mpi4py.futures: MPI-based asynchronous task execution for Python

Marcin Rogowski, Samar A. Aseeri, David E. Keyes and Lisandro Dalcin

Abstract—We present mpi4py.futures, a lightweight, asynchronous task execution framework targeting the Python programming language and using the Message Passing Interface (MPI) for interprocess communication. mpi4py.futures follows the interface of the concurrent.futures package from the Python standard library and can be used as its drop-in replacement, while allowing applications to scale over multiple compute nodes. We discuss the design, implementation, and feature set of mpi4py.futures and compare its performance to other solutions on both shared and distributed memory architectures. On a shared-memory system, we show mpi4py.futures to consistently outperform Python's concurrent.futures with speedup ratios between 1.4X and 3.7X in throughput (tasks per second) and between 1.9X and 2.9X in bandwidth. On a Cray XC40 system, we compare mpi4py.futures to Dask – a well-known Python parallel computing package. Although we note more varied results, we show mpi4py.futures to outperform Dask in most scenarios.

Index Terms—MPI, Python, parallelism, master-worker, parallel programming models, distributed computing, high performance computing, task execution, multiprocessing.

1 INTRODUCTION

1.1 Master-worker pattern

Embarrassingly parallel problems are pervasive in many fields. Over the years, tasks such as text and image processing, web crawling, as well as scientific applications in areas including uncertainty quantification and Monte Carlo simulations led to a magnitude of home-grown solutions. The publication of MapReduce [1] by Google in 2008 was a notable and very successful attempt to provide an optimized and easy-to-use model supporting out-of-the-box parallelization and fault tolerance. Since then, many implementations of the MapReduce concept have emerged, Apache Spark [2] being one notable example. This wave of MapReduce-inspired libraries and their widespread adoption have since formalized the programming model and generalized the “MapReduce” term, which since then no longer refers just to the proprietary package.

1.2 Python and multicore architectures

During the same time as MapReduce became widely accepted in big data processing, Python has been gaining popularity as the general-purpose programming language. Its rise in popularity primarily powered by its ease of use, flexibility, interoperability with C/C++ and Fortran, availability on multiple platforms, and a vast set of scientific and data processing libraries culminated in October 2021, when it was ranked the most popular programming language as measured by the TIOBE Index [3]. Python’s rise in popularity is also owed to its adoption in emerging fields such as machine learning and data science, where it quickly became a dominant language thanks to packages such as TensorFlow [4], PyTorch [5], NumPy [6], Pandas [7], and Project Jupyter [8].

The rise in Python’s popularity was accompanied by a rapid increase in the number of CPU cores per socket. Being first released in 1991, like many other traditional programming languages, Python’s early design choices hindered its capability to fully capitalize on the performance provided by the emerging hardware. More specifically, Python’s global interpreter lock (GIL) present in its reference and default implementation, CPython, prevented multithreaded applications from taking full advantage of multicore architectures [9]. While GIL simplifies programming by preventing many race conditions and greatly increasing the thread safety, it effectively enforces single-threaded execution and yields no actual speedup in CPU-bound tasks. I/O-bound (whether disk or network) tasks, however, can still take advantage of using multiple threads.

One of the solutions proposed to overcome multithreading limitations was the inclusion of the multiprocessing Application Programming Interface (API) in the standard library in 2008 [10]. This interface, inspired by the threading API, allows spawning processes rather than threads. However, as when using threads, it requires explicitly launching processes, constructing special shared data structures, and waiting for completion or termination. Even though interprocess communication (message passing) happens behind the scenes, exposed low-level details may be overwhelming to domain experts who are non-traditional parallel computing users, and such a solution might pose a high barrier to entry. It can be argued that this design, while powerful, hindered the adoption of multiprocessing for parallelization of task-based applications. The space for a simple solution allowing to capitalize on the growing number of CPU cores per socket remained unfilled.
1.3 Futures
The concept of submitting a task for asynchronous execution with the ability to retrieve the result at a later time dates back at least to the 1970s. As [11] (1977) describes “futures”, they note that [12] (1976) discussed “promises” of a result while [13] (1976) talked about “events” to outline the same concept. The use of futures in favor of explicit processes or threads has been implemented in MultiLisp as early as 1985 [14] and later popularized by its inclusion in Java’s java.util.concurrent package [15]. The future construct allows for concurrent execution, as a Future object containing the promise of a result is immediately returned while the evaluation proceeds asynchronously. Once the evaluation concludes, the yielded result is stored as part of the Future object. The future can be queried at any time; however, the execution may be blocked until the value is available. Using such an approach hides cumbersome implementation details from the user while allowing to harness the benefits of multicore architectures.

Concurrent futures have been included in Python since version 3.2 (2011) and the adopted design was proposed by Python Enhancement Proposal (PEP) 3148 [16]. Heavily influenced by the Java predecessor, Python’s concurrent.futures provides two core classes: Executor and Future. The Executor class receives work requests (a callable and its arguments) and returns a Future object representing the execution of that request. Executor interface is implemented by two concrete subclasses: ProcessPoolExecutor and ThreadPoolExecutor. This modular approach allows users to benefit from all the physical cores of their modern hardware while presenting a straightforward API. The abstract interface of the Executor class allows for effortless switching between process and thread pools.

1.4 mpi4py and mpi4py.futures
In large-scale High-Performance Computing (HPC), the Message Passing Interface (MPI) remains a dominant paradigm. While typically paired with compiled programming languages such as C/C++ or Fortran, MPI’s usage in conjunction with Python is gaining popularity. One such solution is mpi4py [17], [18] which has been in continuous development since 2004. It has since been coupled with packages such as ParaView [19] or yt [20] and has become one of the most commonly used ways of using MPI within Python codes.

