OmpSs-2 Specification

Release

BSC Programming Models

Nov 29, 2018

CONTENTS

CHAPTER

ONE

INTRODUCTION

1.1 License and liability disclaimer

Note: There is a PDF version of this document at http://pm.bsc.es/ompss-docs/spec/OmpSs-2-Specification.pdf

1.2 A bit of history

OmpSs-2 is the second generation of the OmpSs programming model. The name originally comes from two other programming models: OpenMP and StarSs. The design principles of these two programming models constitutes the fundamental ideas used to conceive the OmpSs philosophy.

OmpSs takes from OpenMP its viewpoint 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

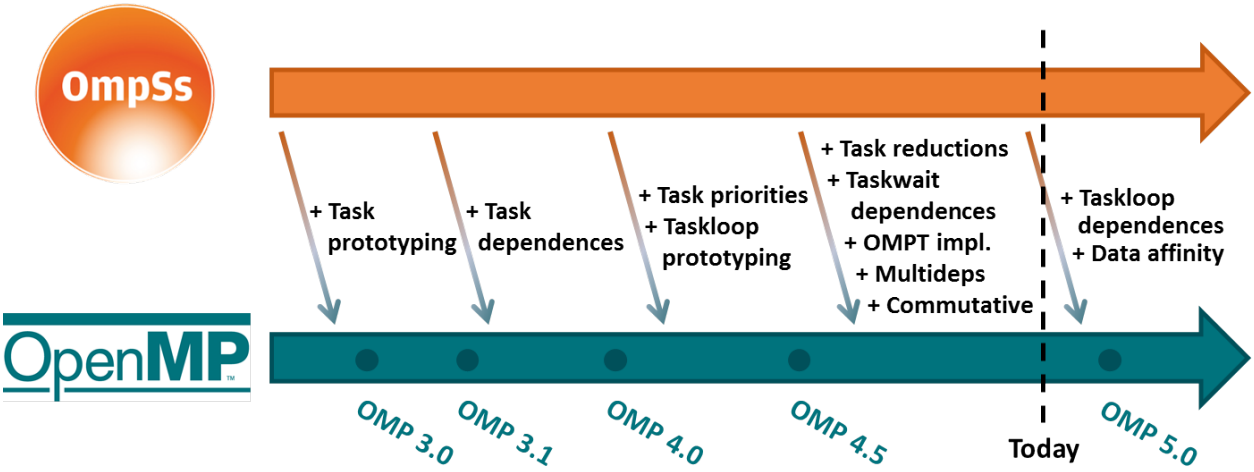
architectures through leveraging native kernels implementation while OpenMP targets accelerator support through direct compiler code generation. Finally StarSs offers asynchronous parallelism as the main mechanism of expressing parallelism whereas OpenMP only started to implement it since its version 3.0. Also StarSs offers task synchronization by means of dependences as the main mechanism of expressing the task execution order, enabling the look-ahead instantiation of task whereas OpenMP included it since its version 4.0.

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.

1.3 Influence in OpenMP

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



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. This feature was tested using Mercurium source-to-source compiler and the Nanos++ RTL.

In OpenMP 4.5 (released on November 2015, and current version), the tasking model was extended with the `taskloop` construct using Nanos++ as the reference implementation to validate these ideas. BSC also contributed to version 4.5 adding the `priority` clause to `task` and `taskloop` constructs.

Upcopming versions of OpenMP will include: `task reductions`, as a way to use tasks as the participants of a reduction operation; `iterators` and `multidependences`, mechanism to specify a variable number of dependences for a task instance; `taskwait dependences`, to relax the set of descendants we need to wait for before proceeding with the taskwait construct; and the `mutexinoutset` (a.k.a. `commutative`), a new type of dependence to support mutually exclusive tasks. The Nanos++ RTL was also used to validate the new interface included in the OpenMP standard with respect third party tools.

