The Scaling of Many-Task Computing Approaches in Python on Cluster Supercomputers

Monte Lunacek
Research Computing
University of Colorado Boulder
Boulder, Colorado 80309–597
Email: monte.lunacek@colorado.edu

Jazcek Braden
Research Computing
University of Colorado Boulder
Boulder, Colorado 80309–597
Email: jazcek.braden@colorado.edu

Thomas Hauser
Research Computing
University of Colorado Boulder
Boulder, Colorado 80309–597
Email: thomas.hauser@colorado.edu

***Abstract*—We compare two packages for performing many-task computing (MTC) in Python: IPython Parallel and Celery. We describe these packages in detail and compare their features as applied to many-task computing on a cluster, including a scaling study using over 12,000 cores and several thousand tasks. We use mpi4py as a baseline for our comparisons. Our results suggest that Python is an excellent way to manage many-task computing and that no single technique is the obvious choice in every situation.**

I. Introduction

Researchers in life and social sciences are increasingly utilizing high performance compute and storage systems to advance their work. One common question that continues to surface for these scientists is: what's the most efficient way to execute thousands of independent tasks on a cluster? This body of research is called Many-Task Computing (MTC) [1], and in this paper we compare the characteristics and scaling of two different ways to address this problem using distributed Python: IPython Parallel [3] and Celery [4]. Our scaling results, using up to 12288 cores, suggest that Python is an excellent language for executing MTC tasks, and that no single technique is always the best choice.

We have recently been working with scientists using single-core software in Microbial Ecology [5], Evolutionary Biology [6], [7], Geology [8], and Geography [9] to help them efficiently execute task-based parallel jobs on a cluster. These scientists are non-traditional parallel computing users and face a high barrier to entry because the software they use for simulation is designed for single-core workstations. A strong background in scientific computing and parallel programming is necessary to transform these serial codes into efficient parallel workflows. Unfortunately, submitting thousands of individual jobs to a resource manager, such as Condor [10] or SLURM [11], is not a very efficient approach [1]. Non-traditional supercomputing users need an efficient and easy to use framework for executing many-tasks on a compute cluster.

A decade ago, distributed computing on a supercomputer usually implied using the Message Passing Interface standard (MPI) [12] with a language like C, C++, or Fortran, possibly running as a hybrid application with OpenMP [13]. Recently, as multi-core computing has becoming increasingly less obscure, new message passing standards, like the Advanced Message Queuing Protocol (AMQP) [14] and new library implementations such as ZeroMQ [15], have emerged. Often these message passing implementations include bindings to other programming languages such as Python. At the same time, languages such as Erlang [16] and Scala [17] are gaining popularity because they come with build-in parallel support, meaning that you don't need an external library, such as MPI, OpenMP, or ZeroMQ, to utilize multiple nodes or cores. The landscape is HPC changing.

These approaches give users of High Performance Computing (HPC) system several new options for distributed computing. First, Python is much more accessible and easier to learn than a traditional HPC language, making it ideal for cluster users with little or no background in programming. Second, many new parallel libraries provide a simple abstraction that allows a user to *map* a set of tasks onto a function in parallel. This is ideal for many-task computing workflows. Finally, new options for message passing, like AMQP and ZeroMQ, were designed to be fault-tolerant. This makes some jobs that are not possible with MPI fairly straight forward with a library like *IPython Parallel* or *Celery*. In short, these new libraries make many-task distributed computing easer to use and offer a different set of characteristics that are not available with MPI.

This paper is organized in the following way. In the next section, we discuss the emerging python technologies that are gaining acceptance in the HPC community, especially with non-traditional users in the many-task computing domain. Then we present and discuss the results of our scaling study. We include mpi4py [2] as a baseline for comparing the scaling of IPython Parallel and Celery. This creates boundaries around the strengths and weaknesses of these different approaches. We discuss related work before concluding the paper.

II. Python for Many-Task Computing

We define a *job* to be a set of *tasks*, where each task represents an independent process. The duration of each task and the time scale for which the job is executed determines how it is classified [1]. When the *job* uses a large number of resources over a short period of time it is classified as a MTC job. On the other hand, if the job is executed over a period of months and the primary performance metric is operations per month, then job is classified as a High Throughput Computing (HTC) job. The goal in MTC is to execute all the tasks in the most efficient way across a given resources. This problem occurs in many

978-1-4799-0898-1/13/$31.00 ©2013 IEEE

areas of science including monte-carlo simulations, parameter optimization, uncertainty quantification, and parameter scans, just to name a few.

The tasks that make up an MTC job do not necessarily need to be single-core or completely independent. The tasks may be, for example, a set of programs that use MPI across many nodes or a set of OpenMP tasks on a single node (or even both). There may also be an order that each task must execute, essentially adding a degree of dependency between the tasks.