Even though the master-worker pattern is generally well-understood and addressed using the technologies mentioned above, the implementation on HPC clusters and using MPI in particular still poses some difficulties. It is especially challenging for users with limited exposure to parallel programming. In fact, the topic has been a recurrent matter of consultation for mpi4py developers. To address the apparent community need and following the same motivation that led to Python’s concurrent.futures – encapsulation of implementation details - mpi4py.futures module was developed. MPIPoolExecutor was added as a subclass of the abstract concurrent.futures.Executor class and the interface defined by PEP 3148 [16] was implemented. As a result of this design, mpi4py.futures can be used as a drop-in replacement for Python’s concurrent futures. No knowledge of MPI is assumed as the user does not use any MPI function calls explicitly. All the parallelization takes place behind the scenes, and the data movement is transparent from the user’s perspective. The mpi4py.futures module is available as part of mpi4py since release 3.0.

Significant differences stemming from the respective implementations - MPI and Python’s multiprocessing module respectively – should be pointed out here. Python’s concurrent.futures module is based on the multiprocessing module, i.e., uses standard capabilities of the operating system to create and manage new processes and to communicate between them. As a part of the Python standard library, it brings no external dependencies. However, the limitation is being constrained to shared memory (single node). On the other hand, mpi4py.futures relies on MPI as a backend for both process spawning and interprocess communication. Hence, it inherits naturally the capability to scale to multiple compute nodes. We do not consider depending on MPI a hindrance to mpi4py.futures’ adoption, as we target users of cloud-based platforms and pre-configured clusters, mostly at supercomputing centers, both of which typically provide an optimized MPI distribution.

Since the mpi4py.futures module is built on top of MPI, which has emphasized performance since its initial release in 1994, the performance we obtained in terms of throughput and bandwidth is typically better than Python’s concurrent.futures for single-node applications. For multi-node benchmarks, we compare our solution to Dask. While there are scenarios where Dask performs better, generally mpi4py.futures performs on-par or better.

The rest of this paper is structured as follows: first, we introduce related work (Section 2), where we discuss alternative packages allowing for asynchronous task execution. There, we emphasize the trade-offs of mpi4py.futures PEP 3148-compliant design and simplicity as compared to more feature-rich counterparts. In Section 3 we introduce details of mpi4py’s implementation pertaining to mpi4py.futures. Then, we talk about the frontend and backend implementations of mpi4py.futures as well as a few unique characteristics of our package. Section 4 contains a code snippet, using which we show the ease of parallelization of Python’s concurrent.futures code with mpi4py.futures. After that, we compare the performance of mpi4py.futures to that of Python’s concurrent.futures on a single node and to Dask on multiple nodes (Section 5). In Section 7, we shortly introduce a few applications that internally rely on mpi4py.futures. The paper ends with the conclusions (Section 8).

2 Related work
Our implementation of asynchronous task execution is by no means the only Python package with such capabilities. Other very capable libraries exist; however, many of them might seem heavyweight for a user pursuing a simple solution such as parallelization of an embarrassingly parallel workload. It is essential to reiterate the goal of mpi4py.futures as we describe similar packages – we aim to provide a performant drop-in replacement for...
Python’s `concurrent.futures` module with the additional capability of parallelization over multiple nodes while following the API defined by PEP 3148 [16]. Users with complex applications—especially those who wish to create task-dependency graphs, have specific scheduling needs, or require fault tolerance—are encouraged to evaluate alternative solutions instead.

Dask [21] is one of the more popular packages targeting parallel computation in Python. Dask authors acknowledge the high cost of rewriting software to target parallel platforms, and thus they created distributed versions of widely used Python data structures such as a `dask.array` (a parallel NumPy `ndarray`), and `dask.dataframe` (a parallel Pandas `dataframe`). In addition to those high-level structures, Dask offers a Futures module extending `concurrent.futures` as does `mpi4py.futures`. Combined with dynamic task scheduling and support for task graphs, Dask allows for efficient parallel execution of Python codes while keeping the barrier to entry low.

PyCOMPSs [22], a Python extension of COMPSs [23], also facilitates the development of parallel workflows. PyCOMPSs uses decorators to annotate functions and define data dependencies between them, which are then used to build a task-dependency graph and spawn tasks to available resources. The focus is put on detecting data dependencies between tasks and exploiting the inherent concurrency of a complex workflow as much as possible.

Like PyCOMPSs, Celery [24] uses decorators to declare tasks and uses its own API. Many protocols, such as RabbitMQ and Redis, can be used for communication between the brokers and workers. Celery focuses its queueing solution on real-time processing and is primarily used for web applications.

Few packages extending PEP 3148 [16] API to clusters exist. SCOOP [25] addresses multi-node clusters as well as standalone systems by using ZeroMQ as a communication backend. On HPC systems, ZeroMQ uses TCP over InfiniBand for transferring data. Additionally, SCOOP can be configured to use RDMA-accelerated sockets. Another full-featured package is torcpy [26], a Python implementation of the TORC C++ runtime library [27], which is built on top of `mpi4py`. Its rich API, while extensive and PEP 3148–like, does not fully conform to the specification, and as such, torcpy can be used as an easy, but not a drop-in replacement for Python’s `concurrent.futures`. However, torcpy does provide several extensions to the standard API and allows for nested parallelism as well as using legacy MPI code within the task functions.

