Best Practice Guide for Writing MPI + OmpSs Interoperable Programs

Version 1.0, 3rd April 2017

Table of Contents

Table of Figures

Table of Codes

1 Introduction

OmpSs is a parallel programming model based on tasks and developed at Barcelona Supercomputing Center. OmpSs also tries to be a test bench for the OpenMP programming model in order to improve its tasking model. In particular, our objective is to extend OpenMP with new directives, clauses and semantics to support asynchronous parallelism.

OmpSs uses preprocessor annotations to express the concurrency of the application. Programmers can create tasks (through task generating constructs) and guarantee data race free programs though synchronization mechanisms (e.g. dependences, taskwaits, atomics, critical, etc.). Tasks are the smallest unit of work which represents a specific instance of an executable code and its associated data. Dependences let the user express the data flow of the program, so that at runtime this information can be used to determine if the parallel execution of two tasks may cause data races or not. Other synchronization constructs, such as critical or atomic, guarantee correct access to these variables when dependences are not completely needed.

MPI has been used, since its appearance in 1994, as one of the most widespread programming models for distributed memory environments. The API has been evolved through the years in order to include more functionality and adapt himself to take into account new hardware architectures. The standard defines a set of library routines that allow writing portable message-passing programs that usually follow the Single Process Multipe Data (SPMD) execution model, but also supports the Multiple Program Multiple Data execution model. (MPMD)

MPI programs execute multiple processes, each one executing their own code and using the MPI communication primitives in order to explicitly communicate data between nodes and synchronize sections of code running in parallel. Each process executes in its own memory address space.

As also discussed in the “Best Practice Guide to Hybrid MPI + OpenMP Programming” [3] there are two principal reasons why a hybrid version of an application MPI plus a shared-memory programming model might be beneficial. The first is to reduce the memory requirements of the application; the second is to improve its performance. The document reference above also provides more details about the advantages and disadvantages when programming with this combination of programming models. In addition, this document can also be considered as an extension of the MPI + OpenMP best practice guide in those aspects concerning the tasking model due most of the issues discussed in this document can also be applied to OpenMP.

1.1 Purpose

The purpose of this document is to establish a set of advice and guidelines to be used when programming hybrid applications using MPI + OmpSs programming models. The document will serve as a guide for application developers that are considering taking advantage of the multilevel parallelism offered by modern High Performance Computing (HPC) systems using these two programming models (i.e. invoking MPI calls within asynchronous tasks).

1.2 Glossary of Acronyms

API	Application Programming Interface
CPU	Central Processing Unit
HPC	High Performance Computing
MPMD	Multiple Program Multiple Data
MPI	Message Passing Interface
OmpSs	OpenMP Super Scalar

ORTE	Open Run-Time Environment
PGAS	Partitioned Global Address Space
RAW	Read After Write (dependence)
SMP	Symmetric Multi-Processor
SMPD	Single Program Multiple Data
WAR	Write After Read (dependence)
WAW	Write After Write (dependence)

2 Interoperability

Currently, MPI and OmpSs/OpenMP[1] are totally decoupled. For instance, the OmpSs runtime does not know whether a MPI call will block or not, and likewise, the MPI runtime does not know if a thread calling an MPI primitive is in a task created by the OmpSs runtime. Each of these programming models is focused on exploiting a different level of parallelism. MPI provides explicit communication primitives to exploit inter-process parallelism (inter- or intra- node) while OmpSs exploits parallelism within a process (always intra-node). Mixing both MPI and OmpSs programming models in the same application is a tricky task, as features of both languages might easily interact in an unexpected way, resulting in dead-locks, incorrect results, fatal errors or performance issues. Sections 3 and 5 describe some of the most common pitfalls when MPI and OmpSs are used together.

Despite the fact these two programming models do not offer (at the moment) any common service, there are some mechanisms and guidelines that can be used to safely combine them, such as to enclose MPI communication primitives inside OmpSs tasks and let the OmpSs dependency system manage the synchronization between them. This methodology facilitates the overlapping of communication and computation phases, generally improving the application performance. These kinds of techniques will be explained in Sections 4, 6 and 8.

[1]The rest of this document will refer to the OmpSs programming model although discussion and techniques may also apply to OpenMP tasking model.

3 MPI Initialization and the multi-threaded levels

MPI programs start with a function call which initializes the message passing library. MPI standard defines two different functions for this purpose [12.4.3]:

- `MPI_Init()`, used when no multi-threading support is needed.
- `MPI_Init_thread()`, specifies a desired level of multi-threading support.

