Performance Meets Programmabilty: Enabling Native Python MPI Tasks In PyCOMPSs

Hatem Elshazly
Department of Computer Sciences
Barcelona Supercomputing Center (BSC)
Barcelona, Spain
hatem.elshazly@bsc.es

Francesc Lordan
Department of Computer Sciences
Barcelona Supercomputing Center (BSC)
Barcelona, Spain
francesc.lordan@bsc.es

Jorge Ejarque
Department of Computer Sciences
Barcelona Supercomputing Center (BSC)
Barcelona, Spain
jorge.ejarque@bsc.es

Rosa M. Badia
Department of Computer Sciences
Barcelona Supercomputing Center (BSC)
Barcelona, Spain
rosa.m.badia@bsc.es

***Abstract*—The increasing complexity of modern and future computing systems makes it challenging to develop applications that aim for maximum performance. Hybrid parallel programming models offer new ways to exploit the capabilities of the underlying infrastructure. However, the performance gain is sometimes accompanied by increased programming complexity. We introduce an extension to PyCOMPSs, a high-level task-based parallel programming model for Python applications, to support tasks that use MPI natively as part of the task model. Without compromising application's programmability, using *Native MPI* tasks in PyCOMPSs offers up to 3x improvement in total performance for compute intensive applications and up to 1.9x improvement in total performance for I/O intensive applications over sequential implementation of the tasks.**

***Index Terms*—Hybrid Programming Models, Distributed Computing, MPI, High Performance Computing, Task-based Parallel Programming Models, Performance, Productivity**

I. Introduction

The growing complexity and heterogeneity of computing systems hinder the development of parallel applications from fully exploiting their capabilities. Application programmers have to handle the complexity of the systems to obtain performance and the code needs to be exposed to the underlying hardware details.

While MPI [1] + X (where X is another parallel programming model) has been proposed and used by the community, we propose a hybrid programming model that combines task-based model + MPI. The task-based programming model offers the necessary abstraction to simplify the application development for large scale execution, and supporting tasks that launch MPI executions enables to exploit the performance capabilities of many-core systems.

In this paper, we present an extension to PyCOMPSs framework [2], a task-based parallel programming model for the execution of Python applications. Throughout this paper, we name the tasks that natively execute MPI code as *Native MPI Tasks*, as opposed to tasks that call external MPI binaries. Having *Native MPI* tasks as part of the programming model means that in the same source file users can have two types of task: tasks that execute MPI code and other tasks that execute non-MPI code. PyCOMPSs organizes the tasks in Directed Acyclic Graph (DAG) and manages their scheduling and execution, hence users can focus only on the logic of the task.

II. Related Work

Unlike task-based parallel programming models that are characterized by fine-grained parallelism such as OpenMP [3], Intel TBB framework [4] and OmpSs [5], PyCOMPSs targets coarse-grained parallelism for execution in large compute clusters.

Other task-based parallel programming models target higher level of granularity such as Ruffus [6] and Luigi [7].However, PyCOMPSs allows more flexibility in expressing parallelism since it does not force the use of a specific parallel paradigm.

On the other hand, hybrid parallel programming models were discussed in previous works [8], [9]. A widely-used hybrid model is the one combining MPI and OpenMP, where OpenMP threads perform compute-intensive work on local data on each node whereas MPI is responsible for the communication between processes.

To the best knowledge of the authors, this is the first work to target enabling Native MPI tasks in a high-level task-based parallel programming model for large scale executions.

III. Native MPI in PyCOMPSs

Tasks are defined in PyCOMPSs by annotating application's method with Python decorators. Through the @task annotation, developers indicate that a function in the code becomes a task.

Using PyCOMPSs, a method is declared as *Native MPI* task by means of the @mpi decorator. The number of MPI processes per *Native MPI* task can be specified using @constraints decorator as shown in the sample code snippet in Figure 1.

```
@constraints(computingUnits=4)
@mpi(runner='mpirun', computingNodes=1)
@task(returns=int)
def return_ranks(random_num):
    from mpi4py import MPI
    rank = MPI.COMM_WORLD.rank
    return rank*random_num
```

Fig. 1. Simple *Native MPI* task in PyCOMPSs. return_ranks task will be executed by 4 MPI processes as specified in computingUnits on 1 node. It returns a list of each MPI rank multiplied by the random_num input value.

PyCOMPSs runtime will manage the input and output data of *Native MPI* tasks like any non-MPI task in a completely transparent manner to the user. The runtime will ensure that all the processes in the MPI environment have access to all the input data of the task. The return output of a *Native MPI* task – if any – is a list containing the output of all the MPI processes invoked for the task.