While Python packages providing multiple approaches to utilize multicore and multi-node architectures exist, the `mpi4py.futures` package is positioned uniquely. As it is an extension of a well-known package with no additional dependencies, users can instantly benefit from the added functionality. Thanks to its fully-conformant PEP 3148 interface, `mpi4py.futures` can be used as a drop-in replacement for the Python standard library `concurrent.futures`, making multinode parallelization effortless. Despite the lack of advanced features in `mpi4py.futures`, its simplicity and interoperability allow for a low-overhead implementation that other libraries and applications can build and rely upon. One such example can be found in Dask since version 2021.03.1, where `concurrent.futures.Executor` subclasses, e.g., executors such as `MPIPoolExecutor` from `mpi4py.futures`, are allowed to be used in conjunction with the Dask scheduler.

3 IMPLEMENTATION

3.1 mpi4py

`mpi4py` strives to provide feature-complete Python to MPI bindings while giving the user as much as possible of MPI interface whilst staying close to the well-known MPI syntax and semantics. In this section, we discuss a few particularities of the core `mpi4py` implementation that the `mpi4py.futures` module builds upon.

For MPI communication, `mpi4py` can operate in two distinct modes: buffer-based and pickle-based. Buffer-based communication is close to that known from C/C++/Fortran—memory buffers and their respective sizes and data types are forwarded to MPI communication routines. This mode of operation ensures low overhead and close-to-native MPI performance. On the other hand, pickle-based communication is simpler and more convenient, although it involves additional overhead. Generic Python objects are transparently serialized and deserialized via standard Python mechanisms before and after MPI communication, allowing communication with point-to-point and collective MPI routines.

The original implementation of the Python `pickle` module required multiple, redundant memory copies. This was not an issue at the time, as the primary motivation was persistence via high-latency on-disk storage. Over time, `pickle` serialization became an essential component in applications requiring fast interprocess and over-the-network communication. These use cases motivated the proposal of the pickle protocol 5 formalized in PEP 574 [28], which allowed for out-of-band handling of memory buffers, thus minimizing redundant memory copies.

`mpi4py` release 3.1 added preliminary support for pickle protocol 5 with out-of-band buffer handling. In its implementation, each individual buffer is communicated with a separate MPI message, thus increasing latency. However, the benefits of avoiding memory allocations and copies are expected to outweigh the additional overhead for large enough buffers.

The `mpi4py.futures` module uses pickle-based communication and supports both in-band and out-of-band buffer handling.

3.2 mpi4py.futures - PEP 3148 API

3.2.1 Frontend interface

The focus of `mpi4py.futures` is to follow closely the interface of `concurrent.futures` as defined by PEP 3148 [16] by providing an `MPIPoolExecutor` class (subclass of `abstract concurrent.futures.Executor` class) a drop-in replacement for the `ProcessPoolExecutor` class provided by the standard `concurrent.futures` package. The same established interface of using `submit()` and `map()` methods for task scheduling is used. Synchronization is achieved using `wait()` and `as_completed()` calls, while the results can be retrieved using the `result()` method on a `Future` object.
3.2.2 Backend implementation

mpi4py.futures uses MPI-2 features to create worker processes and the standard MPI communication mechanisms for interprocess communication between the master process and the workers. As an MPIPoolExecutor is instantiated, and the first task is submitted, the main thread creates an ancillary manager thread, which is in charge of handling workers, task queueing and work distribution. The manager thread uses MPI-2 dynamic process management, namely MPI_Comm_Spawn, to create new MPI processes. By default, the new MPI processes run the same Python interpreter as the master, and they enter a continuous loop waiting for tasks to arrive, processing them on arrival, and sending back the results. The communication takes place via an MPI intercommunication connecting two groups – the master process on one side, and all the workers on the other. The hierarchy of processes and the MPI communicator is sketched in Fig. 1.

Once the MPIPoolExecutor is instantiated, tasks can be submitted. Tasks are simply callable functions and their arguments. A user submit tasks from the main thread, which does not participate in the computations but only feeds the executor with tasks by placing them in a queue. The manager thread continuously tries to retrieve a task from the queue and assign it to any worker. Internally, idle workers are also managed using a queue. This can be thought of as a form of dynamic scheduling, as workers are assigned tasks as they become available. Once a task is assigned, a Future object holding a reference to the task is returned to the user. The task is sent to the assigned worker using pickle-based point-to-point MPI communication. All the scheduling, task management and MPI communication happen in the manager thread, hence from the user’s point of view, all those operations happen asynchronously. The assigned worker eventually receives its task, executes it by calling the provided function with given arguments, and returns the result (or exception in case of error). The manager thread eventually receives the result and places it back into the Future object, where it can be accessed by the user from the main thread. This workflow is illustrated in Fig. 2.

3.3 mpi4py.futures extensions

While mpi4py.futures module aims to be fully compliant with the API of concurrent.futures, it includes a few extensions:

- The executor, MPIPoolExecutor, accepts an additional backoff parameter. Its default value, \(10^3\) seconds, prevents the “busy wait” in loops exchanging messages. Setting it to 0.0 seconds may slightly improve the performance, however, at the cost of decreased power efficiency. This parameter can also be controlled by setting the MPI4PY_FUTURES__BACKOFF environment variable.

- Pickle 5 out-of-band mode can be used for communication between the master and workers. This can be achieved using a combination of use_pkl5 parameter of the MPIPoolExecutor and the MPI4PY__PICKLE_THRESHOLD environment variable defining the minimum message size for out-of-band transfers. Using pickle 5 out-of-band mode can improve the performance of sending large messages and is a subject of the analysis presented in this paper.

- As workers are spawned independently via MPI dynamic process management, they may run a different Python interpreter (or even a different Python implementation) than the master process. Similarly, environment variables such as PYTHONPATH can be changed if needed.