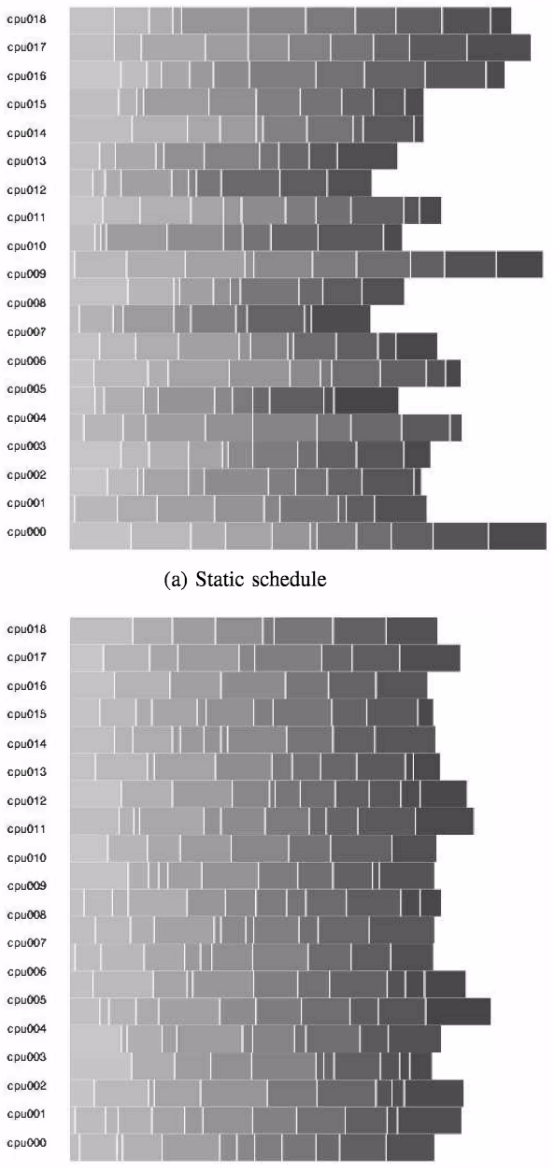
The distinction between MTC and HTC, or any other term that seems to apply here (e.g embarrassingly parallel), is not sharp. Raicu *et* al. [1] argue that the emphasis for MTC is on achieving performance in terms Float Point Operations per Seconds (FLOPS) for the entire job. In this way, we are not just concerned with measuring the performance of each task, but rather the overall performance of the job.

In this paper, we are focusing on executing many single-core tasks with no dependencies or communication between tasks. There are several ways to map a set of tasks across a given resource. The two extreme cases are: 1) every core does it's fair share of the work, which we refer to as a *static schedule*, or 2) every core is given a single task one-at-a-time, which we denote as a *dynamic schedule*. When the runtime time of each task is roughly the same, a static schedule will be as efficient and is less likely to incur a bottleneck when processing and allocating tasks. Dynamic scheduling is often more efficient when there is variation in the amount of time each task executes.

Figure 1 shows an example of a static schedule (top graph) and a dynamic schedule (bottom graph), both executing the same job, and therefore the same tasks. In this example, the dynamic schedule is able to load-balance the tasks more efficiently than the static schedule. Every worker in the static schedule also does the same number of tasks. Executing the overall job faster will obviously result in a better MTC metric of success.

Dynamic scheduling uses a broker to distribute the tasks to each working processor. Sometimes this becomes a bottleneck in the system, resulting in higher latency for task distribution, and reducing the overall efficiency of the job. One way to hide this latency is to allow each worker to queue up additional tasks that can start immediately, while in the background the broker can be processing a worker's response and scheduling additional tasks.

In this section, we describe the characteristics and configuration necessary to solve MTC problems using mpi4py, IPython Parallel, and Celery on a compute cluster. In this paper, we consider jobs comprised of completely independent tasks that are always single-core.



(a) Static schedule

(b) Dynamic schedule

Fig. 1: Static and Dynamic task distribution. The $y-$axis contains 18 cores and the $x-$axis represents time. The tasks (blocks) are arranged from lighter to darker based on the linear order in which they appear to the scheduler. Notice that in the static case, each core does the exact same number of tasks.

A. Background

The first MPI standard was created as a way to incorporate and standardize the characteristics that were being developed concurrently in many, duplicating message passing libraries. These libraries were being used primarily on supercomputers where performance was the main concern. A similar event occurred about 10 years later in the finance industry, where the AMQP standard was created. The goals of MPI and AMQP standards are very different, as are the applications they are designed for. In addition to performance, the AMQP standard is also concerned with fault-tolerance.

Initially AMQP was designed strictly as message broker protocol. Unlike MPI, where individual ranks can, and often

do, communicate with each other, the messages in the initial AMQ protocol all pass go through a centralized *hub*. This implies that no two workers in the system can communicate directly. One advantage in using a broker system is that sender and receiver of the message don't have to be running at the same time. A worker can push a message to the hub and, even if the worker process stops running, the message persists and can still be delivered to the intended recipient. This makes the broker model more fault-tolerant with respect to worker failure. Unfortunately, having a single centralized broker can become a bottleneck in the system and increase latency in message passing.