Similar to non-MPI PyCOMPSs tasks, the execution details of *Native MPI* tasks are completely abstracted from the runtime; the MPI environment is encapsulated within the *Native MPI* task that launched it. Thus, one workflow can have multiple *Native MPI* tasks, each with different configuration parameters (i.e., number of computing nodes and MPI processes) and combine them with other tasks in the task execution graph.

Special working processes called *MPI Worker* are responsible for launching the MPI environment with the configuration set by the user in the arguments of the @mpi and @constraints decorators. In addition to that, they monitor the execution of the task and send a task completion signal to the runtime in case of successful execution or an error signal in case of failed execution. Moreover, MPI workers are also responsible for communicating the inputs and the outputs of the task to the runtime in a totally transparent manner to the user.

PyCOMPSs runtime launches MPI workers for *Native MPI* tasks at the time of the task execution. For each *Native MPI* task in the application, a single MPI worker is launched exclusively for that task to manage its execution. If two *Native MPI* tasks are scheduled for execution at the same time, the runtime launches an exclusive MPI worker for each of them. Hence, each of the tasks will have its own isolated execution environment. Upon successful or failed execution of a *Native MPI* task, the runtime terminates the MPI worker. Figure 2 shows a diagram of the execution behavior when executing *Native MPI* tasks and non-MPI tasks on a 2-node infrastructure. Each node has a worker to host the execution of up to one non-MPI task. When *Native MPI* tasks need to be executed, one MPI worker per *Native MPI* task is launched to handle its execution.

IV. Evaluation

In this section, we evaluate both the productivity and performance benefits of using *Native MPI* tasks in PyCOMPSs.

Experiments were conducted on the MareNostrum4 supercomputer; which includes a set of high-memory computing nodes with 48 cores and 370 GB of memory each. Each node has a local SSD disk with 200 GB and access to General Parallel Filesystem (GPFS).

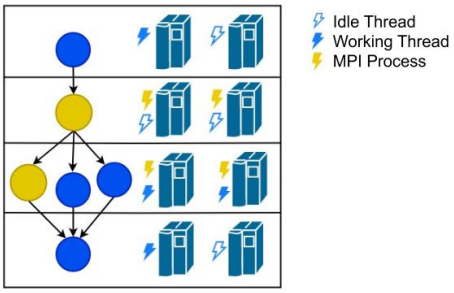


Fig. 2. PyCOMPSs Task Execution Behaviour. Non-MPI tasks (in blue) use the persistent workers of PyCOMPSs (in this case two) whereas *Native MPI* tasks (in yellow) use MPI workers for launching the specified number of MPI processes (in this case each MPI task launches 2 MPI processes).

A. Programmability Evaluation

PyCOMPSs with *Native Python MPI* combines fine-grained MPI parallelism with coarse-grained task parallelism. Since task execution is encapsulated, it allows the combination and interaction of different types of tasks (e.g. *Native Python MPI*, Binary MPI, Non-MPI) each with its own independent configuration and execution requirements. Parallelism can be controlled by simply changing the parameters of PyCOMPSs decorators. Using PyCOMPSs without *Native MPI* parallelism will only allow task parallelism but opportunities for MPI parallelism will not exploited.

On the other hand, in a pure Python approach in which MPI is used, no task parallelism is realized so the ability of executing in a distributed cluster environment is limited. In addition to that, to control parallelism, users have to use conditional statements to specify how many MPI processes will execute a block of code. Other ways to get more parallelism out of this approach is by spawning MPI processes which makes the management of this environment of nested MPI processes a complicated task.

Figure 3 presents the main function of two approaches for parallelizing a Python code. Figure 3(a) is using PyCOMPSs with *Native Python MPI* support whereas Figure 3(b) is presenting one of the ways that MPI can be used to parallelize Pure Python code. Using PyCOMPSs with *Native MPI* task support enables combining task parallelism and MPI parallelism with minimal code modifications.