1.4 Main features

OmpSs-2 is a programming model composed of a set of directives and library routines that can be used in conjunction with a high level programming language 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-2 extends the tasking model of OmpSs/OpenMP to support both task nesting and fine-grained dependences across different nesting levels, which enables the effective parallelization of applications using a top-down methodology.

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-2 is to provide a productive environment to develop applications for modern High-Performance Computing (HPC) systems. Two concepts add to make OmpSs-2 a productive programming model: performance and ease of use. Programs developed in OmpSs-2 must be able to deliver a reasonable performance when compared to other programming models targeting the same architecture(s). Ease of use is a concept difficult to quantify but OmpSs-2 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/or API services or general features to better support asynchronous data-flow parallelism and heterogeneity. These are, currently, the most prominent features introduced in the OmpSs-2 programming model:

- **Lifetime of task data environment**: A task is completed once the last statement of its body is executed. It becomes deeply completed when also all its children become deeply completed. The data environment of a task, which includes all the variables captured when the task is created, is preserved until the task is deeply completed. Notice that the stack of the thread that is executing the task is NOT part of the task data environment.
- **Nested dependency domain connection**: Incoming dependences of a task propagate to its childrean as if the task did not exist. When a task finishes, its outgoing dependences are replaced by those generated by its children
- **Early release of dependences**: By default, once a task is completed it will release all the dependences that are not included on any unfinished descendant task. If the wait clause is specified in the task construct, all its dependences will be released at once when the task becomes deeply completed.
- **Weak dependences**: The weakin/weakout clauses specify potential dependences only required by descendants tasks. These annotations do not delay the execution of the task.
- **Native offload API**: A new asynchronous API to execute OmpSs-2 kernels on a specified set of CPUs from any kind of application, including Java, Python, R, etc.
- **Task Pause/Resume API**: A new API that can be used to programmatically suspend and resume the execution of a task. It is currently used to improve the interoperability and performance of hybrid MPI and OmpSs-2 applications.

1.5 Reference implementation

The reference implementation of OmpSs-2 is based on the Mercurium source-to-source compiler and the Nanos6 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 *Nanos6* 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.

1.6 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.

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-2 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*.

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-2 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).

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.

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 scheduling points*). Tasks in OmpSs-2 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-2 it can occurs when encountering a `task` construct, a `loop` construct or when calling a routine annotated with a `task` declarative directive.

task scheduling point The different points in wich the runtime may suspend the execution of the current task and execute a different one.

CHAPTER

TWO

PROGRAMMING MODEL

2.1 Execution model

The most notable difference from OmpSs-2 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-2 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.

Warning: 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.

The OmpSs-2 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 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-2 directives. The OmpSs-2 programming model is intended to support programs that will execute correctly both as parallel and as sequential programs (if the OmpSs-2 language is ignored).

OmpSs-2 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-2 run-time environment. The OmpSs-2 run-time will eventually execute the task on a parallel resource. The execution of this 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.

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.

Regular OmpSs-2 applications are linked as whole OmpSs-2 applications, however, the runtime also has a library mode. In library mode, there is no implicit task for the whole program. Instead, the user code defines functions that contain regular OmpSs-2 code (i.e. tasks, ...), and offloads them to the runtime through an API. In this case, the code is not linked as an OmpSs-2 application, but as a regular application that is linked to the runtime.

2.2 Data sharing attributes

OmpSs-2 allows to specify the **explicit data sharing attributes** for the variables referenced in a construct using the following clauses:

- private(<list>)
- firstprivate(<list>)
- shared(<list>)

The `private` and `firstprivate` clauses declare one or more variables to be private to the construct (i.e. a new variable will be created). All internal references to the original variable are replaced by references to this new variable. Variables privatized using the `private` clause are uninitialized when the execution of the construct begins. Variables privatized using the `firstprivate` clause are initialized with the value of the corresponding original variable when the construct was encountered.