There are several configurations between a pure broker and a peer-to-peer model. ZeroMQ [15] is a library written in C++ that offers flexibility in how a message passing system is implemented. ZeroMQ doesn't provide the same functionality as AMQP; this is left up to the application developer using the library to decide what functionality is needed.

Erlang is gaining popularity for multi-core parallel programming because it provides fast asynchronous message passing that scales. RabbitMQ [18] is an implementation of AMQP written in Erlang. There are several client bindings to RabbitMQ, including some written in Python. Celery [4] is a python package for managing distributed tasks, including scheduled jobs (like cron), and one of the backend options for Celery is RabbitMQ.

IPython [3] originally started as an alternative to the python shell, and has since evolved into a package containing both parallel support and a web-based notebook. IPython implements message passing using ZeroMQ.

IPython Parallel is widely used and is designed for task-based parallelism, but it's scaling characteristics are either unknown or unpublished. Celery is extremely popular as a web-based task management system, but like IPython Parallel, we were unable to find any scaling characteristics when used for MTC on a cluster. The mpi4py package provides bindings to an MPI library. We chose mpi4py as part of our comparison because it scales well, it's Python based, and MPI represents the standard for distributed high performance computing. In other words, mpi4py forms a baseline for comparing IPython Parallel and Celery.

There are several other options distributed computing with Python. We have experimented with Boost MPI [19], Scoop [20], and Multiprocessing, which is part of the standard library.

B. *mpi4py*

The MPI library does not provide load-balancing capabilities for MTC by default. We created a static schedule by by scattering the tasks in the job to each rank using the `statter` function, and then gathered their results at the end using the `gather` function. Our dynamic schedule implementation uses a single rank that acts as the broker responsible for sending and receiving messages that signal the start and end of each task. This was done on an as-needed basis, meaning that each worker only had a single task at any time. Our results in the next section show that this did not become a bottleneck for the jobs we evaluated.

All of our communication used `numpy` arrays to send and receive the data. We also used simple blocking and non-blocking send and receive methods (`send`, `recv`, `isend`, and `irecv`) for the dynamic implementation.

C. *IPython Parallel*

IPython Parallel allows users to interactively run parallel jobs in a distributed environment. Relevant to this paper is the *task interface*, which provides load-balancing capabilities in parallel. IPython includes a *direct interface* that allows more flexibility and control over the work that each core does, but this is outside the scope of MTC problems.

The user interacts with IPython through a *controller* that provides an interface for working with a set of *engines*. The engines are python processes that run on each core, and that execute python code sent from the controller. The controller assigns tasks to each engine through an asynchronous *scheduler*, which can be configured in a variety of ways. All the engine and client connections are monitored through a *hub*. The hub also keeps track of engine requests and results by storing them in a database.

The controller, which will automatically launch the hub and scheduler, must be running before any of the engines can register. The engine process, `ipengine`, must be launched on each core intended for computing . The characteristics of the controller and the engines are specified in a profile, which is a collection of python scripts. The configurations we used for IPython can be found in Appendix A.

We made several changes to the *controller* configuration. Setting the `c.HubFactory.ip = '*'` specifies that the hub will listen for connections on any `ip`. This is acceptable because we are running on a secure cluster. We increased the default `c.HeartMonitor.period` from 3 seconds to 30 seconds to avoid engine registration timeout when running on several thousand cores [1]. This impacts the initialization time because the engine is not ready until the hub receives a heartbeat. No effort was made to optimize this value. Finally, the scheduler is set to `leastload` and the *high watermark* (`hwm`) is set to either 1, for a dynamic schedule, or 0, for a static schedule.

We also increase the time that the engines wait for the controller to respond to registration requests by setting the value of `c.EngineFactory.timeout` to 600 seconds in the `ipengine_config.py` file. This does not appear to impact initialization performance.

IPython has created a tool, called *ipcluster*, to launch the `ipcontroller` and `ipengine` commands. The `ipcluster` process provides several configurations to launch the remote engines, including using MPI, SSH, and PBS. In this paper, we used the MPI launcher because it is most efficient. See Appendix A for additional details. In applications where fault-tolerance is needed, we prefer the SSH launcher.

D. *Celery and RabbitMQ*

Celery is a Python-based distributed task queue. Behind any Celery instance is a message *broker* that is used to send

[1] http://python.6.x6.nabble.com/IPython-User-Making-an-ipython-cluster-more-stable-td3600781.html

and receive messages. In this paper, we used RabbitMQ as the Celery broker.

The process of starting a Celery instance on a cluster requires several steps. The first step is to start the RabbitMQ server and register each machine (or node) that will execute a celery worker. This is accomplished with the `rabbitmqctl add_vhost` command. In our experience, this was the most time consuming part of the initialization process. There are several environmental variables available that control where RabbitMQ stores the instance information, such as log files, the backend database directoryIPython, and the location of the configuration file. Please see Appendix A for details. Once the RabbitMQ broker is running, you can start the Celery process.