B. Performance Impact

This subsection presents the performance results of two types of applications: a compute intensive application and an I/O intensive application. In both applications we targeted the task that dominates the execution time. Each experiment was run multiple times: using sequential implementation of the targeted tasks and a parallel implementation with an increasing number of MPI processes (2, 4 and 8). In all experiments, the sequential implementation of the task is used as the baseline.

```
1. if __name__ == "__main__":

2.     for i in range(NUM_RANGE):
3.         data = generate_data()
4.         data_sum = mpi_sum(data)
5.
6.         data_sum = compss_wait_on(data_sum)
7.         avg = data_sum/RANGE_LEN
8.
9.         print i, ": ", avg
```

(a) Sample Python Code Parallelized with PyCOMPSs. A set of NUM_RANGE *generate_data* tasks, each has a successor *mpi_sum* Native MPI task. More parallelism achieved with minimal code additions.

```
1. if __name__ == "__main__":

2.     for i in range(NUM_RANGE):
3.         if rank == 0:
4.             data = generate_data()
5.         else:
6.             data = None
7.
8.         data = comm.scatter(data, root=0)
9.
10.        data_sum = mpi_sum(data)
11.        comm.Barrier()
12.
13.        if rank == 0:
14.            avg = data_sum/RANGE_LEN
15.            print i, ": ", avg
```

(b) Sample Python code parallelized with MPI. No task parallelism and MPI parallelism is controlled using If conditionals and distributing the data among Processes is the responsibility of the user.

Fig. 3. main function of a Python code parallelized with PyCOMPSs Native MPI support and with MPI only.

All experiments were launched on 8 nodes of MareNostrum4 Supercomputer.

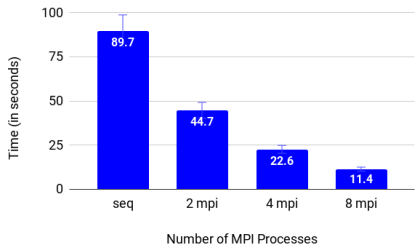
1) Performance Impact Of Parallelizing Compute Tasks: For the purpose of this test, we developed an application that calculates the term frequency (TF-IDF) of a web archive file. We used an input web archive file of a total size of 186 Gbytes The application consists of a reading task which reads a record from the file and a compute task that calculates TF-IDF. The total number of tasks for this application is 1440 tasks; 720 read tasks and 720 corresponding compute tasks.

Figure 4 shows the performance results of the application.

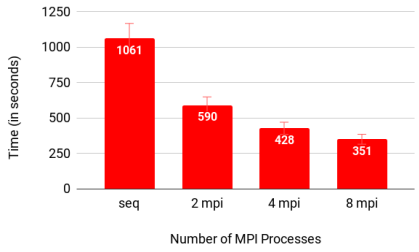
As shown in Figure 4(a) the average time per compute task decreases while increasing the number of MPI processes per compute task. Using 8 MPI processes per compute task, we obtained up to 7x speedup in the average time per compute task. In addition to that, as shown in Figure 4(b), the performance improvement per compute task is reflected as up to 3x speedup improvement in the total execution time.

2) Performance Impact Of Parallelizing I/O Tasks: For testing the performance impact of parallelizing I/O tasks with MPI using *Native MPI* tasks in PyCOMPSs, we used a checkpointing version of a block matrix multiplication application. This application has a total number of 1216 tasks; 192 tasks that generates a block of a given dimension, 512 multiply block tasks that carries out the multiplication process and 512 write tasks that writes the result of each multiplication. Figure 5 shows the average execution time per write task and the overall execution time.

As presented in Figure 5(a) the write time per task decreases when using parallel *Native Python MPI* tasks. Figure 5(a) shows up to 3x improvement in the I/O time per task when using 8 MPI process per write task. This in turn is reflected as up to 1.9x improvement in the total time as shown in Figure 5(b). As the number of MPI processes per write task increases, the load on the parallel file system increases which creates a congestion in I/O bandwidth so we do not get a linear speedup.

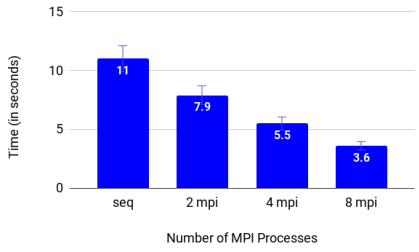


(a) Average Time Per Compute Task

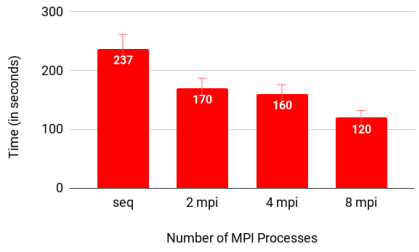


(b) Total Execution Time

Fig. 4. Performance Results for Web Archive Analysis Application



(a) Average Time Per Write Task



(b) Total Execution Time

Fig. 5. Performance Results for Checkpointing Matrix Multiplication Application

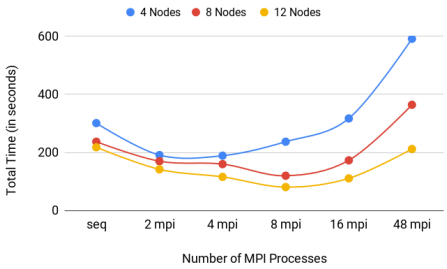
C. MPI Parallelism and Task Parallelism TradeOff

