StarPU Handbook - StarPU Language Bindings

for StarPU 1.4.0

Generated by Doxygen.

This manual documents the usage of StarPU version 1.4.0. Its contents was last updated on 2023-03-28.

Chapter 1

Introduction of StarPU Language Bindings

1.1 Organization

This part shows how StarPU which is natively written in C, has been extended to allow applications written in other languages to use it.

- You can learn to natively access most of StarPU functionalities from Fortran 2008+ codes with some explanations and examples in Chapter The StarPU Native Fortran Support.
- You can find out how to execute Java applications with some important StarPU APIs in Chapter StarPU Java Interface.
- Python interface supports most of the main StarPU functionalities, and new functions especially adapted to Python have been added as well. There are detailed explanations and examples in Chapter Python Interface.
- You can learn how to execute OpenMP tasks with some specific functions in Chapter The StarPU OpenMP Runtime Support (SC

Chapter 2

Native Fortran Support

StarPU provides the necessary routines and support to natively access most of its functionalities from Fortran 2008+ codes.
All symbols (functions, constants) are defined in `fstarpu_mod.f90`. Every symbol of the Native Fortran support API is prefixed by `fstarpu_`.
Note: Mixing uses of `fstarpu_` and `starpu_` symbols in the same Fortran code has unspecified behavior. See Valid API Mixes and Language Mixes for a discussion about valid and unspecified combinations.

2.1 Implementation Details and Specificities

2.1.1 Prerequisites

The Native Fortran support relies on Fortran 2008 specific constructs, as well as on the support for interoperability of assumed-shape arrays introduced as part of Fortran's Technical Specification ISO/IEC TS 29113:2012, for which no equivalent are available in previous versions of the standard. It has currently been tested successfully with GNU GFortran 4.9, GFortran 5.x, GFortran 6.x and the Intel Fortran Compiler >= 2016. It is known not to work with GNU GFortran < 4.9, Intel Fortran Compiler < 2016.
See Section Using StarPU with Older Fortran Compilers for information on how to write StarPU Fortran code with older compilers.

2.1.2 Configuration

The Native Fortran API is enabled and its companion `fstarpu_mod.f90` Fortran module source file is installed by default when a Fortran compiler is found, unless the detected Fortran compiler is known not to support the requirements for the Native Fortran API. The support can be disabled through the `configure` option --disable-fortran. Conditional compiled source codes may check for the availability of the Native Fortran Support by testing whether the preprocessor macro `STARPU_HAVE_FC` is defined or not.

2.1.3 Examples

Several examples using the Native Fortran API are provided in StarPU's `examples/native_fortran/` examples directory, to showcase the Fortran flavor of various basic and more advanced StarPU features.

2.1.4 Compiling a Native Fortran Application

The Fortran module `fstarpu_mod.f90` installed in StarPU's `include/` directory provides all the necessary API definitions. It must be compiled with the same compiler (same vendor, same version) as the application itself, and the resulting `fstarpu_mod.o` object file must be linked with the application executable.
Each example provided in StarPU's `examples/native_fortran/` examples directory comes with its own dedicated Makefile for out-of-tree build. Such example Makefiles may be used as starting points for building application codes with StarPU.

2.2 Fortran Translation for Common StarPU API Idioms

All these examples assume that the standard Fortran module `iso_c_binding` is in use.

- Specifying a `NULL` pointer
```
type(c_ptr) :: my_ptr  ! variable to store the pointer
! [...]
my_ptr = c_null_ptr    ! assign standard constant for null ptr
```

- Obtaining a pointer to some object:
```
real(8), dimension(:), allocatable, target :: va
type(c_ptr) :: p_va  ! variable to store a pointer to array va
! [...]
p_va = c_loc(va)
```

- Obtaining a pointer to some subroutine:
```
! pointed routine definition
recursive subroutine myfunc () bind(C)
! [...]
type(c_funptr) :: p_fun  ! variable to store the routine pointer
! [...]
p_fun = c_funloc(my_func)
```

- Obtaining the size of some object:
```
real(8) :: a
integer(c_size_t) :: sz_a  ! variable to store the size of a
! [...]
sz_a = c_sizeof(a)
```

- Obtaining the length of an array dimension:
```
real(8), dimension(:,:), allocatable, target :: vb
intger(c_int) :: ln_vb_1  ! variable to store the length of vb's dimension 1
intger(c_int) :: ln_vb_2  ! variable to store the length of vb's dimension 2
! [...]
ln_vb_1 = 1+ubound(vb,1)-lbound(vb,1)  ! get length of dimension 1 of vb
ln_vb_2 = 1+ubound(vb,2)-lbound(vb,2)  ! get length of dimension 2 of vb
```

- Specifying a string constant:
```
type(c_ptr) :: my_cl  ! a StarPU codelet
! [...]
! set the name of a codelet to string 'my_codele't:
call fstarpu_codelet_set_name(my_cl, c_char_"my_codelet"//c_null_char)
! note: using the C_CHAR_ prefix and the //C_NULL_CHAR concatenation at the end ensures
! that the string constant is properly '\0' terminated, and compatible with StarPU's
! internal C routines
!
! note: plain Fortran string constants are not '\0' terminated, and as such, must not be
! passed to starpu routines.
```

- Combining multiple flag constants with a bitwise 'or':
```
type(c_ptr) :: my_cl  ! a pointer for the codelet structure
! [...]
! add a managed buffer to a codelet, specifying both the Read/Write access mode and the Locality hint
call fstarpu_codelet_add_buffer(my_cl, fstarpu_rw.ior.fstarpu_locality)
```

A basic example is available in `examples/native_fortran/nf_vector_scal.f90`.

2.3 Uses, Initialization and Shutdown

The snippet below show an example of minimal StarPU code using the Native Fortran support. The program should `use` the standard module `iso_c_binding` as well as StarPU's `fstarpu_mod`. The StarPU runtime engine is initialized with a call to function `fstarpu_init`, which returns an integer status of 0 if successful or non-0 otherwise. Eventually, a call to `fstarpu_shutdown` ends the runtime engine and frees all internal StarPU data structures.

```
program nf_initexit
        use iso_c_binding       ! C interfacing module
        use fstarpu_mod         ! StarPU interfacing module
        implicit none           ! Fortran recommended best practice
        integer(c_int) :: err   ! return status for fstarpu_init
        ! initialize StarPU with default settings
        err = fstarpu_init(c_null_ptr)
        if (err /= 0) then
                stop 1          ! StarPU initialization failure
        end if
        ! - add StarPU Native Fortran API calls here
        ! shut StarPU down
        call fstarpu_shutdown()
end program nf_initexit
```

2.4 Fortran Flavor of StarPU's Variadic Insert_task

Fortran does not have a construction similar to C variadic functions, on which starpu_task_insert() relies at the time of this writing. However, Fortran's variable length arrays of `c_ptr` elements enable to emulate much of the convenience of C's variadic functions. This is the approach retained for implementing `fstarpu_task_insert`. The general syntax for using `fstarpu_task_insert` is as follows:

```
call fstarpu_task_insert((/ <codelet ptr>       &
    [, <access mode flags>, <data handle>]*     &
    [, <argument type constant>, <argument>]*   &
    , c_null_ptr /))
```

There is thus a unique array argument `(/ ... /)` passed to `fstarpu_task_insert` which itself contains the task settings. Each element of the array must be of type `type(c_ptr)`. The last element of the array must be `C_NULL_PTR`.

Example extracted from nf_vector.f90:

```
call fstarpu_task_insert((/ cl_vec,          &    ! codelet
    fstarpu_r, dh_va,                        &    ! a first data handle
    fstarpu_rw.ior.fstarpu_locality, dh_vb,  &    ! a second data handle
    c_null_ptr /))                                ! no more args
```

The full example is available in `examples/native_fortran/nf_vector.f90`.

2.5 Functions and Subroutines Expecting Data Structures Arguments

Several StarPU structures that are expected to be passed to the C API, are replaced by function/subroutine wrapper sets to allocate, set fields and free such structure. This strategy has been preferred over defining native Fortran equivalent of such structures using Fortran's derived types, to avoid potential layout mismatch between C and Fortran StarPU data structures. Examples of such data structures wrappers include `fstarpu_conf_allocate` and alike, `fstarpu_codelet_allocate` and alike, `fstarpu_data_filter_allocate` and alike.

Here is an example of allocating, filling and deallocating a codelet structure:

```
! a pointer for the codelet structure
type(c_ptr) :: cl_vec
! [...]
! allocate an empty codelet structure
cl_vec = fstarpu_codelet_allocate()
! add a CPU implementation function to the codelet
call fstarpu_codelet_add_cpu_func(cl_vec, c_funloc(cl_cpu_func_vec))
! add a CUDA implementation function to the codelet
call fstarpu_codelet_add_cuda_func(cl_vec, c_funloc(cl_cuda_func_vec))
! set the codelet name
call fstarpu_codelet_set_name(cl_vec, c_char_"my_vec_codelet"//c_null_char)
! add a Read-only mode data buffer to the codelet
call fstarpu_codelet_add_buffer(cl_vec, fstarpu_r)
! add a Read-Write mode data buffer to the codelet
call fstarpu_codelet_add_buffer(cl_vec, fstarpu_rw.ior.fstarpu_locality)
! [...]
! free codelet structure
call fstarpu_codelet_free(cl_vec)
```

The full example is available in `examples/native_fortran/nf_vector.f90`.

2.6 Additional Notes about the Native Fortran Support

2.6.1 Using StarPU with Older Fortran Compilers

When using older compilers, Fortran applications may still interoperate with StarPU using C marshalling functions as exemplified in StarPU's `examples/fortran/` and `examples/fortran90/` example directories, though the process will be less convenient.

Basically, the main FORTRAN code calls some C wrapper functions to submit tasks to StarPU. Then, when StarPU starts a task, another C wrapper function calls the FORTRAN routine for the task.

Note that this marshalled FORTRAN support remains available even when specifying `configure` option --disable-fortran (which only disables StarPU's native Fortran layer).

2.6.2 Valid API Mixes and Language Mixes

Mixing uses of `fstarpu_` and `starpu_` symbols in the same Fortran code has unspecified behavior. Using `fstarpu_` symbols in C code has unspecified behavior.

For multi-language applications using both C and Fortran source files:

- C source files must use `starpu_` symbols exclusively
- Fortran sources must uniformly use either `fstarpu_` symbols exclusively, or `starpu_` symbols exclusively. Every other combination has unspecified behavior.

Chapter 3

StarPU Java Interface

The StarPU Java Interface provides the ability to execute Java applications on top of StarPU.
The interface allows to write either StarPU-like applications

```
package fr.labri.hpccloud.starpu.examples;
import fr.labri.hpccloud.starpu.Codelet;
import fr.labri.hpccloud.starpu.StarPU;
import fr.labri.hpccloud.starpu.data.DataHandle;
import fr.labri.hpccloud.starpu.data.IntegerVariableHandle;
import fr.labri.hpccloud.starpu.data.VectorHandle;
import java.util.Random;
import static fr.labri.hpccloud.starpu.data.DataHandle.AccessMode.*;
public class VectorScal
{
        public static final int NX = 10;
        public static final Float factor = 3.14f;
        static final Codelet scal = new Codelet()
        {
                @Override
                public void run(DataHandle[] buffers)
                {
                        VectorHandle<Float> array = (VectorHandle<Float>)buffers[0];
                        int n = array.getSize();
                        System.out.println(String.format("scaling array %s with %d elements", array, n));
                        for (int i = 0; i < n; i++)
                        {
                                array.setValueAt(i, factor * array.getValueAt(i));
                        }
                }
                @Override
                public DataHandle.AccessMode[] getAccessModes()
                {
                        return new DataHandle.AccessMode[]
                        {
                                STARPU_RW
                        };
                }
        };
        public static void main(String[] args) throws Exception
        {
                int nx = (args.length == 0) ? NX : Integer.valueOf(args[0]);
                compute(nx);
        }
        public static void compute(int nx) throws Exception
        {
                StarPU.init();
                System.out.println(String.format("VECTOR[#nx=%d]", nx));
                VectorHandle<Float> arrayHandle = VectorHandle.register(nx);
                System.out.println(String.format("scaling array %s", arrayHandle));
                for(int i=0 ; i<nx ; i++)
                {
                        arrayHandle.setValueAt(i, i+1.0f);
                }
                StarPU.submitTask(scal, false, arrayHandle);
                arrayHandle.acquire();
                for(int i=0 ; i<nx ; i++)
                {
                        System.out.println(String.format("v[%d] = %f", i, arrayHandle.getValueAt(i)));
                }
                arrayHandle.release();
                arrayHandle.unregister();
                StarPU.shutdown();
        }
}
```

or Spark applications.

```
package fr.labri.hpccloud.starpu.examples;
```

```
import fr.labri.hpccloud.starpu.StarPU;
import fr.labri.hpccloud.starpu.data.DataPairSet;
import fr.labri.hpccloud.starpu.data.DataSet;
import fr.labri.hpccloud.starpu.data.Tuple2;
import java.io.File;
import java.io.FileInputStream;
import java.io.InputStream;
import java.util.Arrays;
import java.util.regex.Pattern;
public class WordCount
{
        static InputStream openFile(String filename) throws Exception
        {
                return WordCount.class.getResourceAsStream(filename);
        }
        private static final Pattern SPACE = Pattern.compile(" ");
        public static void main(String[] args ) throws Exception
        {
                InputStream input = new FileInputStream(args[0]);
                StarPU.init();
                compute(input);
                input.close();
                StarPU.shutdown();
        }
        private static void compute(InputStream input) throws Exception
        {
                DataSet<String> lines = DataSet.readFile (input, s->s).splitByBlocks(10);
                DataSet<String> words = lines.flatMap(s ->
       Arrays.asList(SPACE.split(s)).iterator()).splitByBlocks(10);
                DataPairSet<String,Integer> ones = (DataPairSet<String,Integer>)words.mapToPair(w-> new
       Tuple2<>(w,1));
                DataPairSet<String,Integer> counts = ones.reduceByKey((c1,c2)-> c1 + c2);
                for(Tuple2<String,Integer> p : counts.collect())
                {
                        System.out.println("("+p._1()+","+p._2()+")");
                }
        }
}
```

The installation process is not yet included in the StarPU autotools mechanism. However, a file `INSTALL.org` is provided in the `starpujni` directory to explain how to proceed with the installation, and shows how to run some basic examples.