The celery workers are launched on each node with a concurrency level. On our system, each node has 12 cores, so we launch a single worker with a `CELERYD_CONCURRENCY` equal to 12. The result will be 13 celery processes running on the node, one for each worker and a single process that acts as a broker between the workers and the task queue. This provides less contention on the RabbitMQ broker because there are 12 times fewer connections. Other important configuration variables include specifying the broker, `CELERY_RESULT_BACKEND = "amqp"` and which file to have each worker import. In our example, the task for each worker was specified in the `tasks.py` file, so our configuration is the following: `CELERY_IMPORTS = ("tasks", )`. We launched the celery workers remotely using the `pbsdsh -u` command, although this could also be accomplished in a way similar to IPython using either MPI or SSH. The command to launch the workers is: `celery worker --app=tasks --workdir=$PBS_O_WORKDIR`.

III. Performance

In order to better understand the scaling behavior of mpi4py, IPython Parallel, and Celery, we performed a weak-scaling study starting with a single node and doubling the node count of each test until we reach 1024 nodes. For the remainder of the paper, we denote n to be the number of nodes in each test. Since each node on our system has 12 cores, the first test on one node ($n = 1$) used 12 cores, the second test ($n = 2$) used 24 cores, and so on, using a total of $1024 \cdot 12 = 12288$ cores for the final test.

Each test executed a different number of tasks, which is proportional to the number of nodes. The total number of tasks executed by each test is $n \cdot 12 \cdot 10$, so, on average, each core will execute about 10 tasks. If we exclude any parallel overhead, then each test should take about the same amount of time to execute. In other words, weak-scaling studies have the advantage the amount of time needed by each test is approximately the same.

There are several factors that impact the overhead associated with running several thousand tasks in parallel. If the task requires its own directory and configuration files, then the operating system calls to set up this configuration can become noticeable as the number of cores increases. The size of the messages used in communication for both the start and end of the task will also impact the efficiency of the overall job. We also observe that many fast jobs tend to scale poorly when compared to longer running jobs.

Our goal was to keep this experiment simple and easy to interpret, so that we can more easily understand the best case scenario for each of the Python implementations we tested. With this is mind, we used a simple `sleep` function as our target task. This means that there is no file system contention or memory contention when executing tasks (other than the `import` statements). It also limits the extent to which oversubscribed nodes impact each other for CPU resources. For example, when launching Celery workers, we previously mentioned there is a concurrency of 12, and that Celery will launch an additional worker to handle task coordination. This additional process will also share CPU resources. The alternative of using a smaller degree of concurrency doesn't seem necessary, but only with a CPU-bound task could we actually measure this impact.

We varied the amount of time needed to execute each task. Each of our tests sampled randomly from the same uniform distribution that contained a 10% variation from the mean. Specifically, we executed tasks from the following uniform distributions, which we treated as runtime in seconds: $[27, 33]$, $[90, 110]$, and $[270, 330]$. Given that each core executes approximately 10 tasks, the expected runtime for each test is about 300 seconds (5 minutes), 1000 seconds (16.7 minutes), and 3000 seconds (50 minutes) respectively.

A common practice when discussing scaling is to exclude the initialization time. We follow this practice, but report the time it takes to initialize in a section below because of the dramatic difference in start up times for Celery and IPython Parallel when compared to mpi4py.

A. Batch Scheduling

There is nothing that needs to be done to configure mpi4py once it is installed. Simply `import mpi4py` and launch your script as you would any other MPI program. In our experience, this is the easiest way to run parallel python code on a cluster.

We automated the process for launching IPython such that each job creates a new profile, executes the `ipcluster` command, and immediately began running the main code once the IPython engines are ready. We did this so that we could have multiple IPython instances running at the same time, and so that we could measure the initialization time required to start the cluster.

Celery has the least support for initializing. Our configuration involved starting the rabbitmq server, attaching the compute nodes from our `PBS_NODEFILE`, and then finally launching the `celery worker` command on each node using the previously mentioned `pbsdsh` command. As with IPython, each instance had its own custom profile so that we could run multiple instances concurrently.

