrace with "bt" command).
```

2. Execution time debugging:

```
1- Compile your application with -k and -g and with --debug flag
2- Add a sleep(X) at the start of the offload section which gives you enough time␣
↪to do the next steps:
3- Execute your application and make sure it is able to execute the offload␣
↪section/sleep.
4- ssh to the remote machine or execute everything in localhost (for debugging␣
↪purposes).
5- Look for the allocated/offloaded processes. (NX_OFFL_DEBUG=3 will help you to␣
↪identify their hostname and PID).
6- Attach to one of the processes with "gdb -pid YOURPID" (use your preferred␣
↪debugger)
        -Your offload tasks will be executed by the main thread of the␣
↪application.
7- You are now debugging the offload process/thread, you can place breakpoints␣
↪and then write "continue" in gdb.
```

Offloading variables

When offloading, a few rules are used in order to offload variables (also arrays/data-regions):

- *Local variables**: They will be copied as in regular OmpSs/OMP tasks.
- *Non-const Global variables**: They will be copied to and from the offload worker (visible inside the task body and also in external functions).
- *Const global variables**: As they are non-writable, they can only be in/firstprivate (copy of host value will be visible inside task body, external functions will see initial value).
- *Global variables which are never specified in copy/firstprivate* : They will not be modified by OmpSs, so users may use them to store "node-private" values under their responsibility.

Global variables are C/C++ global variables and Fortran Module/Common variables.
Task region refers to the code between the task brackets/definition, external functions are functions which are called from inside task region.

*OmpSs offload does not provide any guarantee about the value or the allocated space for these variables after the task which copies/uses them has finished.

Obtaining traces of offload applications

In order to trace an offload application users must follow these steps:

- Compile with –instrument. An installation using Nanox instrumentation version (–with-extrae) is required.
- Set "export EXTRAE_HOME=/my/extrae/path" and "export EXTRAE_HOME_MIC=/my/extrae/path/mic"
- Optional: if you want to use a configuration file, export EXTRAE_CONFIG_FILE=./extrae.xml. Do not enable merging of traces inside the xml file, as is not supported. EXTRAE_DIR environment variable has to be set if this file points extrae intermediate files to a different folder than current directory ($PWD).
- Call your program with "offload_instrumentation.sh mpirun ..." instead of "mpirun ..."

"offload_instrumentation.sh" script can be found in $NANOX_HOME/bin in case its not available through the PATH. It will setup the environment and then call the real application.

Three configuration environment variables are available for this script:

- *EXTRAE_DIR*: Directory where intermediate files will be generated (if specified with .xml file, this variable has to point to the same folder). [Default value: $PWD]
- *AUTO_MERGE_EXTRAE_TRACES*: If enabled, after the application finishes, the script will check the directory pointed by *EXTRAE_DIR* and merge the intermediate files into a final trace (.prv) [Default value: YES]. If disabled, users can merge these files manually by calling mpi2prv with the right parameters or by executing the script with no arguments (it will merge intermediate files located in *EXTRAE_DIR*).
- *CLEAR_EXTRAE_TMP_FILES*: If enabled, intermediate files will be removed after finishing the execution of the script [Default value: NO].

Offload Troubleshooting

- If you are offloading tasks which use MPI communications on the workers side and they hang, make sure that you launched as many tasks as nodes in the communicator (so every rank of the remote communicator is executing one of the tasks), otherwise all the other tasks will be waiting for that one if they perform collective operations.
- If you are trying to offload to Intel MICs, make sure the variable I_MPI_MIC is set to yes/1 (export I_MPI_MIC=1).
- If your application behaves differently by simply compiling with OmpSs offload, take into account that OmpSs offload initializes MPI with MPI_THREAD_MULTIPLE call.
- If your application hangs inside DEEP Booster alloc calls, check that the hosts provided in the hostfile exist.
- Your application (when using MPI_COMM_SELF as first parameter when calling `deep_booster_alloc` from multiple MPI processes) has to be executed in a folder where you have write permissions, this is needed because Intel MPI implementation of MPI_Comm_spawn is not inter-process safe so we have to use a filesystem lock in order to serialize spawns so it does not crash. As a consequence of this problem spawns using MPI_COMM_SELF will be serialized. This behaviour can be disabled by using the variable `NX_OFFL_DISABLE_SPAWN_LOCK=1`, but then MPI Implementation has to support concurrent spawns or you will have to guarantee inside your application code that multiple calls to `DEEP_Booster_alloc` with MPI_COMM_SELF as first parameter are not performed at the same time. This behaviour is disabled when using OpenMPI as everything works as it should in this implementation.

- If you are trying to offload code which gets compiled/configured with CMake, you have to point the compilers to ones provided by the offload, you can do this with `FC=mpimfc CXX=mpimcxx CC=mpimcc cmake ..` If you are using FindMPI, you should disable it (recommended). Setting MPI and non-MPI compilers pointing to our compiler with `FC=mpimfc CXX=mpimcxx CC=mpimcc cmake . -DMPI_C_COMPILER=mpimcc -DMPI_CXX_COMPILER=mpimcxx -DMPI_Fortran_COMPILER=mpimfc` (CXX compiler is mandatory even if you are not using C++) should be enough.

Apart from offload troubleshooting, there are some problems with Intel implementation of MPI_COMM_SPAWN_MULTIPLE on MICs:

- (Fixed when using Intel MPI 5.0.0.016 or higher) If the same host (only on MICs) appears twice, the call to DEEP_BOOSTER_ALLOC/MPI_COMM_SPAWN may hang, in this case try specifying thread binding manually and interleaving those hosts with other machines.

Example crashing hostfile:

```
knights3-mic0
knights3-mic0
knights4-mic0
knights4-mic0
```

Example workaround *fixed* hostfile:

```
knights3-mic0
knights4-mic0
knights3-mic0
knights4-mic0
```

- More than 80 different hosts can't be allocated by using a single file, in this case DEEP_BOOSTER_ALLOC/MPI_COMM_SPAWN will throw a Segmentation Fault. In order to handle this problem, we made a workaround, in offload host-file, instead of specifying one host, you can specify a host-file which will contain hosts.

Example hostfile using sub-groups in files:

```
#Hostfile1 contains 64 hosts, and all of them will have the same offload variables
#(path should be absolute or relative to working directory of your app)
./hostfile1:64<NX_SMP_WORKERS=1,NX_BINDING_STRIDE=4
knights3-mic0
./hostfile2:128
```

Tuning OmpSs Applications Using Offload for Performance

In general, best performance results is when *write-back* cache policy is used (default).

When offloading, one thread will take care of sending tasks from the host to the device. These threads will only wait for a few operations (for most operations, it just sends orders to the remote nodes) this is enough for most applications, but if you feel that these threads are not enough to offload all your independent tasks. You can increase the number of threads which will handle each alloc by using the variable `NX_OFFL_MAX_WORKERS` as in `NX_OFFL_MAX_WORKERS=12`, which by default has a value of 1.

Formula for N_THREADS_PER_ALLOC = min(NX_OFFL_MAX_WORKERS,number of offload processes/number of master processes).

If you know that you application is not uniform and it will have transfers from one remote node to other remote node while one of them is executing tasks, you may get a small performance improvement by setting

`NX_OFFL_CACHE_THREADS=1`, which will start an extra cache thread at worker nodes. If this option is not enabled by the user, it will be enabled automatically only when this behaviour is detected.

Each worker node can receive tasks from different masters but it will execute only one at a time. Try to avoid this kind of behaviour if you application needs to be balanced.

Other Nanox options for performance tuning can be found at *My application does not run as fast as I think it could*.

Setting environment variables (i.e. number of threads) for offload workers

In regular applications, number of threads will be defined by regular variables, for example NX_SMP_WORKERS=X which sets the number of SMP workers/OMP threads in the master. In offload workers, default value will be the same than cores on the machine that the worker is running, if you want to specify them manually, as you may have more than one architecture, number of threads can be different for each node/architecture. In order to do this we provide a few ways to do it, in case of conflicts, the latest one in this list, will be used:

- Specify it individually on the hostfile on each node as an environment variable (NX_SMP_WORKERS or OFFL_NX_SMP_WORKERS). This value overwrites global and architecture variable for that node/group of nodes.
- Specify it globally using the variable "OFFL_VAR", this means that once the remote worker starts, VAL will be defined for every offload worker with the value specified in OFFL_VAR.
- Specify it per-architecture using the variable "YY_VAR", being YY the suffix of the executable for that architecture (explained in *Naming Convention*.) and VAR the variable you want to define in that architecture. For example, for Intel mic architecture, you can use "MIC_NX_SMP_WORKERS=240" in order to set number of threads to 240 for this architecture. This value overwrites global variable for the concrete architecture.

MPI implementation usually exports all those variables automatically, if this is not the case, users must configure it so variables which are needed are exported.

Other Offload variables

Other offload configuration variables can be obtained by using "nanox –help":

Offload Arch:

Environment variables

NX_OFFL_HOSTS = <string> Defines hosts file where secondary process can spawn in DEEP_Booster_Alloc Same format than NX_OFFLHOSTFILE but in a single line and separated with ';' Example: hostZ hostA<env_vars hostB:2<env_vars hostC:3 hostD:4

NX_OFFL_ALIGNTHRESHOLD = <positive integer> Defines minimum size (bytes) which determines if offloaded variables (copy_in/out) will be aligned (default value: 128), arrays with size bigger or equal than this value will be aligned when offloaded

NX_OFFL_CACHE_THREADS = <true/false> Defines if offload processes will have an extra cache thread, this is good for applications which need data from other tasks so they don't have to wait until task in owner node finishes. (Default: False, but if this kind of behaviour is detected, the thread will be created)

NX_OFFL_ALIGNMENT = <positive integer> Defines the alignment (bytes) applied to offloaded variables (copy_in/out) (default value: 4096)

NX_OFFL_HOSTFILE = <string> Defines hosts file where secondary process can spawn in DEEP_Booster_Alloc The format of the file is: One host per line with blank lines and lines beginning with # ignored Multiple processes per host can be specified by specifying the host name

as follows: hostA:n Environment variables for the host can be specified separated by comma using hostA:n<env_var1,envar2... or hostA<env_var1,envar2...

NX_OFFL_EXEC = <string> Defines executable path (in child nodes) used in DEEP_Booster_Alloc

NX_OFFL_MAX_WORKERS = <positive integer> Defines the maximum number of worker threads created per alloc (Default: 1)

NX_OFFL_LAUNCHER = <string> Defines launcher script path (in child nodes) used in DEEP_Booster_Alloc

NX_OFFL_ENABLE_ALLOCWIDE = <0/1> Alloc full objects in the cache. This way if you only copy half of the array, the whole array will be allocated. This is good/useful when OmpSs copies share data between offload nodes (Default: False)

NX_OFFL_CONTROLFILE = <string> Defines a shared (GPFS or similar) file which will be used to automatically manage offload hosts (round robin). This means that each alloc will consume hosts, so future allocs do not oversubscribe on the same host.

2.3.3 Runtime Modules (plug-ins)

As the main purpose of Nanos++ is to be used in research of parallel programming environments it has been designed to be extensible by means of plugins. Runtime plugins can be selected for each execution using the proper runtime option (e.g. NX_SCHEDULE=cilk). These are:

Scheduling policies

The scheduling policy defines how ready tasks are executed. Ready tasks are those whose dependencies have been satisfied therefore its execution can start immediately. The scheduling policy has to decide the order of execution of tasks and the resource where each task will be executed.

When you inject a dependant task into the runtime system actually you are inserting this task into the dependant task graph. Once all dependencies for a given task are fulfilled this task will be inserted into a ready queue.

Before taking into account any scheduling criteria, we must consider four different scheduler modifiers. They are:

- Throttling policy at task creation. Just when we are going to create a new task, the runtime system determines (according to a throttling policy) whether to create it and push it into the scheduler system or just create a minimal description of the task and execute it right away in the current task context. Throttle mechanism can be configured as a completely independent plugin using the throttle option. See *Throttling policies*.
- Task Priority. When inserting a new task in a priority queue, if the new task has the same priority as another task already in the queue, the new one will be inserted before/after the existent one (according with the FIFO/LIFO behaviour between *tasks with the same priority*). The priority is a number >= 0. Given two tasks with priority A and B, where A > B, the task with priority A will be fetched earlier than the one with B from the queue. Priority is also accumulate from parent to children. When a task T with priority A creates a task Tc that was given priority B by the user, the priority of Tc will be added to that of its parent. In other words, the priority of Tc will be A + B. Smart priority behaviour also propagates the priority to the immediate preceding tasks (when dependant tasks). Ready task priority queues are only available in some schedulers. Check specific scheduler parameters and/or description for more details.
- Immediate successor. Releasing last dependency when exiting a task. When a task is in the dependency graph and its last immediate predecessor finishes execution, we are able to run the blocked tasks immediately instead of adding it to the ready queue. Sometimes immediate successor is not a configurable option. Check specific scheduler description and/or parameters for more details.
- Parent continuation when last child has been executed. When executing nested OpenMP/OmpSs applications (explicit tasks creating more explicit tasks) it may happens that a given task becomes blocked due the execution

of a taskwait or taskwait on directive. When the last child has finished its execution the parent will be promptly resumed instead of being push back into the ready queue (which potentially could delay its execution). Parent continuation is only available in wf scheduler. Check specific scheduler description for more details.

When a thread has no work assigned or when it is waiting until a certain condition is met (e.g. all task's children completes in a taskwait), threads are in idle loop and conditional wait respectively. Users can also specify which is the behaviour of these thread duties through the following options (for more info see *Thread Manager*):

--checks=<n> Set number of checks on a conditional wait loop before spinning on the ready task pool (default = 1).

--spins=<n> Set number of spin iterations before yielding on idle loop and conditional wait loop (default = 1).

--enable-yield Enables thread yielding on idle loop and conditional wait loop (disabled by default).

--yields=<n> Set number of yields before blocking on idle loop and conditional wait loop (default = 1).

--enable-block Enables thread blocking on idle loop and conditional wait loop (disabled by default).

A complete idle loop cycle starts with spinning many times as spins parameter specifies After spinning, if yield is enabled, it will yield once and it will start again spinning. The whole spin/yield process will be repeated as many time as the yields option specifies and, finally, if thread block is enabled, the thread will block. Yield and block can not happen in the same cycle.

The conditional wait loop will start with checking the wait condition as many times as the checks parameter specifies. After that number of checks it will start a idle loop cycle (described in the previous paragraph).

The scheduler plug-in can be specified by the `NX_SCHEDULE` environment variable or the `--schedule` option included in the `NX_ARGS` environment variable:

```
$export NX_SCHEDULE=<module>
$./myProgram

$export NX_ARGS="--schedule[ \|=]<module> ..."
$./myProgram
```

--schedule=<module> Set the scheduling policy to be used during the execution. The argument *module* can be one of the following options: `bf`, `dbf`, `wf`, `socket`, `affinity`, `affinity-smartpriority` or `versioning`. The most suitable scheduling policy can depend on the application and architecture used.

Nanos++ implements several schedule plug-ins. The following sections will show a description of the available plug-ins with respect to scheduling policies. Some of these modules also incorporate specific options.

Breadth First

Important: This is the default scheduler

- *Configuration string:* `NX_SCHEDULE=bf` *or* `NX_ARGS="--schedule=bf"`
- *Description:* This scheduler policy only implements a single/global ready queue. When creating a task with no dependences (or when a task becomes ready after all its dependences has been fulfilled) it is placed in this ready queue. Ready queue is ordered following a FIFO (First In First Out) algorithm by default, but it can be changed

through parameters. Breadth first implements immediate successor mechanism by default (if not in conflict with priority).

- *Parameters:*

  --bf-stack, --no-bf-stack Retrieving from queue's back or queue's front respectively. Using –bf-stack parameters transform ready queue into a LIFO (Last In First Out) structure. Effectively, this means that tasks are inserted in queue's back and retrieved from the same position (queue's back).(default behaviour is –no-bf-stack)

  --bf-use-stack, --no-bf-use-stack This is an alias of the previous parameter.

  --schedule-priority, --no-schedule-priority Use priorities queues.

  --schedule-smart-priority, --no-schedule-smart-priority Use smart priority queues.

Distributed Breadth First

- *Configuration string:* `NX_SCHEDULE=dbf` *or* `NX_ARGS="--schedule=dbf"`
- *Description:* Is a breadth first algorithm implemented with thread local queues instead of having a single/global ready queue. Each thread inserts its created tasks into its own ready queue following a FIFO policy (First In First Out, inserting in queue's front and retrieving from queue's back) algorithm. If thread local ready queue is empty, it tries to execute current task's parent (if it is queued in any other thread ready queue). In the case parent task cannot be eligible for execution it steals from next thread ready queue. Stealing retrieves tasks from queue's front (i.e. the opposite side from local retrieve).

  --schedule-priority, --no-schedule-priority Use priorities queues.

  --schedule-smart-priority, --no-schedule-smart-priority Use smart priority queues.

Work First

- *Configuration string:* `NX_SCHEDULE=wf` *or* `NX_ARGS="--schedule=wf"`
- *Description:* This scheduler policy implements a local ready queue per thread. Once a task is created it chooses to continue with the new created task, leaving current task (creator) into current thread's ready queue. Default behaviour is implemented through FIFO access to local queue, LIFO access on steal and steals parent if available which actually is equivalent with the cilk scheduler behaviour.
- *Parameters:*

  --wf-steal-parent, --no-wf-steal-parent If local ready queue is empty, it tries to steal parent task if available (default: wf-steal-parent).

  --wf-local-policy=<string> Defines queue access for local ready queue. Configuration string can be:

  - `FIFO`: First In First Out queue access (default).
  - `LIFO`: Last In First Out queue access.

  --wf-steal-policy=<string> Defines queue access for stealing. Configuration string can be:

  - `FIFO`: First In First Out queue access (default).
  - `LIFO`: Last In First Out queue access.

Socket-aware scheduler

- *Configuration string:* `NX_SCHEDULE=socket` *or* `NX_ARGS="--schedule=socket"`
- *Description:* This scheduler will assign top level tasks (depth 1) to a NUMA node set by the user before task creation while nested tasks will run in the same node as their parent. To do that, the user must call the nanos_current_socket function before executing tasks to set the NUMA node the task will be assigned to. The queues are sorted by priority, and there are as many queues as NUMA nodes specified (see num-sockets parameter). Besides that, changing the binding start and stride is not supported. Work stealing is optional. By default, work stealing of child tasks is enabled. Upon initialisation, the policy will create lists of valid nodes to steal. For each node, the policy will only steal from the closest nodes. Use numactl –hardware to print the distance matrix that the policy will use. For each node, a pointer to the next node to steal is kept; if a node steals a task, it will not affect where other nodes will steal. There is an option to steal from random nodes as well, and to steal top level tasks instead of child tasks.
- *Parameters:* There are important parameters as the number of sockets and the number of cores per socket. If Nanos++ is linked against the hwloc library, it will be used to get that information automatically. Besides that, if the number of cores per socket, sockets and number of processing elements do not make sense, the number of sockets will be adjusted.

  --num-sockets Sets the number of NUMA nodes.

  --cores-per-socket Sets the number of hardware threads per NUMA node.

  --socket-steal, --no-socket-steal Enable work stealing from the ready queues of other NUMA nodes (default).

  --socket-random-steal, --no-socket-random-steal Instead of round robin stealing, steal from random queues.

  --socket-steal-parents, --no-socket-steal-parents Steal depth 1 tasks instead of child tasks (disabled by default).

  --socket-steal-spin Number of spins before attempting work stealing.

  --socket-smartpriority, --no-socket-smartpriority Enable smart priority propagation (disabled by default).

  --socket-immediate, --no-socket-immediate Use getImmediateSuccessor when prefetching (disabled by default). Note: this is not completely implemented (atBeforeExit).

  --socket-steal-low-priority, --no-socket-steal-low-priority Steal low priority tasks from the other nodes' queues. Note: this is currently broken.

Affinity

> **Warning:** This policy has been discontinued in master development branch, but it is still used in the *cluster* development branch

- *Configuration string:* `NX_SCHEDULE=affinity` *or* `NX_ARGS="--schedule=affinity"`
- *Description:* Take into account where the data used by a task is located. Meaningful only in Multiple Address Space architectures (or SMP NUMA). Affinity implements immediate successor by default.

Affinity Smart Priority

Warning: This policy has been discontinued in master development branch, but it is still used in the *cluster* development branch

- *Configuration string:* `NX_SCHEDULE=affinity-smartpriority` *or* `NX_ARGS="--schedule=affinity-smart-priority"`
- *Description:* Affinity policy with smart priority support. Works exactly like the affinity policy (same number of queues, it also has work stealing) but uses sorted priority queues with priority propagation to immediate preceding tasks. When inserting a new task in the queue, if the new task has the same priority as another task already in the queue, the new one will be inserted after the existent one (note that, unlike in the priority scheduling policy, there is no option to change this behaviour). Affinity implements immediate successor by default.

Versioning

- *Configuration string:* `NX_SCHEDULE=versioning` *or* `NX_ARGS="--schedule=versioning"`
- *Description:* This scheduler can handle multiple implementations for the same task and gives support to the `implements` clause. The scheduler automatically profiles each task implementation and chooses the most suitable implementation each time the task must be run. Each thread has its own task queue, where task implementations are added depending on scheduler's decisions. The main criteria to assign a task to a thread is that it must be the earliest executor of that task (this means that the thread will finish task's execution at an earliest time). In some cases (where the number of tasks in the graph is big enough), idle threads are also allowed to run task implementations even though they are not the fastest executors. Specifically, the profile for each task implementation includes its execution time and the data set size it uses and it is used to estimate the behavior of future executions of the same implementation. Versioning implements immediate successor by default.
- *Parameters:*

  --versioning-stack, --no-versioning-stack Retrieving from queue's back or queue's front respectively. Using –versioning-stack parameters transform ready queue into a LIFO (Last In First Out) structure. Effectively, this means that tasks are inserted in queue's back and retrieved from the same position (queue's back). (default behaviour is –no-versioning-stack)

  --versioning-use-stack, --no-versioning-use-stack This is an alias of the previous parameter.

Note: A detailed explanation of this scheduler can be found at the following conference paper: Judit Planas, Rosa Badia, Eduard Ayguade, Jesus Labarta, "Self-Adaptive OmpSs Tasks in Heterogeneous Environments", Proceedings of 27th IEEE International Parallel and Distributed Processing Symposium (IEEE IPDPS) (2013).

Bottom level-aware scheduler

- *Configuration string:* `NX_SCHEDULE=botlev` *or* `NX_ARGS="--schedule=botlev"`
- *Description:* This scheduler targets single-ISA heterogeneous machines that maintain two kinds of cores (fast and slow, such as ARM big.LITTLE). The scheduler detects dynamically the longest path of the task dependency graph and assigns the tasks that belong to this path (critical tasks) to the fast cores of the system. The detection of the longest path is based on the computation and the usage of bottom-level longest-path priorities, that is

the length of the longest path in the dependency chains from each node to a leaf node. There are two queues for the ready tasks, one per processor kind. Fast cores retrieve tasks from their unique queue and slow cores retrieve tasks from the other queue. The queues are sorted according to the task priority (bottom level). Work stealing is enabled by default for fast cores: a fast core steals a task from the slow-cores' queue if it is idle. Optionally, work stealing can be performed by both sides if the parameter NX_STEALB is enabled. The policy can be flexible or strict, meaning that the flexible policy considers more tasks as critical, while the strict limits the number of critical tasks.

- *Parameters:*

  –update-freq or NX_BL_FREQ Sets the update frequency of the priorities. By default it is set to zero which means update every time a new dependency occurs. It can be set with any positive integer value and affects the priorities of the tasks. By setting this parameter to a positive value the bottom levels of the tasks are less accurate but the policy has slightly less scheduling overheads. –numSpins or NX_NUM_SPINS Sets the number of spins before attempting work stealing (by default it is set to 300). –from or NX_HP_FROM Sets the thread id of the first fast core. –to or NX_HP_TO Sets the thread id of the last fast core. –strict or NX_STRICTB Defines whether the policy is strict or flexible (–strict=0 for flexible or –strict=1 for strict). –steal or NX_STEALB Defines whether the policy uses uni- or bi-directional work stealing (–steal=0 for uni-directional or –steal=1 for bi-directional).

Throttling policies

The throttle policy determines when tasks are created as entities, that can be scheduled and executed asynchronously, or are executed immediately, as if the code were not meant to be executed in parallel. Applications may be sensitive to this behavior so different throttling policies are provided by Nanos++ in order to allow a more precise tuning of the system.

Usage

Description: Sets the throttling policy that will be used during the execution.
Type: string value.
Environment variable: `NX_THROTTLE=<string>`
Command line flag: `--throttle[ |=]<string>`

List of throttling plugins

Throttle policies are provided in the form of plug-ins. Currently Nanos++ comes with the following throttling policies:

Hysteresis

Important: This is the default throttling policy

- *Configuration string:* `NX_THROTTLE=hysteresis` *or* `NX_ARGS="--throttle=hysteresis"`
- *Description:* Based on the hysteresis mechanisms, once we reach to a upper limit we stop creating tasks but we start to create them again when we reach a lower limit. Upper limit and lower limit are configurable through command line flags:
- *Parameters:*

--throttle-upper Defines the maximum number of tasks per thread allowed to create new first level's tasks. (default value 500).

--throttle-lower Defines the number of tasks per thread to re-active first level task creation. (default value 250).

--throttle-type Defines the target counter when taking into account the number of tasks. User can choose among `ready` or `total`. (default value total).

Task depth

- *Configuration string:* `NX_THROTTLE=taskdepth` *or* `NX_ARGS=--throttle=taskdepth"`
- *Description:* This throttle mechanism is based on the task depth. The runtime will not create tasks further than the nested level specified by the limit.
- *Parameters:*

  --throttle-limit Defines maximum depth for tasks. (default value 4).

Ready tasks

- *Configuration string:* `NX_THROTTLE=readytasks` *or* `NX_ARGS="--throttle=readytasks"`
- *Description:* Defines the throttle policy according with the number of ready tasks while creating a new task.
- *Parameters:*

  --throttle-limit Defines the number of ready tasks we use as limit for task creation. When creating a task, if the number of ready tasks is greater than limit task will not be created, task creation will block and issued to scheduler decision. If number of ready tasks is less or equal than limit, task will be created normally. (default value 100).

Idle threads

- *Configuration string:* `NX_THROTTLE=idlethreads` *or* `NX_ARGS="--throttle=idlethreads`
- *Description:* This throttle policy take the decision of create (or not) a task according with the number of idle threads we have at task creation instant.
- *Parameters:*

  --throttle-limit Defines the number of idle threads we use as limit for task creation. When creating a task, if the number of idle threads is less than this value the task will not be created, task creation will block and issued to scheduler decision. If number of idle threads is greater or equal than this limit, the task will be created normally. (default value is 0).

Number of tasks

- *Configuration string:* `NX_THROTTLE=numtasks` *or* `NX_ARGS=--thottle=numtasks`
- *Description:* Defines the throttle policy according with the existing number of tasks at task creation.
- *Parameters:*

--throttle-limit Defines the number of tasks (total tasks already created and not completed, being ready or not) we use as limit for task creation. When creating a task, if the number of on-fly-tasks is greater than limit the task will not be created, task creation will be blocked and issued to scheduler decision. If number of on-fly-tasks is less or equal than this limit, the task will be created normally. (default value 100).

Dummy

- *Configuration string:* `NX_THROTTLE=dummy` *or* `NX_ARGS="--throttle=dummy"`
- *Description:* This throttle policy always takes the decision of create (or not) tasks.
- *Parameters:*

  --throttle-create-tasks Set the behaviour of the dummy throttle to force task creation and deferred execution. (default option).

  --no-throttle-create-tasks Set the behaviour of the dummy throttle to avoid task creation and force undeferred execution.

Throttle policy examples

If we want to execute our program with 4 threads and using a throttling policy which create tasks only if we have at least one thread idle, we will use the following command line:

```
NX_ARGS="--threads=4 --throttle=idlethreads --throttle-limit=1" ./myProgram
```

If we want to execute our program with 8 threads and using a throttling policy which create tasks only if we do not have at least 50 ready tasks per thread, we will use the following command line:

```
NX_ARGS="--threads=8 --throttle=readytasks --throttle-limit=50" ./myProgram
```

Instrumentation modules

The instrumentation flag selects among all available instrumentation plug-ins which one to use. An instrumentation plug-in generates output information which describes one or more aspects of the actual execution. This module is in charge of translate internal Nanos++ events into the desired output: it may be just showing the event description in the screen (as soon as they occur), generates a trace file which can be processed by other software component, build a dependence graph (taking into account only dependence events), or just call an external service when a given type of event occurs. In general the events are taking place inside the runtime but also they can be generated by the application's programmers or introduced by the compiler (e.g. outlined functions created by Mercurium compiler).

The instrumentation plug-in can be specified by the `NX_INSTRUMENTATION` environment variable or the `--instrumentation` option included in the `NX_ARGS` environment variable:

```
$export NX_INSTRUMENTATION=<module>
$./my_program

$export NX_ARGS="--instrumentation[ \|=]<module> ..."
$./my_program
```

--instrumentation=<module> It sets the instrumentation backend to be used during the execution. Module can be one of the following options: `empty`, `print_trace`, `extrae`, `graph`, `ayudame` or `tasksim tdg`.

By default, only a set of Nanos++ events are enabled (see Nanos++ events section for further details). Additionally you can also use following options in order to specify the set of events that will be instrumented for a given execution.

--instrument-default=<mode> Set the instrumentation event set by default. Where `mode` can be one of the following options: `none` disabling all events, or `all` enabling all events. Additionally we can choose one instrumentation profile: `advanced`, `developer` and `user`. Interval and punctual event table shows which of these events are generated when activating one of these levels. (Default value is `user`).

--instrument-enable=<event-type> Add events to the current instrumentation event list. Where `event-type` can be one of the following options: `state` enabling all the **state** events, `ptp` enabling all the **point-to-point** events, or an specific *event-prefix* enabling those **interval** and/or **punctual** events whose name matches with *event-prefix*. Users can also combine several instrument-enable options in the same execution environment (e.g. `NX_ARGS="... --instrument-enable=state --instrument-enable=for"`).

--instrument-disable=<event-type> Remove events to the current instrumentation event list. Where `event-type` can be one of the following options: `state` disabling all the **state** events, `ptp` disabling all the **point-to-point** events, or an specific *event-prefix* disabling those **interval** and/or **punctual** events whose name matches with *event-prefix*. Users can also combine several instrument-disable options in the same execution environment (e.g. `NX_ARGS="... --instrument-disable=state --instrument-disable=for"`).

--instrument-cpuid, --no-instrument-cpuid Add cpuid event when binding is disabled (expensive).

Nanos++ implements several instrumentation plug-ins. The following sections will show a description of each of the available plug-ins with respect to that feature.

Plug-in description

Empty Trace

Important: This is the default instrumentation plugin.

- *Configuration String:* `NX_INSTRUMENTATION=empty_trace` *or* `NX_ARGS="--instrumentation=empty_trace"`
- *Description:* It does not generate any output.

Extrae

- *Configuration String:* `NX_INSTRUMENTATION=extrae` *or* `NX_ARGS="--instrumentation=extrae"`
- *Description:* It generates a Paraver trace using Extrae library

Paraver is a flexible parallel program visualization and analysis tool based on an easy-to-use Motif GUI. Paraver was developed responding to the need of having a qualitative global perception of the application behavior by visual

inspection and then to be able to focus on the detailed quantitative analysis of the problems. Paraver provides a large amount of information useful to decide the points on which to invest the programming effort to optimize an application.

Before getting any trace we will need to compile our application with instrumentation support. Compile your application using Mercurium with the `--instrument` flag:

```
$ mcc --ompss --instrument my_program.c -o my_program
```

In order to get a Paraver's Trace you will need to execute your OmpSs program with the instrumentation plugin Extrae using the environment variable NX_INSTRUMENTATION. That will enable the Extrae plugin which translates all internal events into a Paraver events using the Extrae library. Extrae is completely configured through a XML file that is set through the EXTRAE_CONFIG_FILE environment variable:

```
$ export NX_INSTRUMENTATION=extrae
$ export EXTRAE_CONFIG_FILE=config.xml
$ .\my_program
```

Important: Extrae library installation includes several XML file examples to serve as a basis for the end user (see $EXTRAE_DIR/share/example). Check your Extrae installation to get one of them and devote some time in tuning the parameters. This file is divided in diferent sections in which you can enable and disable some Extrae features.

XML's merge section enabled (recommended)

If the merge section is enabled the merge process will be automatically invoked after the application run. You can enable or disable this Extrae feature by just setting the enabled flag to yes/no. The value of this XML node (in between `<merge>` and `</merge>` tags) is used as the tracefile name (i.e. my_program_trace.prv in the following example). You can check other flags meanings in the Extrae User's Guide:

```
<merge enabled="yes"
   synchronization="default"
   binary="my_program"
   tree-fan-out="16"
   max-memory="512"
   joint-states="yes"
   keep-mpits="no"
   sort-addresses="yes"
   remove-files="yes"
>
  my_program_trace.prv
</merge>
```

XML's merge section disabled

If the merge section is disabled (or you are not using the XML configuration file), once you have run your instrumented program Extrae library will produce several files containing all the information related with the execution. There will be as many .mpit files as tasks and threads where running the target application. Each file contains information gathered by the specified task/thread in raw binary format. A single .mpits file that contain a list of related .mpit files and, if the DynInst based instrumentation package was used, an addition .sym file that contains some symbolic information gathered by the DynInst library.

In order to use Paraver, those intermediate files (i.e., .mpit files) must be merged and translated into a Paraver trace file format. To proceed with any of these translation all the intermediate trace files must be merged into a single trace file using one of the available mergers in the Extrae bin directory.

Using Hardware Counters (PAPI)

In order to get a Paraver trace files including hardware counters (HWC) we need to verify that our Extrae library has been installed with PAPI support. Once we have verified this we can turn on the generation of PAPI hardware counter events through the XML Extrae configuration file section: counters. Here is an example of the counters section in the XML configuration file:

```
<counters enabled="yes">
  <cpu enabled="yes" starting-set-distribution="1">
    <set enabled="yes" domain="all" changeat-time="1s">
      PAPI_TOT_INS,PAPI_TOT_CYC,PAPI_L1_DCM
      <sampling enabled="yes" period="100000000">PAPI_TOT_CYC</sampling>
    </set>
    <set enabled="yes" domain="all" changeat-time="1s">
      PAPI_TOT_INS,PAPI_TOT_CYC,PAPI_FP_INS
    </set>
  </cpu>
  <network enabled="no" />
  <resource-usage enabled="no" />
</counters>
```

Processor performance counters are configured in the <cpu> nodes. The user can configure many sets in the <cpu> node using the <set> node, but just one set will be active at any given time. In the example two sets are defined. First set will read PAPI_TOT_INS (total instructions), PAPI_TOT_CYC (total cycles) and PAPI_L1_DCM (1st level cache misses). Second set is configured to obtain PAPI_TOT_INS (total instructions), PAPI_TOT_CYC (total cycles) and PAPI_FP_INS (floating point instructions). You can get a list of PAPI Standard Events By Architecture here. In order to change the active set you can use the changeat-time attribute specifying the minimum time to hold the set (i.e. changeat-time=”1s” in the example).

See also the list of PAPI Standard Events by Architecture and the Manuals of the BSC Performance Tools.

Task Sim

> **Warning:** This plugin is experimental

- *Configuration String:* `NX_INSTRUMENTATION=tasksim` *or* `NX_ARGS="--instrumentation=tasksim"`
- *Description:* It generates a simulation trace which can be used with TaskSim

Ayudame

Ayudame plug-in allows to interact current program execution with Temanejo graphical debugger. The main goal is to display the task-dependency graph, and to allow simple interaction with the runtime system in order to control some aspects of the parallel execution of an application.

Ayudame plug-in is not included by default in Nanos++ runtime package. In Temanejo’s manual (see references at the end of the section) you will find detailed instructions about how to get and install this plug-in.

Before using Ayudame/Temanejo interaction you will need to compile your application with instrumentation support. Compile your application using Mercurium compiler and the `--instrument` flag:

```
$ mcc --ompss --instrument my_program.c -o myProgram
```

At runtime you will need to enable Ayudame plug-in using instrumentation option:

```
$export NX_INSTRUMENTATION=ayudame
$./myProgram

$export NX_ARGS="--instrumentation[ \|=]ayudame ..."
$./myProgram
```

You can find more information about Ayudame/Temanejo in HLRS web site.

Task Dependency Graph

- *Configuration String:* `NX_INSTRUMENTATION=tdg` *or* `NX_ARGS="--instrumentation=tdg"`
- *Description:* It generates a .dot file with all the real dependences produced in that particular execution.

This plugin also admit some extra parameters:

--node-size=<string> Defines the semantic for the node size. Configuration string can be:

- `constant`: the size of all nodes is the same (default).
- `linear`: the size of the nodes increases linearly with the task time.
- `log`: the size of the nodes increases logarithmically with the task time.

This instrumentation plugin generates a dependency graph using Graphviz (http://www.graphviz.org) format.

When compiling with Mercurium, you will need to use the flag `--instrument`. At runtime you will need to enable the `tdg` plugin using the environment variable `NX_ARGS="--instrument=tdg"`. The execution of the program will generate `graph.dot` file, and also (if `dot` program available) a `graph.pdf` file.

Dependences shown in the graph are real dependences, which means that are dependences that actually happen during that specific execution.

Usually one wants all the theoretical dependences in the program. To do this you have to run your program with a single thread and ensure that the schedule creates all the tasks (i.e. does not decide to immediately run them). To achieve this use `NX_ARGS="--smp-workers=1 --throttle=dummy"`, see example below.

This version of the plugin is thread-safe, so you do not need to execute your program with a unique thread to draw the TDG (although it is recommended to draw properly your program dependences).

In the graph, task nodes are colored according with the function they represent (the plugin includes a legend in the right-top part showing equivalence between colors and functions). The size of the node may also represent time, which means that bigger tasks are more time expensive than the smaller ones. Finally the `tdg` plugin also shows two different kind of arrows. Solid arrows are true dependences, while dashed arrows are used to specify anti/output dependences as defined in the legend. Gray arrows show nested relationships.

Output Example

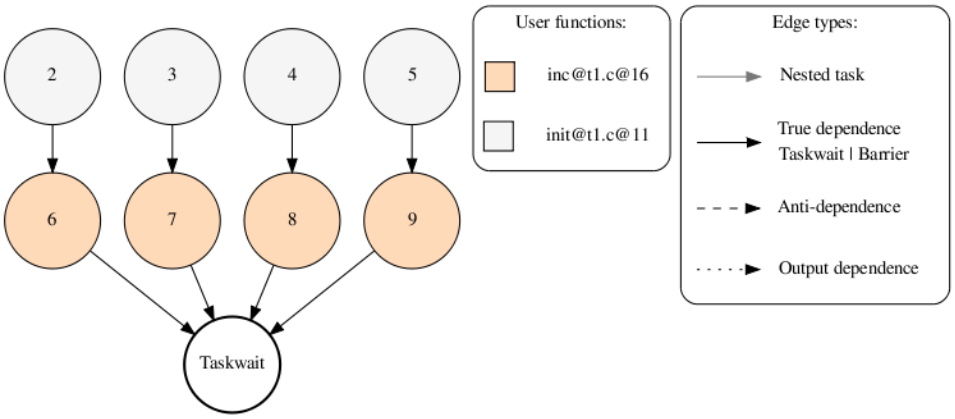
Running the following code with this plugin will produce the next graph:

```
int a[N];

for(int i=0; i<N; ++i) {
   #pragma omp task out(a[i]) label(init) firstprivate(i)
   a[i]=0;
```

```
}

for(int i=0; i<N; ++i) {
   #pragma omp task inout(a[i]) label(inc) firstprivate(i)
   a[i]++;
}

#pragma omp taskwait

for(int i=0; i<N; ++i) {
   printf("%d,",a[i]);
}
```

Command line:

```
$ NX_ARGS="--smp-workers=1 --throttle=dummy --instrumentation=tdg" ./myprogram
```



Nanos++ events

State events

In Nanos++ programming model a thread can represent whether a POSIX thread or a runtime Work Descriptor according with the instrumentation model we were using.

##	Event	Description
00	NANOS_NOT_CREATED	Thread has not been created yet.
01	NANOS_NOT_RUNNING	Thread is executing nothing (e.g. a Work Descriptor which is not in execution or sub state when state is enabled).
02	NANOS_STARTUP	Thread is executing runtime start up.
03	NANOS_SHUTDOWN	Thread is executing runtime shut down.
04	NANOS_ERROR	Error while instrumenting.
05	NANOS_IDLE	Thread is on idle loop.
06	NANOS_RUNTIME	Thread is executing generic runtime code (not defined in other state).
07	NANOS_RUNNING	Thread is executing user code. Main thread executing main(), or when executing an outlined user function. This state is directly related with "user-code" event which register Work Descriptor Id.
08	NANOS_SYNCHRONIZATION	Thread is executing a synchronization mechanism: team barrier, task wait, synchronized condition (wait and signal), wait on dependencies, set/unset/try locks (through Nanos API), acquiring a single region, waitOnCondition idle loop and waking up a blocked Work Descriptor.
09	NANOS_SCHEDULING	Thread is calling a scheduler policy method, submitting a new task or a Work Descriptor is yielding thread to a new one.
10	NANOS_CREATION	Thread is creating a new Work Descriptor or setting translate function.
11	NANOS_MEM_TRANSFER_IN	Thread is copying data to cache.

Point to Point events

##	Event	Description
0	NANOS_WD_DOMAIN	WD's Identifier
1	NANOS_WD_DEPENDENCY	Data Dependency
2	NANOS_WAIT	Hierarchical Dependency (e.g "omp taskwait")
3	NANOS_WD_REMOTE	Remote Workdescriptor Execution (among nodes)
4	NANOS_XFER_PUT	Transfer PUT (Data Send)
5	NANOS_XFER_GET	Transfer GET (Data Request)

Only in cluster branch:

##	Event	Description
6	NANOS_AM_WORK	Sending WDs (master -> slave)
7	NANOS_AM_WORK_DONE	WD is done (slave->master)

Interval/Punctual events

###	Event	Description	ADV	DEV	USR
001	api	Nanos Runtime API (interval)	X	X	
002	wd-id	Work Descriptor id (interval)	X	X	
003	cache-copy-in	Transfer data into device cache (interval)			
004	cache-copy-out	Transfer data to main memory (interval)			
005	cache-local-copy	Local copy in device memory (interval)			
006	cache-malloc	Memory allocation in device cache (interval)			
007	cache-free	Memory free in device cache (interval)			
008	cache-hit	Hit in the cache (interval)			
009	copy-in	Copying WD inputs (interval)			
010	copy-out	Copying WD outputs (interval)			
011	user-funct-name	User Function Name (interval)	X	X	X
012	user-code	User Code (wd) (interval)			
013	create-wd-id	Create WD Id: (punctual)			
014	create-wd-ptr	Create WD pointer: (punctual)	X		
015	wd-num-deps	Create WD num. deps. (punctual)	X		
016	wd-deps-ptr	Create WD dependence pointer (punctual)	X		
017	lock-addr	Lock address (interval)	X		
018	num-spins	Number of Spins (punctual)	X	X	
019	num-yields	Number of Yields (punctual)	X	X	
020	time-yields	Time on Yield (in nsecs) (punctual)	X	X	
021	user-funct-location	User Function Location (interval)	X	X	X
022	num-ready	Number of ready tasks in the queues (punctual)	X	X	X
023	graph-size	Number tasks in the graph (punctual)	X	X	X
024	loop-lower	Loop lower bound (punctual)	X	X	
025	loop-upper	Loop upper (punctual)	X	X	

###	Event	Description	ADV	DEV	USR
026	loop-step	Loop step (punctual)	X	X	
027	in-cuda-runtime	Inside CUDA runtime (interval)	X	X	
028	xfer-size	Transfer size (punctual)	X	X	
029	cache-wait	Cache waiting for something (interval)			
030	chunk-size	Chunk size (punctual)	X	X	
031	num-sleeps	Number of sleeps (punctual)	X	X	
032	time-sleeps	Time on Sleep (in nsecs) (punctual)	X	X	
033	num-scheds	Number of scheduler operations (punctual)	X	X	
034	time-scheds	Time in scheduler operations (in nsecs) (punctual)	X	X	
035	sched-versioning	Versioning scheduler decisions (punctual)	X		
036	dependence	Dependence analysis, system have found a new dependence (punctual)	X	X	
037	dep-direction	Dependence direction (punctual)	X	X	
038	wd-priority	WD's priority (punctual)	X	X	
039	in-opencl-runtime	In OpenCL runtime (interval)	X	X	
040	taskwait	Taskwait (interval)	X	X	X
041	set-num-threads	Set/change number of threads (punctual)	X	X	X
042	cpuid	CPU id (punctual) - Xdisabled by default	X	X	X
043	dep-address	Dependence address (punctual)	X	X	
044	copy-data-in	Work descriptor id that is copying data in	X	X	
045	cache-copy-dada-in	Work descriptor id that is copying data in	X	X	
046	cache-copy-data-out	Work descriptor id that is copying data out	X	X	
047	sched-affinity-const	Constraint used in affinity scheduler	X	X	
048	in-mpi-runtime	Inside MPI runtime	X	X	
049	wd-ready	Workdescriptor becomes ready	X		
050	wd-blocked	Workdescriptor becomes blocked	X		

###	Event	Description	ADV	DEV	USR
051	parallel-outline-fct	Parallel outline function	X		
052	async-thread	Asynchronous thread state events	X	X	
053	copy-in-gpu	Asynchronous memory copy from host to device			
054	copy-out-gpu	Asynchronous memory copy from device to host			
055	gpu-wd-id	GPU's work descriptor id	X	X	
056	wd-criticality	Work descriptor criticality	X	X	
057	blev-overheads	Total overheads of botlev scheduler	X	X	
058	blev-overheads-break	Overheads of botlev scheduler	X	X	
059	critical-wd-id	A critical work descriptor is submitted	X	X	
060	copy-dir-devices	Asynchronous memory copy between host and device	X	X	X
061	concurrent-task	Number of concurrent task in the ready queue	X	X	
062	network-transfer	Network transfer to node	X	X	
064	thread-numa-node	NUMA node of the worker thread	X	X	
065	wd-numa-node	NUMA node assigned to the work descriptor	X	X	
066	steal	If the work descriptor to be executed is the result of a steal operation	X		

Barrier algorithms

Usage

Description: Selects the barrier algorithm used during the execution.
Type: string value.
Environment variable: `NX_BARRIER=<string>`
Command line flag: `--barrier[ |=]<string>`

List of barrier plugins

Centralized

Important: This is the default barrier algorithm

- *Configuratin string:* `NX_BARRIER=centralized` *or* `NX_ARGS="--barrier=centralized`
- *Description:* All the threads are synchronized following a centralized structure.

Tree

Warning: This barrier is experimental.

- *Configuration string:* `NX_BARRIER=tree` *or* `NX_ARGS="--barrier=tree"`
- *Description:* Threads are synchronized following a tree structure.

Dependence managers

Nanos++ provides several plugins to handle task dependencies, with different performance and features.

Usage

Description: Changes the dependency plugin to use.
Type: string value
Environment variable: `NX_DEPS=<string>`
Command line flag: `--deps[ |=]<string>`

List of dependence plugins

Plain

Important: This is the default dependence plugin

- *Configuration string:* `NX_DEPS=plain` *or* `NX_ARGS="--deps=plain"`
- *Description:* This plugin uses single memory addresses to compute dependences. In most of the cases the programmer can use a single memory point as a sentinel for a whole region.

Regions

- *Configuration string:* `NX_DEPS=regions` *or* `NX_ARGS="--deps=regions"`
- *Description:* This plugin does not partition regions. It is more suitable for task with halos.

Perfect-regions

- *Configuration string:* `NX_DEPS=perfect-regions` *or* `NX_ARSG="--deps=perfect-regions"`
- *Description:* Partitions regions into perfect regions. It is recommended for applications like multisort.

Contiguous regions

Important: This is a test and functional plugin, so it's performance may not be the best

- *Configuration string:* `NX_DEPS=cregions` *or* `NX_ARSG="--deps=cregions"`
- *Description:* Detects dependencies between regions defined by start address and end address.

Contiguous regions nocache

Important: This is a test and functional plugin, so it's performance may not be the best

- *Configuration string:* `NX_DEPS=cregions_nocache` *or* `NX_ARSG="--deps=cregions_nocache"`
- *Description:* Similar to Contiguous regions but small regions go to a dedicated structure (performance may be better in some applications).

Dependence management examples

The default plugin is called plain, and it can only handle simple dependencies that do not overlap. The following is allowed:

```
#pragma omp task inout( [NB]block )
task_1();

#pragma omp task inout( [NB]block )
task_2();
```

```
#pragma omp task inout( block[0;NB] )
task_3();
```

The following requires regions support, as provided by the regions and perfect-regions plugins:

```
#pragma omp task inout( block[0;3] )
task_4();