`hadoop` needs to be installed before running the installation process.

Chapter 4

Python Interface

This section presents the StarPU Python Interface. It provides for those used to the Python language a more concise and easy-to-use StarPU interface.
This interface supports most of the main StarPU functionalities. All the functionalities of the C API are not provided, however, new functions especially adapted to Python have been added. Several examples using the Python API are provided in the directory `starpupy/examples/`.

4.1 Installation of the Python Interface

The python modules `joblib` and `cloudpickle` are mandatory and should be installed before calling `configure`. The python module `numpy` is recommended, but not mandatory.
If all requirements are met, calling `configure` will enable by default the Python Interface. You can also specify the option --enable-starpupy which will fail if some requirements are missing.

```
$ pip3 install joblib
$ pip3 install cloudpickle
$ pip3 install numpy
$  ../configure --enable-starpupy --enable-blocking-drivers --prefix=$HOME/usr/starpu
$ make
$ make install
```

You can then go to the directory in which StarPU is installed, and test the provided Python examples.

```
$ cd $HOME/usr/starpu
$ . ./bin/starpu_env
Setting StarPU environment for ...
$ cd lib/starpu/python
$ python3 starpu_py.py
Example 1:
Hello, world!
...
$
```

4.2 Python Parallelism

Python interpreters share the Global Interpreter Lock (GIL), which requires that at any time, one and only one thread has the right to execute a task. With the latest version of Python (3.11), if the application is pure Python script, even enable multi-interpreters, the program cannot be executed in parallel. GIL makes the multiple interpreters execution of Python actually serial rather than parallel, and the execution of Python program is single-threaded essentially.
For the pure Python script, the only way to achieve parallelism is to use the master-slave mechanism (Section Master Slave Support). Parallelism may be implemented with multi-interpreters in the future Python version. Details can be found in Section Multiple Interpreters. Otherwise parallelism can be achieved when external C applications are called or external APIs e.g. BLAS API is used for Numpy objects.

4.3 Using StarPU in Python

The StarPU module should be imported in any Python code wanting to use the StarPU Python interface.

```
import starpu
```

Before using any StarPU functionnality, it is necessary to call `starpu.init()`. The function `starpu.shutdown()` should be called after all StarPU functions have been called.

```
import starpu
starpu.init()
# ...
starpu.shutdown()
```

4.3.1 Submitting Tasks

One of the most important functionalities in StarPU is the task submission. The Python interface simplifies its usage, as the function can directly be called without any extra steps.
The Python function to submit tasks is `task_submit(options)(func, *args, **kwargs)`. `func` is any Python function, `args` and `kwargs` are the function arguments. The function can also be provided as a string. In order to let StarPU make optimizations for your program, you should submit all tasks to allow StarPU to efficiently schedule the underlying tasks. Submitted tasks will not be executed immediately, and you can only get the return value once the task has been executed.
In the first set of parentheses, the following options may be specified. Please note that all options have a default value, the parentheses must be kept even without any option.

- `name` (string, default: `None`)

  Set the name of the task. This can be useful for debugging purposes.

- `synchronous` (unsigned, default: 0)

  If this flag is set, the function `task_submit` only returns when the task has been executed (or if no worker is able to process the task). Otherwise, `task_submit` returns immediately.

- `priority` (int, default: 0)

  Set the level of priority for the task. This is an integer value whose value must be greater than the return value of the function `starpu.sched_get_min_priority()` (for the least important tasks), and lower or equal to the return value of the function `starpu.sched_get_max_priority()` (for the most important tasks). Default priority is defined as 0 in order to allow static task initialization. Scheduling strategies that take priorities into account can use this parameter to take better scheduling decisions, but the scheduling policy may also ignore it.

- `color` (unsigned, default: `None`)

  Set the color of the task to be used in `dag.dot`.

- `flops` (double, default: `None`)

  Set the number of floating points operations that the task will have to achieve. This is useful for easily getting GFlops/s curves from the function `starpu.perfmodel_plot`, and for the hypervisor load balancing.

- `perfmodel` (string, default: `None`)

  Set the name of the performance model. This name will be used as the filename where the performance model information will be saved. After the task is executed, one can call the function `starpu.perfmodel_plot()` by giving the symbol of perfmodel to view its performance curve.

4.3.2 Returning Future Object

In order to realize asynchronous frameworks, the `task_submit()` function returns a Future object. This is an extended use for the Python interface. A Future represents an eventual result of an asynchronous operation. It is an awaitable object, Coroutines can await on Future objects until they either have a result or an exception set, or until they are canceled. Some basic examples are available in script `starpupy/examples/starpu_py.py`. This feature needs the `asyncio` module to be imported.

```
import starpu
import asyncio
starpu.init()
def add(a, b):
    return a+b
async def main():
    fut = starpu.task_submit()(add, 1, 2)
    res = await fut
    print("The result of function is", res)
```

```
asyncio.run(main())
starpu.shutdown()
```

Execution:

```
The result of function is 3
```

When using at least the version 3.8 of python, one can also use the parameter `-m asyncio` which allows to directly use `await` instead of `asyncio.run()`.

```
$ python3 -m asyncio
>>> import asyncio

import starpu
starpu.init()
def add(a, b):
        print("The result is ready!")
        return a+b
fut = starpu.task_submit()(add, 1, 2)

The result is ready!

res = await fut
res

3
```

You can also use the decorator `starpu.delayed` to wrap a function. The function can then directly be submitted to StarPU and will automatically create a Future object.

```
@starpu.delayed
def add_deco(a, b):
        print("The result is ready!")
        return a+b
fut = add_deco(1, 2)

The result is ready!

res = await fut
res

3
```

To specify options when using the decorator, just do as follows:

```
@starpu.delayed(name="add", color=2, perfmodel="add_deco")
def add_deco(a, b):
        print("The result is ready!")
        return a+b
fut = add_deco(1, 2)

The result is ready!

res = await fut
res

3
```

The Future object can also be used for the next step calculation even before being ready. The eventual result will be awaited until the Future has a result.
In this example, after submitting the first task, a Future object `fut1` is created, and it is used as an argument of a second task. The second task is submitted even without having the return value of the first task.

```
import asyncio
import starpu
import time
starpu.init()
def add(a, b):
        time.sleep(10)
        print("The first result is ready!")
        return a+b
def sub(x, a):
        print("The second result is ready!")
        return x-a
fut1 = starpu.task_submit()(add, 1, 2)
fut2 = starpu.task_submit()(sub, fut1, 1)

The first result is ready!
The second result is ready!

res = await fut2
res

2
```

4.3.3 Submit Python Objects Supporting The Buffer Protocol

The Python buffer protocol is a framework in which Python objects can expose raw byte arrays to other Python objects. This can be extremely useful to efficiently store and manipulate large arrays of data. The StarPU Python Interface allows users to use such objects as task parameters.

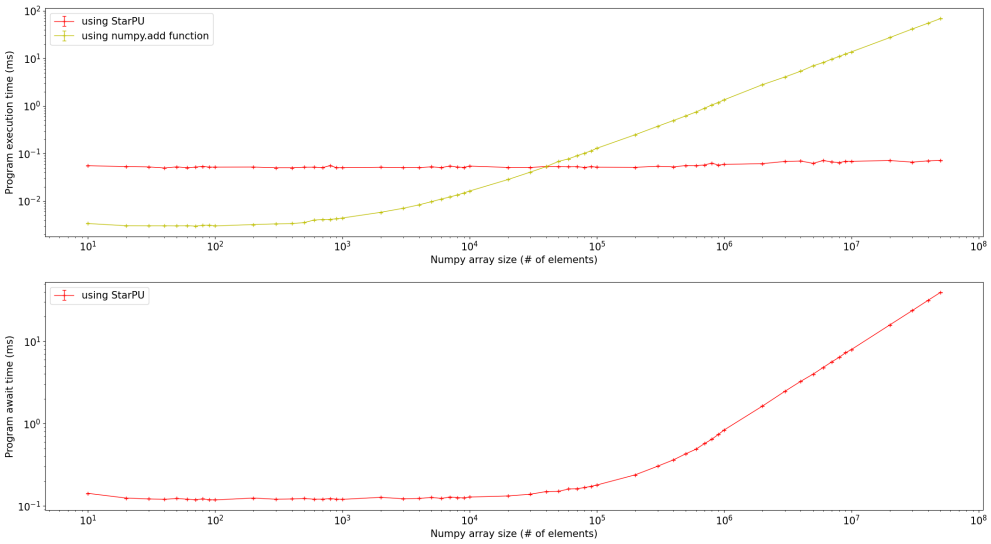
```
import asyncio
import starpu
import time
import numpy as np
starpu.init()
def add(a,b):
        c = np.zeros(np.size(a))
        for i in range(np.size(a)):
                c[i] = a[i] + b[i]
        return c
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
fut = starpu.task_submit()(add, a, b)
res = await fut
res

array([5., 7., 9.])
```

StarPU uses a specific data interface to handle Python objects supporting buffer protocol, such python objects are then managed by the StarPU data management library which allows minimizing data transfers between accelerators, and avoids copying the object each time.
We show the performances below of the `numpy` addition (`numpy.add` running the script `test_perf.sh`) with different array sizes (10, 20, ..., 100, 200, ..., 1000, 2000, ..., 10000, 20000, ..., 100000, 200000, ..., 1000000, 2000000, ..., 10000000, ..., 50000000). We compare two cases:

1. Using StarPU,
2. Without using StarPU tasks, but directly calling the `numpy.add` function.

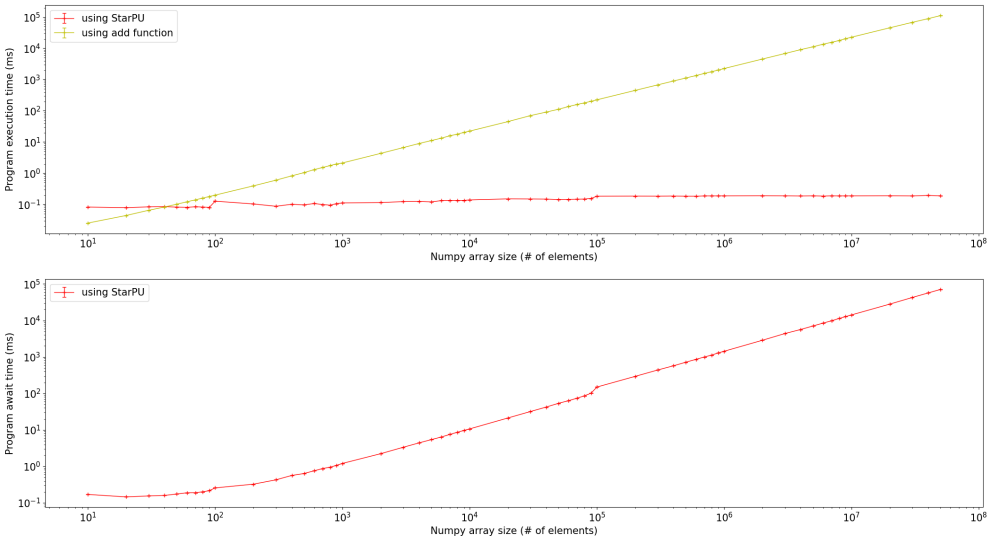
The first plot compares the task submission time when using StarPU and the program execution time without using StarPU. We can see that there is an obvious optimization using StarPU when the test array size is large. The task has not finished its execution yet as shown in second figure, the time can be used to perform other operations.



We can also define our own function to do the `numpy` operation, e.g. the element addition:

```
def add(a, b):
        for i in range(np.size(a)):
                a[i] = a[i] + b[i]
```

We will compare operation performances with the same two cases, but based on our custom function `add(a, b)`. We can see that the custom function is not as efficient as the `numpy` function overall. The optimization for large arrays is the same when using StarPU.



4.3.3.1 Access Mode Annotation

StarPU defines different access modes for a data, it can be readable (access mode is `R`), writable (access mode is `W`), or both readable and writable (access mode is `RW`). The default access mode is `R`.
For the Python interface, these modes can be defined as shown below.