- By default, map() returns an iterator yielding results in-order. When the keyword argument, unordered=True is passed to the MPIPoolExecutor, results are yielded as soon as a task completes, regardless of submission order. This may be especially important in applications such as statistical analysis of simulations exploring parameter space where the execution time is unknown or unpredictable. Once enough tasks to make an informed
decision is finished, the remaining tasks can be canceled as the executor is shutting down (invoking `shutdown(cancel_futures=True)`).

### 3.4 MPI-1 legacy mode

The legacy MPI-1 implementation and even some of the more recent MPI implementations do not support dynamic process management features of MPI-2. Additionally, the job schedulers used in some HPC facilities may impose restrictions on MPI process spawning at runtime. As MPI-based workers are central to the design of `mpi4py.futures`, this issue had to be circumvented. To achieve this, `mpi4py.futures` includes support for running using MPI-1 features only. To use this mode, all the workers can be spawned at the beginning of the execution, i.e., using `mpiexec -n NWORKERS` command. The MPI world communicator is then split into a master process, and many worker processes and the same configuration as described in Section 3.2 is obtained. As an implication of using this mode, the number of workers cannot be adjusted during the runtime; however, all other features remain available.

### 4 Usage example

To illustrate the changes required to parallelize the code using the `ProcessPoolExecutor` with the MPI4PoolExecutor, we use an example adapted from the Python `concurrent.futures` documentation [29].

```python
# from concurrent.futures import \
# ProcessPoolExecutor as Executor
from mpi4py.futures import \
MPIPoolExecutor as Executor
import math

PRIMES = [112272535095293, 112582705942171, 
115797848077099, 1099726899285419]

def is_prime(n):
    if n < 2: return False
    if n == 2: return True
    if n % 2 == 0: return False
    sqrt_n = int(math.floor(math.sqrt(n)))
    for i in range(3, sqrt_n + 1, 2):
        if n % i == 0: return False
    return True

def main():
    with Executor() as executor:
        return True

if __name__ == "__main__":
    main()
```

In order to convert code to use `mpi4py.futures`, only one change is required – `ProcessPoolExecutor()` is no longer imported as `Executor` in favor of `MPIPoolExecutor()` (see lines 1–4). Alternatively, `MPIPoolExecutor` could be used explicitly in the `with` statement in line 23. As a result of either change, tasks – `is_prime()` function calls with arguments from the `PRIMES` array – will be distributed over MPI workers which may now be placed on separate nodes. This simple example does not highlight the benefits of `concurrent.futures` and `mpi4py.futures` sufficiently. Rather than a simple integer, the arguments can be complex Python data structures, such as NumPy arrays or objects of user-defined classes, which will then be serialized using pickle and communicated using the multiprocessing.Queue (`concurrent.futures`) or MPI `Isend/Irecv` (`mpi4py.futures`).

### 5 Performance

#### 5.1 Experimental setup

For the purposes of evaluating the performance of `mpi4py.futures`, we compare it to `concurrent.futures` on a workstation and to Dask (`dask.distributed`) on a Cray XC40 supercomputer. Our benchmarks are designed to measure the throughput, i.e., the number of tasks executed per second, and the bandwidth, i.e., the communication speed between the master process and workers.

We have not evaluated the startup time of the different solutions, as we do not consider it to be a critical metric for a typical production application of tasking frameworks. However, from our experience, the cost is dominated by the mechanism used to create new worker processes. The `concurrent.futures` package, being limited to shared-memory systems, takes advantage of the fast copy-on-write semantics of the `fork()` system call on POSIX platforms. Multi-node solution using `mpi4py.futures` involves a more expensive process creation mechanism.

Each of our experiments comprises of a Python script submitting tasks to an `Executor`. Tasks are Python code with no dependencies or communication between them. The function run by the tasks is trivial: it accepts a single argument and immediately returns it to the caller. The argument is either `None` for the throughput experiment or a NumPy array of a given length in the bandwidth tests. Batching was not used, i.e., the chunk size (the `chunksize` argument of the `map()` method) was set to 1.

The startup time of spawning workers has been excluded. In single-node experiments, 26 iterations of each test have been performed, with the timing of the first iteration explicitly omitted to ignore any delayed setup costs. The values reported are based on the arithmetic average of the remaining 25 measurements. In multi-node experiments, the timings are based on the arithmetic average of 100 measurements. Those measurements were performed with 11 iterations (the first iteration is discarded) executed in 10 separate jobs submitted via SLURM [1]. The design of this experimental setup accommodates for run-to-run variability often seen on supercomputers using the Dragonfly network topology [30].

1. We made two practical exceptions to this. In the throughput experiment, due to Dask’s performance issues when submitting many tasks, we used only 1 job with 11 repetitions for the scenarios with 2,048 workers and 32,768, 65,536, 131,072 and 262,144 tasks. This was done for purely pragmatic reasons, as the cost of running underperforming jobs on 2,050 compute nodes became prohibitively expensive without providing any further insights. In the bandwidth experiment, the Dask tests with 10 MB messages consistently failed when using 11 repetitions. Instead, we timed different jobs with 3 repetitions each with the first timing discarded.
Initial single-node experiments performed using simultaneous multithreading (SMT) showed performance lower than that achieved using only physical cores. Hence, in the reported results, the number of workers per node never exceeds the number of available physical CPU cores. In multi-node experiments, each node was assigned only one process in order to stress internode communication and factor out any intranode shared-memory effects. In mpi4py.futures experiments, the total number of nodes used is equal to the number of workers + 1 to account for the master process. As in Dask the scheduler is separate from a client, both the scheduler and the client were assigned separate nodes, for a total of number of workers + 2. Dask workers were set to use 1 thread only. In the mpi4py.futures configuration, the backoff parameter was set to 0.0 seconds, rather than the default value of 10^{-3} seconds. This results in a more efficient “busy wait” behavior at the cost of energy consumption. The backoff parameter is one of the mpi4py.futures-specific features described earlier in Section 5.5.

mpi4py main development branch (commit b971ddf) was used for all the experiments. Packages Dask 2022.9.0, Dask.distributed 2022.9.0, and Dask-MPI 2022.4.0 were installed. In single-node experiments, Python 3.10.6 and MPICH 3.4.3 were used. Both packages were those bundled with Fedora 36. In multi-node experiments, Python 3.10.1 and Cray MPICH 7.7.18 modules were employed.