#pragma omp task inout( block[1;3] )
task_5();
```

Here task 5 will not depend on task 4 when using the plain plugin.

Regions are also required when you have non-contiguous memory, such as matrices. Although regions are much more powerful, there is a drawback with the current regions plugins: alignment. Keep in mind that if your data is not properly aligned, there will be a potential task serialisation and therefore a huge performance toll.

Thread Manager

The Thread Manager module controls the amount of working threads needed for a specific amount of workload. It could be useful to block idle threads when they are not needed or even to create auxiliary threads in a heavy work load scenario if we have resources to spare.

Thread Manager options

--thread-manager=<none,nanos,dlb> Select which Thread Manager will be used

--enable-dlb Enable inter-process management with Dynamic Load Balancing (DLB) library

--enable-block Enable thread blocking on idle loop

--enable-sleep Enable thread sleeping on idle loop

--sleep-time=<n> Set the amount of time (in nsec) in each sleeping phase

--enable-yield Enable thread yielding on idle loop

--yields=<n> Set number of yields before blocking

--force-tie-master Force Master WD (user code) to run on Master Thread

--warmup-threads Force the creation of as many threads as available CPUs at initialization time, then block them immediately if needed.

List of thread managers

Nanos++ provides the selection of the thread manager to use through the option `--thread-manager`. The default value is *none*, unless one or more of the options `--enable-dlb`, `--enable-yield`, `--enable-block` or `--enable-sleep` are selected, which in this case the default value is *nanos*.

Thread Manager: Nanos

If `--enable-block` is used, Nanos++ will temporarily block those threads that are not considered useful. As soon as Nanos++ detects an increase in the workload, the threads will be activated again.

Alternatively, if `--enable-sleep` is used, Nanos++ will temporarily sleep for a fixed amount of time those threads that are not considered useful. After that, the threads are activated again.

Thread Manager: DLB

Thread manager DLB (do not confuse with `--enable-dlb`, which only enables some DLB features), yields the complete thread control to the DLB library. Each thread can be individually blocked as in the Nanos Thread Manager but DLB can give the resources associated to this thread to another process.

2.3.4 Extra Modules (plug-ins)

Plugins are also used in some other library modules. This allows to implement several versions for the same algorithm and also offers an abstraction layer to access them. These extra plugins modules are:

- Slicers: tasks that can be eventually divided in more tasks.
- Worksharings: a list of work items which can be executed in parallel without creating a new task (e.g. in loops, a chunk of iterations will be a work item).

These plug-ins can be set in user source code using the proper OmpSs mechanism. Slicers implement task generating loops constructs (i.e. an OmpSs loop construct). Worksharing implement work distribution among the team of threads encountering the construct (e.g. OpenMP loop constructs).

2.4 Installation of OmpSs from git

Sometimes it may happen that you have to compile OmpSs from the git repository.

Note: There is no need to compile from git, you can always use a tarball. See *Installation of OmpSs*.

2.4.1 Additional requirements when building from git

- Automake 1.9 or better. Get it at http://ftp.gnu.org/gnu/automake
- Autoconf 2.60 or better. Get it at http://ftp.gnu.org/gnu/autoconf
- Libtool 2.2 or better. Get it at http://ftp.gnu.org/gnu/libtool
- Git. Get it at http://git-scm.com/download/linux

Note: It is likely that your Linux distribution or system already contains these tools packaged or installed. Check the documentation of your Linux distribution or ask your administrator.

2.4.2 Nanos++ from git

1. Make sure you fulfill the requirements of Nanos++. See *Nanos++ build requirements*.
2. Clone the Nanos++ repository:

```
$ git clone http://pm.bsc.es/git/nanox.git
Cloning into 'nanox'...
remote: Counting objects: 22280, done.
remote: Compressing objects: 100% (7567/7567), done.
remote: Total 22280 (delta 17397), reused 18399 (delta 14290)
```

```
Receiving objects: 100% (22280/22280), 3.50 MiB, done.
Resolving deltas: 100% (17397/17397), done.
```

3. Run `autoreconf` in the newly created `nanox` distribution:

```
$ cd nanox
$ autoreconf -fiv
autoreconf: Entering directory `.'
autoreconf: configure.ac: not using Gettext
autoreconf: running: aclocal --force -I m4
autoreconf: configure.ac: tracing
autoreconf: running: libtoolize --copy --force
libtoolize: putting auxiliary files in `.'.
libtoolize: copying file `./ltmain.sh'
libtoolize: putting macros in AC_CONFIG_MACRO_DIR, `m4'.
libtoolize: copying file `m4/libtool.m4'
libtoolize: copying file `m4/ltoptions.m4'
libtoolize: copying file `m4/ltsugar.m4'
libtoolize: copying file `m4/ltversion.m4'
libtoolize: copying file `m4/lt~obsolete.m4'
autoreconf: running: /usr/bin/autoconf --force
autoreconf: running: /usr/bin/autoheader --force
autoreconf: running: automake --add-missing --copy --force-missing
configure.ac:149: installing './ar-lib'
configure.ac:153: installing './compile'
configure.ac:10: installing './config.guess'
configure.ac:10: installing './config.sub'
configure.ac:15: installing './install-sh'
configure.ac:15: installing './missing'
Makefile.am: installing './INSTALL'
src/apis/c/debug/Makefile.am: installing './depcomp'
autoreconf: Leaving directory `.'
```

4. Now follow steps in *Installation of Nanos++*, starting from the step where you have to run `configure`.

2.4.3 Mercurium from git

You can find the instructions to build the latest version of Mercurium from our GitHub repository in the following link: https://github.com/bsc-pm/mcxx/blob/master/README.md

2.5 FAQ: Frequently Asked Questions

2.5.1 What is the difference between OpenMP and OmpSs?

Initial team and creation

You must compile with `--ompss` flag to enable the OmpSs programming model. While both programming models are pretty similar in many aspects there are some key differences.

In OpenMP your program starts with a team of one thread. You can create a new team of threads using `#pragma omp parallel` (or a combined parallel worksharing like `#pragma omp parallel for` or `#pragma omp parallel sections`).

In OmpSs your program starts with a team of threads but only one runs the `main` (or `PROGRAM` in Fortran). The remaining threads are waiting for work. You create work using `#pragma omp task` or `#pragma omp for`. One of the threads (including the one that was running `main`) will pick the created work and execute it.

This is the reason why `#pragma omp parallel` is ignored by the compiler in OmpSs mode. Combined worksharings like `#pragma omp parallel for` and `#pragma omp parallel sections` will be handled as if they were `#pragma omp for` and `#pragma omp sections`, respectively.

Mercurium compiler will emit a warning when it encounters a `#pragma omp parallel` that will be ignored.

Worksharings

In OpenMP mode, our worksharing implementation for `#pragma omp for` (and `#pragma omp parallel for`) uses the typical strategy of:

```
begin-parallel-loop
  code-of-the-parallel-loop
end-parallel-loop
```

In OmpSs mode, the implementation of `#pragma omp for` exploits a Nanos++ feature called *slicers*. Basically the compiler creates a task which will create internally several more tasks, each one implementing some part of the iteration space of the parallel loop.

These two implementations are mostly equivalent except for the following case:

```
int main(int argc, char** argv)
{
    int i;
#pragma omp parallel
    {
        int x = 0;
#pragma omp for
        for (i = 0; i < 100; i++)
        {
            x++;
        }
    }

    return 0;
}
```

In OmpSs, since `#pragma omp parallel` is ignored, there will not be an `x` variable per thread (like it would happen in OpenMP) but just an `x` shared among all the threads running the `#pragma omp for`.

2.5.2 How to create burst events in OmpSs programs

Burst events are such with begin/end boundaries. They are useful to measure when we are starting to execute a code and when we are finalizing. Lets imagine we have the following code:

```
#include <stdio.h>

#define ITERS 10

#pragma omp task
void f( int n )
{
```

```
   usleep(100);

   fprintf(stderr,"[%3d]",n);

   usleep(200);
}

int main (int argc, char *argv[] )
{
   for (int i=0; i<ITERS; i++ )
      f(i);
#pragma omp taskwait
   fprintf(stderr,"\n");
}
```

And we want to distinguish the time consumed in the first `usleep` from the second. We will call them *Phase 1* and *Phase 2* respectively. So we will need one new type of events (Key) and 2 new values with description. So first we need to declare a `nanos_event_t` variable and initializes type, key and value members. The type member will be `NANOS_BURST_START` or `NANOS_BURST_END` depending if we are open or closing the burst. We also will use `nanos_instrument_register_key` and `nanos_instrument_register_value` allowing Nanos++ to generate the appropriate Paraver configuration file. Finally we have to raise the events from the proper place. In our case we want to surround `usleep` function calls.

The code in the function task `f` will be:

```
#pragma omp task
void f( int n )
{
   nanos_event_t e1, e2;

   // Registering new event key
   nanos_instrument_register_key ( &e1.key, "phases-of-f", "Phases of f()", false );
   nanos_instrument_register_key ( &e2.key, "phases-of-f", "Phases of f()", false );

   // Registering new event values (for key "phases-of-f")
   nanos_instrument_register_value ( &e1.value, "phases-of-f", "phase-1", "Phase 1",␣
↪false);
   nanos_instrument_register_value ( &e2.value, "phases-of-f", "phase-2", "Phase 2",␣
↪false);

   // First phase
   e1.type = NANOS_BURST_START;
   nanos_instrument_events( 1, &e1);

   usleep(100);

   e1.type = NANOS_BURST_END;
   nanos_instrument_events( 1, &e1);

   fprintf(stderr,"[%3d]",n);

   // Second phase
   e2.type = NANOS_BURST_START;
   nanos_instrument_events( 1, &e2);

   usleep(200);

   e1.type = NANOS_BURST_END;
```

```
   nanos_instrument_events( 1, &e2);
}
```

Additionally you can use the `nanos_instrument_begin_burst()` and `nanos_instrument_end_burst()` which actually wrap the behaviour of opening and closing the events. And if you don't need to register the values you can use the functions `nanos_instrument_begin_burst_with_val()` and `nanos_instrument_end_burst_with_val`:

```
#pragma omp task
void f( int n )
{
   // Raising open event for phase-1
   nanos_instrument_begin_burst ( "phases-of-f", "Phases of f()", "phase-1", "Phase 1
↪" );

   usleep(100);

   // Raising close event for phase-1
   nanos_instrument_end_burst ( "phases-of-f", "phase-1" );

   fprintf(stderr,"[%3d]",n);

   // Raising open event for phase-2
   nanos_instrument_begin_burst ( "phases-of-f", "Phases of f()", "phase-2", "Phase 2
↪" );

   usleep(200);

   // Raising close event for phase-2
   nanos_instrument_end_burst ( "phases-of-f", "phase-2" );

   for(int i=0; i<n; i++) {
      nanos_event_value_t ev = i;
      // Raising open event for iteration i
      nanos_instrument_begin_burst_with_val ( "iterations-of-f", "Iterations of f()",␣
↪&ev );
      usleep(100);
      // Raising close event for iteration i
      nanos_instrument_end_burst_with_val ( "iterations-of-f", &ev );
   }
}
```

This mechanism can also be used in Fortran codes as in the follow example:

```
!$OMP TASK
SUBROUTINE F()
    IMPLICIT NONE
    CHARACTER(LEN=*) :: KEY = "phase-of-f" // ACHAR(0)
    CHARACTER(LEN=*) :: KEY_DESCR = "phase of f()" // ACHAR(0)
    CHARACTER(LEN=*) :: VAL = "phase-1" // ACHAR(0)
    CHARACTER(LEN=*) :: VAL_DESCR = "Phase 1" // ACHAR(0)
    INTEGER :: ERROR

    INTEGER, EXTERNAL :: NANOS_INSTRUMENT_BEGIN_BURST
    INTEGER, EXTERNAL :: NANOS_INSTRUMENT_END_BURST

    ERROR = NANOS_INSTRUMENT_BEGIN_BURST(KEY,KEY_DESCR,VAL, VAL_DESCR)
    CALL SLEEP(1)
```

```
    ERROR = NANOS_INSTRUMENT_END_BURST(KEY,VAL)
END SUBROUTINE F
```

2.5.3 How to execute hybrid (MPI+OmpSs) programs

You have an MPI source code annotated with OmpSs directives and you wonder how to compile and execute it. There are some additional steps in order to obtain a Paraver trace file.

Compilation

The idea here is to compile and link the source code with the Mercurium compiler adding any library, include file and flags that are normally used by the MPI compilers. Usually MPI compilers offer some option to get all the information needed. For example, OpenMPI compiler offers the flags `-showme:compile` and `-showme:link` (others have `-compile-info` and `-link-info`, consult your specific compiler documentation). To compile your application you should do something like:

```
$ mcc --ompss $(mpicc -showme:compile) -c hello.c
$ mcc --ompss $(mpicc -showme:link) hello.o -o hello
```

If you need instrumentation just remember to include the `--instrumentation` flag.

Execution

Just use the usual commands to execute an MPI application (consult your specific documentation). Regarding OmpSs the only requirements are that you export all needed variables in each MPI node. For example: To execute the previous binary with 2 MPI nodes and 3 threads per node using a SLURM queue management:

```
#@ job_name = hello
#@ total_tasks = 2
#@ wall_clock_limit = 00:20:00
#@ tasks_per_node = 1
#@ cpus_per_task = 3

export NX_ARGS="--pes 3"
srun ./hello
```

When running more than one process per node, it is recommended to double-check the CPUs used by each OmpSs process. Job schedulers usually place MPI processes into different CPU sets so they don't overlap, but mpirun does not by default. You can get which CPUs are being used by each process by adding the flag `--summary` to the NX_ARGS environment variable:

```
$ NX_ARGS="--summary" mpirun -n 4 --cpus-per-proc 4 ./a.out 2>&1 | grep CPUs
MSG: [?] === Active CPUs:    [ 4, 5, 6, 7, ]
MSG: [?] === Active CPUs:    [ 8, 9, 10, 11, ]
MSG: [?] === Active CPUs:    [ 12, 13, 14, 15, ]
MSG: [?] === Active CPUs:    [ 0, 1, 2, 3, ]
```

Instrumentation

The required steps to instrument both levels of parallelism are very similar to the process described here: *Extrae*. The only exception is that we have to preload the Extrae MPI library to intercept the MPI events.

When an MPI application is launched through an MPI driver (mpirun) or a Job Scheduler specific command (srun) it is not clear whether the environment variables are exported, or may be you need to define a variable after the MPI launcher has already done the setup but just before your binary starts. This is why we consider a good practice to define the environment variables we wish to set into a shell script file, which each spawned process will run. For instance, we create a script file trace.sh with execution permission that contains:

```
#!/bin/bash
### Other options ###
export NX_ARGS= ...
#####################
export NX_INSTRUMENTATION=extrae
export EXTRAE_CONFIG_FILE=extrae.xml
export LD_PRELOAD=$EXTRAE_HOME/lib/libnanosmpitrace.so  # or libnanosmpitracef.so for␣
↪Fortran
$*
```

We then run our MPI application like this:

```
$ mpirun --some-flags ./trace.sh ./hello_instr
```

If the `extrae.xml` was configured as recommended in the *Extrae* section, a hybrid MPI+OmpSs Paraver trace will be automatically merged after the execution is ended. Otherwise just run:

```
$ mpi2prv -f TRACE.mpits
```

2.5.4 How to exploit NUMA (socket) aware scheduling policy using Nanos++

In order to use OmpSs in NUMA system we have developed a special scheduling policy, which also supports other OmpSs features as task priorities.

We have tested this policy in machines with up to 8 NUMA nodes (48 cores), where we get about 70% of the peak performance in the Cholesky factorisation. We would appreciate if you shared with us the details of the machine where you plan to use OmpSs.

Important: This scheduling policy works best with the Portable Hardware Locality (hwloc) library. Make sure you enabled it when compiling Nanos++. Check *Nanos++ configure flags*.

Important: Memory is asigned to nodes in pages. A whole page can only belong to a single NUMA node, thus, you must make sure memory is aligned to the page size. You can use the aligned attribute or other allocation functions.

This policy assigns tasks to threads based on either data copy information or programmer hints indicating in which NUMA node that task should run.

You must select the NUMA scheduling policy when running your application. You can do so by defining NX_SCHEDULE=socket or by using –schedule=socket in NX_ARGS. Example:

```
$ NX_SCHEDULE=socket ./my_application
$ NX_ARGS="--schedule=socket" ./my_application
```

Automatic NUMA node discovery

The NUMA scheduling policy has the ability to detect initialisation tasks and track where your data is located. This option is disabled by default and can be selected by supplying –socket-auto-detect in NX_ARGS. Like this:

```
$ NX_ARGS="--schedule=socket --socket-auto-detect" ./my_application
```

This automatic feature has a few requirements and assumptions:

- First-touch NUMA policy is in effect (default in most systems).
- Data is initialised by OmpSs.
- Copies are enabled (either manually, using copy_in|out|inout; or copy_deps, enabled automatically by the compiler).
- Initialisation tasks can only be detected if they are SMP tasks with at least one output whose produced version will be one.

Important: Initialisation tasks will be assigned to NUMA nodes so that your data is initialised in round-robin. If this does not suit you, head over to the programming hints section below.

Some examples:

```
// Default OmpSs configuration: copy_deps enabled by the compiler
#pragma omp task out( [N][N]block )
void init_task( float * block )
{
}

// Alternate OmpSs configuration: copy_deps manually activated
#pragma omp target device(smp) copy_deps
#pragma omp task out( [N][N]block )
void init_task( float * block )
{
}

// No dependencies defined, copies manually defined
#pragma omp target device(smp) copy_out( [N][N]block )
#pragma omp task
void init_task( float * block )
{
}

// This one will not be detected
#pragma omp task
void not_valid_init_task( float * block )
{
}

// Nor will be this one
#pragma omp task inout( [N][N]block )
void not_init_task( float * block )
{
}
```

The rest of the application would be like a normal OmpSs one:

```
#pragma omp task inout( [N][N] block )
void compute_task( float* block )
{
    for( /* loop here */ )
    {
        // Do something
    }
}

int main(int argc, char* argv[])
{

  // Allocate matrix A

  // Call init tasks
  for( int i = 0; i < nb*nb; ++i )
  {
      init_task( A+i );
  }

  // Now compute
  for( int i = 0; i < nb*nb; ++i )
  {
      compute_task( A+i );
  }

  #pragma omp taskwait
  return 0;
}
```

Using programmer hints

This approach provides a more direct control on where to run tasks. Data copies are not required although it has the same first touch policy requirement, and your data must be also initialised by an OmpSs task.

For instance, you probably have initialisation tasks and want to spread your data over all the NUMA nodes. You must use the function `nanos_current_socket` to specify in which node the following task should be executed. Example:

```
#pragma omp task out( [N][N]block )
void init_task( float * block )
{
  // First touch NUMA policy will assign pages in the node of the current
  // thread when they are written for the first time.
}

int main(int argc, char* argv[])
{
  int numa_nodes;
  nanos_get_num_sockets( &numa_nodes );

  // Allocate matrix A

  // Call init tasks
  for( int i = 0; i < nb*nb; ++i )
  {
      // Round-robin assignment
```

```
      nanos_current_socket( i % numa_nodes );
      init_task( A+i );
  }

#pragma omp taskwait
  return 0;
}
```

Now you have the data where you want it to be. Your computation tasks must be also told where to run, to minimise access to out of node memory. You can do this the same way you do for init tasks:

```
int main( int argc, char *argv[] )
{
  // Allocation and initialisation goes above this
  for( /* loop here */ )
  {
      // Once again you must set the socket before calling the task
      nanos_current_socket( i % numa_nodes );
      compute_task( A+i );
  }
#pragma omp taskwait
}
```

Nesting

If you want to use nested tasks, you don't need to (and you should not) call nanos_current_socket() when creating the child tasks. Tasks created by another one will be run in the same NUMA node as their parent. For instance, let's say that compute_task() is indeed nested:

```
#pragma omp task inout( [N][N] block )
void compute_task( float* block )
{
  for( /* loop here */ )
  {
      small_calculation( block[i] );
  }
#pragma omp taskwait
}

#pragma omp task inout( [N]data )
void small_calculation( float* data )
{
  // Do something
}
```

In this case the scheduling policy will run small_calculation in the same NUMA node as the parent compute_task.

Other

Deepening a bit into the internals, this scheduling policy has task queues depending on the number of nodes. Threads are expected to work only on tasks that were assigned to the NUMA node they belong to, but as there might be imbalance, we have implemented work stealing.

When using stealing (enabled by default), you must consider that:

- Threads will only steal child tasks. If you have an application without nested tasks, you must change a parameter to steal from first level tasks (–socket-steal-parents).
- Threads will steal only from adjacent nodes. If you have 4 NUMA nodes in your system, a thread in NUMA node #0 will only steal from nodes 1 and 2 if the distance to 1 and 2 is (for instance) 22, and the distance to node 3 is 23. This requires your system to have a valid distance matrix (you can use numactl –hardware to check it).
- In order to prevent stealing from the same node, it will be performed in round robin. In the above example, the first time a thread in node 0 will steal from node 1, and the next time it will use node 2.

You can get the full list of options of the NUMA scheduling policy here Or using `nanox --help`.

2.5.5 My application crashes. What can I do?

Does it crash at the beginning or at the end?

Check if Nanos++ was built with the **allocator disabled**.

Get a backtrace

The first step would be to obtain a **backtrace** using `gdb`. (see GNU GDB website).

Ways to run gdb

A typical approach could be:

```
$ gdb --args program parameters
```

If your applications needs some `NX_ARGS` you can pass them like this:

```
$ NX_ARGS="..." gdb --args program parameters
```

or set in the environment using `export` or `setenv` as usual before running `gdb`.

If you are running in a batch queue (not interactively) you will have to use `gdb` in batch mode. Pass the flags `--batch` and `-ex` like this:

```
$ NX_ARGS="..." gdb --batch -ex "run" -ex "some-gdb-command" program parameters
```

You may pass more than one `-ex` command and they will be run sequentially.

Instead of `-ex` you may write the commands in a file and pass the parameter `--command`. Check the GNU GDB documentation.

Backtrace

Once the program crashes inside `gdb` it is time to get a backtrace. If your program has more than one thread the usual `backtrace` (or its short form `bt`) will not give all the information we want. Make sure you use `thread apply all backtrace` command (or its short form `thread apply all bt`).

If you are running in batch, your command will look as follows:

```
$ NX_ARGS="..." gdb --batch -ex "run" -ex "thread apply all bt" program parameters
```

I cannot get a backtrace

If there is no backtrace, this can be a symptom of a **low stack size**. You can increase it with *NX_STACK_SIZE* (see *Running OmpSs Programs*).

If this does not solve your problem, we suggest to follow these steps:

- Comment all tasks.
- Run with only one thread.
- Activate the tasks again, but follow them by a `#pragma omp taskwait`
- Remove the unneeded `taskwait` one by one.
- Increase the number of threads.

Chances are that you will be able to diagnose your problem after one of these steps.

2.5.6 I am trying to use regions, but tasks are serialised and I think they should not

The two available region plugins have an important limitation: you have to **align memory** to a power of 2. This is an optimisation trade-off (that might be lifted in the distant future).

If you have two matrices of size (32-bit floats), and you want to work with blocks of elements, you must align to . The block size does not affect us in this case.

You must carefully choose a way to align memory, depending on your system. We have found problems with static alignment, while posix_memalign usually works for us.

To see if there is, effectively, aliasing, you may want to run you program with the **graph** instrumentation plugin (*Task Dependency Graph*). If you see lots of anti/output dependencies, you are likely not aligning your memory properly.

Alignment rules: For every dimension, you must ensure that a rule is true:

Where is an address, is the block size **in bytes** in dimension , the size **in number of elements** in dimension .

For the following example of 2 dimensions:

- The address of a block module the block width must be equal to 0.
- The address of a block module the total width multiplied by the block height must be also 0.

In some cases, if your application uses overlapping blocks with halos, you'll need to make sure the start position of each block is aligned. Consider the next matrix (6x6):

0	0	0	0	0	0
0	1	1	2	2	0
0	1	1	2	2	0
0	3	3	4	4	0
0	3	3	4	4	0
0	0	0	0	0	0

There are 4 blocks (1, 2, 3 and 4) of 4 elements each. If the application has a halo as an input, and the block as an output dependency, it would look like this for the first block:

X	I	I	X	X	X
I	O	O	I	X	X
I	O	O	I	X	X
X	I	I	X	X	X
X	X	X	X	X	X
X	X	X	X	X	X

X, I, O denotes no dependency, input and output, respectively.

A first approach would be aligning to the next power of 2 of , but it is not aligned to a power of 2.

Besides that, the blocks are not properly aligned. The first block (1,1) would be in address (0x1C). is 4, not a power of 2.

To solve this, you must pad the data. In this case, we need to allocate an 8x8 matrix and pad data the following way (although we only need 6 rows to comply with the alignment rules):

P	P	P	P	P	P	P	P
P	0	0	0	0	0	0	P
P	0	1	1	2	2	0	P
P	0	1	1	2	2	0	P
P	0	3	3	4	4	0	P
P	0	3	3	4	4	0	P
P	0	0	0	0	0	0	P
P	P	P	P	P	P	P	P

P is the padded data. It does not need to be initialised (thus some those pages will not be used).

Then, for the first block the dependencies will be:

X	X	X	X	X	X	X	X
X	X	I	I	X	X	X	X
X	I	O	O	I	X	X	X
X	I	O	O	I	X	X	X
X	X	I	I	X	X	X	X
X	X	X	X	X	X	X	X
X	X	X	X	X	X	X	X
X	X	X	X	X	X	X	X

The address of the output block (2,2) is . Its mod is 0, so the rule is satisfied for the first dimension.

For the second rule, . Now we apply the first rule for 8 as the address, and its module is 0. It will now work.

2.5.7 My application does not run as fast as I think it could

There are a few runtime options you can tweak in order to get better results.

- Compile in performance mode (do not use *–instrument* or *–debug*).
- Adjust your program parameters (such as problem size, block size, etc.). Watch out for too fine grain tasks.
- Disable thread sleep, or reduce/increase sleep time.
- Reduce or increase the spin time. If you increase it, this might put more stress on the tasks queue(s).

2.5.8 How to run OmpSs on Blue Gene/Q?

Installation

Note: If you want IBM compilers support, make sure that the full-name drivers (e.g. powerpc64-bgq-linux-xlc) are in the PATH. You can usually do so by setting the PATH variable before the configuration step:

```
$ export PATH=/bgsys/drivers/ppcfloor/gnu-linux/powerpc64-bgq-linux/bin/:$PATH
```

You don't need to keep this change after the configuration step.

Follow the instructions described here: *Installation of Nanos++* and *Installation of Mercurium C/C++/Fortran source-to-source compiler*, and add these options for each configuration:

- For Nanos++:

```
$ ./configure --host=powerpc64-bgq-linux ...
```

- For Mercurium:

```
$ ./configure --target=powerpc64-bgq-linux ...
```

Finally compile and install as usual.

Compiling your application

Mercurium will install the drivers using the target keyword as the prefix, so to build your application:

```
$ powerpc64-bgq-linux-mcc --ompss test.c -o test
```

For MPI applications, you can use the MPI wrapper as long as the MPI implementation allows to set the native compiler. For example, if using MPICH:

```
$ export MPICH_CC="powerpc64-bgq-linux-xlmcc --ompss"
$ mpicc test.c -o test.c
```

Instrumenting

For non-MPI applications, you can follow the same instructions described here: *Extrae*

For MPI applications, you should follow the same steps but preloading the Extrae MPI library so that it can intercept the MPI calls. Blue Gene/Q does not allow this method so you will have to link your application with Extrae. You will find an example of how to do it in the Extrae installation directory for BG/Q '$EXTRAE_HOME/share/example/MPI/'.

2.5.9 Why macros do not work in a #pragma?

OpenMP compilers usually expand macros inside `#pragma omp`, so why macros do not work in Mercurium?

The reason is that when Mercurium compiles an OmpSs/OpenMP program we do not enable OpenMP in the native compiler to reduce the chance of unwanted interferences. This implies that the `#pragma omp` is not known to the native preprocessor, which leaves the `#pragma omp` as is. As a consequence, macros are not expanded and they become unusable inside pragmas.

Alternatives

Although macros are not expanded in clauses, most of the cases they are used for integer constants:

```
#define N 100

#pragma omp task out([N]a)
void f(int *a);

void g()
{
  int a[N];

  f(a);
#pragma omp taskwait
}
```

The example shown above will not work but fortunately in these cases one can avoid macros. A first approach replaces the macro with an enumerator. This is the most portable approach:

```
enum { N = 100 };
// Rest of the code
```

Another approach uses an `const` integer:

```
const int N = 100;
// Rest of the code
```

Note: In C a `const` qualified variable is not a constant expression whereas in C++ it may be (more information). Mercurium, though, allows this case as a constant expression in both languages.

In case you need to change the value at compile time, you can use the following strategy. Here assume that `N_` is a macro that you change through the invocation of the compiler (typically with a compiler flag like `-DN_=100`):

```
enum { N = N_ };
// Rest of the code
```

Another feasible scenario where macros and pragmas are combined is when we want to generate the pragma itself via macros. To do that, we could use the `_Pragma` operator introduced in `C99` and `C++11`:

```
#define STRINGIFY(X) #X
#define MY_PRAGMA(X) _Pragma(STRINGIFY(X))
#define MY_OPENMP_PARALLEL_LOOP omp parallel for

int main() {
    MY_PRAGMA(MY_OPENMP_PARALLEL_LOOP)
    for(int i = 0; i < 10; ++i) {
        // ...
    }
}
```

Why rely on the native preprocessor, then?

We could provide our own preprocessor, but this adds a lot of complication just to support this case (and as shown in *Alternatives* the typical use case can be worked around).

Complications include, but do not limit to:

- it is not trivial to know where are the system headers. While `gcc` provides good support to know which paths it is using, not all compiler vendors provide this information. Even worse, some parameters may change the used paths between invocations of the native compiler
- subtleties among preprocessor implementations. Not all preprocessors behave the same way and sometimes these differences are relevant for the native compiler headers

2.5.10 How to track dependences for a given task using paraver

Once we have a Paraver trace we can track task execution by filtering the proper events or using the appropriate Paraver configuration file. Nanos also include an script that can dynamically generate a Paraver configuration file filtering a task (using its task id), the dependences it has during the execution and where the task was generated.

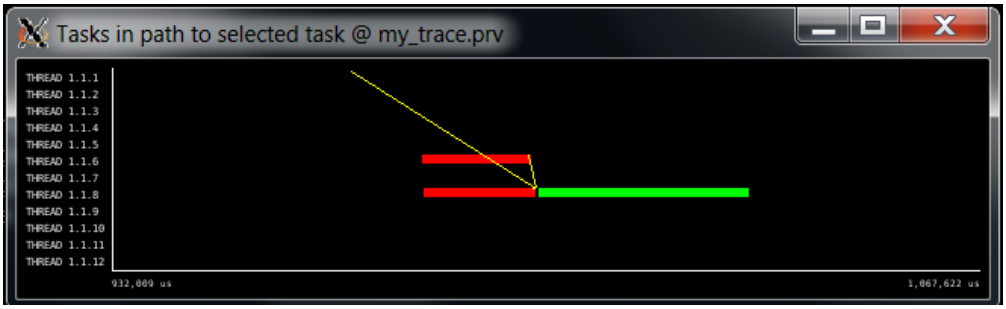
Before using the command you should have the trace loaded in Paraver. You will also need to load the configuration file displaying task numbers and find out the task number of the task you are interested in. This command will help you identify the incoming dependences of the selected task(s).

Usage: track_deps.sh trace.prv list_of_task_numbers

Functionality

The script generates and loads a configuration file that shows only the specified tasks, the tasks on which they depend and the dependences between them. It writes to standard output the numbers of the task so that the command can be used iteratively to explore the full dependency chain.

The following figure shows the results of executing the track_deps.sh script over a Cholesky kernel Paraver trace. We have asked to show the dependences of a task (in the example task id = 25). We can see where the task was created and which tasks it depends on.



- genindex

CHAPTER

THREE

OMPSS EXAMPLES AND EXERCISES

The information included in this document is provided "as is", with no warranties whatsoever, including any warranty of merchantability, fitness for any particular purpose, or any warranty otherwise arising out of any proposal, specification, or sample. The document is not guaranteed to be complete and/or error-free at this stage and it is subject to changes without furthernotice. Barcelona Supercomputing Center will not assume any responsibility for errors or omissions in this document. Please send comments, corrections and/or suggestions to pm-tools at bsc.es. This document is provided for informational purposes only.

Note: A PDF version of this document is available in http://pm.bsc.es/ompss-docs/examples/OmpSsExamples.pdf, and all the example source codes in http://pm.bsc.es/ompss-docs/examples/ompss-ee.tar.gz

3.1 Introduction

This documentation contains examples and exercises using the OmpSs programming model. The main objective of this document is to provide guidance in learning OmpSs programming model and serve as teaching materials in courses and tutorials. To find more complete applications please visit our BAR (BSC Application Repository) in the URL:

http://pm.bsc.es/projects/bar

3.1.1 System configuration

In this section we describe how to tune your configure script and also how to use it to configure your environment. If you have a pre-configured package you can skip this section and simply run the Linux command `source` using the provided configure script:

```
$source configure.sh
```

The configure script is used to set all environment variables you need to properly execute OmpSs applications. Among other things it contains the PATH where the system will look for to find Mercurium compiler utility, and the MKL installation directory (if available) to run specific OmpSs applications (e.g. Cholesky kernel).

Once you have configured the script you will need to to run the Linux command `source` using your configure script as described in the beginning of this section.

3.1.2 Building the examples

The make utility

In software development, Make is a utility that automatically builds executable programs and libraries from source code by reading files called makefiles which specify how to derive the target program. Though integrated development environments and language-specific compiler features can also be used to manage a build process, Make remains widely used, especially in Unix.

Make searches the current directory for the makefile to use, e.g. GNU make searches files in order for a file named one of GNUmakefile, makefile, Makefile and then runs the specified (or default) target(s) from (only) that file.

A makefile consists of rules. Each rule begins with a textual dependency line which defines a target followed by a colon (:) and optionally an enumeration of components (files or other targets) on which the target depends. The dependency line is arranged so that the target (left hand of the colon) depends on components (right hand of the colon). It is common to refer to components as prerequisites of the target:

```
target [target...] : [component...]
[<TAB>] command-1
[<TAB>] command-2
...
[<TAB>] command-n
[target
```

Below is a very simple makefile that by default compiles the program helloworld (first target: all) using the GCC C compiler (CC) and using "-g" compiler option (CFLAGS). The makefile also provides a ''clean" target to remove the generated files if the user desires to start over (by running make clean):

```
CC = gcc
CFLAGS = -g
LDFLAGS =

all: helloworld

helloworld: helloworld.o
   # Commands start with TAB not spaces !!!
   $(CC) $(LDFLAGS) -o $@ $^

helloworld.o: helloworld.c
   $(CC) $(CFLAGS) -c -o $@ $<

clean:
   rm -f helloworld helloworld.o
```

Examples Makefiles

All the examples and exercises comes with a makefile (Makefile) configured to compile 3 different versions for each program. Each of the binary file name created by running make ends with a suffix which determines the version:

- program-p: performance version
- program-i: instrumented version
- program-d: debug version

You can actually select which version you want to compile by executing: ''make program-version" (e.g. in the Cholesky kernel you can compile the performance version executing ''make cholesky-p''. By default (running make with no parameters) all the versions are compiled.

Apart of building the program's binaries, the make utility will also build shell scripts to run the program. Each exercise have two running scripts, one to run a single program execution (''run-once.sh''), the other will run multiples configurations with respect the number of threads, data size, etc (''multirun.sh''). Before submitting any job, make sure all environment variables have the values you expect to. Here is an example of the ''run-once.sh" script:

```
#!/bin/bash
export NX_SMP_WORKERS=4

./cholesky-p 4096 512 1
```

In some cases the shell script will contain job scheduler variables declared in top of the script file. A job scheduler script must contain a series of directives to inform the batch system about the characteristics of the job. These directives appear as comments in the job script file and the syntax will depend on the job scheduler system used.

With the running scripts it also comes a ''trace.sh" file, which can be used to include all the environment variables needed to get an instrumentation trace of the execution. The content of this file is:

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

Additionally, you will need to change your running script in order to invoke the your program through the ''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 in between instrumented and non-instrumented executions. When you want to instrument you will need to include ''trace.sh" before your program execution command line:

```
#!/bin/bash
export NX_SMP_WORKERS=1

./trace.sh ./cholesky-i 4096 512 1
```

Finally, the make utility will generate (if not already present in the directory) other configuration files as it is the case of ''extrae.xml" file (used to configure Extrae plugin in order to get a Paraver trace, see ''trace.sh" file).

3.1.3 Job Scheduler: Minotauro

The current section has a short explanation on how to use the job scheduler systems installed in BSC's Minotauro machine. Slurm is the utility used in this machine for batch processing support, so all jobs must be run through it. These are the basic directives to submit jobs:

- mnsubmit job_script submits a ''job script" to the queue system (see below for job script directives).
- mnq: shows all the submitted jobs.
- mncancel <job_id> remove the job from the queue system, cancelling the execution of the processes, if they were still running.

A job must contain a series of directives to inform the batch system about the characteristics of the job. These directives appear as comments in the job script, with the following syntax:

```
# @ directive = value.
```

The job would be submitted using: ''mnsubmit <job_script>''. While the jobs are queued, you can check their status using the command ''mnq" (it may take a while to start executing). Once a job has been executed you will get two files. One for console standard output (with .out extension) and other for console standard error (with .err extension).

3.1.4 Job Scheduler: Marenostrum

LSF is the utility used at MareNostrum III for batch processing support, so all jobs must be run through it. This section provides information for getting started with job execution at the Cluster. These are the basic commands to submit, control and check your jobs:

- bsub < job_script: submits a ''job script" passed through standard input (STDIN) to the queue system.
- bjobs: shows all the submitted jobs
- bkill <job_id>: remove the job from the queue system, canceling the execution of the processes, if they were still running.
- bsc_jobs: shows all the pending or running jobs from your group.

3.1.5 Document's contributions

The OmpSs Examples and Exercises document is written using Sphinx

http://www.sphinx-doc.org/

1. Make sure you have sphinx-doc in your machine

   Ubuntu/Debian:

   $ sudo apt-get install sphinx-doc python-sphinx texlive-latex-extra texlive-fonts-recommended

   (Note: texlive- packages are required to build PDF documentation).

2. Make changes to .rst files

   Start from index.rst to see the structure. Look at the .. toctree::, it lists the included files used to generate the documentation (toctree stands for "tree of the table of contents").

   Syntax of .rst is reStructuredText. You may want to read a quick introduction at

   http://www.sphinx-doc.org/rest.html

   The official reStructuredText documentation (if you want to dig further in the details) is in:

   http://docutils.sourceforge.net/rst.html#user-documentation

3. Generate the documentation

3.1. Generate the HTML

$ make html

Now open your browser to .build/html/index.html and behold your contribution.

3.2. Generate the PDF

$ make latexpdf

Now open your PDF viewer to the .build/html/<docfile>.pdf (the file depends on the directory you chose in the step 0 above)

4. Commit your changes using git

   $ git commit -a $ git push

It may happen that the remote repository changed where you were editing your local one. In that case, first do

$ git pull –rebase

and then proceed as above.

$ git commit -a $ git push

3.2 Writing OmpSs programs

Following examples are written in C/C++ or Fortran using OmpSs as a programming model. With each example we provide simple explanations on how they are annotated and, in some cases, how they can be compiled (if a full example is provided).

3.2.1 Data Management

Reusing device data among same device kernel executions

Although memory management is completely done by the runtime system, in some cases we can assume a predefined behaviour. This is the case of the following Fortran example using an OpenCL kernel. If we assume runtime is using a `write-back` cache policy we can also determine that second kernel call will not imply any data movement.

kernel_1.cl:

```
__kernel void vec_sum(int n, __global int* a, __global int* b, __global int* res)
{
    const int idx = get_global_id(0);

    if (idx < n) res[idx] = a[idx] + b[idx];
}
```

test_1.f90:

```
! NOTE: Assuming write-back cache policy

SUBROUTINE INITIALIZE(N, VEC1, VEC2, RESULTS)
    IMPLICIT NONE
    INTEGER :: N
    INTEGER :: VEC1(N), VEC2(N), RESULTS(N), I
    DO I=1,N
    VEC1(I) = I
    VEC2(I) = N+1-I
    RESULTS(I) = -1
    END DO
END SUBROUTINE INITIALIZE

PROGRAM P
    IMPLICIT NONE
    INTERFACE
        !$OMP TARGET DEVICE(OPENCL) NDRANGE(1, N, 128) FILE(kernel_1.cl) COPY_DEPS
        !$OMP TASK IN(A, B) OUT(RES)
        SUBROUTINE VEC_SUM(N, A, B, RES)
            IMPLICIT NONE
            INTEGER, VALUE :: N
            INTEGER :: A(N), B(N), RES(N)
        END SUBROUTINE VEC_SUM
    END INTERFACE
    INTEGER, PARAMETER :: N  = 20
    INTEGER :: VEC1(N), VEC2(N), RESULTS(N), I

    CALL INITIALIZE(N, VEC1, VEC2, RESULTS)
```

```
    CALL VEC_SUM(N, VEC1, VEC2, RESULTS)
    !  The vectors VEC1 and VEC2 are sent to the GPU. The input transfers at this
    !   point are: 2 x ( 20 x sizeof(INTEGER)) = 2 x (20 x 4) = 160 B.

    CALL VEC_SUM(N, VEC1, RESULTS, RESULTS)
    !  All the input data is already in the GPU. We don't need to send
    !   anything.

    !$OMP TASKWAIT
    !  At this point we copy out from the GPU the computed values of RESULTS
    !   and remove all the data from the GPU

    ! print the final vector's values
    PRINT *, "RESULTS: ", RESULTS
END PROGRAM P

!  Expected IN/OUT transfers:
!   IN = 160B
!   OUT = 80B
```

Compile with:

```
oclmfc -o test_1 test_1.f90 kernel_1.cl --ompss
```

Forcing data back using a taskwait

In this example, we need to copy back the data in between the two kernel calls. We force this copy back using a `taskwait`. Note that we are assuming `write-back` cache policy.

kernel_2.cl:

```
__kernel void vec_sum(int n, __global int* a, __global int* b, __global int* res)
{
    const int idx = get_global_id(0);

    if (idx < n) res[idx] = a[idx] + b[idx];
}
```

test_2.f90:

```
! NOTE: Assuming write-back cache policy

SUBROUTINE INITIALIZE(N, VEC1, VEC2, RESULTS)
    IMPLICIT NONE
    INTEGER :: N
    INTEGER :: VEC1(N), VEC2(N), RESULTS(N), I
    DO I=1,N
    VEC1(I) = I
    VEC2(I) = N+1-I
    RESULTS(I) = -1
    END DO
END SUBROUTINE INITIALIZE

PROGRAM P
    IMPLICIT NONE
    INTERFACE
```

```
        !$OMP TARGET DEVICE(OPENCL) NDRANGE(1, N, 128) FILE(kernel_2.cl) COPY_DEPS
        !$OMP TASK IN(A, B) OUT(RES)
        SUBROUTINE VEC_SUM(N, A, B, RES)
            IMPLICIT NONE
            INTEGER, VALUE :: N
            INTEGER :: A(N), B(N), RES(N)
        END SUBROUTINE VEC_SUM
    END INTERFACE
    INTEGER, PARAMETER :: N  = 20
    INTEGER :: VEC1(N), VEC2(N), RESULTS(N), I

    CALL INITIALIZE(N, VEC1, VEC2, RESULTS)

    CALL VEC_SUM(N, VEC1, VEC2, RESULTS)
    !  The vectors VEC1 and VEC2 are sent to the GPU. The input transfers at this
    !   point are: 2 x ( 20 x sizeof(INTEGER)) = 2 x (20 x 4) = 160 B.

    !$OMP TASKWAIT
    !  At this point we copy out from the GPU the computed values of RESULT
    !   and remove all the data from the GPU

    PRINT *, "PARTIAL RESULTS: ", RESULTS

    CALL VEC_SUM(N, VEC1, RESULTS, RESULTS)
    !  The vectors VEC1 and RESULT are sent to the GPU. The input transfers at this
    !   point are: 2 x ( 20 x sizeof(INTEGER)) = 2 x (20 x 4) = 160 B.

    !$OMP TASKWAIT
    !  At this point we copy out from the GPU the computed values of RESULT
    !   and remove all the data from the GPU

    ! print the final vector's values
    PRINT *, "RESULTS: ", RESULTS
END PROGRAM P

!  Expected IN/OUT transfers:
!   IN = 320B
!   OUT = 160B
```

Compile with:

```
oclmfc -o test_2 test_2.f90 kernel_2.cl --ompss
```

Forcing data back using a task

This example is similar to the example 1.2 but instead of using a `taskwait` to force the copy back, we use a task with copies. Note that we are assuming `write-back` cache policy.

kernel_3.cl:

```
__kernel void vec_sum(int n, __global int* a, __global int* b, __global int* res)
{
    const int idx = get_global_id(0);

    if (idx < n) res[idx] = a[idx] + b[idx];
}
```

test_3.f90:

```
! NOTE: Assuming write-back cache policy

SUBROUTINE INITIALIZE(N, VEC1, VEC2, RESULTS)
    IMPLICIT NONE
    INTEGER :: N
    INTEGER :: VEC1(N), VEC2(N), RESULTS(N), I
    DO I=1,N
    VEC1(I) = I
    VEC2(I) = N+1-I
    RESULTS(I) = -1
    END DO
END SUBROUTINE INITIALIZE

PROGRAM P
    IMPLICIT NONE
    INTERFACE
        !$OMP TARGET DEVICE(OPENCL) NDRANGE(1, N, 128) FILE(kernel_3.cl) COPY_DEPS
        !$OMP TASK IN(A, B) OUT(RES)
        SUBROUTINE VEC_SUM(N, A, B, RES)
            IMPLICIT NONE
            INTEGER, VALUE :: N
            INTEGER :: A(N), B(N), RES(N)
        END SUBROUTINE VEC_SUM

        !$OMP TARGET DEVICE(SMP) COPY_DEPS
        !$OMP TASK IN(BUFF)
        SUBROUTINE PRINT_BUFF(N, BUFF)
            IMPLICIT NONE
            INTEGER, VALUE :: N
            INTEGER :: BUFF(N)
        END SUBROUTINE VEC_SUM
    END INTERFACE

    INTEGER, PARAMETER :: N  = 20
    INTEGER :: VEC1(N), VEC2(N), RESULTS(N), I

    CALL INITIALIZE(N, VEC1, VEC2, RESULTS)

    CALL VEC_SUM(N, VEC1, VEC2, RESULTS)
    !  The vectors VEC1 and VEC2 are sent to the GPU. The input transfers at this
    !   point are: 2 x ( 20 x sizeof(INTEGER)) = 2 x (20 x 4) = 160 B.

    CALL PRINT_BUFF(N, RESULTS)
    !  The vector RESULTS is copied from the GPU to the CPU. The copy of this vector␣
↪in
    !   the memory of the GPU is not removed because the task 'PRINT_BUFF' does not␣
↪modify it.
    !   Output transfers: 80B.
    !   VEC1 and VEC2 are still in the GPU.

    CALL VEC_SUM(N, VEC1, RESULTS, RESULTS)
    !  The vectors VEC1 and RESULTS are already in the GPU. Do not copy anything.

    CALL PRINT_BUFF(N, RESULTS)
    !  The vector RESULTS is copied from the GPU to the CPU. The copy of this vector␣
↪in
    !   the memory of the GPU is not removed because the task 'PRINT_BUFF' does not␣
↪it.
```

```
    !   Output transfers: 80B.
    !   VEC1 and VEC2 are still in the GPU.

    !$OMP TASKWAIT
    !  At this point we remove all the data from the GPU. The right values of the␣
↪vector RESULTS are
    !   already in the memory of the CPU, then we don't need to copy anything from␣
↪the GPU.

END PROGRAM P

SUBROUTINE PRINT_BUFF(N, BUFF)
    IMPLICIT NONE
    INTEGER, VALUE :: N
    INTEGER :: BUFF(N)

    PRINT *, "BUFF: ", BUFF
END SUBROUTINE VEC_SUM

!  Expected IN/OUT transfers:
!   IN = 160B
!   OUT = 160B
```

Compile with:

```
oclmfc -o test_3 test_3.f90 kernel_3.cl --ompss
```

3.2.2 Application's kernels

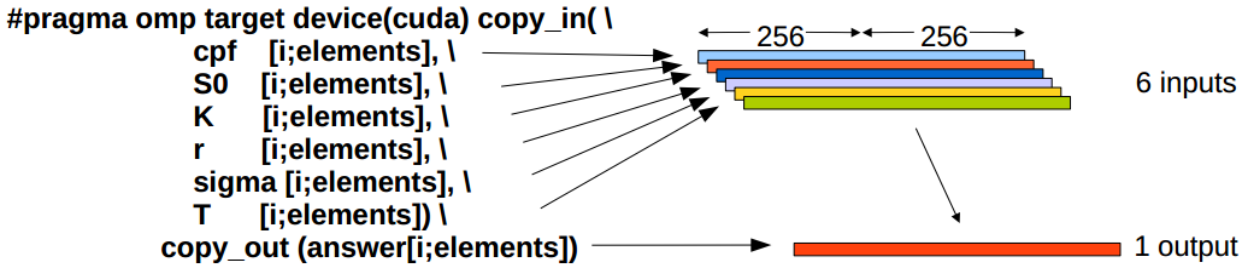
BlackScholes

This benchmark computes the pricing of European-style options. Its kernel has 6 input arrays, and a single output. Offloading is done by means of the following code:

```
for (i=0; i<array_size; i+= chunk_size ) {
  int elements;
  unsigned int * cpf;
  elements = min(i+chunk_size, array_size ) - i;
  cpf = cpflag;
#pragma omp target device(cuda) copy_in( \
                     cpf    [i;elements], \
                     S0     [i;elements], \
                     K      [i;elements], \
                     r      [i;elements], \
                     sigma  [i;elements], \
                     T      [i;elements]) \
                 copy_out (answer[i;elements])
#pragma omp task firstprivate(local_work_group_size, i)
     {
         dim3 dimBlock(local_work_group_size, 1 , 1);
         dim3 dimGrid(elements / local_work_group_size, 1 , 1 );
         cuda_bsop <<<dimGrid, dimBlock>>> (&cpf[i], &S0[i], &K[i],

                      &r[i], &sigma[i], &T[i], &answer[i]);
     }
```

```
  }
#pragma omp taskwait
```

Following image shows graphically the annotations used to offload tasks to the GPUs available. Data arrays annotated with the copy_in clause are automatically transferred by the Nanos++ runtime system onto the GPU global memory. After the CUDA kernel has been executed, the copy_out clause indicates to the runtime system that the results written by the GPU onto the output array should be synchronized onto the host memory. This is done at the latest when the host program encounters the `taskwait` directive.



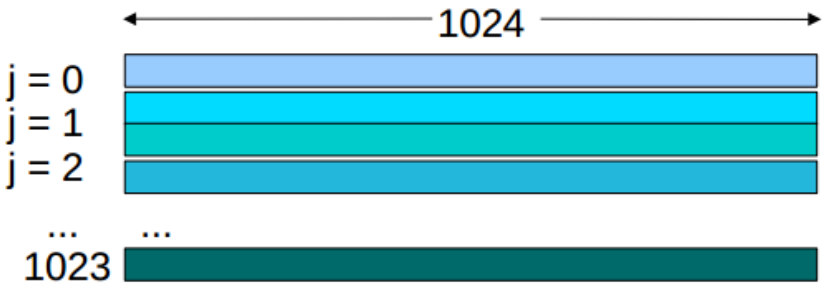
Perlin Noise

This benchmark generates an image consisting of noise, useful to be applied to gaming applications, in order to provide realistic effects. The application has a single output array, with the generated image. Annotations are shown here:

```
for (j = 0; j < img_height; j+=BS) {
  // Each task writes BS rows of the image
  #pragma omp target device(cuda) copy_deps
  #pragma omp task output (output[j*rowstride:(j+BS)*rowstride-1])
  {
    dim3 dimBlock;
    dim3 dimGrid;
    dimBlock.x = (img_width < BSx) ? img_width : BSx;
    dimBlock.y = (BS < BSy) ? BS : BSy;
    dimBlock.z = 1;
    dimGrid.x = img_width/dimBlock.x;
    dimGrid.y = BS/dimBlock.y;
    dimGrid.z = 1;

    cuda_perlin <<<dimGrid, dimBlock>>> (&output[j*rowstride], time, j, rowstride);
  }
}
#pragma omp taskwait noflush
```

In this example, the `noflush` clause eliminates the need for the data synchronization implied by the `taskwait` directive. This is useful when the programmer knows that the next task that will be accessing this result will also be executed in the GPUs, and the host program does not need to access it. The runtime system ensures in this case that the data is consistent across GPUs.

Following image shows the graphical representation of the data, and the way annotations split it across tasks.