B. Initialization

One advantage of using mpi4py is its initialization time. The `mpirun` command is very efficient at launching thousands of processors and MPI initiIPythonalizes quickly. The other Python packages require a remote process to be running and registered on each node and, this process takes substantially

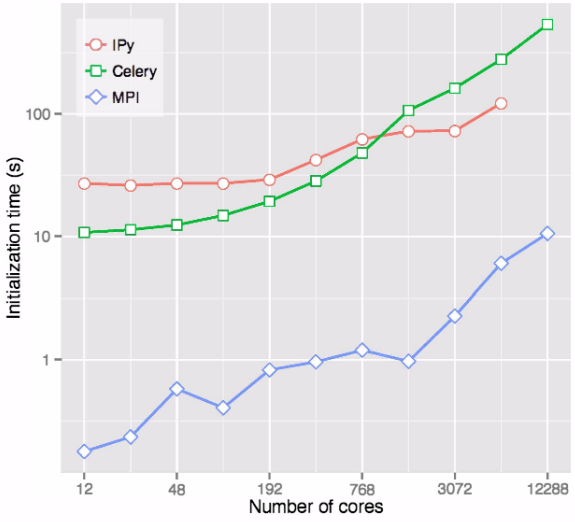


Fig. 2: The time required for each package to initialize as a function of the total number of cores. The mpi4py packages is dramatically more efficient than the other two Python instances.

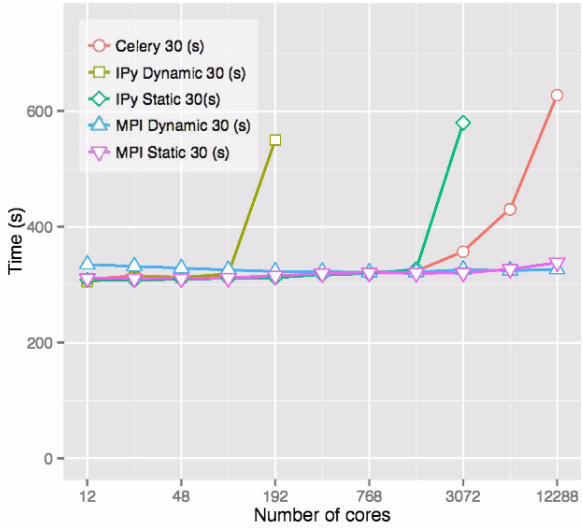


Fig. 3: The time required for for the five configurations we tested needed to complete each job.

more time. Figure 2 shows the initialization time for each package as a function of the number of cores used. IPython and Celery are anywhere from 8 to 100 times slower. The noticeable difference in initialization time may be a limitation for some users. We observe that when using up to approximately 1500 cores, all the python implementations are ready for parallel execution in less than a two minutes.

C. Scaling

We tested five configurations in total: An IPython dynamic and static schedule, and mpi4py dynamic and static schedule, and the default Celery schedule, which is dynamic. Figure 3 shows the total task execution time (without initialization) for these five configurations necessary to complete the set of tasks drawn from the $[27, 33]$ distribution.

All of the configurations perform well up to 96 processors. The mpi4py dynamic instance is less efficient initially because a single core is set aside to broker the tasks and is not used for any work. The other tests did not sacrifice a core to use as a broker.

With the exception of the IPython dynamic instance, the performance for the remaining test cases are very similar using up to 1536 cores. This is where the IPython static schedule and the Celery schedule begin to diverge. Celery degrades less quickly than IPython. The mpi4py instances both perform well throughout the entire test.

Another way to look at scaling is *efficiency*. This is a measure of how much work each CPU in the system is doing when compared to an ideal scenario. We compute efficiency by dividing the *minimum* time reported for 12 cores in Figure 3 by the time required to complete each task. The assumption

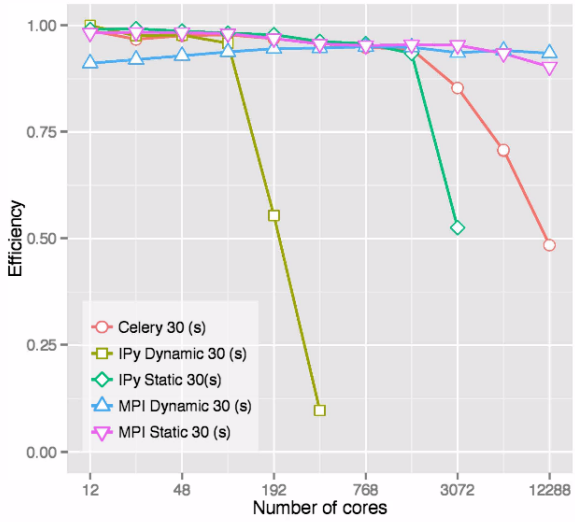


Fig. 4: An efficiency perspective on Figure 3.

here is that the IPython dynamic, the minimum time for 12 cores in Figure 3, is 100% efficient. This is not exactly true, but probably not too far off.

The efficiency for each instance of our weak-scaling study as a function of the number of cores is shown in figure 4. The mpi4py instances always maintain an efficiency greater than 90%. Celery is also extremely efficient using up to 1536 cores, and still maintains an efficiency of 85% when doubling that number. The efficiency for IPython drops quickly in the dynamic instance after 96 cores and in the static instance after

1536 cores. We assume that the *hub* becomes a bottleneck in both instances. The concurrency of Celery workers probably prolongs the degradation in scaling.

Efficiency is important in the context of MTC because it directly impacts the performance metric of the overall job. For example, if each task in the job operates at a high efficiency level and our goal is to execute the overall job at this same level of efficiency, then we need to use a framework that can manage our tasks at an efficiency close to 100%.