These routines must be called by one thread only. That thread is called the main thread and must be the thread that calls `MPI_Finalize()`.

The programmer must provide a desired level of multi-threading support to the `MPI_Init_thread()` which, in turn, may return a value lower than requested. This is because different library implementations may be restricted to different levels (e.g. absence of locking mechanisms for efficiency in single-threaded programs).

Valid multi-threading level values are:

- `MPI_THREAD_SINGLE`, only one thread will execute the program.
- `MPI_THREAD_FUNNELED`, the process may be multi-threaded, but only the main thread will make MPI calls (all MPI calls are funnelled to the main thread).
- `MPI_THREAD_SERIALIZED`, the process may be multi-threaded, and multiple threads may make MPI calls, but only one at a time: MPI calls are not made concurrently from two distinct threads (all MPI calls are serialized).
- `MPI_THREAD_MULTIPLE`, multiple threads may call MPI, with no restrictions.

Multithreading MPI programs should never use MPI_THREAD_SINGLE multithreading level. It is strongly advised to always check threading support return value after a call to `MPI_Init_thread()` as done in Code 1.

```
int main( int argc, char* argv[] ) {
   int ierr, provided, required = MPI_THREAD_SERIALIZED;
   ierr = MPI_Init_thread( &argc, &argv, required, &provided );

   if( provided < required ) {
      fprintf( stderr, "Error: MPI implementation does not "
                       "support serialized multithreading.\n" );
      abort();
   }
   // [...]
}
```

Code 1: Initializing the MPI communication library (multi-threading level)

Note that the check is performed using an integral comparison, that satisfies `MPI_THREAD_SINGLE < MPI_THREAD_FUNNELED < MPI_THREAD_SERIALIZED < MPI_THREAD_MULTIPLE`.

4 MPI Communication and computation overlap

Communication and computation overlap can improve performance of applications, by performing useful work while the message data is still in transit.

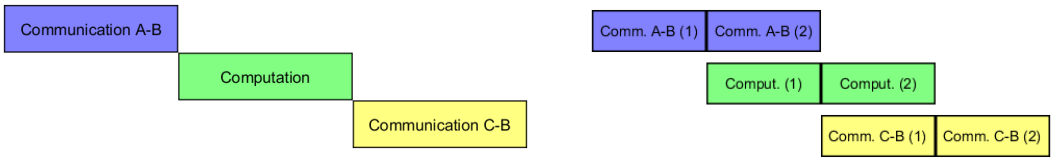

Figure 1: MPI non-overlapping vs. overlapping communication

```
// Perform computation on a single element.
// Only modifies left operand.
void compute( element_t* left, const element_t* right );

void solve( element_t* local, size_t length,
            MPI_Datatype mpi_type_element ){
   int rank, nprocs, prev, next;
   element_t* remote;

   MPI_Comm_rank( MPI_COMM_WORLD, &rank );
   MPI_Comm_size( MPI_COMM_WORLD, &nprocs );

   prev = (rank-1) % nprocs;
   next = (rank+1) % nprocs;

   remote = (element_t*) malloc( sizeof(element_t) * length );

   // Send local elements to the next process
   MPI_Send( local, length, mpi_type_element, next, 0/*tag*/ );

   for( int s = 0; s < steps; ++s ) {
      // Communication phase: wait until remote elements arrive
      MPI_Recv( remote,  length, mpi_type_element, prev,
                0 /*tag*/, MPI_COMM_WORLD, MPI_STATUS_IGNORE );

      // Computation phase
      #pragma omp parallel for
      for( size_t i = 0; i < length; ++i ) {
         compute( local[i], remote[i] );
      }
      // implicit thread synchronization barrier

      // Skip communication in last iteration
      if( s != steps-1 ) {
         // Communication phase: forward elements to next process
         MPI_Send( remote,  length, mpi_type_element, next,
                   0 /*tag*/, MPI_COMM_WORLD, MPI_STATUS_IGNORE );
      }
   }
}
```