Throughout this paper, we use units following the IEC 60027-2 standard [31], i.e., 1 gigabyte = 1 GB = 10^{10} MB = 1,000 MB, while 1 gigabyte = 1 GiB = 2^{10} MiB = 1,024 MiB.

### 5.1.2 Hardware description

For single-node runs, a workstation with AMD Ryzen Threadripper PRO 3995WX and 512 GiB of RAM was used. For multi-node experiments, Shaheen II, a Cray XC40 supercomputer hosted by King Abdullah University of Science and Technology (KAUST) was used [32]. Each of Shaheen’s 6,174 nodes contains two Intel Haswell (Xeon E5-2698v3) CPUs with 16 cores each and 128 GiB of memory. Nodes are connected using a Cray Aries interconnect with Dragonfly topology. Nodes were assigned by the scheduler in an exclusive mode, but as in a typical production environment, the network was shared with other users. Even though it is difficult to predict the performance of the Aries interconnect due to its multi-level topology and adaptive routing, we consider the node-to-router PCIe connection to be the most limiting in a master-worker setting. In our testing stressing this connection, the MPI-1 ping-pong test from the Intel MPI Benchmark [33] achieves a sustained bandwidth of 8.5 GB/s. This figure is consistent with the results reported for Cray XC series [34].

### 5.2 Single-node throughput

In the single-node throughput experiment, we compare the performance of Python’s concurrent.futures (ProcessPoolExecutor) to mpi4py.futures (MPIPoolExecutor) using MPI dynamic process management. We measure the time it takes a given number of workers to execute a specified number of trivial function evaluations accepting and returning a None object. Based on the number of tasks and the elapsed time, we report the throughput, defined as the number of tasks executed per second. In the experiments, we vary the number of workers in powers of 2, between 1 and up to 2^{16} = 64, which is the number of physical CPU cores of the host. We vary the number of tasks from 1 to 65,536, doubling the number for each successive experiment. After excluding the scenarios where the number of workers exceeds the number of tasks, we evaluate 98 different combinations for each concurrent.futures and mpi4py.futures.

![Fig. 3. Single-node task execution throughput using mpi4py.futures (higher is better). Values shown in cells are thousands of tasks per second. The color palette is common for Fig. 3 and Fig. 4, i.e., based on the minimum (red) and maximum (green) values regardless of the execution backend.](image-url)

![Fig. 4. Single-node task execution throughput using concurrent.futures (higher is better). Values shown in cells are thousands of tasks per second. The color palette is common for Fig. 3 and Fig. 4, i.e., based on the minimum (red) and maximum (green) values regardless of the execution backend.](image-url)
minimum (red) and maximum (green). As the heatmaps show, mpi4py.futures has a performance advantage in every scenario, with speedup ratios ranging from 1.4× to
3.7× with an average of 2.1×.

As shown in Fig. 3 the performance of mpi4py.futures is not correlated with the number of tasks. Additionally, we do not notice a significant performance degradation when increasing the number of workers. The performance is only noticeably lower when using 64 workers. However, this behavior can be partially attributed to oversubscription, as the total number of processes involved in the test exceeds the number of physical CPU cores. The average performance is 17,652 tasks per second, with a peak of 30,343 tasks per second.

Python’s concurrent.futures exhibits a different behavior (Fig. 4). Its performance oscillates around 8,000–10,000 tasks per second when using 1–16 workers. We also see a noticeable performance degradation as the number of workers increases above 16, with the average throughput of 7,726 tasks per second for 32 workers and only 5,835 tasks per second for 64 workers. As with mpi4py.futures, we do not observe a clear performance trend as the number of tasks is increased. Overall, the average throughput is 8,452 tasks per second, with a peak of 11,661 tasks per second.

5.3 Single-node bandwidth

The second experiment is designed to measure the bandwidth, i.e., the speed of data transmission between the master process and workers. For the purpose of this experiment, tasks involve the execution of a trivial function accepting and returning a NumPy array. After receiving all the results from workers, we calculate the effective aggregated bandwidth based on the total data size transferred and the elapsed time.

To limit the number of variables, we choose one combination of the number of workers and the number of tasks as we vary the NumPy array length. Based on the results shown in Fig. 3 and 4, we pick 16 workers and 8,192 tasks – a scenario in which both mpi4py.futures and Python’s concurrent.futures performed above their respective averages – as we vary the array size from 10³ to 10⁶ bytes (1 KB to 10 MB), increasing it by a factor of 10 for each successive experiment.

Both mpi4py.futures and Python’s concurrent.futures use the pickle module for serialization and deserialization of the function and its arguments. mpi4py.futures, however, also includes support for pickle protocol 5 out-of-band mode (see Section 3.3). In this section, we first evaluate the performance of mpi4py.futures vs. concurrent.futures, both using in-band buffer handling. Next, we look at the effects of using the pickle 5 out-of-band approach within mpi4py.futures.