The `shared` clause declare one or more variables to be shared to the construct (i.e. the construct still will refer the original variable). Programmers must ensure that shared variables do not reach the end of their lifetime before other constructs referencing them have finished.

There are a few exceptions in which the data sharing clauses can not be used on certain variables due the nature of the symbol. In these cases we talk about **pre-determined data sharing attributes** and they are defined by the following rules:

- Dynamic storage duration objects are shared (malloc, new,...).
- Static data members are shared.
- Variables declared inside the construct with static storage duration are shared.
- Variables declared inside the construct with automatic storage duration are private.
- The loop iteration variable(s) of a loop construct is/are private.

When the variable does not have a *pre-determined behaviour* and it is not referenced by any of the *explicit data sharing clauses* it is considered to have an **implicit data sharing attribute** according with the following rules:

- If the variable appears in a `depend` clause, the variable will be shared.
- If a `default` clause is present in the construct, the implicit data sharing attribute will be the one defined as a parameter of this clause.
- If no `default` clause is present and the variable was private/local in the context encountering the construct, the variable will be firstprivate
- If no `default` clause is present and the variable was shared/global in the context encountering the construct, the variable will be shared.

2.3 Dependence model

Asynchronous parallelism is enabled in OmpSs-2 by the use data-dependencies between the different tasks of the program. OmpSs-2 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-2 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.

OpenMP introduced tasks in version 3.0 and added support for task dependences in version 4.0. In this section we briefly describe the OpenMP tasking model and define some terms that we use throughout the rest of this text to reason about it.

In the general case, the statement that follows the pragma is to be executed asynchronously. Whenever a thread encounters a task, it instantiates it and resumes its execution after the construct. The task instance can be executed either by that same thread at some other time or by another thread. The semantics can be altered through additional clauses and through the properties of the enclosing environment in which the task is instantiated.

Dependencies allow tasks to be scheduled after other tasks using the depend clause to define them. 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 `depend` clause admit the following keywords: `in`, `out`, `inout` or `concurrent`. The keyword is followed by a colon and a comma separated list of elements (memory references). 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.

The semantics of the `depend` clause are also extended with the `in` (standing for input), `out` (standing for output), `inout` (standing for input/output), and `concurrent` clauses to this end. All these clauses also admit a comma separated list of elements (memory references) as described in the previous paragraph.

The meaning of each clause/keyword 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` or `concurrent` 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` or `concurrent` 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` and `inout` 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.

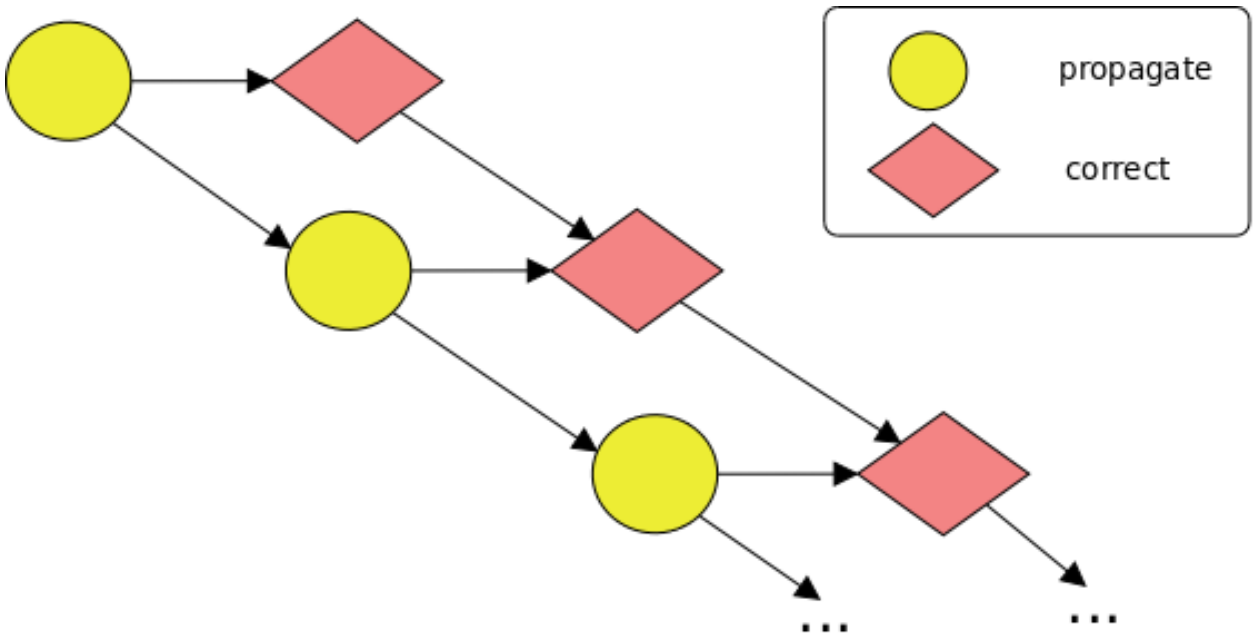
The usual scope of the dependency calculation is restricted to that determined by the enclosing (possibly implicit) task. That is, the contents of the depend clause of two tasks can determine dependencies between them only if they share the same parent task. In this sense, tasks define an inner and independent dependency domain into which to calculate the dependencies between its direct children.