Code 2: Non-overlapping communication and computation program

```
// Computation on single element. Only modifies left operand.
void compute( element_t* left, const element_t* right );

void solve( element_t* local, size_t tot_len,
            MPI_Datatype mpi_type_element ) {
  int rank, nprocs, prev, next;
  element_t* remote;

  int tags[2];
  element_t* loc_chunks[2], rem_chunks[2];
  size_t lengths[2];

  MPI_Request send_reqs[2], recv_reqs[2];

  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  MPI_Comm_size( MPI_COMM_WORLD, &nprocs );

  prev = (rank-1) % nprocs;       next = (rank+1) % nprocs;
  remote = (element_t*) malloc( sizeof(element_t) * tot_len );

  tags[0]      = 0;          tags[1]      = 1;
  lengths[0]   = tot_len/2;  lengths[1]   = tot_len - lengths[0];
  loc_chunks[0]= &local[0];  loc_chunks[1]= &loc[lengths[0]];
  rem_chunks[0]= &remote[0]; rem_chunks[1]= &rem[lengths[0]];

  // Communication phase A: send local elements next process.
  for( int b = 0; b < 2; ++b )
    MPI_Isend( loc_chunks[b], lengths[b], mpi_type_element,
               next, tags[b], MPI_COMM_WORLD, &send_reqs[b] );

  for( int s = 0; s < steps; ++s ) {
    // Communication phase B: receive from the previous process.
    for( int b = 0; b < 2; ++b ) {
      // Wait until remote buffer is available (reusing it).
      if( s > 0 ) MPI_Wait( send_reqs[b], MPI_STATUS_IGNORE );

      MPI_Irecv( rem_chunks[b], lengths[b], mpi_type_element,
                 prev, tags[b], MPI_COMM_WORLD, &recv_reqs[b] );
    }

    // Computation phase: Wait until element buffers are released
    // from Comm phases A and B.
    for( int b = 0; b < 2; ++b ) {
      MPI_Request comm_reqs[2] = { send_reqs[b], recv_reqs[b] };
      MPI_Waitall( 2, comm_reqs, MPI_STATUSES_IGNORE );

      // Perform first half computations (barrier at the end)
      #pragma omp parallel for
      for( size_t i = 0; i < lengths[b]; ++i )
         compute( loc_chunks[b][i], rem_chunks[b][i] );

      // Communication phase C: forward received elements to next
      // process, since computation is over. Skip last iteration.
      if( s != steps-1 )
         MPI_Isend( rem_chunks[b], lengths[b], mpi_type_element,
                    next, tags[b], MPI_COMM_WORLD, &send_reqs[b] );
    }
  }
}
```

Code 3: Overlapping communication and computation program

The example shown in Code 2 illustrates a simple hybrid MPI + OmpSs algorithm with well-defined communication and computation phase. Note that `MPI_Recv()` routine blocks program execution until the message reception is completed and the `remote` buffer is usable. On the other hand, communication cannot start until all threads involved in the parallel region finish, due to the implicit synchronization barrier at the end of this construct.

For this purpose, MPI provides a mechanism known as non-blocking communication that allows routines to return immediately, regardless if they can be considered as completed. This gives the user the opportunity to do other independent work in the meantime. In addition to regular blocking routines, these return an additional request object that will be used to check if they have finished.

In the example in Code 3 we have modified the program so that it now uses double buffering to allow communication and computation overlap. In this version, computation and communication phases have been split into two halves, allowing the program to start the second computation phase without waiting to receive data from the first one. Note how the request objects are used to make the program wait whenever the incoming/outgoing buffer is going to be used.

Further improvements are possible, though. Code 3 still forces threads to synchronize at the end of the computation phase. Breaking this synchronization could be possible by means of asynchronous parallelism (e.g. tasks and data dependences).

5 Avoiding two-sided communication false-matching

In multi-threaded applications where both complex communication patterns and communication-computation overlap take place, it is important to understand the general properties of point-to-point communications, so that errors can be avoided.

Code 3 (i.e. communication and computation overlap) can potentially use two different message sizes in the case where the total amount of elements is not evenly divisible by 2. The reception of the second half of elements using the first reception buffer `rem_chunks[0]` is considered erroneous, as its size is not big enough to hold the entire message. For this reason, it is really important to identify messages as uniquely as possible, so that the non-determinism caused by concurrent execution cannot produce this kind of errors.

> *"A message can be received by a receive operation if its envelope matches the* `source`*,* `tag` *and* `comm` *values specified by the receive operation. The receiver may specify a wildcard* `MPI_ANY_SOURCE` *value for* `source`*, and/or a wildcard* `MPI_ANY_TAG` *value for* `tag`*, indicating that any source and/or tag are acceptable."*
>
> MPI: A Message-Passing Interface Standard 3.1 (p. 29: 23-27)

Code 3 (in the previous section) differentiates the communication of the two halves by using a different value for the tag, since both of them share the same source, destination and communicator.

MPI guarantees that two matching messages (same source, communicator and tag) are received in the same order that they were sent. In the case where the process is multi-threaded, there may not be a defined relative order of messages (there is no synchronization between them). In this case, two receives can match these messages in any order.