To further understand the performance and behaviour of *Native Python MPI* tasks in PyCOMPSs, several experiments were conducted on the Web Archive Analysis and the Checkpointing Matrix Multiplication applications. Every experiment is launched multiple times with a sequential implementation task and then a parallel *Native MPI* task implementation with different numbers of MPI processes (2, 4, 8, 16 and 48) on different number of nodes (4, 8 and 12). Figure 6(a) presents the results of the Checkpointing Matrix Multiplication application and Figure 6(b) presents the results of the Web Archive Analysis application.

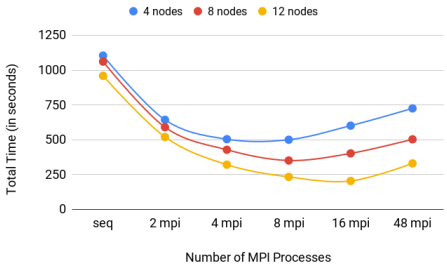
As shown in Figure 6, as the number of nodes increases, task parallelism increases so the total execution time of both applications improves.

For a specific number of nodes, total execution time decreases until it reaches a point after which it starts to increase as the number of MPI processes per *Native MPI* task increases. For the checkpointing matrix multiplication application in figure 6(a), this point is 4 MPI processes for 4 nodes and 8 MPI processes for 8 and 12 node. For the Web Archive Analysis application in figure 6(b) this point is 8 MPI processes for 4, 8 nodes and 16 MPI processes for 12 nodes. This is because *Native Python MPI* tasks use the @constraint decorator of PyCOMPSs to specify the number of MPI processes per task. Increasing the number of MPI processes per task (i.e. increasing task constraints) decreases task parallelism.

This effect is mitigated as the number of resources increases because there are enough resources to maintain the same level or allow for more task parallelism. This can be noted in Figure 6(a), for 4 nodes the performance degrades when more than 4 MPI processes per task are requested. When the number of execution resources increase to 8 and 12 nodes, the total execution time starts degrading at a later point when more than 8 MPI processes per task are requested. The same can be noted in Figure 6(b) where for 4 and 8 nodes the total execution time degrades at 8 MPI processes but when the number of nodes is increased to 12, this point shifts to 16 MPI processes.



(a) Matrix Multiplication Application



(b) Web Archive Analysis Application

Fig. 6. Scalability Results

V. Conclusion and Future Work

Enabling the execution of MPI code natively in PyCOMPSs tasks offers great benefits in terms of both programmability and performance for Python applications. However, a tradeoff arises between MPI parallelism per task and task parallelism that may negatively affect the total time of the application. As future work, we plan to improve the scheduling of tasks to better utilize the underlying infrastructure.

Acknowledges

This work is partially supported by the European Union through the Horizon 2020 research and innovation programme under contracts 721865 (EXPERTISE Project) and 800898 (ExaQUte project), by the Spanish Government (TIN2015-65316-P) and the Generalitat de Catalunya (contract 2014-SGR-1051).

References

[1] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: portable parallel programming with the message-passing interface*, vol. 1. MIT press, 1999.

[2] E. Tejedor, Y. Becerra, G. Alomar, A. Queralt, R. M. Badia, J. Torres, T. Cortes, and J. Labarta, "Pycompss: Parallel computational workflows in python," *International Journal of High Performance Computing Applications*, 2015.

[3] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[4] J. Reinders, "Intel threading building blocks - outfitting c++ for multi-core processor parallelism," 2007.

[5] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: a proposal for programming heterogeneous multi-core architectures.," *Parallel Processing Letters*, vol. 21, pp. 173–193, 06 2011.

[6] L. Goodstadt, "Ruffus: a lightweight python library for computational pipelines," *Bioinformatics*, vol. 26, no. 21, pp. 2778–2779, 2010. Exported from https://app.dimensions.ai on 2019/02/23.

[7] "spotify/luigi," tech. rep., 2016.

[8] R. Rabenseifner, G. Hager, and G. Jost, "Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes," PDP '09, 2009.

[9] J. Diaz, C. Muñoz-Caro, and A. Niño, "A survey of parallel programming models and tools in the multi and many-core era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, pp. 1369–1386, Aug 2012.