5.3.1 mpi4py.futures vs. concurrent.futures

In this section, we analyze the bandwidth achieved using the in-band pickle-based serialization and deserialization approach. In Fig. 5 we present the results as message sizes vary between 1 KB and 10 MB. Given the number of messages exchanged, this is equivalent to transferring between ≈16 MB and ≈160 GB, respectively. In the scenarios analyzed, MPI-based communication as used by mpi4py.futures is always faster than Python’s interprocess communication mechanisms based on sockets or pipes. For a 1 KB message, the bandwidth difference is as high as 2.0×. As the message size increases and the memory bandwidth saturates, the bandwidth difference remains between 1.9× to 2.9× with both mpi4py.futures and concurrent.futures achieving their peak with 10 MB messages at 3.0 GB/s and 1.0 GB/s, respectively.

Fig. 5. Single-node bandwidth (MB/s) as the message size is increased (higher is better). Error bars show the 95% confidence interval.

5.3.2 In-band vs. out-of-band communication modes

Pickle protocol 5, as described in PEP 574 [28], allowed for out-of-band handling of memory buffers to minimize redundant memory copies. mpi4py 3.1 added preliminary support for out-of-band buffer handling, and the feature can be used within the mpi4py.futures module.

When using the out-of-band mode, each individual buffer is communicated with a separate MPI message. Therefore, we expect increased latency; however, for large buffers, the extra latency will most likely be outweighed by the benefits of reduced memory allocations and copies.

The results we present in Fig. 6 confirm our expectations. We present bandwidth as a function of message size, which we vary from 10³ bytes to 10⁶ bytes. Generally, the bandwidth achieved using out-of-band buffer handling is approximately 40% to 50% lower than that of the default in-band mode. Only once the message size reaches ≈200 KB, the performance of the out-of-band mode starts to show its benefits. As the message size further increases, the advantage of using the out-of-band mode becomes evident, and it is 1.8× higher for 10 MB messages (3.0 GB/s vs. 5.4 GB/s).

It is worth noting that mpi4py uses a threshold value for the message size to decide whether to handle buffers in-band or out-of-band. The threshold value defaults to 256 KiB and can be controlled by users by setting the MPI4PY_PICKLE_THRESHOLD environment variable (see

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.Authorized licensed use limited to: KAUST. Downloaded on November 30,2022 at 06:07:12 UTC from IEEE Xplore. Restrictions apply.
Table 1: Time required for MPI tasks to complete (in seconds) using mpi4py.futures.

<table>
<thead>
<tr>
<th>Number of workers</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (s)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>256</td>
<td>512</td>
<td>1024</td>
<td>2048</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>0.7</td>
<td>0.7</td>
<td>1.8</td>
<td>1.8</td>
<td>1.7</td>
<td>1.8</td>
<td>1.8</td>
<td>3.9</td>
<td>3.9</td>
<td>3.5</td>
</tr>
</tbody>
</table>

Section 3.3. In this experiment, we aimed to use the out-of-band mode regardless of the message size, thus we set the threshold value to 0.

5.4 Multi-node throughput

For the multi-node experiments, we reuse most of the single-node testing infrastructure. As the concurrent.futures package does not support distributed environments, we use dask.distributed instead. In particular, we use Dask’s Client class to instantiate the executor object. Rather than placing multiple workers on each node, we emphasize communication performance by assigning only one process per compute node. The total number of nodes involved is then \( n_{\text{workers}} + 2 \) for mpi4py.futures and \( n_{\text{workers}} + 2 \) for dask.distributed, as mpi4py.futures only requires a single node for the master process, and we assign separate nodes to Dask’s scheduler and client processes.

The results we obtained for mpi4py.futures are presented in Fig. 7. Throughput measured in thousands of tasks per second is reasonably consistent across all the scenarios, with the values varying between 16,532 and 27,420 tasks per second. The average throughput is 20,469 tasks per second.

The throughput of Dask, however, exhibits much higher variability. Results for the same scenarios are presented in Fig. 8. While the throughput does not seem to vary significantly with the number of workers, the total number of tasks dramatically affects the results. Throughput for 2 to 512 tasks is lower than what we observed using mpi4py.futures. Even though the difference is clear and somewhat unexpected, it is not of practical relevance for production scenarios. For 1,024 to 2,048 tasks, throughput is consistently above 20,000 tasks per second and is comparable to that of mpi4py.futures. However, as the number of tasks increases, throughput declines rapidly (see rows corresponding to 8,192 and more tasks in Fig. 8).

5.5 Multi-node bandwidth

As with the single-node bandwidth experiment, we pick a combination of number of tasks and workers based on the results of the throughput experiment. Both mpi4py.futures and dask.distributed performed near their peak with 1,024 tasks distributed over 128 workers. Therefore, we choose that combination for further experimentation by varying the message size from 2 KB to 1024 KB.

2. As of November 23, 2022 this behavior has been reported upstream to Dask developers and is under investigation [https://github.com/dask/distributed/issues/5715].
sage size between 1 KB and 10 MB. The results of this distributed experiment are shown in Fig. 9. For messages of size 1-10 KB, dask.distributed is faster than mpi4py.futures by 1.3 x to 1.4 x. However, once the message size further increases, mpi4py.futures outperforms dask.distributed with a maximum bandwidth ratio of almost 4.5 x for a 1 MB message (5.1 GB/s and 1.1 GB/s, respectively).