Therefore, since Code 2 program communication order is well defined (the master thread performs all sends and receives in order) is not required to use different tags for the first and second halves, as it always performs the send and receive in the same order (first half then second). However, when using asynchronous parallelism, the second computation phase could finish before the first one, reversing the original order of the messages. Using two different tags ensures that each receive matches its own half.

The MPI standard defines several other properties that must be satisfied by any implementation. See section 3.5 for details.

> *"Advice to users. The use of different communicators offers some flexibility regarding the matching of non-blocking collective operations. In this sense, communicators could be used as an equivalent to tags. However, communicator construction might induce overheads so that this should be used carefully. (End of advice to users.)"*
>
> MPI: A Message-Passing Interface Standard 3.1 (p. 291: 5-7)

In addition, using different communicators also improves flexibility in point-to-point communications if the program uses a wide range of tag values or if tag generation is not trivial.

6 Asynchronous communication and tasks

The user can extend OmpSs data driven dependences in hybrid OmpSs + MPI programs by *taskifying* MPI communication routines. These will allow the program to synchronize task execution beyond the process level, since a task will not finish until the data has arrived or has been sent as seen in Figure 2.

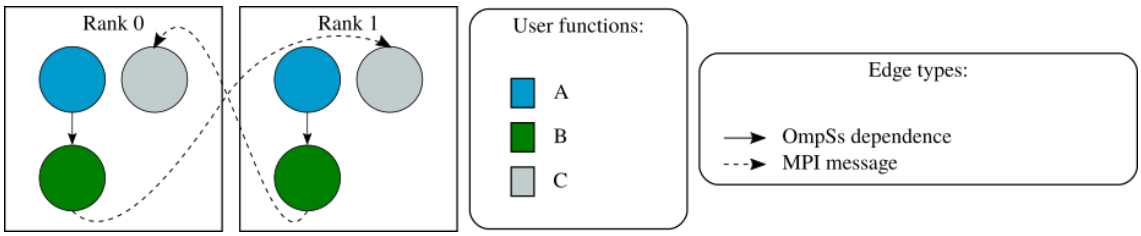

Figure 2: Task Dependence Graph of two MPI processes

Even though it is possible to completely wrap MPI routines with OmpSs tasks, the user should be aware that doing so may introduce the possibility of deadlock.

```
int buddy = ...; /* rank of another MPI process */

int b;
int a;

void example() {
   // Task A
   #pragma omp task out(a) priority(1)
   a = 10;

   // Task B
   #pragma omp task in(a) priority(1)
   MPI_Send(&a, 1, MPI_INT, buddy, 0/*tag*/, MPI_COMM_WORLD);

   // Task C
   #pragma omp task out(b) priority(2)
   MPI_Recv( &a, 1, MPI_INT, buddy, 0/*tag*/, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE );

   #pragma omp taskwait
}
```

Code 4: Wrapping MPI routines in OmpSs tasks.

Code 4 shows a case where a specific order of task scheduling can produce the single thread execution to hang: all processes execute task C first, making the thread wait for a message that is never sent. This thread enters the MPI routine and cannot leave it until the communication is completed. Using MPI non-blocking routines instead, we make sure that this deadlock effect is moved inside the task body, using the following MPI code:

```
MPI_Recv( ... );
```

```
int flag = 0;
MPI_Request req;
MPI_Irecv( ..., &req );
while( flag == 0 ) {
  MPI_Test( &req, &flag,
            MPI_STATUS_IGNORE );
}
```

This transformation is not enough because the thread still waits inside the task. We can suspend the execution of this task in favor of others using the `taskyield` directive. Code 5 shows an alternative implementation of `Task C` using this technique.

```
// Task C
#pragma omp task out(b) priority(2)
{
   int flag = 0;
   MPI_Request req;
   MPI_Irecv( &b, 1, MPI_INT, buddy, 0/*tag*/,
              MPI_COMM_WORLD, &req );

   MPI_Test( &req, &flag, MPI_STATUS_IGNORE );
   while( flag == 0 ) {
      #pragma omp taskyield
      MPI_Test( &req, &flag, MPI_STATUS_IGNORE );
   }
}
```

Code 5: Alternative implementation using non-blocking MPI routines and taskyield.

The main drawback of this technique is that modifying every single MPI routine of the application is cumbersome. In addition, tasks can be resumed even though the requests they are waiting for have not been completed yet. Therefore, they are resumed specifically to check the completion of the communication, which is not really efficient, as they may just be suspended again.