1. Using the decorator `starpu.access(arg="R/W/RW")` to wrap the function.

```
a = np.array([1, 2, 3, 4, 5, 6])
e = np.array([0, 0, 0, 0, 0, 0])
@starpu.access(a="R", b="W")
def assign(a,b):
        for i in range(min(np.size(a), np.size(b))):
                b[i]=a[i]
fut = starpu.task_submit()(assign, a, e)
starpu.acquire(e)


array([1, 2, 3, 4, 5, 6, 0])

starpu.release(e)
```

2. Using the decorator `starpu.delayed(options, arg="R/W/RW")`.

```
@starpu.delayed(a="R", b="W")
def assign(a,b):
        for i in range(min(np.size(a), np.size(b))):
                b[i]=a[i]
fut = assign(a, e)
starpu.acquire(e)


array([1, 2, 3, 4, 5, 6, 0])

starpu.release(e)
```

3. Using the method `starpu.set_access(func, arg="R/W/RW")` that will create a new function.

```
def assign(a,b):
        for i in range(min(np.size(a), np.size(b))):
                b[i]=a[i]
assign_access=starpu.set_access(assign, a="R", b="W")
fut = starpu.task_submit()(assign_access, a, e)
starpu.acquire(e)


array([1, 2, 3, 4, 5, 6, 0])

starpu.release(e)
```

4.3.3.2 Methods

Once the access mode of one argument is set to at least `W`, it may be modified during the task execution. We should pay attention that before the task is finished, we cannot get the up-to-date value of this argument by simply using `print` function. For example:

```
import asyncio
import starpu
import time
import numpy as np
starpu.init()
a = np.array([1, 2, 3, 4, 5, 6])
e = np.array([0, 0, 0, 0, 0, 0, 0])
@starpu.access(a="R", b="W")
def assign(a,b):
        time.sleep(10)
        for i in range(min(np.size(a), np.size(b))):
                b[i]=a[i]
fut = starpu.task_submit()(assign, a, e)
print(e) # before the task is finished

[0 0 0 0 0 0 0]
```

We `print` argument `e` right after submitting the task, but since the task is not finished yet, we can only get its unchanged value. If we want to get its up-to-date value, we need extra functions.
In order to access data registered to StarPU outside tasks, we provide an acquire and release mechanism.

- The `starpu.acquire(data, mode)` method should be called to access registered data outside tasks (Refer to the method `starpu_data_acquire()` in C interface). StarPU will ensure that the application will get an up-to-date copy of handle in main memory located where the data was originally registered, and that all concurrent accesses (e.g. from tasks) will be consistent with the access mode specified with the given mode (`R` the default mode, `W` or `RW`).
- The `starpu.release(data)` method must be called once the application no longer needs to access the piece of data (Refer to the method `starpu_data_release()` in C interface).
- The `starpu.unregister(data)` method must be called to unregister the Python object from StarPU. (Refer to the method `starpu_data_unregister()` in C interface). This method waits for all calculations to be finished before unregistering data.

With `acquire`, even we ask to access the argument right after submitting the task, the up-to-date value will be printed once the task is finished.

```
starpu.acquire(e) # before the task is finished

array([1, 2, 3, 4, 5, 6, 0])
```

In order to complete the addition operation example, execution steps are:

```
import asyncio
import starpu
import time
import numpy as np
starpu.init()
@starpu.access(a="RW", b="R")
def add(a,b):
        time.sleep(10)
        for i in range(np.size(a)):
                a[i] = a[i] + b[i]
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
starpu.acquire(a, mode="R")

array([1, 2, 3])

starpu.release(a)
fut = starpu.task_submit()(add, a, b)
starpu.acquire(b, mode="R")

array([4, 5, 6])

starpu.acquire(a, mode="R") # before the task is finished

array([5, 7, 9])
```

```
starpu.release(a)
starpu.release(b)
starpu.unregister(a)
starpu.unregister(b)
```

The result of `b` is printed directly right after calling `acquire`, but the up-to-date value of `a` is printed after the task is finished. Here we need to pay attention that if we want to modify an argument during the task execution and get its up-to-date value for the future operation, we should set the access mode of this argument to at least `W`, otherwise this argument object is not synchronous, and the next task which needs this object will not wait its up-to-date value to execute.

If we call `acquire` but not `release` before the task submission, the task will not start to execute until the object is released.

An example is shown below:

```
import asyncio
import starpu
import numpy as np
import time
starpu.init()
@starpu.access(a="RW")
def add(a,b):
        print("This is the addition function")
        time.sleep(10)
        for i in range(np.size(a)):
                a[i] = a[i] + b[i]
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
starpu.acquire(a, mode="R")

array([1, 2, 3])

fut = starpu.task_submit()(add, a, b)
starpu.release(a)

This is the addition function   # The task will not start until "a" is released

starpu.acquire(a, mode="R") # Before the task is finished

array([5, 7, 9])                # After the task is finished

starpu.release(a)
starpu.unregister(a)
starpu.unregister(b)
```

4.4 StarPU Data Interface for Python Objects

StarPU uses data handles to manage a piece of data. A data handle keeps track of replicates of the same data (registered by the application) over various memory nodes. The data management library manages to keep them coherent. That also allows minimizing the data transfers, and avoids copying the object each time. Data handles are managed through specific data interfaces. Some examples applying this specific interface are available in script `starpupy/examples/starpu_py_handle.py`.

4.4.1 Interface for Ordinary Python Objects

A specific data interface has been defined to manage Python objects, such as constant (integer, float...), string, list, etc. This interface is defined with the class `Handle`. When submitting a task, instead of specifying a function and its arguments, we specify a function and the handles of its arguments.

In addition to returning a Future object, it is also possible to return a StarPU handle object when submitting a function. To do so, you need to set the `starpu.task_submit` option `ret_handle` to `True`, its default value is `False`.

```
import starpu
from starpu import Handle
starpu.init()
def add(x, y):
   return x + y
x = Handle(2)
y = Handle(3)
res = starpu.task_submit(ret_handle=True)(add, x, y)
```

We then need to call the method `get()` to get the latest version of this Python Object.

```
res.get()

5
```

When not setting the parameter `ret_handle`, the return object is a Future.

```
res_fut = starpu.task_submit()(add, x, y)
await res_fut
```

If the Python object is immutable (such as int, float, str, tuple...), registering the same object several times is authorised. That means you can do this:

```
x = Handle(2)
x1 = Handle(2)
```

x and x1 are two different Handle objects.

4.4.2 Interface for Python Objects Supporting Buffer Protocol

This StarPU data interface can also be used to manage Python objects supporting buffer protocol, i.e `numpy` array, bytes, bytearray, array.array and memoryview object.

```
import numpy as np
import starpu
from starpu import Handle
starpu.init()
def add(a,b):
   for i in range(np.size(a)):
      a[i] = a[i] + b[i]
   return a
a = np.array([1, 2, 3])
b = np.array([2, 4, 6])
a_h = Handle(a)
b_h = Handle(b)
res = starpu.task_submit(ret_handle=True)(add, a_h, b_h)
res.get()

array([3, 6, 9])
```

Different from immutable Python object, all Python objects supporting buffer protocol are mutable, and registering the same object one more time is not authorized. If you do this:

```
a = np.array([1, 2, 3])
a_h = Handle(a)
a1_h = Handle(a)
```

You will get an error message:

```
starpupy.error: Should not register the same mutable python object once more.
```

You may refer to Section Submit Python Objects Supporting The Buffer Protocol, and realize that StarPU Python interface uses data handles to manage Python objects supporting buffer protocol by default. These objects are usually relatively large, such as a big NumPy matrix. We want to avoid multiple copies and transfers of this data over various memory nodes, so we set the default `starpu.task_submit()` option `arg_handle` to `True` for users to allow their applications to get the most optimization. To deactivate the use of this data interface, you need to set the option `arg_handle` to `False`.

Since we use data handles by default, registration is implemented in the step of task submission. Therefore, you should be careful not to register again the same object after the task submission, like this:

```
a = np.array([1, 2, 3])
b = np.array([2, 4, 6])
res = starpu.task_submit(ret_handle=True)(add, a, b)
a_h = Handle(a)
```

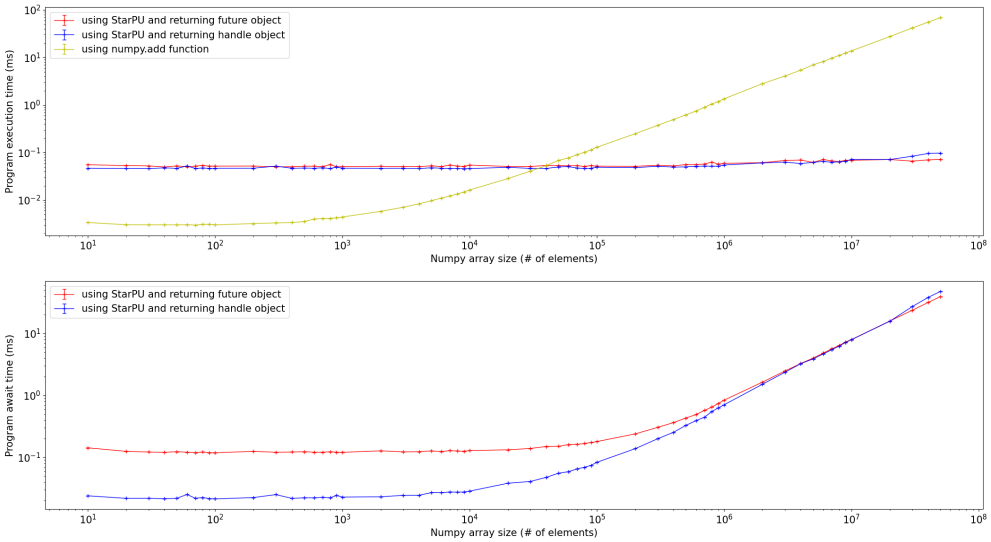
You will get the error message:

```
starpupy.error: Should not register the same mutable python object once more.
```

As performances, we showed in Section Submit Python Objects Supporting The Buffer Protocol, we add one case to compare with the others two cases. We still test the `numpy` addition (`numpy.add` running the script `test_handle_perf.sh`) with different array sizes (10, 20, ..., 100, 200, ..., 1000, 2000, ..., 10000, 20000, ..., 100000, 200000, ..., 1000000, 2000000, ..., 10000000, ..., 50000000). Three cases are:

1. Using StarPU and returning future object,
2. Using StarPU and returning handle object,
3. Without using StarPU tasks, but directly calling the `numpy.add` function.

The first plot compares the task submission time when using StarPU either returning a Future or a handle object and the program execution time without using StarPU. We can see that there is an obvious optimization using StarPU, either returning a Future or a handle object when the test array size is large. The task has not finished its execution yet as shown in second figure, the time can be used to perform other operations. When array size is not very large, returning a handle has a better execution performance than returning a Future.

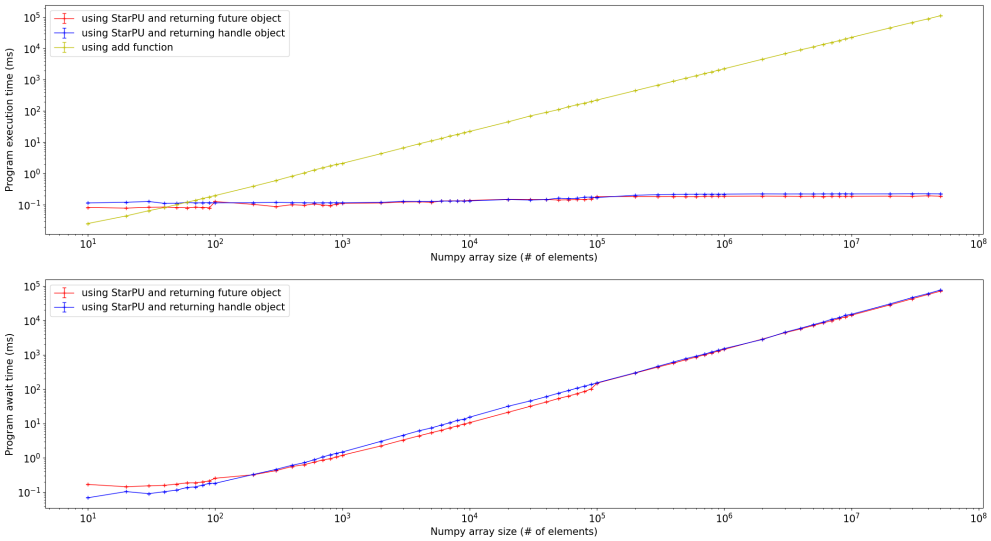


We can also define our own function to do the `numpy` operation, e.g. the element addition:

```
def add(a, b):
        for i in range(np.size(a)):
                a[i] = a[i] + b[i]
```

We will compare operation performances with the same three cases but based on our custom function `add(a, b)`.

We can see that the custom function is not as efficient as the `numpy` function overall. The optimisation for large arrays is the same when using StarPU.



4.4.2.1 Methods

As in Section Methods, the `Handle` class defines methods to provide an acquire and release mechanism.

- The method `Handle::acquire(mode)` should be called before accessing the object outside tasks (Refer to the method `starpu_data_acquire()` in C interface). The access mode can be `"R"`, `"W"`, `"RW"`, the default value is "R". We will get an up-to-date copy of Python object by calling this method.

- The method `Handle::release()` must be called once the application no longer needs to access the registered data (Refer to the method `starpu_data_release()` in C interface).
- The method `Handle::unregister()` to unregister the Python object handle from StarPU (Refer to the method `starpu_data_unregister()` in C interface). This method will wait for all calculations to be finished before unregistering data.

The previous example can be coded as follows:

```
import numpy as np
import starpu
from starpu import Handle
starpu.init()
@starpu.access(a="RW", b="R")
def add(a,b):
   for i in range(np.size(a)):
      a[i] = a[i] + b[i]
a = np.array([1, 2, 3])
b = np.array([2, 4, 6])
a_h = Handle(a)
b_h = Handle(b)
a_h.acquire(mode = "R")
array([1, 2, 3])
a_h.release()
starpu.task_submit(ret_handle=True)(add, a_h, b_h)
a_h.acquire(mode = "R") # we get the up-to-date value

array([3, 6, 9])

a_h.release()
a_h.unregister()
```

4.4.3 Interface for Empty Numpy Array

We can register an empty `numpy` array by calling `HandleNumpy(size, type)`. The default value for `type` is `float64`.
You will find below an example which defines the function `assign` taking two arrays as parameters, the second one being an empty array which will be assigned the values of the first array.

```
import numpy as np
import starpu
from starpu import Handle
from starpu import HandleNumpy
starpu.init()
@starpu.access(b="W")
def assign(a,b):
   for i in range(min(np.size(a,0), np.size(b,0))):
      for j in range(min(np.size(a,1), np.size(b,1))):
         b[i][j] = a[i][j]
   return b
a = np.array([[1, 2, 3], [4, 5, 6]])
a_h = Handle(a)
e_h = HandleNumpy((5,10), a.dtype)
res = starpu.task_submit(ret_handle=True)(assign, a_h, e_h)
e_h.acquire()

array([[1, 2, 3, 0, 0, 0, 0, 0, 0, 0],
       [4, 5, 6, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])

e_h.release()
```

4.4.4 Array Partitioning

A n-dim `numpy` array can be split into several sub-arrays by calling the method `Handle::partition(nchildren, dim, chunks_list)` (Refer to the method `starpu_data_partition_plan()` in C interface).