These results only apply to tasks that take approximately 30 seconds. As mentioned, longer running tasks often produce better scaling. Figure 5 shows the total time for each set of tasks for the IPython dynamic schedule as a function of the number of cores. IPython dynamic scales well using up to 384 processors when the mean task execution time is 300 seconds.

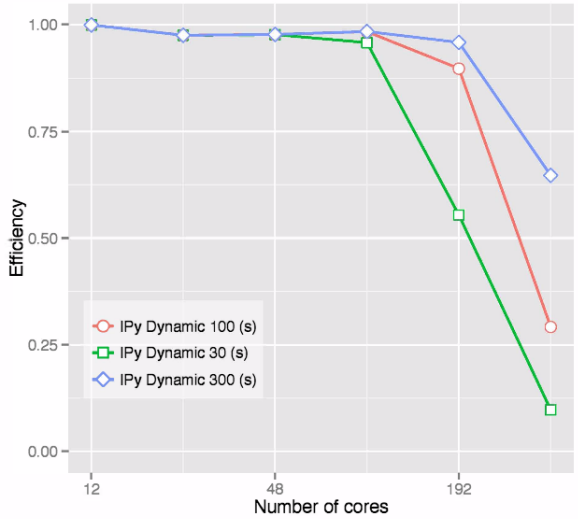


Fig. 5: Longer running tasks are more efficient. The initial impact of scale is increasingly noticeable as the number of cores increases. For example, using 192 cores impacts the job that consists of time 30 tasks more than it does the other two jobs.

D. Discussion of Features

Our results indicate that mpi4py has the most efficient initialization and scaling, and for many scientists wanted to use Python, building a MTC tool with mpi4py may be the easiest way to get something up and running quickly. Scaling is just one characteristic that is important to a user with a MTC job. Some other important characteristics include *fault-tolerance*, *elasticity*, *persistence*, *retries*, and *task time-boundaries*.

We refer to a libraries ability to complete a job when one or more nodes or cores fail as *fault-tolerance*. One of the disadvantages of using MPI for MTC is that if a single core fails, the entire job will fail. A broker message passing design can continue operating when a portion of the resource fails, and this is a strong argument for using either IPython Parallel or Celery.

Elasticity refers to a programs ability to grow and shrink it's resources while executing. The MPI-2 standard does not address resource control. Libraries such as ZeroMQ and AMQP have built in functionality to allow elastic resources. This is certainly an advantage when considering node failure, and offers the ability to add the node back at a later time.

The RabbitMQ broker stores it's messages in a database by default. IPython Parallel has the ability to connect to a database also, but this is not the default behavior. Storing messages in a database means that if a job partially finishes, it can be restarted without managing which tasks have finished and which still need to be executed. This *persistence* is kind of a checkpoint mechanism for MTC jobs. This type of feature could easily be built into a system using mpi4py.

IPython Parallel or Celery also have built-in mechanisms to handle multiple submissions, or *retries*, should a task fail, or fail to start the first time. We have worked with code that exhibits this quality and this feature is very useful in enabling a job to finish. There are also ways to put parameters around the maximum duration of a task. Is is especially important for simulations exploring portions of the parameter space where the behavior of the simulation, especially in terms of execution time, is unknown.

Consider the example job from Figure 1, but this time we add a 10% task failure rate. Furthermore, we remove three cores from the resource pool randomly about half way through the job. IPython Parallel and Celery make finishing this job very easy. The tasks are retried until they finish, and when resources are removed, the brokers reschedule the incomplete tasks on the remaining cores. Figure 6 shows the time-line for this test job. The large empty spaces occur because we removed the core from the pool or resources. The smaller intermittent gaps are from instances where a task failed at least once.

IV. Related Work

Submitting thousands of jobs to a compute cluster directly to a resource manager such as Condor [10], SLURM [11], SGE [21], or some of the commercial batch schedulers [22], is not a very efficient approach. First, starting a task directly through a batch scheduler may take several seconds which is not a trivial overhead compared to short running tasks. Second, policies implemented in a scheduler may not support the submission of thousands of compute jobs simultaneously. One successful abstraction to address these restrictions is the concept of Pilot-Jobs [23]. Luckow et. al. [23] define a pilot job as "an abstraction that generalizes the reoccurring concept of utilizing a placeholder job as a container for a set of compute tasks; an instance of that placeholder job is commonly referred to as Pilot-Job or pilot."

A variety of Pilot-Job frameworks have been developed in the context of high-throughput computing. A few are listed here: Falkon [24], Swift [25], myCluster [26], DIANE[27], Condor Glide-in [28], Nimrod/G[29], DIRAC[30], DIANE [27], SAGA BigJob [31].

Many-task computing is also being explored at the petascale level. Raicu et. al. [1] have extended the Falkon task execution framework for petascale systems. Rynge et. al. [32] apply an MPI based master/worker approach to petascale system for earthquake data analysis.