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 oss task in(a[i-1]) inout(a[i]) out(b[i])
      propagate(&a[i-1], &a[i], &b[i]);

#pragma oss 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.

2.3.1 Combining task nesting and dependencies

Being able to combine task nesting and dependences is important for programmability. The following code shows an example that combines nesting and dependencies with two levels of tasks:

```
#pragma oss task depend(inout: a, b) // Task T1
{
  a++; b++;
  #pragma oss task depend(inout: a) // Task T1.1
  a += ...;
  #pragma oss task depend(inout: b) // Task T1.2
  b += ...;
  #pragma oss taskwait
}
#pragma oss task depend(in: a, b) depend(out: z, c, d) // Task T2
{
  z = ...;
  #pragma oss task depend(in: a) depend(out: c) // Task T2.1
  c = ... + a + ...;
  #pragma oss task depend(in: b) depend(out: d) // Task T2.2
  d = ... + b + ...;
  #pragma oss taskwait
}
#pragma oss task depend(in:a, b, d) depend(out:e, f) // Task T3
{
  #pragma oss task depend(in:a, d) depend(out:e) // Task T3.1
  e = ... + a + d + ...;
  #pragma oss task depend(in:b) depend(out:f) // Task T3.2
  f = ... + b + ...;
  #pragma oss taskwait
}
#pragma oss task depend(in: c, d, e, f) // Task T4
{
  #pragma oss task depend(in: c, e) // Task T4.1
  ... = ... + c + e + ...;
  #pragma oss task depend(in: d, f) // Task T4.2
  ... = ... + d + f + ...;
  #pragma oss taskwait
}
```

For brevity and without loss of generality, the inner tasks do not have dependencies between their siblings.

To parallelize the original code using a top-down approach we would perform the following steps. First, we would add the pragmas of the outer tasks. Since they have conflicting accesses over some variables, we add the depend clause. In general it is a good practice to have entries to protect all the accesses of a task. Doing this reduces the burden on the programmer, improves the maintainability of the code and reduces the chances to overlook conflicting accesses.

Then inside each task, we identify separate functionalities, convert each into a task, and add a taskwait at the end of each outer task. If we follow the same approach, the depend clause of each inner task will contain entries to protect its own accesses.

When we consider how the depend clauses are composed, we observe that outer tasks contain a combination of elements to protect their own accesses and elements that are only needed by their subtasks. This is necessary to avoid data-races between subtasks with different parents. In this sense, the latter defer the execution of the outer task until all the dependencies of its subtasks have been fulfilled. In addition, the taskwait at the end of each outer task delays the release of its dependencies until its subtasks have finished.