- `nchildren` is the number of sub-handles,
- `dim` is the dimension that we want to partition along, it can be 0 for vertical dimension, 1 for horizontal dimension, 2 for depth dimension, 3 for time dimension, ...etc.
- `chunks_list` is a list containing the size of each segment. The total length of segments in this list must be equal to the length of the selected dimension.

The method will return a sub-handle list, each of the sub-handles can be used when submitting a task with `task_submit()`. This allows to process an array in parallel, once the execution of each sub-handle is finished, the result will be directly reflected in the original n-dim array.
When the sub-handles are no longer needed, the method `Handle::unpartition(handle_list, nchildren)` should be called to clear the partition and unregister all the sub-handles (Refer to the method `starpu_data_partition_clean()` in C interface).

- `handle_list` is the sub-handle list which was previously returned by the method `Handle::partition()`,
- `nchildren` is the number of sub-handles.

Here is an example to use these methods.

```
import numpy as np
import starpu
from starpu import Handle
starpu.init()
@starpu.access(a="RW", b="R")
def add(a,b):
   np.add(a,b,out=a)
n, m = 20, 10
arr = np.arange(n*m).reshape(n, m)
arr_h = Handle(arr)
arr_h.acquire(mode='RW')
```

```
 [[  0   1   2   3   4   5   6   7   8   9]
 [ 10  11  12  13  14  15  16  17  18  19]
 [ 20  21  22  23  24  25  26  27  28  29]
 [ 30  31  32  33  34  35  36  37  38  39]
 [ 40  41  42  43  44  45  46  47  48  49]
 [ 50  51  52  53  54  55  56  57  58  59]
 [ 60  61  62  63  64  65  66  67  68  69]
 [ 70  71  72  73  74  75  76  77  78  79]
 [ 80  81  82  83  84  85  86  87  88  89]
 [ 90  91  92  93  94  95  96  97  98  99]
 [100 101 102 103 104 105 106 107 108 109]
 [110 111 112 113 114 115 116 117 118 119]
 [120 121 122 123 124 125 126 127 128 129]
 [130 131 132 133 134 135 136 137 138 139]
 [140 141 142 143 144 145 146 147 148 149]
 [150 151 152 153 154 155 156 157 158 159]
 [160 161 162 163 164 165 166 167 168 169]
 [170 171 172 173 174 175 176 177 178 179]
 [180 181 182 183 184 185 186 187 188 189]
 [190 191 192 193 194 195 196 197 198 199]]
```

```
arr_h.release()
split_num = 3
arr_h_list = arr_h.partition(split_num, 1, [3,2,5]) # split into 3 sub-handles, and partition along the
       horizontal dimension
for i in range(split_num):
   res=starpu.task_submit(ret_handle=True)(add, arr_h_list[i], arr_h_list[i])
arr_h.acquire(mode='RW')
```

```
[[  0   2   4  12  16  40  48  56  64  72]
 [ 80  88  96 104 112 120 128 136 144 152]
 [160 168 176 184 192 200 208 216 224 232]
 [240 248 256 264 272 280 288 296 304 312]
 [320 328 336 172 176 180 184 188 192 196]
 [200 204 208 212 216 220 224 228 232 236]
 [120 122 124 126 128 130 132 134 136 138]
 [140 142 144 146 148 150 152 154 156 158]
 [160 162 164 166 168 170 172 174 176 178]
 [180 182 184 186 188 190 192 194 196 198]
 [200 202 204 206 208 105 106 107 108 109]
 [110 111 112 113 114 115 116 117 118 119]
 [120 121 122 123 124 125 126 127 128 129]
 [130 131 132 133 134 135 136 137 138 139]
 [140 141 142 143 144 145 146 147 148 149]
 [150 151 152 153 154 155 156 157 158 159]
 [160 161 162 163 164 165 166 167 168 169]
 [170 171 172 173 174 175 176 177 178 179]
 [180 181 182 183 184 185 186 187 188 189]
 [190 191 192 193 194 195 196 197 198 199]]
```

```
arr_h.release()
arr_h.unpartition(arr_h_list, split_num)
arr_h.unregister()
```

The method `Handle::get_partition_size(handle_list)` can be used to get the array size of each sub-array.

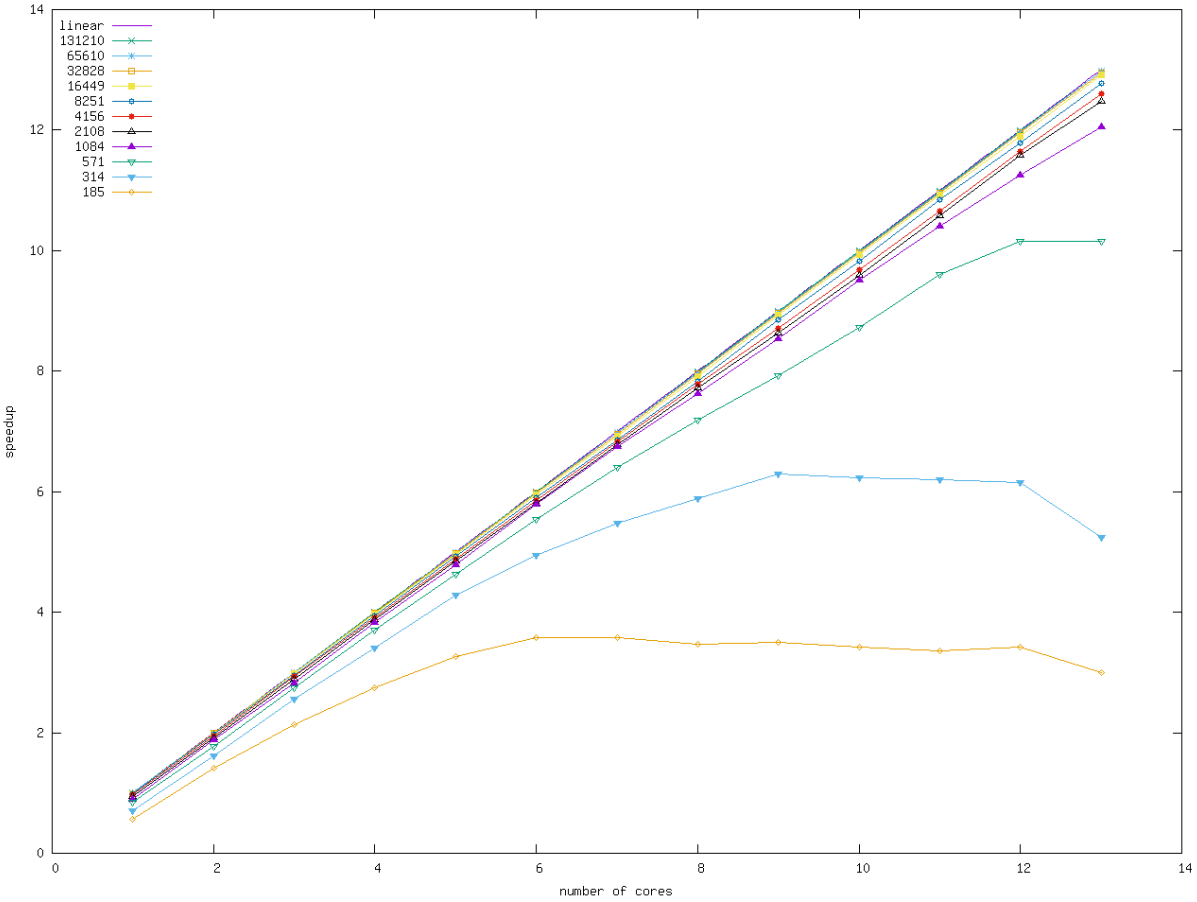
```
arr_h_list = arr_h.partition(split_num, 1, [3,2,5])
arr_h.get_partition_size(arr_h_list)
```

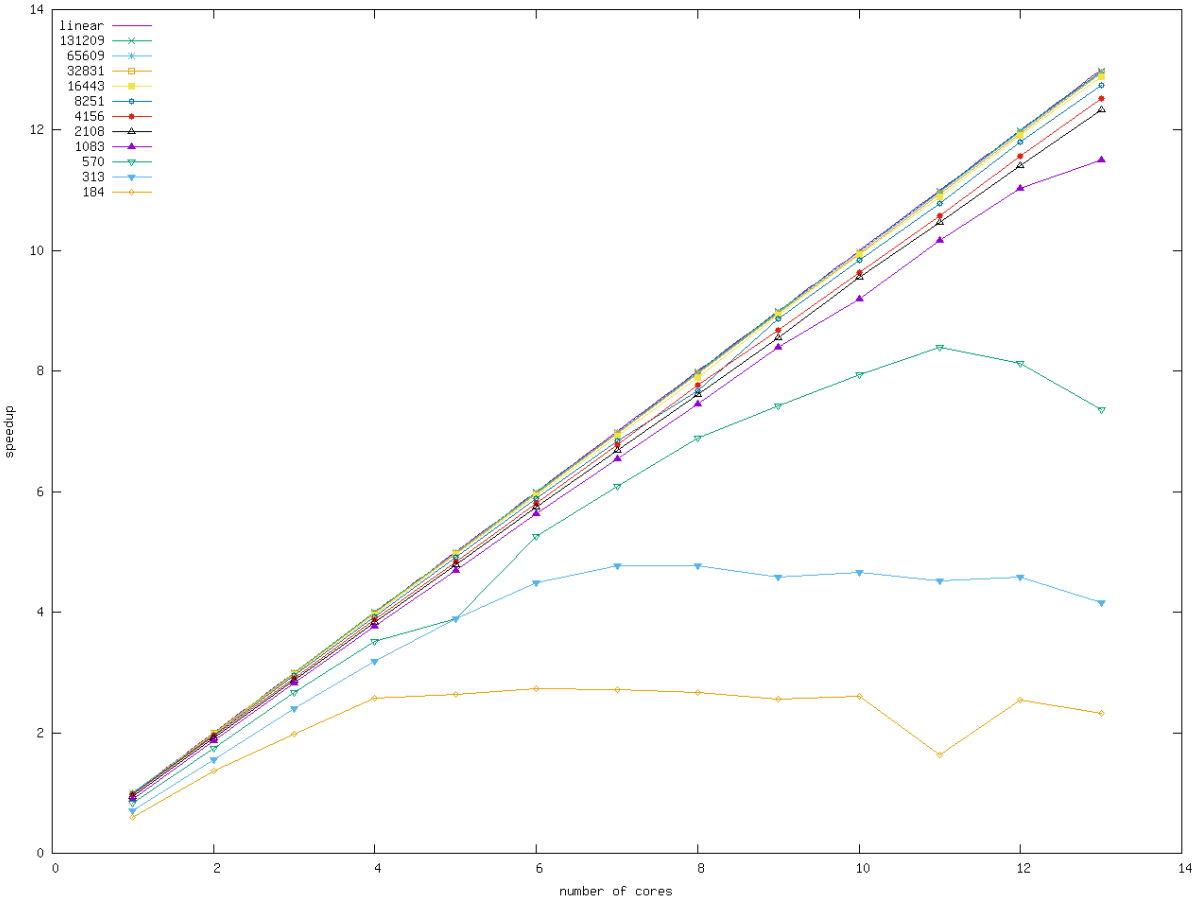
```
[60, 40, 100]
```

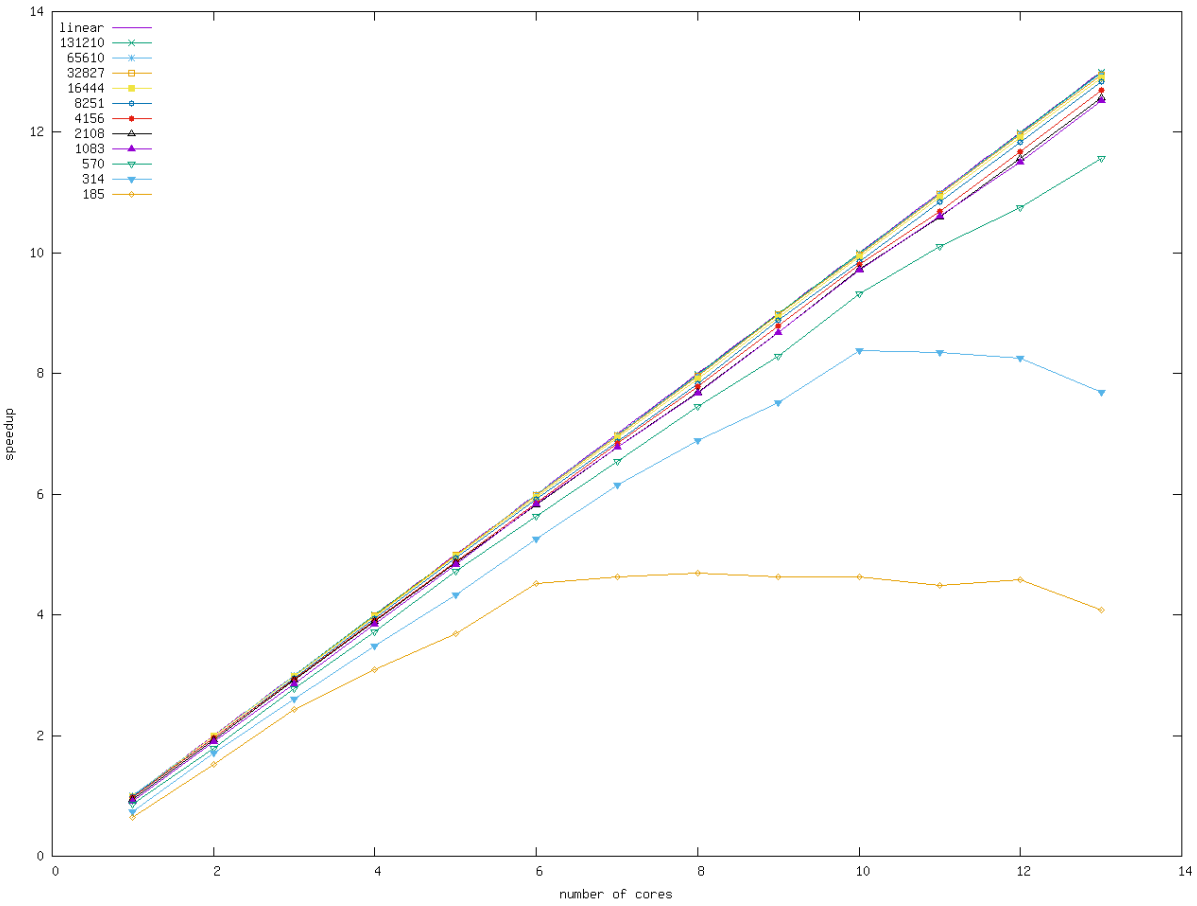
The full script is available in `starpupy/examples/starpu_py_partition.py`.