```
for (j = 0; j < img_height; j+=BS) {

#pragma omp target device(cuda)  copy_deps
#pragma omp task output (output[j*rowstride:(j*BS)*rowstride-1])
```

N-Body

This benchmark implements the gravitational forces among a set of particles. It works with an input array (this_particle_array), and an output array (output_array). Mass, velocities, and positions of the particles are kept updated alternatively in each array by means of a pointer exchange. The annotated code is shown here:

```
void Particle_array_calculate_forces_cuda ( int number_of_particles,
              Particle this_particle_array[number_of_particles],
              Particle output_array[number_of_particles],
              float time_interval )
{
  const int bs = number_of_particles/8;
  size_t num_threads, num_blocks;
  num_threads = ((number_of_particles < MAX_NUM_THREADS) ?
                Number_of_particles :  MAX_NUM_THREADS );
  num_blocks = ( number_of_particles + MAX_NUM_THREADS ) / MAX_NUM_THREADS;
  #pragma omp target device(cuda) copy_deps
  #pragma omp task output( output_array) input(this_particle_array )
    calculate_forces_kernel_naive <<< num_blocks, MAX_NUM_THREADS >>>
                              (time_interval, this_particle_array, number_of_particles,
                               &output_array[first_local], first_local, last_local);
  #pragma omp taskwait
}
```

3.3 Examples Using OmpSs

3.3.1 Introduction

In this section we include several OmpSs applications that are already parallelized (i.e. annotated with OmpSs directives). Users have not to change the code, but they are encouraged to experiment with them. You can also use that source directory to experiment with the different compiler and runtime options, as well as the different instrumentation plugins provided with your OmpSs installation.

3.3.2 Cholesky kernel

This example shows the Cholesky kernel. This algorithm is a decomposition of a Hermitian, positive-definite matrix into the product of a lower triangular matrix and its conjugate transpose.

Note: You can dowload this code visiting the url http://pm.bsc.es *OmpSs Examples and Exercises*'s (code) link. The Cholesky kernel is included inside the *01-examples*'s directory.

The kernel uses four different linear algorithms: potrf, trsm, gemm and syrk. Following code shows the basic pattern for a Cholesky factorisation:

```
for (int k = 0; k < nt; k++) {
   // Diagonal Block factorization
   omp_potrf (Ah[k][k], ts, ts);

   // Triangular systems
   for (int i = k + 1; i < nt; i++)
      omp_trsm (Ah[k][k], Ah[k][i], ts, ts);

   // Update trailing matrix
   for (int i = k + 1; i < nt; i++) {
      for (int j = k + 1; j < i; j++)
         omp_gemm (Ah[k][i], Ah[k][j], Ah[j][i], ts, ts);

      omp_syrk (Ah[k][i], Ah[i][i], ts, ts);
   }

}
```

In this case we parallelize the code by annotating the kernel functions. So each call in the previous loop becomes the instantiation of a task. The following code shows how we have parallelized Cholesky:

```
#pragma omp task inout([ts][ts]A)
void omp_potrf(double * const A, int ts, int ld)
{
   ...
}

#pragma omp task in([ts][ts]A) inout([ts][ts]B)
void omp_trsm(double *A, double *B, int ts, int ld)
{
   ...
}

#pragma omp task in([ts][ts]A) inout([ts][ts]B)
void omp_syrk(double *A, double *B, int ts, int ld)
{
   ...
}

#pragma omp task in([ts][ts]A, [ts][ts]B) inout([ts][ts]C)
void omp_gemm(double *A, double *B, double *C, int ts, int ld)
{
   ...
}
```

Note that for each of the dependences we also specify which is the matrix (block) size. Although this is not needed,

due there is no overlapping among the different blocks, it will allow the runtime to compute dependences using the region mechanism.

Goals of this exercise

- Code is completely annotated: you DON'T need to modify it
- Review source code and check the different directives and their clauses. Try to understand what they mean.
- Check different compiled versions (performance, instrumented & debug)
- Check other runtime options (schedulers, throttle,...)
- Check (scalability), execute the program using different number of threads and compute the speed-up
- Get a task dependency graph to analyse dependences
- Get different paraver traces and visualize them: thread state, task name,...

3.3.3 Stream Benchmark

The stream benchmark is part of the HPC Challenge benchmarks (http://icl.cs.utk.edu/hpcc/) and here we present two versions: one that inserts barriers and another without barriers. The behavior of version with barriers resembles the OpenMP version, where the different functions (Copy, Scale, ...) are executed one after another for the whole array while in the version without barriers, functions that operate on one part of the array are interleaved and the OmpSs runtime keeps the correctness by means of the detection of data-dependences.

Note: You can dowload this code visiting the url http://pm.bsc.es *OmpSs Examples and Exercises*'s (code) link. The Stream benchmark is included inside the *01-examples*'s directory.

3.3.4 Array Sum Benchmark (Fortran version)

This benchmark computes the sum of two arrays and stores the result in an other array.

Note: You can dowload this code visiting the url http://pm.bsc.es *OmpSs Examples and Exercises*'s (code) link. The Array Sum benchmark is included inside the *01-examples*'s directory.

In this case we annotate the algorithm using the Fortran syntax. The benchmark compute a set of array sums. The first inner loop initializes one array, that will be computed in the second inner loop. Dependences warrant proper execution and synchronization between initialization and compute results:

```
DO K=1,1000
   IF(MOD(K,100)==0) WRITE(0,*) 'K=',K
   ! INITIALIZE THE ARRAYS
   DO I=1, N, BS
      !$OMP TASK OUT(VEC1(I:I+BS-1), VEC2(I:I+BS-1), RESULTS(I:I+BS-1))&
      !$OMP PRIVATE(J) FIRSTPRIVATE(I, BS) LABEL(INIT_TASK)
      DO J = I, I+BS-1
         VEC1(J) = J
         VEC2(J) = N + 1 - J
         RESULTS(J) = -1
      END DO
      !$OMP END TASK
   ENDDO
   ! RESULTS = VEC1 + VEC2
```

```
   DO I=1, N, BS
      !$OMP TASK OUT(VEC1(I:I+BS-1), VEC2(I:I+BS-1)) IN(RESULTS(I:I+BS-1))&
      !$OMP PRIVATE(J) FIRSTPRIVATE(I, BS) LABEL(ARRAY_SUM_TASK)
      DO J = I, I+BS-1
         RESULTS(J) = VEC1(J) + VEC2(J)
      END DO
      !$OMP END TASK
   ENDDO
ENDDO ! K
!$OMP TASKWAIT
```

Goals of this exercise

- Code is completely annotated, you DON'T need to modify it.
- Review source code, check different directives and their clauses. Try to understand what they mean.
- Check different compiled versions (performance, instrumented & debug).
- Check other runtime options (schedulers, throttle,...).
- Check (scalability), execute using different number of threads and compute speed-up.
- Get a task dependency graph to analyse dependences.
- Get different paraver traces and analyse them: use paraver configure files (.pcf) in order to visualize thread state, task name,...

3.4 Beginners Exercises

3.4.1 Matrix Multiplication

This example performs the multiplication of two matrices (A and B) into a third one (C). Since the code is not optimized, not very good performance results are expected. Think about how to parallelize (using OmpSs) the following code found in compute() function:

```
for (i = 0; i < DIM; i++)
  for (j = 0; j < DIM; j++)
    for (k = 0; k < DIM; k++)
      matmul ((double *)A[i][k], (double *)B[k][j], (double *)C[i][j], NB);
```

This time you are on your own: you have to identify what code must be a task. There are a few hints and that you may consider before do the exercise:

- Have a look at the compute function. It is the one that the main procedure calls to perform the multiplication. As you can see, this algorithm operates on blocks (to increase memory locality and to parallelize operations on those blocks).
- Now go to the matmul function. As you can see, this function performs the multiplication on a block level.
- When creating tasks do not forget to ensure that all of them have finished before returning the result of the matrix multiplication (would it be necessary any synchronization directive to guarantee that result has been already computed?).

Goals of this exercise

- Look for candidates to become a task and taskify them
- Include synchroniztion directives when required

- Check scalability (for different versions), use different runtime options (schedulers,...)
- Get a task dependency graph and/or paraver traces

3.4.2 Dot Product

The dot product is an algebraic operation that takes two equal-length sequences of numbers and returns a single number obtained by multiplying corresponding entries and then summing those products. A common implementation of this operation is shown below:

```
double dot_product(int N, int v1[N], int v2[N]) {
  double result = 0.0;
  for (long i=0; i<N; i++)
    result += v1[i] * v2[i];

  return result;
}
```

The example above is interesting from a programming model point of view because it accumulates the result of each iteration on a single variable called `result`. As we have already seen in this course, this kind of operation is called reduction, and it is a very common pattern in scientific and mathematical applications.

There are several ways to parallelize operations that compute a reduction:

- Protect the reduction with a lock or atomic clause, so that only one thread increments the variable at the same time. Note that locks are expensive.
- Specify that there is a dependency on the reduction variable, but choose carefully, you don't want to serialize the whole execution! In this exercise we are incrementing a variable, and the sum operation is commutative. OmpSs has a type of dependency called ''commutative'', designed specifically for this purpose.
- Use a vector to store intermediate accumulations. Tasks operate on a given position of the vector (the parallelism will be determined by the vector length), and when all the tasks are completed, the contents of the vector are summed.

Once we have introduced the dot product operation and the different ways of parallelizing a reduction, let's start this exercise. If you open the *dot-product.c* file, you will see that the `dot_product` function is a bit more complicated than the previous version.

```
double result = 0.0;
long j = 0;
for (long i=0; i<N; i+=CHUNK_SIZE) {
    actual_size = (N - i >= CHUNK_SIZE) ? CHUNK_SIZE : N - CHUNK_SIZE;
    C[j] = 0;

    #pragma omp task label( dot_prod ) firstprivate( j, i, actual_size )
    {
        for (long ii=0; ii<actual_size; ii++)
            C[j] +=  A[i+ii] * B[i+ii];
    }

    #pragma omp task label( increment ) firstprivate( j )
    result += C[j];

    j++;
}
```

Basically we have prepared our code to parallelize it, creating a private storage for each chunk and splitting the main loop into two different nested loops to adjust the granularity of our tasks (see `CHUNK_SIZE` variable). Apart from that, we have also annotated the tasks for you, but this parallel version is not ready, yet.

Goals of this exercise

- Find all the `#pragma omp` lines. As you can see, there are tasks, but we forgot to specify their dependencies.
- Tasks are executed asynchronously. Thus, at some point we have to wait for them. Where should we do that?
- There is a task with a label `dot_prod`. What are the inputs of that task? Does it have an output? What is the size of the inputs and outputs? Annotate the input and output dependencies.
- Below the `dot_prod` task, there is another task labeled as `increment`. What does it do? Do you see a difference from the previous? You have to write the dependencies of this task again, but this time think if there is any other clause (besides in and out) that you can use in order to maximize parallelism.
- Think in other parallelization approaches using other types of dependencies.
- Check scalability (for different versions), use different runtime options (schedulers,. . .)
- Get a task dependency graph and/or paraver trackes (analysis)

3.4.3 Multisort application

Multisort application, sorts an array using a divide and conquer strategy. The vector is split into 4 chunks, and each chunk is recursively sorted (as it is recursive, it may be even split into other 4 smaller chunks), and then the result is merged. When the vector size is smaller than a configurable threshold (MIN_SORT_SIZE) the algorithm switches to a sequential sort version:

```
if (n >= MIN_SORT_SIZE*4L) {

  // Recursive decomposition
  multisort(n/4L, &data[0], &tmp[0]);
  multisort(n/4L, &data[n/4L], &tmp[n/4L]);
  multisort(n/4L, &data[n/2L], &tmp[n/2L]);
  multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);

  // Recursive merge: quarters to halves
  merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
  merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);

  // Recursive merge: halves to whole
  merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);

} else {
  // Base case: using simpler algorithm
  basicsort(n, data);
}
```

As the code is already annotated with some task directives, try to compile and run the program. Is it verifying? Why do you think it is failing? Running an unmodified version of this code may also raise a `Segmentation Fault`. Investigate which is the cause of that problem. Although it is not needed, you can also try to debug program execution using gdb debugger (with the OmpSs debug version):

```
$NX_SMP_WORKERS=4 gdb --args ./multisort-d 4096 64 128
```

Goals of this exercise

- Solve the existant bug, program is not properly annotated.

- Think how the tasks must be synchronized and annotate the source file.
- Check different parallelization approaches: taskwait/dependences.
- Check scalability (for the different versions), use other runtime options (schedulers,...)
- Get a task dependency graph (different domains) and/or paraver traces

3.5 GPU Device Exercises

3.5.1 Introduction

Almost all the programs in this section is available both in OpenCL and CUDA. From the point of view of an OmpSs programmer, the only difference between them is the language in which the kernel is written.

As OmpSs abstracts the user from doing the work in the host part of the code. Both OpenCL and CUDA have the same syntax. You can do any of the two versions, as they are basically the same, when you got one of them working, same steps can be done in the other version.

3.5.2 Saxpy kernel

In this exercise we will work with the Saxpy kernel. This algorithm sums one vector with another vector multiplied by a constant.

The sources are not complete, but the standard structure for OmpSs CUDA/Kernel is complete:

- There is a kernel in the files (kernel.cl/kernel.cu) in which the kernel code (or codes) is defined.
- There is a C-file in which the host-program code is defined.
- There is a kernel header file which declares the kernel as a task, this header must be included in the C-file and can also be included in the kernel file (not strictly needed).

Kernel header file (kernel.h) have:

```
#pragma omp target device(cuda) copy_deps ndrange( /*???*/ )
#pragma omp task in([n]x) inout([n]y)
__global__ void saxpy(int n, float a,float* x, float* y);
```

As you can see, we have two vectors (x and y) of size n and a constant a. They specify which data needs to be copied to our runtime. In order to get this program working, we only need to specify the `ndrange` clause, which has three members:

- First one is the number of dimensions on the kernel (1 in this case).
- The second one is the total number of kernel threads to be launched (as one kernel thread usually calculates a single index of data, this is usually the number of elements, of the vectors in this case).
- The third one is the group size (number of threads per block), in this kind of kernels which do not use shared memory between groups, any number from 16 to 128 will work correctly (optimal number depends on hardware, kernel code...).

When the `ndrange` clause is correct. We can proceed to compile the source code, using the command 'make'. After it (if there are no compilation/link errors), we can execute it using one of the running scripts.

Goals of this exercise

- Complete the OmpSs annotation (NDRange clause)
- Check execution and behaviour for different thread hierarchy configurations

- Check different runtime options (devices, max mem, prefetch, overlap,...)

3.5.3 Krist kernel

Krist kernel is used on crystallography to find the exact shape of a molecule using Rntgen diffraction on single crystals or powders. We'll execute the same kernel many times.

The sources are not complete, but the standard structure for OmpSs CUDA/Kernel is complete:

- There is a kernel in the files (kernel.cl/kernel.cu) in which the kernel code (or codes) is defined.
- There is a C-file in which the host-program code is defined.
- There is a kernel header file (krist.h) which declares the kernel as a task, this header must be included in the C-file and can also be included in the kernel file (not strictly needed).

Krist header file (krist.h) have:

```
#pragma omp target device(cuda) copy_deps //ndrange?
#pragma omp task //in and outs?
__global__ void cstructfac(int na, int number_of_elements, int nc, float f2, int NA,
                           TYPE_A* a, int NH, TYPE_H* h, int NE, TYPE_E* output_
↪array);
```

As you can see, now we need to specify the `ndrange` clause (same procedure than previous exercise) and the inputs and outputs. As we have done before for SMP (hint: Look at the source code of the kernel in order to know which arrays are read and which ones are written). The total number of elements which we'll process (not easy to guess by reading the kernel) is 'number_of_elements'.

Remind: ND-range clause has three members:

- First one is the number of dimensions on the kernel (1 in this case).
- The second one is the total number of kernel threads to be launched (as one kernel threads usually calculates a single index of data, this is usually the number of elements, of the vectors in this case).
- The third one is the group size (number of threads per block), in this kind of kernels which do not use shared memory between groups, any number from 16 to 128 will work correctly (optimal number depends on hardware, kernel code...)

Once the `ndrange` clause is correct and the input/outputs are correctly defined. We can proceed to compile the source code, using the command 'make'. After it (if there are no errors), we can execute it using one of the provided running scripts. Check if all environment variables are set to the proper values.

Goals of this exercise

- Complete the target annotation (copy info, NDRange clause,...)
- Complete the task annotation (dependences,...)
- Check execution and behaviour for different thread hierarchy configurations
- Check different runtime options (devices, max mem, prefetch, overlap,...)

3.5.4 Matrix Multiply

In this exercise we will work with the Matmul kernel. This algorithm is used to multiply two 2D-matrices and store the result in a third one. Every matrix has the same size.

This is a blocked-matmul multiplications, this means that we launch many kernels and each one of this kernels will multiply a part of the matrix. This way we can increase parallelism, by having many kernels which may use as many GPUs as possible.

Sources are not complete, but the standard file structure for OmpSs CUDA/Kernel is complete:

- There is a kernel in the files (kernel.cl/kernel.cu) in which the kernel code (or codes) is defined.
- There is a C-file in which the host-program code is defined.
- There is a kernel header file which declares the kernel as a task, this header must be included in the C-file and can also be included in the kernel file (not strictly needed).

Matrix multiply header file (kernel.h) have:

```
//Kernel declaration as a task should be here
//Remember, we want to multiply two matrices, (A*B=C) where all of them have size␣
↪NB*NB
```

In this header, there is no kernel declared as a task, you have to search into the kernel.cu/cl file in order to see which kernel you need to declare, declare the kernel as a task, by placing its declaration and the pragmas over it.

Note: In this case as we are multiplying a two-dimension matrix, so the best approach is to use a two-dimension `ndrange`.

In order to get this program working, we need to specify the `ndrange` clause, which has five members:

- First one is the number of dimensions on the kernel (2 in this case).
- The second and third ones are the total number of kernel threads to be launched (as one kernel threads usually calculates a single index of data, this is usually the number of elements, of the vectors in this case) per dimension.
- The fourth and fifth ones are the group size (number of threads per block), in this kind of kernels which do not use shared memory between groups, any number from 16 to 32 (per dimension) should work correctly (depending on the underlying Hardware).

Once the `ndrange` clause is correct and the input/outputs are properly defined. We can proceed to compile the source code, using the command 'make'. After it (if there are no errors), we can execute it using one of the running scripts.

Goals of this exercise

- Write the target directive (device, copies, thread hierarchy,...)
- Write the task directive (dependences,...)
- Check program execution verification
- Try different complile- (ndrange,...) and run- time options (devices, prefetch,...)

3.5.5 NBody kernel

In this exercise we will work with the NBody kernel. This algorithm numerically approximates the evolution of a system of bodies in which each body continuously interacts with every other body. In this case we want to port a traditional SMP source code to another one which can exploit the benefits of CUDA/OpenCL. Someone already ported the kernel, so it does the same calculations than the previous SMP function.

Sources are not complete, we only have the C code which is calling a SMP function and a CUDA/OCL kernel, they do not interact with each other, nbody.c file have:

```
// Call the kernel
calculate_force_func(bs,time_interval,number_of_particles,this_particle_array, &
↪output_array[i], i, i+bs-1);
```

In this case there is nothing but a kernel ported by someone and a code calling a smp function. We'll need to declare the kernel as an OmpSs task as we have done in previous examples.

Note: Use an intermediate header file and include it, it will work if we declare it on the .c file.

Once the kernel is correctly declared as a task, we can call it instead of the 'old' smp function. We can proceed to compile the source code, using the command 'make'. After it (if there are no errors), we can execute it using one of the running scripts. In order to check results, you can use the command 'diff nbody_out-ref.xyz nbody_out.xyz'.

Note: If someone is interested, you can try to do a NBody implementation which works with multiple GPUs can be done if you have finished early, you must split the output in different parts so each GPU will calculate one of this parts.

If we check the whole source code in nbody.c (not needed), you can see that the 'Particle_array_calculate_forces_cuda' function in kernel.c is called 10 times, and in each call, the input and output array are swapped, so they act like their counter-part in the next call. So when we split the output, we must also split the input in as many pieces as the previous output.

3.5.6 Cholesky kernel

This kernel is just like the SMP version found in the examples, but implemented in CUDA. It uses CUBLAS kernels for the `syrk`, `trsm` and `gemm` kernels, and a CUDA implementation for the potrf kernel (declared in a different file).

Your assignment is to annotate all CUDA tasks in the source code under the section "TASKS FOR CHOLESKY".

3.6 MPI+OmpSs Exercises

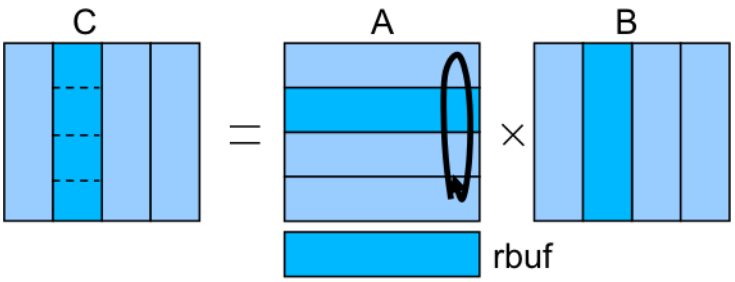
3.6.1 Matrix multiply

This codes performs a matrix - matrix multiplication by means of a hybrid MPI/OmpSs implementation.

Note: You can dowload this code visiting the url http://pm.bsc.es *OmpSs Examples and Exercises*'s (code) link. This version of matrix multiply kernel is included inside the *04-mpi+ompss*'s directory.

Groups of rows of the matrix A are distributed to the different MPI processes. Similarly for the matrix B, groups of columns are distributed to the different MPI process. In each iteration, each process performs the multiplication of the A rows by the B columns in order to compute a block of C. After the computation, each process exchanges with its neighbours the set of rows of A (sending the current ones to the process i+1 and receiving the ones for the next iteration from the process i-1).

An additional buffer rbuf is used to exchange the rows of matrix A between the different iterations.



In this implementation of the code, two tasks are defined: one for the computation of the block of C and another for the communication. See the sample code snippet:

```
for( it = 0; it < nodes; it++ ) {

  #pragma omp task // add in, out and inout
  cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, n, n, m, 1.0, (double *)a, m,
↪ (double *)B, n, 1.0, (double *)&C[i][0], n);

  if (it < nodes-1) {
     #pragma omp task // add in, out and inout
     MPI_Sendrecv( a, size, MPI_DOUBLE, down, tag, rbuf, size, MPI_DOUBLE, up, tag,␣
↪MPI_COMM_WORLD, &stats );
  }

  i = (i+n)%m;                  //next C block circular
  ptmp=a; a=rbuf; rbuf=ptmp;    //swap pointers
}
```

The exercise is provided without the in, out and inout dependence clauses.

- Complete the pragmas in order to define the correct dependences
- Compile and run the example and check that the results are correct (the output of the computation already checks the correctness)
- Generate the tracefile and task graph of the execution

3.6.2 Heat diffusion (Jacobi solver)

This codes performs a . . . hybrid MPI/OmpSs implementation.

Note: You can dowload this code visiting the url http://pm.bsc.es *OmpSs Examples and Exercises*'s (code) link. This version of matrix multiply kernel is included inside the *04-mpi+ompss*'s directory.

Note: You need to specify the number of MPI tasks per node. In Marenostrum you can do this by adding <<#BSUB -R "span[ptile=1]">> to your job script.

3.7 OmpSs+DLB Exercises

3.7.1 PILS (Parallel ImbaLance Simulator)

PILS is an MPI+OpenMP/OmpSs synthetic benchmark that measures the execution time of imbalanced MPI ranks.

Usage:

```
./mpi_ompss_pils <loads-file> <parallel-grain> <loops> <task_size>
    loads-file:     file with load balance (number of tasks per iteration) per␣
↪process, [100, 250] if /dev/null
    parallel-grain: parallelism grain, factor between 0..1 to apply sub-blocking␣
↪techniques
    loops:          number of execution loops
    task_size:      factor to increase task size
```

Goals of this exercise

- Run the instrumented version of PILS and generate a Paraver trace.
  - Analyse the load imbalance between MPI ranks.
- Enable DLB and compare both executions.
  - Observe the dynamic thread creation when other processes suffer load imbalance.
  - Analyse the load imbalance of the new execution. Does it improve?
- Enable DLB MPI interception and trace again. Analyse the new trace.
- Run the multirun.sh script and compare the execution performance with and without DLB.
- Modify the inputs of PILS to reduce load imbalance and see when DLB stops improving performance.

3.7.2 Lulesh

Lulesh is a benchmark from LLNL, it represents a typical hydrocode like ALE3D.

Usage:

```
./lulesh2.0 -i <iterations> -b <balance> -s <size>
```

Goals of this exercise

- Run the instrumented version of Lulesh and analyse the Paraver trace.
- Enable DLB options, MPI interception included. Run and analyse the Paraver trace.
- Run the multirun.sh script and compare the execution performance with and without DLB.

3.7.3 LUB

LUB is an LU matrix decomposition by blocks

Usage:

```
./LUB <size-matrix> <size-block>
```

Goals of this exercise

- Run the instrumented version of LUB and analyse the Paraver trace.
- Enable DLB options. Run and analyse the Paraver trace.
- Run the multirun.sh script and compare the execution performance with and without DLB.

3.7.4 PILS - multiapp example

This example demonstrates the capabilities of DLB sharing resources with two different unrelated applications. The run-once.sh script executes two instances of PILS without MPI support, each one in a different set of CPUs. DLB is able to automatically lend resources from one to another.

Goals of this exercise

- Run the script run-once.sh with tracing and DLB enabled, and observe how two unrelated applications share resources.

BIBLIOGRAPHY

[OPENMP30] OpenMP Application Program Interface, version 3.0 May 2008

INDEX

A

affinity
  scheduling, 49
affinity smart priority
  scheduling, 49
algorithms
  barrier, 61
ancestor tasks, **5**
architectures
  CUDA, runtime, 36
  Offload, runtime, 38
  runtime, 36
Ayudame
  instrumentation plugins, 56

B

barrier
  algorithms, 61
  centralized, 62
  plugins, 61
  tree, 62
base language, **5**
breadth first
  scheduling, 47
build requirements
  Nanos++, 26

C

centralized
  barrier, 62
child task, **5**
cluster
  options, 36
  Point-to-point events, 59
common parameters
  scheduling plugins, 47
compilation
  problems, 34
compile
  OmpSs, 28
  OmpSs CUDA, 30
  OmpSs OpenCL, 32
  OmpSs shared-memory, 29
configure flags
  Nanos++, 27
construct, **5**
CUDA
  compile OmpSs, 30
  options, 35
  runtime architectures, 36

D

data environment, **5**
declarative directive, **5**
default
  scheduler, 47
dependence, **5**
dependences
  plugins, 62
  plugins examples, 63
  plugins perfect-regions, 63
  plugins plain, 62
  plugins regions, 63
  regions troubleshooting, 76
Dependency Graph
  instrumentation plugins, 57
descendant tasks, **5**
device, **5**
directive, **5**
distributed breadth first
  scheduling, 48
dummy
  throttle, 53
  throttling, 53
dynamic extent, **5**

E

empty trace
  instrumentation plugins, 54
Events
  Nanos State Events, 58
examples
  dependences plugins, 63
  throttle, 53

throttling, 53
executable directive, **5**
execute
hybrid
MPI, 70
expression, **6**
Extrae
installation, 25
instrumentation plugins, 54

F

FAQ, 66
application crash, 75
bgq, 77
burst events, 67
macros, 78
NUMA
scheduling, 71
performance, 77
regions, 76
tracking dependences, 80
function task, **6**

G

general
options, 35

H

hardware counters, 55
histeresys
throttle, 51
throttling, 51
host device, **6**
hybrid
MPI OmpSs, 70

I

idle threads
throttle, 52
throttling, 52
immediate successor
scheduling, 46
inline task, **6**
installation
Extrae, 25
Mercurium, 28
Nanos++, 26
Ompss, 25
instrument
hybrid
MPI, 70
instrumentation
plugins, 53
plugins Ayudame, 56
plugins Dependency Graph, 57
plugins empty trace, 54
plugins Extrae, 54
plugins Task Sim, 56
tdg, 57

L

lexical scope, **6**

M

Mercurium
common flags, 28
help, 28
installation, 28
vendor-specific flags, 29
MPI
OmpSs hybrid, 70

N

Nanos++
build requirements, 26
configure flags, 27
installation, 26
nanox
tool, 34
NUMA
scheduling, 71
number of tasks
throttle, 52
throttling, 52

O

Offload
runtime architectures, 38
OmpSs
compile, 28
CUDA, compile, 30
first program, 29
hybrid, MPI, 70
OpenCL, compile, 32
running, 34
shared-memory, compile, 29
Ompss
installation, 25
OpenCL
compile OmpSs, 32
options
cluster, 36
CUDA, 35
general, 35
runtime, 35
outline tasks, **6**

P

Paraver
    Point-to-point events (cluster), traces, 59
    Point-to-point events, traces, 59
Paraver, Extrae, traces, 54
parent continuation
    scheduling, 46
parent task, **6**
perfect-regions
    dependences plugins, 63
performance
    troubleshooting, 77
plain
    dependences plugins, 62
plugins
    Ayudame, instrumentation, 56
    barrier, 61
    common parameters, scheduling, 47
    dependences, 62
    Dependency Graph, instrumentation, 57
    empty trace, instrumentation, 54
    examples, dependences, 63
    Extrae, instrumentation, 54
    instrumentation, 53
    perfect-regions, dependences, 63
    plain, dependences, 62
    regions, dependences, 63
    scheduling, 46
    Task Sim, instrumentation, 56
    throttle, 51
    throttling, 51
Point-to-point events
    cluster, 59
    traces Paraver, 59
Point-to-point events (cluster)
    traces Paraver, 59
predecessor task, **6**
priority
    scheduling, 46
problems
    compilation, 34
punctual events, 59

R

ready task pool, **6**
ready tasks
    throttle, 52
    throttling, 52
regions
    dependences plugins, 63
running
    OmpSs, 34
runtime
    architectures, 36
    architectures CUDA, 36
    architectures Offload, 38
    options, 35

S

scheduler
    default, 47
scheduling
    affinity, 49
    affinity smart priority, 49
    breadth first, 47
    distributed breadth first, 48
    immediate successor, 46
    NUMA, 71
    parent continuation, 46
    plugins, 46
    plugins common parameters, 47
    priority, 46
    threadmanager, 64
    throttling, 46
    versioning, 50
    work first, 48
shared-memory
    compile OmpSs, 29
sliceable task, **6**
slicer policy, **6**
structured block, **6**
successor task, **6**

T

target device, **6**
task, **6**
task dependency graph, **6**
task depth
    throttle, 52
    throttling, 52
task generating code, **6**
Task Sim
    instrumentation plugins, 56
tdg
    instrumentation, 57
thread, **6**
threadmanager
    scheduling, 64
throttle
    dummy, 53
    examples, 53
    histeresys, 51
    idle threads, 52
    number of tasks, 52
    plugins, 51
    ready tasks, 52
    task depth, 52
throttling

dummy, 53
examples, 53
histeresys, 51
idle threads, 52
number of tasks, 52
plugins, 51
ready tasks, 52
scheduling, 46
task depth, 52
tool
nanox, 34
traces
Paraver Point-to-point events, 59
Paraver Point-to-point events (cluster), 59
tree
barrier, 62

V

versioning
scheduling, 50

W

work first
scheduling, 48