In order to solve these problems, OmpSs offers the user the possibility to link against an external library: the OmpSs-MPI interoperability library. This library intercepts the most common MPI routines, transforms them into the equivalent non-blocking counterparts and automatically blocks them until the MPI request object is marked as completed. Thanks to the MPI profiling interface [MPI 14.2], the calls are intercepted transparently to the user, so that the application looks exactly like Code 4, although it behaves like Code 5 and avoids unnecessary context-switches.

7 Executing MPI + OmpSs applications

In order to execute MPI programs users must use the `mpirun` command. This command is actually a shell script (a.k.a. `mpiexec` and `orterun`) that attempts to hide the specific details of the environment when starting MPI processes on a cluster of workstations. The command controls several aspects of the MPI program execution. It relies on the Open Run-Time Environment (ORTE) in order to launch jobs and it closely works with the job scheduler (if used to submit the job to queues). In the case no job scheduler is being used the `mpirun` command relies on a `hostfile` provided by the user and starts remote MPI programs using `ssh`.

The `mpirun` command allows exporting environment variables to remote nodes. This option can be useful when combined with OmpSs to export environment variables which configure the execution of OmpSs (e.g. number of threads, scheduler policy, instrumentation or debug options, etc.).

The following examples are based on Open MPI version 1.2.9.1, but similar options (if not the same) can be found in other implementations of this tool.

From the `mpirun` help command line:

```
-x <env>

Export the specified environment variables to the remote nodes before executing the program. Only one environment variable can be specified per -x option. Existing environment variables can be specified or new variable names specified with corresponding values. For example: % mpirun -x DISPLAY -x OMP_NUM_THREADS=16

The parser for the -x option is not very sophisticated; it does not even understand quoted values. Users are advised to set variables in the environment, and then use -x to export (not define) them.
```

7.1 Binding processes to CPUs

When executing MPI programs is a good practice to keep information about how the processes have been mapped to nodes, as it allows matching this information with performance anomalies, oversubscription, or CPU binding information. The `mpirun` command provides several options that allow users to show a brief summary about their program deployment.

```
-display-map, --display-map

Display a table showing the mapped location of each process prior to launch.

-display-devel-map, --display-devel-map

Display a more detailed table showing the mapped location of each process prior to launch (usually of interest to developers).

-display-allocation, --display-allocation

Display the detected resource allocation.
```

When running multi-threaded applications we need also to ensure that created threads are properly bound to CPUs. In this section we present a simple example in order to check where our MPI processes are executed. Although this code is presented to illustrate the different options, programmers may write their own tests in order to verify how and where their programs are executed.

```
int main ( int argc, char *argv[] ) {
   cpu_set_t mask;
   sched_getaffinity( 0, sizeof(mask), &mask );

   printf("(%d):",getpid());
   for ( int i=0; i<NCPUS; i++ )
      printf("%d", CPU_ISSET( i, &mask ));
   printf("\n");

   return 0;
}
```

Code 6: Getting and printing the affinity mask of the process

This program will print the process id followed by a sequence of 0's and 1's (being 0 – disabled and 1 – enabled) according with the use of the CPUs. Running `mpirun` with 4 processes this is a possible output:

```
$ mpirun -np 2 a.out
(31431):100000000000
(31432):010000000000
```

In this case each MPI process is bound to a different CPU (in a one-to-one fashion). A different output will be printed when using the `-bysocket` flag. In this case MPI processes will be placed in the initial CPU of each memory socket, producing this output:

```
$ mpirun -np 2 -bysocket a.out
(25072):100000000000
(25073):000000100000
```

If your application uses threads, then you probably want to ensure that you are either not bound at all (by specifying `-bind-to-none`), or bound to multiple cores using an appropriate binding level, or specific number of processing elements per application process. In the following screen snapshot we show the output when using the `mpirun`'s option -bind-to-none and when combining `-bysocket` and `-bind-to-socket` options.

```
$ mpirun -np 2 -bysocket -bind-to-none a.out
(20583):111111111111
(20584):111111111111

$ mpirun -np 2 -bysocket -bind-to-socket a.out
(20773):111111000000
(20774):000000111111
```

Programmers may want to run their applications in more complex scenarios. They can combine different `mpirun` options in order to assign sets of CPUs to processes. In the following examples we based the distribution of CPUs per-core (not per-socket, like the previous example) and assigning 3 CPUs per process. The result is:

```
$ mpirun -np 4 -cpus-per-proc 3  -bind-to-core a.out
(897):111000000000
(898):000111000000
(899):000000111000
(900):000000000111
```