4.5 Benchmark

This benchmark gives a glimpse into how long a task should be (in µs) for the StarPU Python interface overhead to be low enough to keep efficiency. Running `starpupy/benchmark/tasks_size_overhead.sh` generates a plot of the speedup of tasks of various sizes, depending on the number of CPUs being used.
In the first figure, the return value is a handle object. In the second figure, the return value is a future object. In the third figure, the return value is `None`.
For example, in the figure of returning handle object, for a 571 µs task (the green line), StarPU overhead is low enough to guarantee a good speedup if the number of CPUs is not more than 12. But with the same number of CPUs, a 314 µs task (the blue line) cannot have a correct speedup. We need to decrease the number of CPUs to about 8 if we want to keep efficiency.




linear
131209
65609
32831
16443
8251
4156
2108
1083
570
313
184
speedup
number of cores
0
2
4
6
8
10
12
14


linear
131210
65610
32827
16444
8251
4156
2108
1083
570
314
185
speedup
number of cores
0
2
4
6
8
10
12
14

4.6 Running Python Functions as Pipeline Jobs (Imitating Joblib Library)

The StarPU Python interface also provides parallel computing for loops using multiprocessing, similarly to the `Joblib Library` that can simply turn out Python code into parallel computing code and thus increase the computing speed.

4.6.1 Examples

- The most basic usage is to parallelize a simple iteration.

```
from math import log10
[log10(10 ** i) for i in range(10)]
```

```
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
```

In order to spread it over several CPUs, you need to import the `starpu.joblib` module, and use its `Parallel` class:

```
import starpu.joblib
from math import log10
starpu.init()
starpu.joblib.Parallel(n_jobs=2)(starpu.joblib.delayed(log10)(10**i)for i in range(10))
```

```
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
```

It is also possible to first create an object of the `Parallel` class, and then call `starpu.joblib.delayed` to execute the generator expression.

```
import starpu.joblib
from math import log10
starpu.init()
parallel=starpu.joblib.Parallel(n_jobs=2)
parallel(starpu.joblib.delayed(log10)(10**i)for i in range(10))
```

```
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
```

- Instead of a generator expression, a list of functions can also be submitted as a task through the `Parallel` class.

```
import starpu.joblib
starpu.init()
#generate a list to store functions
g_func=[]
#function no input no output print hello world
def hello():
    print ("Example 1: Hello, world!")
g_func.append(starpu.joblib.delayed(hello)())
#function has 2 int inputs and 1 int output
def multi(a, b):
    res_multi = a*b
    print("Example 2: The result of ",a,"*",b,"is",res_multi)
    return res_multi
g_func.append(starpu.joblib.delayed(multi)(2, 3))
#function has 4 float inputs and 1 float output
def add(a, b, c, d):
    res_add = a+b+c+d
    print("Example 3: The result of ",a,"+",b,"+",c,"+",d,"is",res_add)
    return res_add
g_func.append(starpu.joblib.delayed(add)(1.2, 2.5, 3.6, 4.9))
#function has 2 int inputs 1 float input and 1 float output 1 int output
def sub(a, b, c):
    res_sub1 = a-b-c
    res_sub2 = a-b
    print ("Example 4: The result of ",a,"-",b,"-",c,"is",res_sub1,"and the result
 of",a,"-",b,"is",res_sub2)
    return res_sub1, res_sub2
g_func.append(starpu.joblib.delayed(sub)(6, 2, 5.9))
#input is iterable function list
starpu.joblib.Parallel(n_jobs=2)(g_func)
```

Execution:

```
Example 3: The result of  1.2 + 2.5 + 3.6 + 4.9 is 12.200000000000001
Example 1: Hello, world!
Example 4: The result of  6 - 2 - 5.9 is -1.9000000000000004 and the result of 6 - 2 is 4
Example 2: The result of  2 * 3 is 6
[None, 6, 12.200000000000001, (-1.9000000000000004, 4)]
```

- The function can also take array parameters.

```
import starpu.joblib
import numpy as np
starpu.init()
def multi_array(a, b):
        for i in range(len(a)):
                a[i] = a[i]*b[i]
A = np.arange(10)
B = np.arange(10, 20, 1)
starpu.joblib.Parallel(n_jobs=2)(starpu.joblib.delayed(multi_array)((i for i in A), (j for j in B)))
A
```

  Here the array `A` has not been modified.

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

  If we pass `A` directly as an argument, its value is updated

```
starpu.joblib.Parallel(n_jobs=2)(starpu.joblib.delayed(multi_array)(A, B))
A
```

```
array([ 0, 11, 24, 39, 56, 75, 96, 119, 144, 171])
```

  In the next call, the value of `A` is also updated.

```
starpu.joblib.Parallel(n_jobs=2)(starpu.joblib.delayed(multi_array)(b=(j for j in B), a=A))
A
```

```
array([ 0, 121, 288, 507, 784, 1125, 1536, 2023, 2592, 3249])
```

  The above three writing methods are equivalent and their execution time are very close. However, when using directly a `numpy` arrays, its value will be updated, this does not happen when generators are provided. When using a `numpy` array, it will be handled by StarPU with a data interface.

- Here an example mixing scalar objects and `numpy` arrays or generator expressions.

```
import starpu.joblib
import numpy as np
starpu.init()
def scal(a, t):
    for i in range(len(t)):
        t[i] = t[i]*a
A = np.arange(10)
starpu.joblib.Parallel(n_jobs=2)(starpu.joblib.delayed(scal)(2, (i for i in A)))
starpu.joblib.Parallel(n_jobs=2)(starpu.joblib.delayed(scal)(2,A))
```

  Again, the value of `A` is modified by the 2nd call.

```
A
```

```
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

The full script is available in `starpupy/examples/starpu_py_parallel.py`.

4.6.2 Parallel Parameters

The `starpu.joblib.Parallel` class accepts the following parameters:

- `mode` (string, default: `"normal"`)

  A string with the value `"normal"` or `"future"`. With the `"normal"` mode, you can call `starpu.joblib.Parallel` directly without using the `asyncio` module, and you will get the result when the task is executed. With the `"future"` mode, when calling `starpu.joblib.Parallel`, you will get a Future object as a return value. By setting the parameter `end_msg`, the given message will be displayed when the result is ready, then you can call `await` to get the result. The `asyncio` module should be imported in this case.

```
import starpu
import asyncio
from math import log10
starpu.init()
fut = starpu.joblib.Parallel(mode="future", n_jobs=3, end_msg="The result is
 ready!")(starpu.joblib.delayed(log10)(10**i)for i in range(10))
The result is ready! <_GatheringFuture finished result=[[0.0, 1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0,
 8.0, 9.0]]>
await fut
```

```
[[0.0, 1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]]
```

- `end_msg` (string, default: `None`)

  A message that will be displayed when the task is executed and the result is ready. When the parameter is unset, no message will be displayed when the result is ready. In any case, you need to perform awaiting to get the result.

- `n_jobs` (int, default: `None`)

  The maximum number of concurrently running jobs. If -1 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. For `n_jobs` below -1, (`n_cpus` + 1 + `n_jobs`) are used. Thus, for `n_jobs` = -2, all CPUs but one are used. `None` is a marker for 'unset' that will be interpreted as `n_jobs=1` (sequential execution). `n_cpus` is the number of CPUs detected by StarPU on the running device.

- `perfmodel` (string, default : `None`)

  Set the name of the performance model. This name will be used as the filename where the performance model information will be saved. After the task is executed, one can call the function `starpu.perfmodel_plot()` by giving the symbol of perfmodel to view its performance curve.

4.6.3 Performances

- We compare the performances of the two methods for passing arguments to `the` starpu.joblib.delayed function. The first method defines a function that contains only scalars calculations, and then we pass a generator expression as an argument. The second method defines a function that contains arrays calculations, and then we pass either `numpy` arrays or generators as arguments. The second method takes less time.

```
import starpu.joblib
import numpy as np
import time
starpu.init()
N=1000000
def multi(a,b):
    res_multi = a*b
    return res_multi
print("--First method")
A = np.arange(N)
B = np.arange(N, 2*N, 1)
start_exec1 = time.time()
start_cpu1 = time.process_time()
starpu.joblib.Parallel(n_jobs=-1)(starpu.joblib.delayed(multi)(i,j) for i,j in zip(A,B))
end_exec1 = time.time()
end_cpu1 = time.process_time()
print("the program execution time is", end_exec1-start_exec1)
print("the cpu execution time is", end_cpu1-start_cpu1)
def multi_array(a, b):
    for i in range(len(a)):
        a[i] = a[i]*b[i]
    return a
print("--Second method with Numpy arrays")
A = np.arange(N)
B = np.arange(N, 2*N, 1)
start_exec2 = time.time()
start_cpu2 = time.process_time()
starpu.joblib.Parallel(n_jobs=-1)(starpu.joblib.delayed(multi_array)(A, B))
end_exec2 = time.time()
end_cpu2 = time.process_time()
print("the program execution time is", end_exec2-start_exec2)
print("the cpu execution time is", end_cpu2-start_cpu2)
print("--Second method with generators")
A = np.arange(N)
B = np.arange(N, 2*N, 1)
start_exec3 = time.time()
start_cpu3 = time.process_time()
starpu.joblib.Parallel(n_jobs=-1)(starpu.joblib.delayed(multi_array)((i for i in A), (j for j in B)))
end_exec3 = time.time()
end_cpu3 = time.process_time()
print("the program execution time is", end_exec3-start_exec3)
print("the cpu execution time is", end_cpu3-start_cpu3)
```

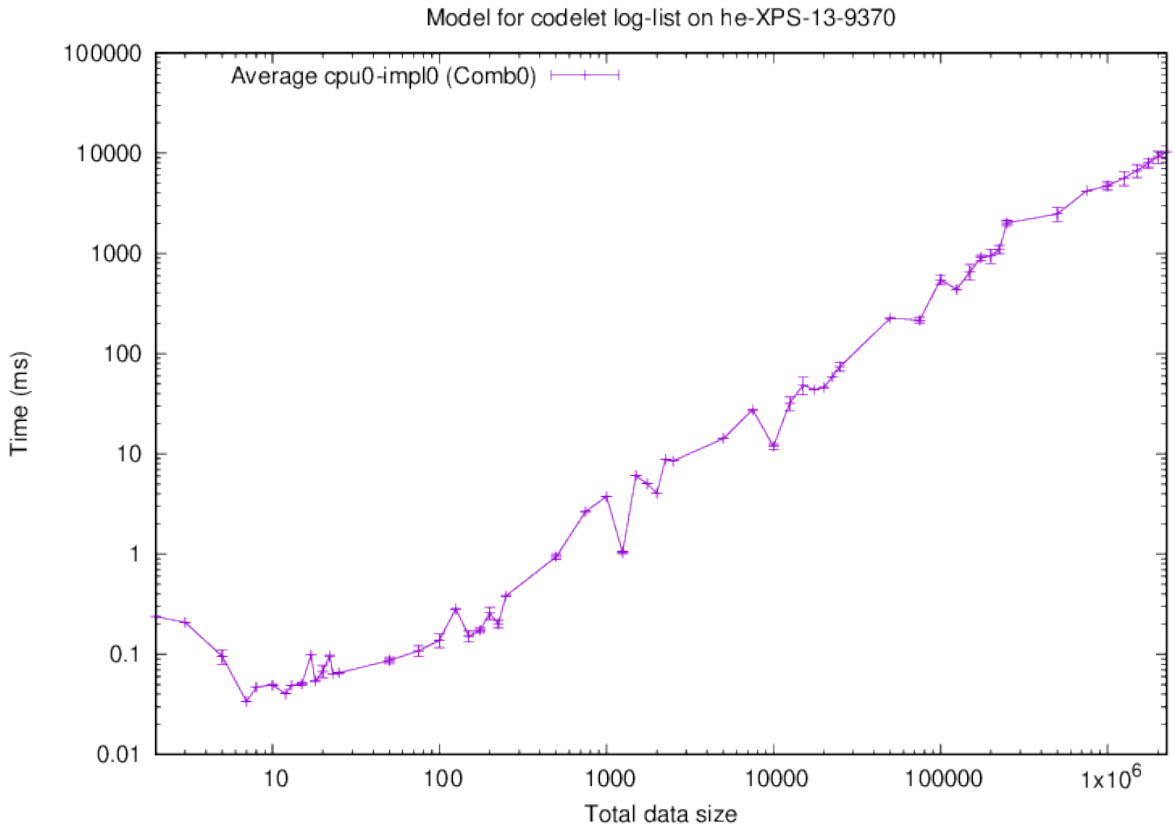
Execution:

```
--First method
the program execution time is 3.000865936279297
the cpu execution time is 5.17138062
```

```
--Second method with Numpy arrays
the program execution time is 0.7571873664855957
the cpu execution time is 0.9166007309999991
--Second method with generators
the program execution time is 0.7259719371795654
the cpu execution time is 1.1182918959999988
```

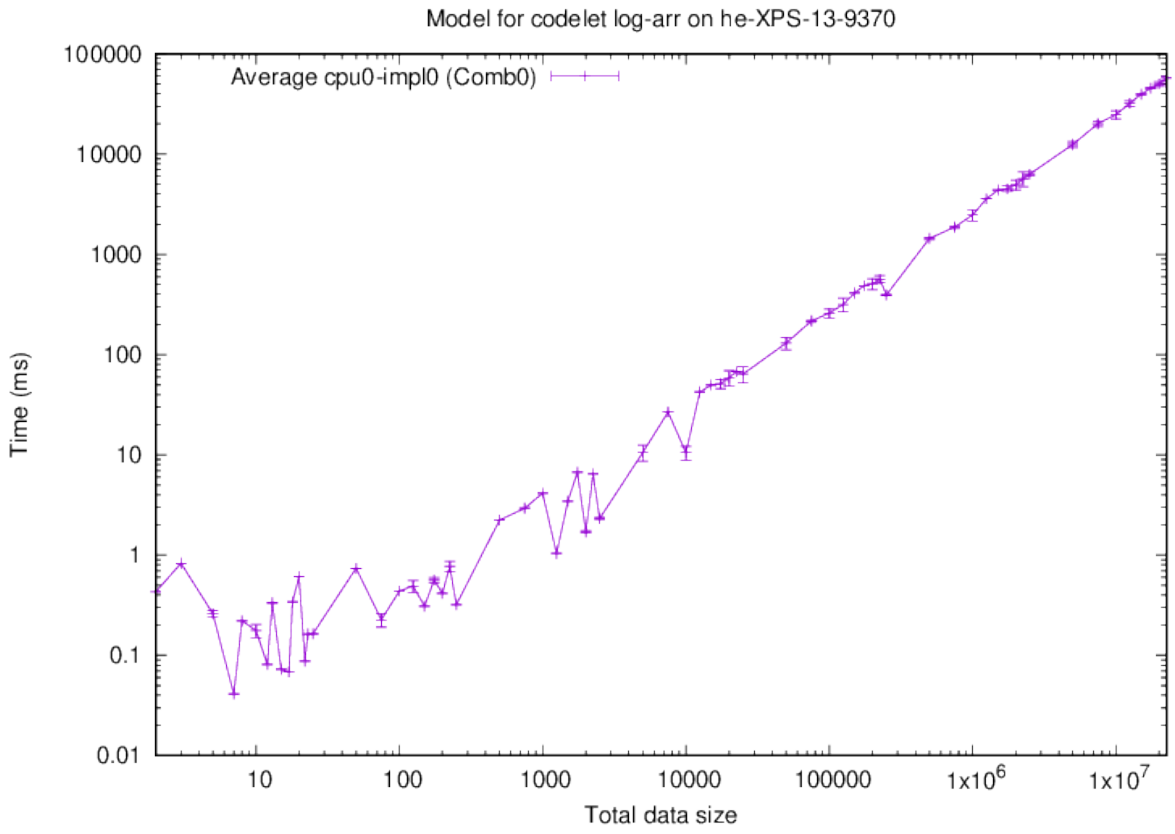
- Performance can also be shown with the performance model. Here an example with the function `log10`.

```
from math import log10
for x in [10, 100, 1000, 10000, 100000, 1000000]:
    for X in range(x, x*10, x):
        starpu.joblib.Parallel(n_jobs=-1, perfmodel="log_list")(starpu.joblib.delayed(log10)(i+1)for i
 in range(X))
starpu.perfmodel_plot(perfmodel="log_list")
```



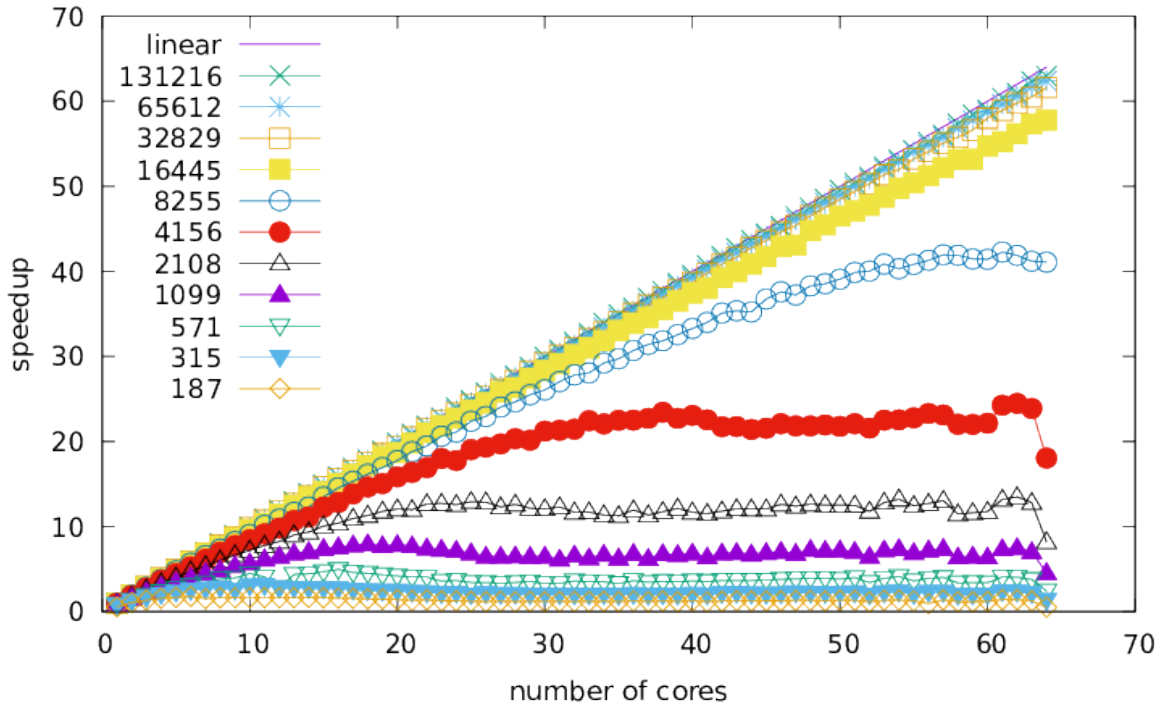
If we use a `numpy` array as parameter, the calculation can withstand larger size, as shown below.

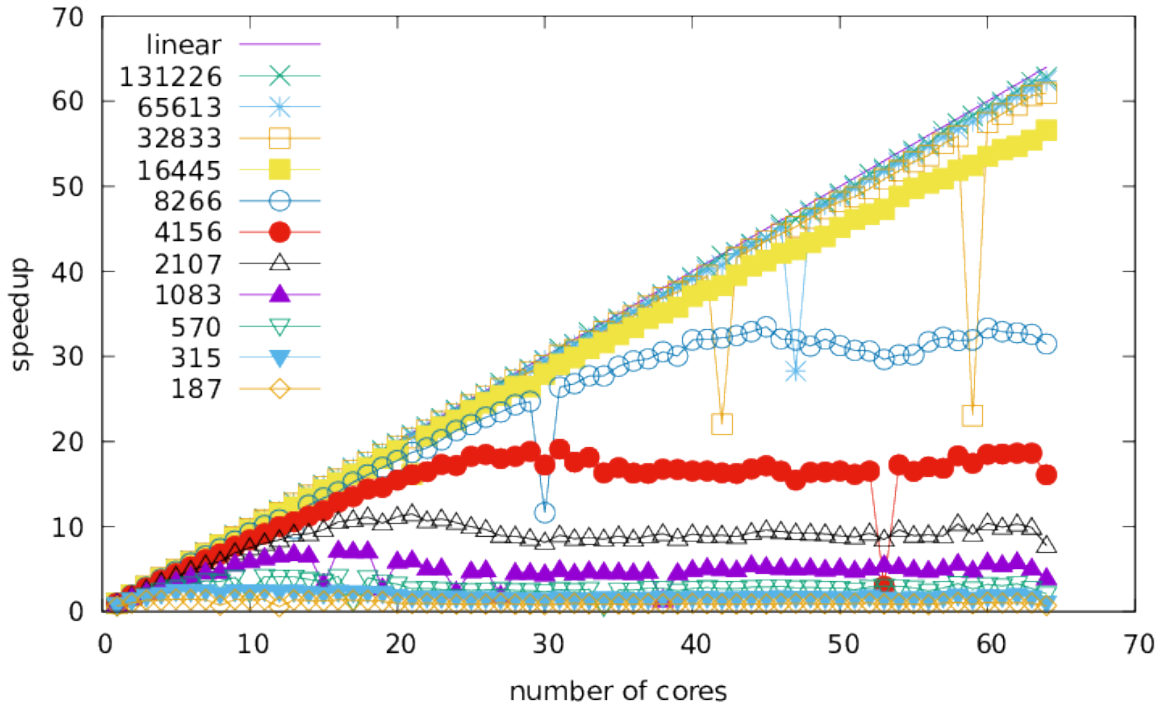
```
from math import log10
def log10_arr(t):
    for i in range(len(t)):
        t[i] = log10(t[i])
    return t
for x in [10, 100, 1000, 10000, 100000, 1000000, 10000000]:
    for X in range(x, x*10, x):
        A = np.arange(1,X+1,1)
        starpu.joblib.Parallel(n_jobs=-1, perfmodel="log_arr")(starpu.joblib.delayed(log10_arr)(A))
starpu.perfmodel_plot(perfmodel="log_arr")
```

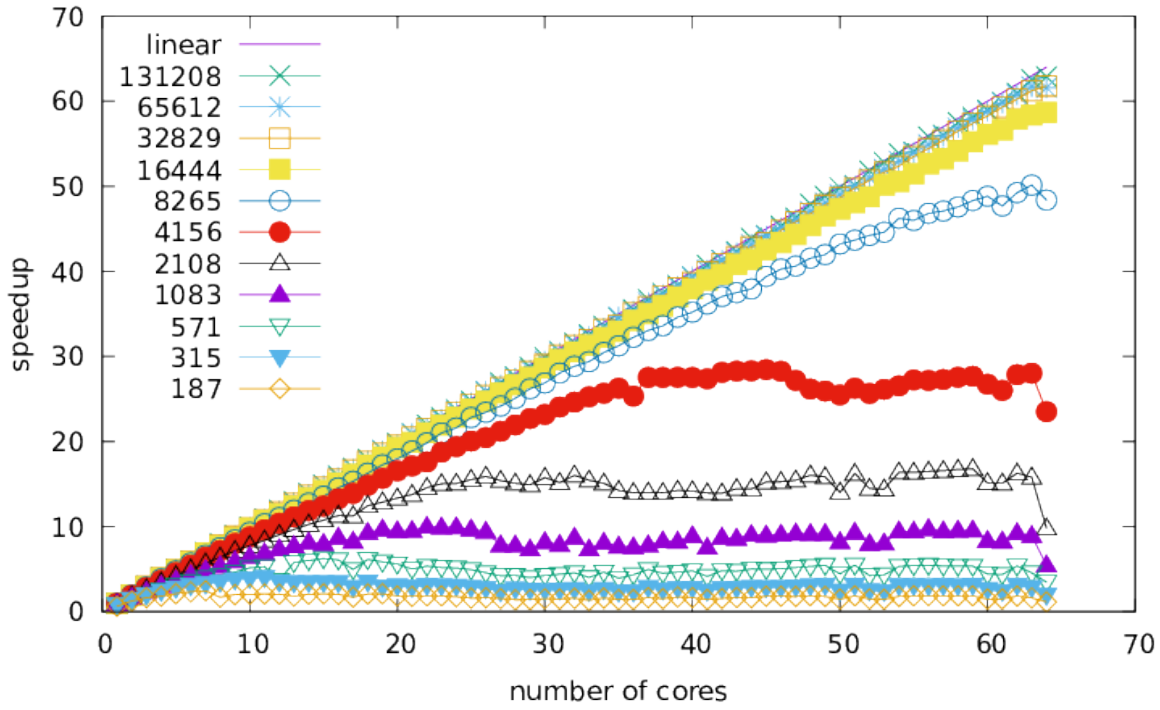


4.7 Multiple Interpreters

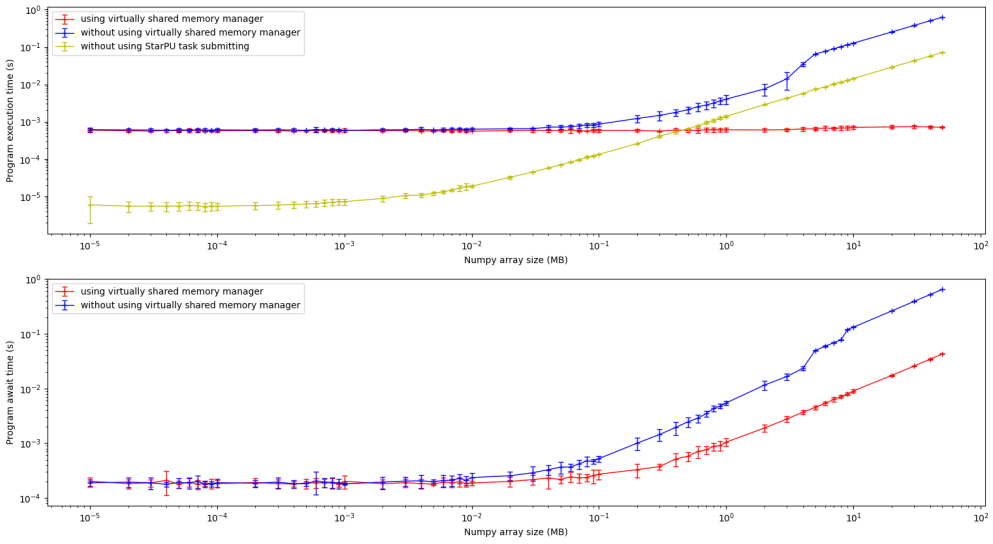
It is possible to use multiple interpreters when running python applications. To do so, you need to set the variable STARPUPY_MULTI_INTERPRETER when running a StarPU Python application.
Python interpreters share the Global Interpreter Lock (GIL), which requires that at any time, one and only one thread has the right to execute a task. In other words, GIL makes the multiple interpreters execution of Python actually serial rather than parallel, and the execution of Python program is single-threaded essentially. Therefore, if the application is pure Python script, even with multi-interpreters, the program cannot be executed in parallel, unless an external C application is called.
Fortunately now there is a quite positive development. Python developpers are preparing to implement stop sharing the GIL between interpreters (`https://peps.nogil.dev/pep-0684/`) or even make GIL optional so that Python code can be run without GIL (`https://peps.nogil.dev/pep-0701/`), that will facilitate true parallelism with the next Python version.
In order to transfer data between interpreters, the module `cloudpickle` is used to serialize Python objects in contiguous byte array. This mechanism increases the overhead of the StarPU Python interface, as shown in the following plots, to be compared to the plots given in Benchmark.
In the first figure, the return value is a handle object. In the second figure, the return value is a future object. In the third figure, the return value is `None`.


linear
131216
65612
32829
16445
8255
4156
2108
1099
571
315
187
speedup
number of cores
0
10
20
30
40
50
60
70


linear
131226
65613
32833
16445
8266
4156
2107
1083
570
315
187
speedup
number of cores
0
10
20
30
40
50
60
70



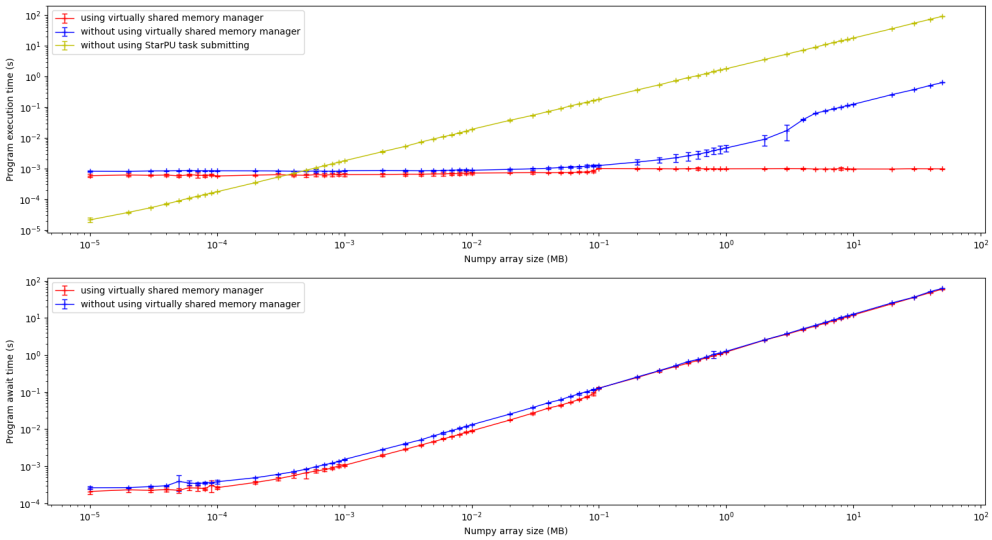
In order to reflect this influence more intuitively, we make a performance comparison.
By default, StarPU uses virtually shared memory manager for Python objects supporting buffer protocol that allows to minimize data transfers. But in the case of multi-interpreter, if we do not use virtually shared memory manager, data transfer can be realized only with the help of cloudpickle.
We will show the operation performances below (Running `test_handle_perf_pickle.sh`). The operation that we test is `numpy` addition (`numpy.add`), and the array size is 10, 20, ..., 100, 200, ..., 1000, 2000, ..., 10000, 2000, ..., 100000,200000, ..., 1000000, 2000000, ..., 10000000, ..., 50000000. We compared three cases: first, using virtually shared memory manager, second, without using virtually shared memory manager, third, without using StarPU task submitting, but directly calling `numpy.add` function.
In the first figure, we compare the submission time when using StarPU and the execution time without using Star↩ PU. We can see that there is still an obvious optimization using StarPU virtually shared memory manager when the test array size is large. However, if only using cloudpickle, StarPU Python interface cannot provide an effective optimization. And in the second figure, we can see that the same operation will take more time to finish the program execution when only using cloudpickle.



We can also define our own function to do the `numpy` operation, e.g. the element addition:

```
def add(a, b):
    for i in range(np.size(a)):
        a[i] = a[i] + b[i]
```

We will compare operation performances of the same three cases, but based on the custom function `add(a, b)`. We can see that the custom function takes more time than `numpy` function overall. Although the same operation still takes more time to submit the task when only using cloudpickle than with virtually shared memory manager, there is still a better optimization. The operation takes less time than only calling a custom function even when the array is not very large.



4.8 Master Slave Support

StarPU Python interface provides MPI master slave support as well. Please refer to MPIMasterSlave for the specific usage.

When you write your Python script, make sure to import all required functions before the `starpu` module. Functions imported after the `starpu` module can only be submitted using their name as a string when calling `task_submit()`, this will decrease the submission efficiency.

(TODO)

Chapter 5

The StarPU OpenMP Runtime Support (SORS)

StarPU provides the necessary routines and support to implement an OpenMP (http://www.openmp.org/) runtime compliant with the revision 3.1 of the language specification, and compliant with the task-related data dependency functionalities introduced in the revision 4.0 of the language. This StarPU OpenMP Runtime Support (SORS) has been designed to be targeted by OpenMP compilers such as the Klang-OMP compiler. Most supported OpenMP directives can both be implemented inline or as outlined functions.
All functions are defined in OpenMP Runtime Support.
Several examples supporting OpenMP API are provided in StarPU's `tests/openmp/` directory.

5.1 Implementation Details and Specificities

5.1.1 Main Thread

When using SORS, the main thread gets involved in executing OpenMP tasks just like every other threads, in order to be compliant with the specification execution model. This contrasts with StarPU's usual execution model, where the main thread submit tasks but does not take part in executing them.

5.1.2 Extended Task Semantics

The semantics of tasks generated by SORS are extended with respect to regular StarPU tasks in that SORS' tasks may block and be preempted by SORS call, whereas regular StarPU tasks cannot. SORS tasks may coexist with regular StarPU tasks. However, only the tasks created using SORS API functions inherit from extended semantics.

5.2 Configuration

SORS can be compiled into `libstarpu` through the `configure` option --enable-openmp. Conditional compiled source codes may check for the availability of the OpenMP Runtime Support by testing whether the C preprocessor macro `STARPU_OPENMP` is defined or not.

5.3 Initialization and Shutdown

SORS needs to be executed/terminated by the starpu_omp_init() / starpu_omp_shutdown() instead of starpu_init() / starpu_shutdown(). This requirement is necessary to make sure that the main thread gets the proper execution environment to run OpenMP tasks. These calls will usually be performed by a compiler runtime. Thus, they can be executed from a constructor/destructor such as this:

```
__attribute__((constructor))
static void omp_constructor(void)
{
        int ret = starpu_omp_init();
        STARPU_CHECK_RETURN_VALUE(ret, "starpu_omp_init");
}
__attribute__((destructor))
static void omp_destructor(void)
{
        starpu_omp_shutdown();
}
```

Basic examples are available in the files `tests/openmp/init_exit_01.c` and `tests/openmp/init_exit_02.c`.

See also

starpu_omp_init()

starpu_omp_shutdown()

5.4 Parallel Regions and Worksharing

SORS provides functions to create OpenMP parallel regions, as well as mapping work on participating workers. The current implementation does not provide nested active parallel regions: Parallel regions may be created recursively, however only the first level parallel region may have more than one worker. From an internal point-of-view, SORS' parallel regions are implemented as a set of implicit, extended semantics StarPU tasks, following the execution model of the OpenMP specification. Thus, SORS' parallel region tasks may block and be preempted, by SORS calls, enabling constructs such as barriers.

5.4.1 Parallel Regions

Parallel regions can be created with the function starpu_omp_parallel_region() which accepts a set of attributes as parameter. The execution of the calling task is suspended until the parallel region completes. The field starpu_omp_parallel_region_attr::cl is a regular StarPU codelet. However, only CPU codelets are supported for parallel regions. Here is an example of use:

```
void parallel_region_f(void *buffers[], void *args)
{
        (void) buffers;
        (void) args;
        pthread_t tid = pthread_self();
        int worker_id = starpu_worker_get_id();
        printf("[tid %p] task thread = %d\n", (void *)tid, worker_id);
}
void f(void)
{
        struct starpu_omp_parallel_region_attr attr;
        memset(&attr, 0, sizeof(attr));
        attr.cl.cpu_funcs[0] = parallel_region_f;
        attr.cl.where        = STARPU_CPU;
        attr.if_clause       = 1;
        starpu_omp_parallel_region(&attr);
        return 0;
}
```

A basic example is available in the file `tests/openmp/parallel_01.c`.

See also

struct starpu_omp_parallel_region_attr

starpu_omp_parallel_region()

5.4.2 Parallel For

OpenMP `for` loops are provided by the starpu_omp_for() group of functions. Variants are available for inline or outlined implementations. SORS supports `static`, `dynamic`, and `guided` loop scheduling clauses. The `auto` scheduling clause is implemented as `static`. The `runtime` scheduling clause honors the scheduling mode selected through the environment variable `OMP_SCHEDULE` or the starpu_omp_set_schedule() function. For loops with the `ordered` clause are also supported. An implicit barrier can be enforced or skipped at the end of the worksharing construct, according to the value of the `nowait` parameter.

The canonical family of starpu_omp_for() functions provide each instance with the first iteration number and the number of iterations (possibly zero) to perform. The alternate family of starpu_omp_for_alt() functions provide each instance with the (possibly empty) range of iterations to perform, including the first and excluding the last. A corresponding example is available in the file `tests/openmp/parallel_for_01.c`.

The family of starpu_omp_ordered() functions enable to implement OpenMP's ordered construct, a region with a parallel for loop that is guaranteed to be executed in the sequential order of the loop iterations. A corresponding example is available in the file `tests/openmp/parallel_for_ordered_01.c`.

```
void for_g(unsigned long long i, unsigned long long nb_i, void *arg)
{
        (void) arg;
```

```
        for (; nb_i > 0; i++, nb_i--)
        {
                array[i] = 1;
        }
}
void parallel_region_f(void *buffers[], void *args)
{
        (void) buffers;
        (void) args;
        starpu_omp_for(for_g, NULL, NB_ITERS, CHUNK, starpu_omp_sched_static, 0, 0);
}
```

See also

- starpu_omp_for()
- starpu_omp_for_inline_first()
- starpu_omp_for_inline_next()
- starpu_omp_for_alt()
- starpu_omp_for_inline_first_alt()
- starpu_omp_for_inline_next_alt()
- starpu_omp_ordered()
- starpu_omp_ordered_inline_begin()
- starpu_omp_ordered_inline_end()

5.4.3 Sections

OpenMP `sections` worksharing constructs are supported using the set of starpu_omp_sections() variants. The general principle is either to provide an array of per-section functions or a single function that will redirect the execution to the suitable per-section functions. An implicit barrier can be enforced or skipped at the end of the worksharing construct, according to the value of the `nowait` parameter.

```
void parallel_region_f(void *buffers[], void *args)
{
        (void) buffers;
        (void) args;
        section_funcs[0] = f;
        section_funcs[1] = g;
        section_funcs[2] = h;
        section_funcs[3] = i;
        section_args[0] = arg_f;
        section_args[1] = arg_g;
        section_args[2] = arg_h;
        section_args[3] = arg_i;
        starpu_omp_sections(4, section_f, section_args, 0);
}
```

A corresponding example is available in the file `tests/openmp/parallel_sections_01.c`.

See also

- starpu_omp_sections()
- starpu_omp_sections_combined()

5.4.4 Single

OpenMP `single` workharing constructs are supported using the set of starpu_omp_single() variants. An implicit barrier can be enforced or skipped at the end of the worksharing construct, according to the value of the `nowait` parameter. A corresponding example is available in the file `tests/openmp/parallel_single_nowait_01.c`.