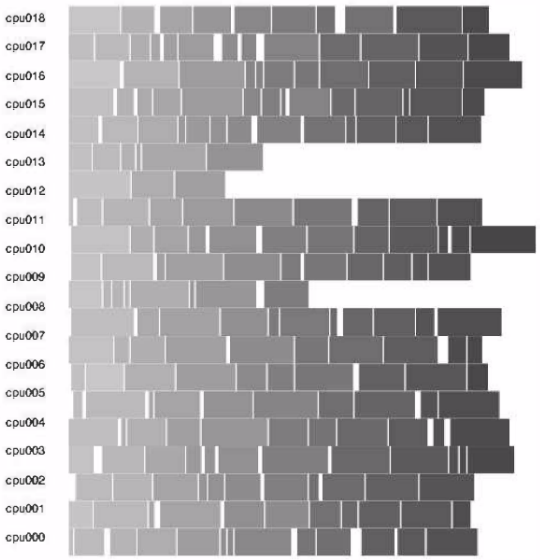


Fig. 6: Some MTC jobs are difficult to complete. In this case, each task fails 10% of the time and three cores are removed from the pool or resources about half way through the job. IPython Parallel and Celery gracefully handle this situation that is not possible with mpi4py.

V. Conclusion

Python is an excellent way to manage MTC jobs. When the tasks at hand are reliable, and the time they need to execute is predictable, a quick mpi4py implementation is an efficient way to load-balance thousands of tasks quickly and easily. If the runtime characteristics of the tasks in a particular job are unknown or unreliable, then both the IPython Parallel and Celery packages provide a solution. Having fault tolerance for individual processes running on remote cores increases the utility for running task-based parallel problems at a large scale.

Our results are the first step toward understanding the scaling of these Python packages when applied to a task-based parallel problem. We have intentionally ignored the impact of a fast interconnect, like Infiniband, because the messages we sent and received are very small and the benefit of a high-performance network would likely be negligible in this experiment. For example, we turned off the Inifiband when running the mpi4py tests and did not see a significant difference in results. We have also ignored how CPU intensive tasks impact our scaling. Addressing these issues are an important direction for future work.

Finding an efficient way to launch remote processes is an important area of development. Improving the efficiency of the registration process will increase the usability of these tools.

Our results are best interpreted as a baseline. Multiple RabbitMQ brokers can be launched as a cluster that acts as a single broker. This will likely improve the scaling of Celery. IPython also supports a *pure* schedule that improves efficiency, but sacrifices some functionality. This may also improve performance.

One of the advantages of mpi4py is that it's built on top of an established standard that is already well supported with parallel debuggers and profilers. Moving forward, there is a need for a general profiling tool that could be used by these new approaches to better understand where the performance bottlenecks are. In the simple case, where there is a single broker, an individual profile may provide useful insight. But as multiple brokers are used together to increase scaling, profiling tools like TAU and the HPCToolkit will be needed for bringing new technologies like IPython and Celery to bare on scientific problems at an even greater scale.

Acknowledgment

This work utilized the Janus supercomputer, which is supported by the National Science Foundation (award number CNS-0821794), the University of Colorado Boulder, the University of Colorado Denver, and the National Center for Atmospheric Research. The Janus supercomputer is operated by the University of Colorado Boulder.

References

[1] I. Raicu, Z. Zhang, M. Wilde, I. Foster, P. Beckman, K. Iskra, and B. Clifford, "Toward loosely coupled programming on petascale systems," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, Nov. 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1413393

[2] L. Dalcín, R. Paz, and M. Storti, "Mpi for python," *Journal of Parallel and Distributed Computing*, vol. 65, no. 9, pp. 1108–1115, 2005.

[3] F. Perez and B. E. Granger, "Ipython: a system for interactive scientific computing," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 21–29, 2007.

[4] "Celery: Distributed task queue." [Online]. Available: http://www.celeryproject.org/

[5] J. G. Caporaso, J. Kuczynski, J. Stombaugh, K. Bittinger, F. D. Bushman, E. K. Costello, N. Fierer, A. G. Pena, J. K. Goodrich, J. I. Gordon, G. A. Huttley, S. T. Kelley, D. Knights, J. E. Koenig, R. E. Ley, C. A. Loft, R.zupone, D. McDonald, B. D. Muegge, M. Pirrung, J. Reeder, J. R. Sevinsky, P. J. Turnbaugh, W. A. Walters, J. Widmann, T. Yatsunenko, J. Zaneveld, and R. Knight, "Qiime allows analysis of high-throughput community sequencing data," *Nature methods*, vol. 7, no. 5, pp. 335–336, 2010. [Online]. Available: http://www.nature.com/nmeth/journal/v7/n5/full/nmeth.f.303.html