Finally we want to finalize the list of examples on how to launch MPI programs by commenting on the option `-report-bindings`, which provides similar information to our small example in Code 6:

```
$ mpirun -np 4 -cpus-per-proc 3  -bind-to-core --report-bindings a.out
[log1:2773] rank 0 bound to socket 0[core 0-2]: [B B B . . .][. . . . . .]
[log1:2773] rank 1 bound to socket 0[core 3-5]: [. . . B B B][. . . . . .]
[log1:2773] rank 2 bound to socket 1[core 0-2]: [. . . . . .][B B B . . .]
[log1:2773] rank 3 bound to socket 1[core 3-5]: [. . . . . .][. . . B B B]
```

Note that information provided by these two examples is almost the same. We launch 4 processes each one uses 3 CPUs. The `–report-bindings` option also gives us information about the socket where the CPUs reside.

Option names provided in this section are based on Open MPI version 1.2.9.1. Future versions of Open MPI will use `–map-by` command line option.

```
--------------------------------------------------------------------------
The following command line options and corresponding MCA parameter have
been deprecated and replaced as follows:

  Command line options:
    Deprecated:  --cpus-per-proc, -cpus-per-proc,
                 --cpus-per-rank, -cpus-per-rank
    Replacement: --map-by <obj>:PE=N

  Equivalent MCA parameter:
    Deprecated:  rmaps_base_cpus_per_proc
    Replacement: rmaps_base_mapping_policy=<obj>:PE=N

The deprecated forms *will* disappear in a future version of Open MPI.
Please update to the new syntax.
--------------------------------------------------------------------------
```

We have preferred to use older option names instead of the new ones because they are still working in all Open MPI implementations, but users should consider checking the specific manual of their actual installed version. There may be other options listed with:

```
$ mpirun --help.
```

7.2 Binding threads to CPUs

If in the previous section we have discussed how to bind processes to a set of CPUs: in this section we will discuss how to bind threads to CPUs. This feature is handled by the task-based runtime system (in general by the multi-threaded runtime). In the specific case of OmpSs threads are usually bound one-to-one to CPUs.

By default, OmpSs will use a logical thread per CPU found in the environment execution. The OmpSs runtime will use the CPU mask used to execute the process to determine the number of threads and the thread binding. Depending on the specific OmpSs version there are some options available that can be used in order to modify this. Check the user guide in order to know what options apply for your installed options.

8 Synchronize communication with dependences

One of the first approaches that programmers adopt when starting to program hybrid MPI + OmpSs applications is to work at independent levels of parallelism: MPI works at cluster level (i.e. inter-process) and OmpSs at node level (i.e. intra-process). In this approach there is no interaction between the MPI calls and the OmpSs directives, and usually the phases of computation are alternated with phases of communication. Further details can be found at "Best Practice Guide to Hybrid MPI + OpenMP Programming" [3]. The first attempt to combine services of both programming models usually consists of the creation of OmpSs tasks wrapping the MPI communication calls.

Using this programming pattern we are able to interconnect the tasks that receive messages (communication tasks) with the parts of code that will later on use them. In the same way we can connect the pieces of code that calculate certain results with the tasks that will later communicate them through the network with the rest of processes. We can see an example of this use in the example of Code 7.

```
#pragma omp task
MPI_Receive(X)

#pragma omp taskwait

for (;;;) {
   #pragma omp task
   for (;;;) {
     compute (X[i][j])
   }
}

#pragma omp taskwait

#pragma omp task
MPI_Send(X)
```

Code 7: Synchronizing communication tasks with taskwait

In order to make this type of code work properly, we must include the corresponding synchronization mechanisms. We have to guarantee that when we start the computation on a given piece of data, it must be present in our MPI process. On the other hand, when we want to send data through the interconnection network we have to ensure that the data has already been updated before sending it. A first approach would be to ensure that all the tasks being instantiated so far have ended (including those that were in charge of communication). The solution in this case is simple: include a taskwait before the computation or the send that requires the receipt or computation of this data respectively (as it is done in Code 7).

OmpSs implements a dependency system that lets the programmer relax the synchronization among these code phases, allowing a more fine-grain synchronization in a point-to-point fashion.

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

Each time a new task is created its dependencies are matched against of those of existing tasks. If a dependency, either Read-after-Write (RaW), Write-after-Write (WaW) or Write-after-Read (WaR), is found the task becomes a successor of the corresponding tasks. This process creates a task dependency graph at runtime. Tasks are scheduled for execution as soon as all their predecessors in the graph have

finished (which does not mean they are executed immediately), or at creation if they have no predecessors.