The inclusion in the depend clause of elements only required by subtasks and the taskwait at the end effectively link the dependency domain of the task with that of its subtasks, which otherwise would be disconnected. However, the elements of the depend clause that the task does not need for itself delay the execution of the task and thus the instantiation of its subtasks. Moreover, the taskwait causes the whole set of dependencies to be released at once. Hence, these two aspects hinder the discovery of parallelism, by delaying and partially hiding it.

2.3.2 Fine-grained release of dependencies across nesting levels

Detaching the taskwait at the end of a task from the task code allows the runtime to be made aware earlier of the fact that the task code has finished and that it will not create further subtasks. In an scenario with task nesting and dependencies, this knowledge allows it to make assumptions about dependencies. Since the task code is finished, the task will no longer perform by itself any action that may require the enforcement of its dependencies. Only the dependencies needed by its live subtasks need to be preserved. In most cases, these dependencies are the ones associated to an element of the depend clause that also appears in a live subtask. Therefore, the dependencies that do not need to be enforced anymore could be released.

In this sense, when a task with the `wait` clause exits from its code, the effects over the dependencies are equivalent to replacing the effects of its depend clause by that of the sequence of its unfinished subtasks. Moreover, this is equivalent to merging its inner dependency domain into that of its parent. To enable tasks to release their dependencies in this way programmers can use the `weakwait` clause. The clause is an alternative to the `wait` clause that indicates that the dependencies can be released incrementally as soon as the inner instantiated tasks finish.

Notice that this improvement is only possible once the runtime is aware of the end of the code of a task. However, it may also be desirable to trigger this mechanism earlier. For instance, a task may use certain data only at the beginning and then perform other lengthy operations that delay the release of the dependencies associated to that data. The `release` directive asserts that a task will no longer perform accesses that conflict with the contents of the associated `depend` clause. The contents of the `depend` clause must be a subset of that of the task construct that is no longer referenced in the rest of the lifetime of the task and its future subtasks.

2.3.3 Weak dependences

Some of the elements of the depend clause may be needed by the task itself, others may be needed only by its subtasks, and others may be needed by both. The ones that are only needed for the subtasks only serve as a mechanism to link the outer domain of dependencies to the inner one. In this sense, allowing them to defer the execution of the task is unnecessary, since the task does not actually perform any conflicting accesses by itself.

The dependence model in OmpSs-2 has been extended to define the `weak` counterparts of `in`, `out` and `inout` dependences. Their semantics are analogous to the ones without the weak prefix. However, the weak variants indicate that the task does not perform by itself any action that requires the enforcement of the dependency. Instead those actions can be performed by any of its deeply nested subtasks. Any subtask that may directly perform those actions needs to include the element in its depend clause in the non-weak variant. In turn, if the subtask delegates the action to a subtask, the element must appear in its depend clause using at least the weak variant.

Weak variants do not imply a direct dependency, and thus do not defer the execution of tasks. Their purpose is to serve as linking point between the dependency domains of each nesting level. Until now, out of all the tasks with the

same parent, the first one with a given element in its depends clause was assumed to not have any input dependency. However, this assumption is no longer true since the dependency domains are no longer isolated. Instead, if the parent has that same element with a weak dependency type, there may actually be a previous and unfinished task with a depend clause that has a dependency to it. If we calculated dependencies as if all types were non-weak, in such a case, the source of the dependency, if any, would be the source of the non-enforced dependency on its parent over the same element.

This change, combined with the fine-grained release of dependencies, merges the inner dependency domain of a task into that of its parent. Since this happens at every nesting level, the result is equivalent to an execution in which all tasks had been created in a single dependency domain.

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

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.

2.3.5 Dependences on the taskwait construct

In addition to the dependencies mechanism, there is a way to set synchronization points in an OmpSs-2 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-2 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.