![Multi-node bandwidth (MB/s) as the message size is increased. Error bars show the 95% confidence interval.](image)

It should be noted here that using mpi4py.futures with pickle 5 out-of-band mode did not yield any measurable performance benefits in this scenario. This behavior is expected as, even on supercomputers, networks are significantly slower than local memory, and therefore the cost of communication is dominated by internode data transfers. The in-node savings resulting from reduced memory copies of out-of-band mode are dwarfed by the off-node network communication cost.

5.6 Additional considerations

5.6.1 Overhead in real-world workloads

The throughput tests that we presented above are admittedly synthetic and not representative of real workloads; in fact, they are just saturation tests designed to stress the task management implementation. However, the resulting metric of number of trivial tasks executed per second is very relevant to developers of tasking frameworks and it can be used to drive optimization efforts. Even though the metric might not seem to be of great importance to end users, there is a crucial correlation between the throughput and efficient use of computational resources.

Based on the saturation test, we can calculate the task execution overhead as the inverse of the throughput. This value, in turn, can be used to calculate the maximum number of workers that the framework is able to keep busy executing tasks without idle-waiting. Dividing the estimated task duration by the task execution overhead (alternatively, multiplying task duration by throughput) yields an upper bound for the number of workers that can effectively contribute to reduce the total execution wall time. For example, when using mpi4py.futures on Shaheen II (Fig. 7), the overhead is between 36 µs and 60 µs. For a large number of 0.05 second tasks, it can be expected that up to 833 workers (0.05s/60µs for the worst-case overhead) can be used efficiently. Similarly, for an average task duration of 3 seconds, the maximum number of workers increases up to 50,000.

5.6.2 Number of processes per node

As we mentioned in Section 5.1.2, we used one process per node in multi-node experiments. Our goal was to emphasize internode communication and exclude any shared-memory effects. In our testing, however, using more processes per node did not change the performance significantly, that is, the performance of mpi4py.futures does not depend on the placement of worker processes. We recommend to choose the number of workers per node to accommodate for the nature of the submitted workload. Typically, users would limit the number of processes per node below the CPU core count only in specific scenarios, such as memory-bound applications.

5.6.3 Shared memory and MPI dynamic process management

mpi4py.futures favors the use of MPI dynamic process management to spawn the worker processes at runtime upon the creation of MPIPoolExecutor objects. This mode of operation is the closest to that of Python’s concurrent.futures and makes for the most natural and fair comparison. It also fits well with many use cases such as interactive computing platforms like the popular Jupyter Notebooks. However, dynamic process management is not always the most performant option. In our testing, we found that spawning all the workers from the beginning of the execution (legacy mode, see Section 5.4) and binding processes to CPU cores increases performance substantially. In the throughput experiment, mpi4py.futures ran in legacy mode averaged at 39,255 tasks per second (2.2 x speedup) with a peak of 53,807 tasks per second (1.8 x speedup). In the bandwidth experiment ran in legacy mode, the maximum bandwidth achieved stands at 3.3 GB/s for 100 KB messages (1.7 x speedup) and 6.1 GB/s for 1,000 KB messages (2.1 x speedup).

This behavior is highly dependent on the MPI implementation. When using the reported version of MPICH, the communication between the parent and the spawned child processes is achieved through network modules. On the other hand, processes started as part of the world group (MPI_COMM_WORLD) communicate through a faster shared memory channel. As the performance optimization of MPI implementations is a continuous process, future enhancements to the MPI dynamic process management support are expected to reduce the performance gap between these two modes of execution. Note that all these details only apply to shared memory systems. In a distributed environment all the master-worker communication is channeled through the network.

6 APPLICATIONS

In Section 5, we presented synthetic experiments designed to isolate and stress different aspects of the frameworks under evaluation. Here, we present a more realistic application example showing how the throughput and bandwidth differences interplay and affect the time-to-solution. In this experiment, we use 10,000 images randomly chosen from the ImageNet dataset [35], with sizes ranging from 1 KB to 3.84 MB with an average of 116 KB and a standard deviation of 128 KB. The master process preloads the images, and the worker processes preload the pre-trained MobileNetV2 [36] TensorFlow [4] model. After these initialization steps, the master submits the images to workers for processing. The workers resize the input image to the required size; next, they perform inference using the machine learning model and finally return the predicted class of the image.

The setup is kept as close as possible as in the multi-node experiments described in Section 5.1.1. However, here we use 10 jobs submitted using SLURM with no repetitions within the job. We also use the more realistic setting of assigning one worker to each CPU core, for a total of 32 processes per compute node.

The results of our machine learning inference experiment are presented in Fig. 10. The higher task throughput and bandwidth (for large messages) of mpi4py.futures results in an overall faster time-to-solution and allows to use more workers efficiently, thus improving strong scalability.

7 APPLICATIONS

An initial version of mpi4py.futures has been first released in 2017 as part of the parent mpi4py package. Since its first release, incremental enhancements and routine software maintenance work have led to its current state and feature set as reported in this work. Even though mpi4py.futures has been in continuous development, many projects took early advantage of its capabilities. In the following, we summarize some of the most prominent application examples we found in the literature. Many other examples can be found through code search in public source code repositories.

Delta [37] is a Python framework enabling advanced data analysis for near real-time decisions in magnetic fusion experiments. The authors use mpi4py.futures to perform spectral data analysis and they report a speedup of over 100× over a single-core implementation.

DDFacet is a wide-field direction-dependent spectral deconvolution framework used to process the data originating from radio telescopes. Recently it has been parallelized using mpi4py.futures [38] with the ultimate goal of addressing the processing needs of the Square Kilometer Array (SKA). The authors benchmark their code using a dataset generated by LOFAR (an SKA precursor). They report a 5.7× speedup of their parallelization approach.

