Optimizing Execution on Large-scale Infrastructures by Integrating Task-based workflows and MPI

Hatem Elshazly*†,Francesc Lordan*†,Jorge Ejarque*†, Rosa M. Badia*†
*Barcelona Supercomputing Center, Barcelona, Spain
†Universitat Politècnica de Catalunya, Barcelona, Spain
E-mail: {hatem.elshazly, francesc.lordan, jorge.ejarque, rosa.m.badia}@bsc.es

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

I. Extended Abstract

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. Task-based workflows offer 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. Hence, application programmers can get the maximum performance out of the underlying systems without compromising the programmability of the application.

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.

A. 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. Following the same approach, 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.

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.

```
@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.

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.

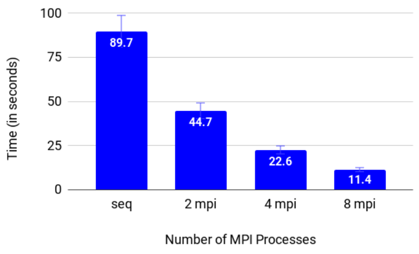
PyCOMPSs runtime launches special Python worker processes for *Native MPI* tasks at the time of the task execution to launch the MPI environment and manage the task execution. If two *Native MPI* tasks are scheduled for execution at the same time, the runtime launches an exclusive MPI worker for each of them. Hence, each of the tasks will have its own isolated execution environment.

B. Evaluation

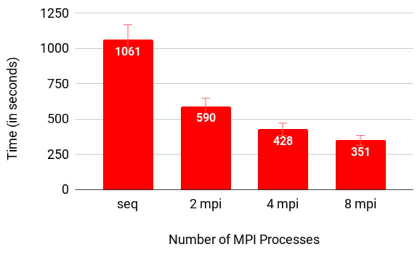
In this section, we evaluate performance benefits and trade-offs of using *Native MPI* tasks in PyCOMPSs. Experiments were conducted on the MareNostrum 4 supercomputer; which includes a set of high-memory computing nodes with 48 cores and 370 GB of memory each. Each experiment was run multiple times: 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.

For the purpose of this evaluation, 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 2 shows the performance results of the application. As shown in Figure 2(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 2(b), the performance improvement per compute task is reflected as up to 3x speedup improvement in the total execution time.



(a) Average Time Per Compute Task



(b) Total Execution Time

Fig. 2. Performance Results for Web Archive Analysis Application

To further understand the performance and behaviour of *Native Python MPI* tasks in PyCOMPSs, several experiments were conducted on the Web Archive Analysis. Each experiment was 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).

As shown in Figure 3, 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. 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 3 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.

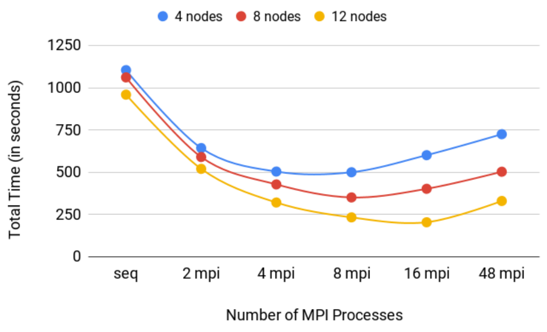


Fig. 3. Scalability Results

C. Conclusion

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.

II. Acknowledgment

This work has been published in the 28th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP), 2020 [3].

References

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

[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] H. Elshazly, F. Lordan, J. Ejarque, and R. M. Badia, “*Performance Meets Programmability: Enabling Native Python MPI In PyCOMPSs*,” in *28th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, 2020.

Hatem Elshazly received his BSc degree in Computer Science from Cairo University, Egypt in 2012. The following year, he joined as a Masters student and research software engineer at Nile University, Egypt. He completed his MSc degree in Optimizing Data Intensive Applications in 2016. Since 2018, he has been with the workflows and distributed computing group of Barcelona Supercomputing Center (BSC) as well as a PhD student at the department of computer architecture of Universitat Politècnica de Catalunya (UPC), Spain.