2.3.6 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 oss 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])
    !$oss task inout([v(i), i=1, n])
        v = v + 1
    !$omp end task
end subroutine foo
```

> **Warning:** Check multidependences syntax

2.4 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 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).

2.5 Conditional compilation

In implementations that support a preprocessor, the `_OMPSS_2` macro name is defined to have a value. In order to keep a non OmpSs-2 programs still compiling programmers should use the symbol definition to guard all OmpSs-2 routine calls or any specific mechanism used by the OmpSs-2 programming model.

2.6 Runtime options

OmpSs-2 uses the runtime library routines to check and configure different Control Variables (CVs) during the program execution. These variables define certain aspects of the program and runtime behaviour. A set of shell environment variables may also be used to configure the CVs before executing the program. A list of library routines can be found in chapter Library Routines.

Some library routines and environment variables are implementation defined and programmers will find them documented in correspondant runtime library user guide (e.g. Nanos6).

CHAPTER

THREE

LANGUAGE DIRECTIVES

3.1 Introduction

This chapter describes the OmpSs-2 language, this is, all the necessary elements to understand how an OmpSs-2 application executes and/or behaves in a parallel architecture. OmpSs-2 provides a simple path for users already familiarized with the OpenMP programming model to easily write (or port) their programs to OmpSs-2.

This description is completely guided by the list of OmpSs-2 directives. In each of the following sections we will find a short description of the directive, its specific syntax, the list of 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-2 program.

As is the case of OpenMP in C and C++, OmpSs-2 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-2 is *oss*. Compilers will typically ignore OmpSs-2 directives if support is disabled or not provided.

C/C++ format:

```
#pragma oss directive-name [clause[ [,] clause] ... ] new-line
```

Fortran format:

```
sentinel directive-name [clause[ [,] clause]...]
```

Where depend on Fortran fixed/free form:

- The sentinels for *fixed form* can be: `!$oss`, `c$oss` or `*$oss`. Sentinels must start in column 1. Continued directive line must have a character other than a space or a zero in column 6.
- The sentinel for *free form* must be `!$oss`. This sentinel can appear in any column as long as is not preceeded by any character different than space. Continued directive line must have an ampersand (&).

3.2 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 oss 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>)
- priority(<expresion>)
- cost(<expresion>)
- if(<scalar-expression>)
- final(<scalar-expresion>)
- label(<string>)
- [wait | weakwait]

The `private`, `firstprivate` and `shared` clauses allow to specify the data sharing of the variables referenced in the construct. A description of these clauses can be found in Data sharing attributes section.

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 model* in the previous chapter. In addition to this syntax, OmpSs-2 allows to specify this information using as the name of the clause the type of dependence. Then, the following code:

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

Is equivalent to this one:

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

The `priority` clause indicates a priority hint for the task. Greater numbers indicate higher priority, and lower numbers indicate less priority. By default, tasks have priority 0. The expression of the priority is evaluated as a singed integer. This way, strictly positive priorities indicate higher priority than the default, and negative priorities indicate lower than default priority.

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.

Tasks with the `wait` clause will perform a taskwait-like operation immediately after exiting from the task code. Since it is performed outside the scope of the code of the task, this happens once the task has abandoned the stack. For this reason, its use is restricted to tasks that upon exiting do not have any subtask accessing its local variables. Otherwise, the regular taskwait shall be used instead.

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 oss task
  do_computation(x);

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

  return 0;
}
```

When the control flow reaches `#pragma oss 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-2 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 oss 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.

3.3 Release directive

The `release` directive asserts that a task will no longer perform accesses that conflict with the contents of the associated `depend` clause. The contents of the `depend` clause must be a subset of that of the task construct that is no longer referenced in the rest of the lifetime of the current task and its future subtasks. The `release` directive has not associated structured block.