```
void single_f(void *arg)
{
        (void) arg;
        pthread_t tid = pthread_self();
        int worker_id = starpu_worker_get_id();
        printf("[tid %p] task thread = %d -- single\n", (void *)tid, worker_id);
}
void parallel_region_f(void *buffers[], void *args)
{
        (void) buffers;
        (void) args;
        starpu_omp_single(single_f, NULL, 0);
}
```

SORS also provides dedicated support for `single` sections with `copyprivate` clauses through the starpu_omp_single_copyprivate() function variants. The OpenMP `master` directive is supported as well, using the starpu_omp_master() function variants. A corresponding example is available in the file `tests/openmp/parallel_single_copyprivate_01.c`.

See also

- starpu_omp_master()
- starpu_omp_master_inline()
- starpu_omp_single()
- starpu_omp_single_inline()
- starpu_omp_single_copyprivate()
- starpu_omp_single_copyprivate_inline_begin()
- starpu_omp_single_copyprivate_inline_end()

5.5 Tasks

SORS implements the necessary support of OpenMP 3.1 and OpenMP 4.0's so-called explicit tasks, together with OpenMP 4.0's data dependency management.

5.5.1 Explicit Tasks

Explicit OpenMP tasks are created with SORS using the starpu_omp_task_region() function. The implementation supports `if`, `final`, `untied` and `mergeable` clauses as defined in the OpenMP specification. Unless specified otherwise by the appropriate clause(s), the created task may be executed by any participating worker of the current parallel region.
The current SORS implementation requires explicit tasks to be created within the context of an active parallel region. In particular, an explicit task cannot be created by the main thread outside a parallel region. Explicit OpenMP tasks created using starpu_omp_task_region() are implemented as StarPU tasks with extended semantics, and may as such be blocked and preempted by SORS routines.
The current SORS implementation supports recursive explicit tasks creation, to ensure compliance with the OpenMP specification. However, it should be noted that StarPU is not designed nor optimized for efficiently scheduling of recursive task applications.
The code below shows how to create 4 explicit tasks within a parallel region.

```
void task_region_g(void *buffers[], void *args)
{
        (void) buffers;
        (void) args;
        pthread tid = pthread_self();
        int worker_id = starpu_worker_get_id();
        printf("[tid %p] task thread = %d: explicit task \"g\"\n", (void *)tid, worker_id);
}
void parallel_region_f(void *buffers[], void *args)
{
        (void) buffers;
        (void) args;
        struct starpu_omp_task_region_attr attr;
        memset(&attr, 0, sizeof(attr));
        attr.cl.cpu_funcs[0]  = task_region_g;
        attr.cl.where         = STARPU_CPU;
        attr.if_clause        = 1;
        attr.final_clause     = 0;
        attr.untied_clause    = 1;
        attr.mergeable_clause = 0;
        starpu_omp_task_region(&attr);
        starpu_omp_task_region(&attr);
        starpu_omp_task_region(&attr);
        starpu_omp_task_region(&attr);
}
```

A corresponding example is available in the file `tests/openmp/parallel_01.c`.

See also

- struct starpu_omp_task_region_attr
- starpu_omp_task_region()

5.5.2 Data Dependencies

SORS implements inter-tasks data dependencies as specified in OpenMP 4.0. Data dependencies are expressed using regular StarPU data handles (starpu_data_handle_t) plugged into the task's `attr.cl` codelet. The family of starpu_vector_data_register() -like functions, the starpu_omp_handle_register() and starpu_omp_handle_unregister() functions, and the starpu_omp_data_lookup() function may be used to register a memory area and to retrieve the current data handle associated with a pointer respectively. The testcase `./tests/openmp/task_02.c` gives a detailed example of using OpenMP 4.0 tasks dependencies with SORS implementation.
Note: the OpenMP 4.0 specification only supports data dependencies between sibling tasks, that are tasks created by the same implicit or explicit parent task. The current SORS implementation also only supports data dependencies between sibling tasks. Consequently, the behavior is unspecified if dependencies are expressed between tasks that have not been created by the same parent task.

5.5.3 TaskWait and TaskGroup

SORS implements both the `taskwait` and `taskgroup` OpenMP task synchronization constructs specified in OpenMP 4.0, with the starpu_omp_taskwait() and starpu_omp_taskgroup() functions, respectively.
An example of starpu_omp_taskwait() use, creating two explicit tasks and waiting for their completion:

```
void task_region_g(void *buffers[], void *args)
{
        (void) buffers;
        (void) args;
        printf("Hello, World!\n");
}
void parallel_region_f(void *buffers[], void *args)
{
        (void) buffers;
        (void) args;
        struct starpu_omp_task_region_attr attr;
        memset(&attr, 0, sizeof(attr));
        attr.cl.cpu_funcs[0]  = task_region_g;
        attr.cl.where         = STARPU_CPU;
        attr.if_clause        = 1;
        attr.final_clause     = 0;
        attr.untied_clause    = 1;
        attr.mergeable_clause = 0;
        starpu_omp_task_region(&attr);
        starpu_omp_task_region(&attr);
        starpu_omp_taskwait();
```

The corresponding example is available in the file `tests/openmp/taskwait_01.c`.
An example of starpu_omp_taskgroup() use, creating a task group of two explicit tasks:

```
void task_region_g(void *buffers[], void *args)
{
        (void) buffers;
        (void) args;
        printf("Hello, World!\n");
}
void taskgroup_f(void *arg)
{
        (void)arg;
        struct starpu_omp_task_region_attr attr;
        memset(&attr, 0, sizeof(attr));
        attr.cl.cpu_funcs[0]  = task_region_g;
        attr.cl.where         = STARPU_CPU;
        attr.if_clause        = 1;
        attr.final_clause     = 0;
        attr.untied_clause    = 1;
        attr.mergeable_clause = 0;
        starpu_omp_task_region(&attr);
        starpu_omp_task_region(&attr);
}
void parallel_region_f(void *buffers[], void *args)
{
        (void) buffers;
        (void) args;
        starpu_omp_taskgroup(taskgroup_f, (void *)NULL);
}
```

The corresponding example is available in the file `tests/openmp/taskgroup_01.c`.

See also

- starpu_omp_task_region()
- starpu_omp_taskwait()
- starpu_omp_taskgroup()

starpu_omp_taskgroup_inline_begin()

starpu_omp_taskgroup_inline_end()

5.6 Synchronization Support

SORS implements objects and method to build common OpenMP synchronization constructs.

5.6.1 Simple Locks

SORS Simple Locks are opaque starpu_omp_lock_t objects enabling multiple tasks to synchronize with each others, following the Simple Lock constructs defined by the OpenMP specification. In accordance with such specification, simple locks may not be acquired multiple times by the same task, without being released in-between; otherwise, deadlocks may result. Codes requiring the possibility to lock multiple times recursively should use Nestable Locks (NestableLock). Codes NOT requiring the possibility to lock multiple times recursively should use Simple Locks as they incur less processing overhead than Nestable Locks. A corresponding example is available in the file `tests/openmp/parallel_simple_lock_01.c`.

See also

starpu_omp_lock_t

starpu_omp_init_lock()

starpu_omp_destroy_lock()

starpu_omp_set_lock()

starpu_omp_unset_lock()

starpu_omp_test_lock()

5.6.2 Nestable Locks

SORS Nestable Locks are opaque starpu_omp_nest_lock_t objects enabling multiple tasks to synchronize with each others, following the Nestable Lock constructs defined by the OpenMP specification. In accordance with such specification, nestable locks may be acquired multiple times recursively by the same task without deadlocking. Nested locking and unlocking operations must be well parenthesized at any time, otherwise deadlock and/or undefined behavior may occur. Codes requiring the possibility to lock multiple times recursively should use Nestable Locks. Codes NOT requiring the possibility to lock multiple times recursively should use Simple Locks (SimpleLock) instead, as they incur less processing overhead than Nestable Locks. A corresponding example is available in the file `tests/openmp/parallel_nested_lock_01.c`.

See also

starpu_omp_nest_lock_t

starpu_omp_init_nest_lock()

starpu_omp_destroy_nest_lock()

starpu_omp_set_nest_lock()

starpu_omp_unset_nest_lock()

starpu_omp_test_nest_lock()

5.6.3 Critical Sections

SORS implements support for OpenMP critical sections through the family of starpu_omp_critical functions. Critical sections may optionally be named. There is a single, common anonymous critical section. Mutual exclusion only occur within the scope of single critical section, either a named one or the anonymous one. Corresponding examples are available in the files `tests/openmp/parallel_critical_01.c` and `tests/openmp/parallel_critical_inline_01.c`.

See also

starpu_omp_critical()

starpu_omp_critical_inline_begin()

starpu_omp_critical_inline_end()

5.6.4 Barriers

SORS provides the starpu_omp_barrier() function to implement barriers over parallel region teams. In accordance with the OpenMP specification, the starpu_omp_barrier() function waits for every implicit task of the parallel region to reach the barrier and every explicit task launched by the parallel region to complete, before returning. A corresponding example is available in the file `tests/openmp/parallel_barrier_01.c`.

See also

starpu_omp_barrier()

5.7 Example: An OpenMP LLVM Support

SORS has been used to implement an OpenMP LLVM Support. This allows to seamlessly run OpenMP applications on top of StarPU.
To enable this support, one just needs to call `configure` with the option --enable-openmp-llvm.
After installation, the directory `lib/starpu/examples/starpu_openmp_llvm` contains a OpenMP application, its source code and the executable compiled with the StarPU OpenMP LLVM support, as well as a README file explaining how to use the support for your own application.
One just needs to compile an OpenMP application with `clang` and to execute it the StarPU OpenMP LLVM support library file instead of the default `libomp.so`.

5.8 OpenMP Standard Functions in StarPU

StarPU provides severals functions which are very similar to their OpenMP counterparts but are adapted to the StarPU runtime system. These functions are:

- starpu_omp_set_num_threads()
- starpu_omp_get_num_threads()
- starpu_omp_get_thread_num()
- starpu_omp_get_max_threads()
- starpu_omp_get_num_procs() which is used to get the number of available StarPU CPU workers.
- starpu_omp_in_parallel()
- starpu_omp_set_dynamic()
- starpu_omp_get_dynamic()
- starpu_omp_set_nested()
- starpu_omp_get_nested()
- starpu_omp_get_cancellation()
- starpu_omp_set_schedule()
- starpu_omp_get_schedule()
- starpu_omp_get_thread_limit()
- starpu_omp_set_max_active_levels()

- starpu_omp_get_max_active_levels()
- starpu_omp_get_level()
- starpu_omp_get_ancestor_thread_num()
- starpu_omp_get_team_size()
- starpu_omp_get_active_level()
- starpu_omp_in_final()
- starpu_omp_get_proc_bind()
- starpu_omp_get_num_places()
- starpu_omp_get_place_num_procs()
- starpu_omp_get_place_proc_ids()
- starpu_omp_get_place_num()
- starpu_omp_get_partition_num_places()
- starpu_omp_get_partition_place_nums()
- starpu_omp_set_default_device()
- starpu_omp_get_default_device()
- starpu_omp_get_num_devices()
- starpu_omp_get_num_teams()
- starpu_omp_get_team_num()
- starpu_omp_is_initial_device()
- starpu_omp_get_initial_device()
- starpu_omp_get_max_task_priority()
- starpu_omp_get_wtime()
- starpu_omp_get_wtick()

Part I

Appendix

Chapter 6

The GNU Free Documentation License

Version 1.3, 3 November 2008

1. PREAMBLE

   The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

   This License is a kind of ``copyleft'', which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

   We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

2. APPLICABILITY AND DEFINITIONS

   This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The ``Document'', below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as ``you''. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

   A ``Modified Version'' of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

   A ``Secondary Section'' is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

   The ``Invariant Sections'' are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The ``Cover Texts'' are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A ``Transparent'' copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not `Transparent'' is called`Opaque''.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The ``Title Page'' means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, ``Title Page'' means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The ``publisher'' means any person or entity that distributes copies of the Document to the public.

A section ``Entitled XYZ'' means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as `Acknowledgements'', Dedications'', `Endorsements'', or`History''.) To ``Preserve the Title'' of such a section when you modify the Document means that it remains a section ``Entitled XYZ'' according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

3. VERBATIM COPYING

   You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

   You may also lend copies, under the same conditions stated above, and you may publicly display copies.

4. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a

computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

5. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

(a) Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

(b) List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

(c) State on the Title page the name of the publisher of the Modified Version, as the publisher.

(d) Preserve all the copyright notices of the Document.

(e) Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

(f) Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

(g) Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

(h) Include an unaltered copy of this License.

(i) Preserve the section Entitled ``History'', Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled ``History'' in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

(j) Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the ``History'' section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

(k) For any section Entitled `Acknowledgements'' or`Dedications'', Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

(l) Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

(m) Delete any section Entitled ``Endorsements''. Such a section may not be included in the Modified Version.

(n) Do not retitle any existing section to be Entitled ``Endorsements'' or to conflict in title with any Invariant Section.

(o) Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled ``Endorsements'', provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-↩ Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

6. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled ``History'' in the various original documents, forming one section Entitled `History''; likewise combine any sections EntitledAcknowledgements", and any sections Entitled `Dedications''. You must delete all sections EntitledEndorsements."

7. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

8. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an ``aggregate'' if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

9. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled `Acknowledgements'', Dedications", or ``History'', the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

10. TERMINATION

    You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

    However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

    Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

    Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

11. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License `‘or any later version’’ applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

12. RELICENSING

    `Massive Multiauthor Collaboration Site'' (or`MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A `Massive Multiauthor Collaboration'' (or`MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

    `‘CC-BY-SA’’ means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

    `‘Incorporate’’ means to publish or republish a Document, in whole or in part, as part of another Document.

    An MMC is `‘eligible for relicensing’’ if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

    The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

6.1 ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (C) *year your name*. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled `‘GNU Free Documentation License’’.

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the `‘with...Texts.’’ line with this:

with the Invariant Sections being *list their titles*, with the Front-Cover Texts being *list*, and with the Back-Cover Texts being *list*.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.
If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.