With this mechanism we are able to connect in a predecessor and successor scheme the communication and computation done by tasks working with the same data. Code 8 shows how the previous example can be adapted to use dependences instead of splitting phases with a taskwait. Figure 3 shows the correspondent Task Dependence Graph (TDG).

```
#pragma omp task out(X)
MPI_Recv(X);

for (;;;) {
   #pragma omp task concurrent(X)
   for (;;;) {
     compute (X[i][j]);
   }
}

#pragma omp task in(X)
MPI_Send(X);
```

Code 8: Synchronizing communication tasks with dependences

In general using dependences will increase the locality of data, since the runtime's implementation of the dependence system is strongly related to the task scheduler, and so that it can execute future tasks working with the same data in the same CPUs (or CPUs close to) where previous ones using this data were executed. This behavior will exploit the memory hierarchy of the system.

Dependence synchronization can be used to implement a producer-consumer approach, as we have seen so far in this section, but can also be used to ensure the correct use (actually reuse) of the memory buffers. An example is shown in Code 9. Here we see that, although the loop in the code re-initializes the memory buffer for future use, this phase cannot begin until the sending has been guaranteed, and we also need a synchronization mechanism between these two phases. In the next chapter we will see that there are other techniques that could break this dependence by temporarily copying the buffer into the data environment of the task (using the `firstprivate` clause). This latter approach can be seen in following section (Code 11).

```
#pragma omp task shared(A) input(A)
{
   int dst = (world_rank + 1) % world_size;
   MPI_Send( A, SIZE, MPI_INT, dst, 0, MPI_COMM_WORLD );
}

[...]

#pragma omp task output(A)
   for (i=0;i<SIZE;i++) A[i] = 0;
```

Code 9: Dependence between MPI send() and re-initialize buffer.

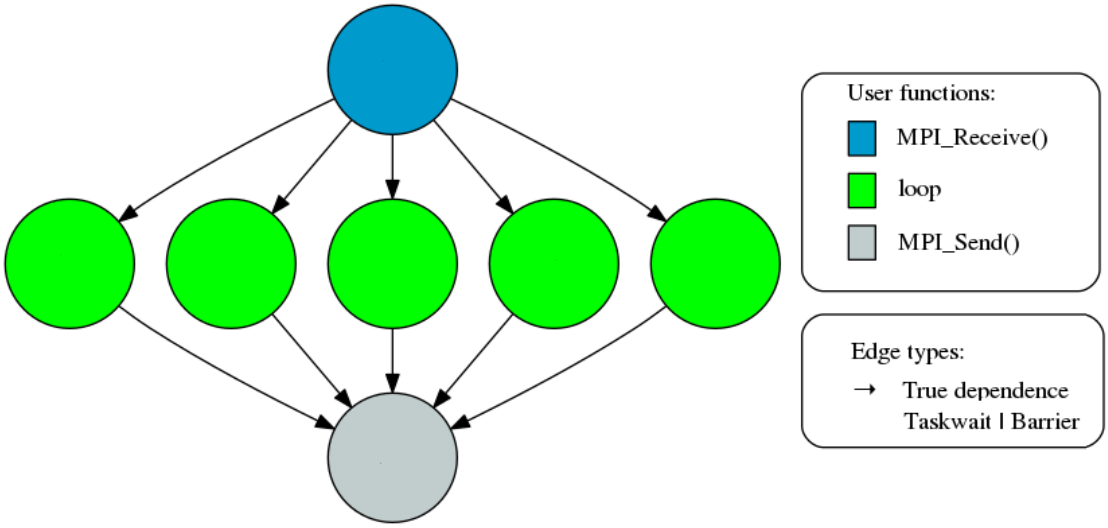

Figure 3: Task Dependence Graph (showing task order relation of Code 6).

9 Data sharing attributes and buffers

When programming with OmpSs we need to take into account how data sharing attributes may impact on MPI structures. On one hand, we may let buffers go out of scope that we need to maintain before the corresponding MPI communication has finalized, which will lead to an error. In other cases, we can leverage the privatization of certain variables to create an intermediate copy of a buffer, thus releasing this buffer from potential dependences. In the following examples we will see how data sharing attributes can be used in these cases.