Within the field of uncertainty quantification (UQ), [39] presents a comparison of various scheduling strategies for parallel UQ scenarios using the non-intrusive polynomial chaos approach. The authors use mpi4py.futures to implement the “dynamic work package” strategy. They report such approach to decrease the overall runtime and attribute it to significantly reduced idling time and improved load balancing as compared to static scheduling.

Modin OpenMPI Compute Engine [40] is an MPI backend for Modin – a drop-in substitute for Pandas, a popular data science package. Modin allows to parallelize Pandas dataframe workloads. It is based on top of mpi4py.futures. The authors report consistent strong scaling results in almost all the tested scenarios.

Cogent3 [41], a descendant of PyCogent [42] is a Python library for the analysis of genomic sequence data. While authors focus on providing the best experience within the Jupyter Notebook environment, the algorithms in cogent3 are reported to support parallel execution on thousands of compute nodes. mpi4py.futures is used as the parallel execution backend.

The ptreeopt package [43] implements the policy tree optimization method and provides a framework for formulating policies as binary trees using a simulation-optimization approach. The parallelization is achieved using either the Python’s multiprocessing module (single-node) or mpi4py.futures (multi-node).

Other projects using mpi4py.futures in their backends include DeepHyper [44], AidData’s Global Chinese Development Finance Dataset [45], sndust [46], ParaCopasi [47], quoFEM [48], and Xopt [49].

8 CONCLUSIONS

In this work, we introduced mpi4py.futures, discussed its design, implementation, and feature set. We compared mpi4py.futures performance to competing solutions – Python standard library’s concurrent.futures on a shared-memory system, and Dask on a distributed-memory supercomputer. We showed that mpi4py.futures performs on par or better than other solutions in most scenarios. On a shared-memory system, mpi4py.futures consistently outperforms Python’s concurrent.futures with speedup ratios from 1.4× to 3.7× in throughput, from 1.9× to 2.9× in bandwidth when using the default in-band communication mode and up to 4.3× in...
out-of-band communication mode. On a Cray XC40 system, we compare mpi4py.futures to Dask—a well-known Python parallel computing package. While the performance of dask.distributed is within 15% of that of mpi4py.futures for scenarios with 256–2,048 tasks, we noted unexpected behavior in other cases (2–128, 8,192–262,144 tasks) where its throughput is as much as 1.3× up to 87.4× lower. This behavior is under investigation by the Dask developer community. We showed that the improved throughput and bandwidth of mpi4py.futures translates to better strong scalability of an application involving machine learning inference of images.

Even though mpi4py.futures is very performant in terms of raw throughput and bandwidth, we would like to reiterate that the feature set provided is limited by design: no support for advanced scheduling, task dependencies, or fault tolerance is provided. Users looking for such features are advised to research the solutions offered by other frameworks, including the possibility of combining them, such as using Dask with the mpi4py.futures executor. With these caveats and the performance characteristics we reported in mind, users should consider the peculiarities of their particular workloads (e.g., number of tasks, expected wall time, available compute resources) to make the final decision about the most appropriate solution to address their needs.

ACKNOWLEDGMENTS

The research reported in this paper was funded by King Abdullah University of Science and Technology (KAUST). We are thankful to the KAUST Supercomputing Laboratory for their computing resources. We would like to thank the Dask developer community, and especially John Kirkham, for their feedback. Some discussions in this work were inspired by a series of blog posts by Matthew Rocklin, the initial author of Dask.

REFERENCES


Marcin Rogowski is a Ph.D. student in the Extreme Computing Research Center at King Abdullah University of Science and Technology (KAUST). He received a M.S. degree from KAUST and a B.Sc. from the University of Aberdeen. Marcin worked on the performance and scalability of a massively parallel reservoir simulator during his time at Saudi Aramco from 2013 to 2019. His current research interests include communication aspects of parallel computing and high-performance computing in computational fluid dynamics.

Samar A. Aseeri has been working as a Computational Scientist at King Abdullah University of Science and Technology (KAUST) since October 2010. She received a Bachelor, Master and Ph.D. degrees in Applied Mathematics from Umm Al-Qura University in Saudi Arabia in 2001, 2005 and 2009 respectively. Before joining KAUST, she was trained in supercomputing at IBM in NY. Her present research interest is in the Fast Fourier Transform (FFT) library benchmarks and implementation. She had provided support to the Shaheen user community at the KAUST Supercomputing Laboratory (KSL) and currently focusing more on research at the KAUST Extreme Computing Research Center (ECRC).

David Keyes earned a B.S.E. in aerospace and mechanical sciences from Princeton in 1978 and a Ph.D. in applied mathematics from Harvard in 1984. He directs the Extreme Computing Research Center at KAUST. He works at the interface between parallel computing and the numerical analysis of PDEs, with a focus on scalable implicit solvers and exploiting data sparsity. He helped develop and popularize the Newton-Krylov-Schwarz (NKS) and Additive Schwarz Preconditioned Inexact Newton (ASPIN) methods. He has been awarded the ACM Gordon Bell Prize and the IEEE Sidney Fernbach Prize and is a fellow of the SIAM, AMS, and AAAS.

Lisandro Dalcin is a senior research scientist with King Abdullah University of Science and Technology (KAUST), Thuwal, Saudi Arabia. He held an adjunct research associate position with National Scientific and Technical Research Council (CONICET), Argentina, from 2010 to 2017. He received a Ph.D. degree in engineering sciences and computational mechanics from National University of the Litoral (UNL), Santa Fe, Argentina in 2008. His research interests include numerical methods in computational fluid mechanics, scalable solution methods for partial differential equations, and parallel computing on distributed memory architectures.