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

```
#pragma oss release [clauses]
```

The valid clauses for the `release` directive are:

- depend(<type>: <memory-reference-list>)

- <depend-type>(<memory-reference-list>)

The following C code shows an example of partial release of the task dependences using the `release` directive:

```
#define SIZE 4096

float x[SIZE];
float y[SIZE];

int main() {

  #pragma oss task depend(out:x,y)
  {
     for (int i=0; i<SIZE; i++) x[i] = 0.0;
     #pragma oss release depend(out:x)

     for (int i=0; i<SIZE; i++) y[i] = 0.0;
  }
}
```

3.4 Taskwait construct

Apart from implicit synchronization (task dependences) OmpSs-2 also offers mechanism which allow users to synchronize task execution. The `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 oss 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. The `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 oss task out(result1)
  result1 = compute1();

  #pragma oss task out(result2)
  result2 = compute2();

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

  #pragma oss taskwait on(result2)
```

```
  printf("result2 = %d\n",result2);

  return 0;
}
```

3.5 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 oss critical
structured-block
```

The syntax also allows named criticals with the following syntax:

```
#pragma oss critical(<name>)
structured-block
```

Named criticals prevent concurrency between threads with respect to all critical regions with the same name. Unnammed criticals prevent concurrency between threads with respect to all unnamed critical regions.

The critical construct has no related clauses.

> **Warning:** Should we include syntax and semantics of UDRs in the spec. Should we mark this option as future work.

CHAPTER

FOUR

LIBRARY ROUTINES

4.1 Introduction

This chapter describes the set of OmpSs-2 runtime library routines available while executing an OmpSs-2 program. Programmers must guarantee the correctness of compiling and linking programs withouth OmpSs-2 support by the use of the conditional compilation mechanism (see Conditional compilation section).

For each library routine we provide a short description of the C/C++ and Fortran formats respectively. C/C++ routines are provided through "C" linkage and prototypes are declared in a header file named `oss.h`. Fortran routines are external procedures and interface declarations should be included through the `oss_lib` module.

4.2 nanos6_get_current_blocking_context

The `nanos6_get_current_blocking_context` returns an opaque pointer that is used for blocking and unblocking the current task.

C/C++ format:

```
void *nanos6_get_current_blocking_context();
```

The underlying implementation may or may not return the same value for repeated calls to this function.

Once the handler has been used once in a call to `nanos6_block_current_task` and a call to `nanos6_unblock_task`, the handler is discarded and a new one must be obtained to perform another cycle of blocking and unblocking.

4.3 nanos6_block_current_task

The `nanos6_block_current_task` function blocks the execution of the current task.

C/C++ format:

```
void nanos6_block_current_task(void *blocking_context);
```

The current task will block at least until a thread calls `nanos6_unblock_task` with its blocking context. The runtime may choose to execute other tasks within the execution scope of this call.

4.4 nanos6_unblock_task

The `nanos6_unblock_task` routine mark as unblocked a task previously or about to be blocked.

C/C++ format:

```
void nanos6_block_current_task(void *blocking_context);
```

While this function can be called before the actual to `nanos6_block_current_task`, only one call to it may precede its matching call to `nanos6_block_current_task`.

The return of this function does not guarantee that the blocked task has resumed yet its execution. It only guarantees that it will be resumed.

4.5 nanos6_spawn_function

The `nanos6_spawn_function` allows to asynchrously spawn a new function using the runtime resources.

C/C++ format:

```
void nanos6_spawn_function( void (*function)(void *), void *args,
   void (*completion_callback)(void *), void *completion_args, char const *label
);
```

Where each of this routine's parameters are:

- `function` the function to be spawned.
- `args` a parameter that is passed to the function.
- `completion_callback` an optional function that will be called when the function finishes.
- `completion_args` a parameter that is passed to the completion callback.
- `label` an optional name for the function.

The routine will create a new OmpSs-2 task executing the `function` code and receiving the `args` parameters. Once the task finishes the runtime will invoke the registered callback service (i.e. `completion_callback` using `completion_args` parameters). The callback is used as the provided synchronization mechanism.

The `label` string will be used for debugging/instrumentation purposes.

INDEX