The first case shows how an undesired privatization can makes our program incorrect. In Code 10 we (unintentionally) privatize variable `request`. As the `MPI_ISend()` call is working with a different copy than the one declared in the code, the following `MPI_Wait()` call will not be able to match the synchronous wait with the proper operation.

```
   MPI_Request request;

#pragma omp task firsprivate(A)
{
   int dst = (world_rank + 1) % world_size;
   MPI_Isend( A, SIZE, MPI_INT, dst, 0, MPI_COMM_WORLD, &request );
}

[...]

   MPI_Status status;
   MPI_Wait(&request,&status);
```

Code 10: Default data-sharing attribute rules may cause undesired behavior

In a second case we have a task in charge of sending a buffer to another process. At some point, the program reuses the buffer (initializing it again). The idea is to prevent sending an incorrect version of the buffer and to honor the actual WaR (Write after Read) dependence. Two different solutions can be used. The first one is to synchronize task execution (by forcing the task to finish before re-initializing the buffer) using dependences. This approach has been explained in a previous section. The second solution will be to use task's data-sharing attributes in order to copy the contents of the buffer before sending it. In the following example we can see how the programmer can duplicate the buffer by using the `firstprivate(A)` clause. This approach is shown in Code 11. The explicit task uses this clause in order to privatize the contents of the array `A`, allowing the next loop to start its execution before waiting for the send operation.

Programmers should use this technique with caution as big buffers may result in a huge overhead when duplicating them. There is no written rule about the maximum buffer size that is worth to privatizing in order to allow the scheduler to advance the execution of the following task: this value will be application dependent.

In general is always a good idea to double check the resulting data-sharing attributes of all the variables used within the task region. A good practice is to use the OmpSs clause default set to none. With this option the compiler will require that each variable that is referenced in the task must have its data-sharing attribute explicitly determined (i.e. must appear in one of the data-sharing attribute clauses).

```
   MPI_Request request;

#pragma omp task firsprivate(A) shared(request) output(request)
{
   int dst = (world_rank + 1) % world_size;
   MPI_Isend( A, SIZE, MPI_INT, dst, 0, MPI_COMM_WORLD, &request );
}

[...]

   MPI_Status status;

   // here was the original wait:
   // MPI_Wait(request,&status);

#pragma omp for
   for (i=0;i<SIZE;i++) A[i] = 0;

#pragma omp taskwait on(request)
   MPI_Wait(&request,&status);
```

Code 11: Privatizing buffers to break dependences (firstprivate)

10 References

[1] MPI Documents. Accessed on March 21st, 2017. Available at: https://www.mpi-forum.org/docs/

[2] Barcelona Supercomputing Center. OmpSs Specification. Accessed on March 23rd, 2017. Available at: https://pm.bsc.es/ompss-docs/spec/

[3] Best Practice Guide to Hybrid MPI + OpenMP Programming. Milestone of INTERTWinE project.

[4] OpenMP Specifications. Accessed on March 21st, 2017. Available at: http://www.openmp.org/specifications/

[5] Open MPI web site. mpirun man page. Accessed on March 21st, Available at: https://www.open-mpi.org/doc/v2.0/man1/mpirun.1.php

Annex A. Summary of possible problems

Problem	Possible cause	Solution
Program reports wrong results, is unstable or crashes.	Initialization does not have the proper multi-threaded level. Multiple threads are sending messages when MPI multi-thread level is single. Multiple threads are sending data simultaneously when level is serialized, etc.	Verify multi-thread level when initializing MPI library.
Segmentation fault when executing a MPI_Recv(). The MPI library points out that message is larger than receive buffer.	Tags are not properly set and messages do not match the order	Generate different tag values for messages in order to guarantee message matching.
Segmentation fault when executing a MPI_Send()	Buffer address is not correct in a communication task	Verify data-sharing attributes of buffer pointers. The task-based programming model may privatize a pointer within the context of the task and use it in an uninitialized state. If buffer is placed in the stack, make sure it is not *unwound* due to the parent task being finished (use taskwait at the end of the function).
Received message contains garbage or out of date values	Send buffer has not being properly synchronized and has been updated/changed before sent	Verify synchronization among tasks
	Send buffer has the wrong data sharing attribute. It may be private when it should be shared, or shared when it should be *firstprivate* (capture contents)	Verify data sharing attributes of communication tasks
Performance degradation or high variability.	Lack of parallelism.	Application needs to expose more parallelism: reduce the granularity, overlap communication and computation, use another data decomposition, etc.
	Thread oversubscription	Verify amount of computational resources: MPI process number and CPU binding. Verify task-based runtime system binding mechanism.
	Message unusually delayed before sending in a communication task	Verify data sharing attributes of this communication task. Is it *firstprivatizing* (capturing contents) of a huge buffer? Large buffers are usually shared due the cost of doing extra copies.
Program hangs	Send and receive tags may not match	Verify message tags