[6] S. M. Flaxman, J. L. Feder, and P. Nosil, "Genetic hitchhiking and the dynamic buildup of genomic divergence during speciation with gene flow," *Evolution*, 2013. [Online]. Available: http://onlinelibrary.wiley.com/doi/10.1111/evo.12055/full

[7] S. Flaxman, J. Feder, and P. Nosil, "Spatially explicit models of divergence and genome hitchhiking," *Journal of evolutionary biology*, vol. 25, no. 12, pp. 2633–2650, 2012. [Online]. Available: http://onlinelibrary.wiley.com/doi/10.1111/jeb.12013/full

[8] G. Tucker and P. van der Beek, "A model for post-orogenic development of a mountain range and its foreland," *Basin Research*, 2012. [Online]. Available: http://onlinelibrary.wiley.com/doi/10.1111/j.1365-2117.2012.00559.x/full

[9] D. Folch and S. Spielman, "Regionalization approach to reduce small area margins of error in the american community survey," presented at the Association of American Geographers Annual Meeting. Los Angeles, CA, 2013.

[10] E. Robinson and D. DeWitt, "Turning Cluster Management into Data Management: A System Overview," *arXiv.org*, Dec. 2006. [Online]. Available: http://arxiv.org/abs/cs/0612137

[11] A. B. Yoo, M. A. Jette, and M. Grondona, "SLURM: Simple Linux Utility for Resource Management - Springer," *Job Scheduling Strategies for Parallel ...*, 2003. [Online]. Available: http://link.springer.com/chapter/10.1007/10968987_3

[12] W. Gropp, E. L. Lusk, and A. Skjellum, *Using MPI-: Portable Parallel Programming with the Message Passing Interface*. MIT press, 1999, vol. 1.

[13] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[14] S. Vinoski, "Advanced message queuing protocol," *Internet Computing, IEEE*, vol. 10, no. 6, pp. 87–89, 2006.

[15] P. Hintjens, *ZeroMQ: Messaging for Many Applications*. O'Reilly Media, 2013.

[16] J. Armstrong, R. Virding, C. Wikstr, M. Williams *et al.*, "Concurrent programming in erlang," 1996.

[17] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala*. Artima Inc, 2008.

[18] A. Videla and J. J. Williams, *RabbitMQ in action*. Manning, 2012.

[19] P. Kambadur, D. Gregor, A. Lumsdaine, and A. Dharurkar, "Modernizing the c++ interface to mpi," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer, 2006, pp. 266–274.

[20] "Scoop: Scalable concurrent operations in python." [Online]. Available: https://code.google.com/p/scoop/

[21] W. Gentzsch, "Sun Grid Engine: towards creating a compute power grid," in *Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on*, 2001, pp. 35–36. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=923173

[22] Y. Etsion and D. Tsafrir, "A short survey of commercial cluster batch schedulers," *... of Computer Science and Engineering*, 2005. [Online]. Available: http://leibniz.cs.huji.ac.il/tr/742.pdf

[23] A. Luckow, M. Santcroos, A. Merzky, O. Weidner, P. Mantha, and S. Jha, "P*: A model of pilot-abstractions," in *E-Science (e-Science), 2012 IEEE 8th International Conference on*, 2012, pp. 1–10. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=6404423&matchBoolean%3Dtrue%26rowsPerPage%3D30%26searchField%3DSearch_All%26queryText%3D%28%22Pilot+Job%22%29

[24] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde, "Falkon: a Fast and Light-weight tasK executiON framework," in *Supercomputing, 2007. SC '07. Proceedings of the 2007 ACM/IEEE Conference on*, 2007, pp. 1–12. [Online]. Available: http://dl.acm.org/citation.cfm?id=1362680

[25] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster, "Swift: A language for distributed parallel scripting," *Parallel Computing*, vol. 37, no. 9, pp. 633–652, Sep. 2011. [Online]. Available: https://trac.ci.uchicago.edu/swift/export/5720/text/parco10submission/original_submission.pdf

[26] E. Walker, J. P. Gardner, V. Litvin, and E. L. Turner, "Creating personal adaptive clusters for managing scientific jobs in a distributed computing environment," *Challenges of Large Applications in Distributed Environments, 2006 IEEE*, pp. 95–103, 2006. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1652061

[27] J. T. Moscicki, "DIANE - distributed analysis environment for GRID-enabled simulation and analysis of physics data," in *Nuclear Science Symposium Conference Record, 2003 IEEE*, 2003, pp. 1617–1620. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1352187

[28] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke, "Condor-G: A Computation Management Agent for Multi-Institutional Grids - Springer," *Cluster Computing*, 2002. [Online]. Available: http://link.springer.com/article/10.1023/A%3A1015617019423

[29] R. Buyya, D. Abramson, and J. Giddy, "Nimrod/G: an architecture for a resource management and scheduling system in a global computational grid," in *High Performance Computing in the Asia-Pacific Region, 2000. Proceedings. The Fourth International Conference/Exhibition on*, 2000, pp. 283–289. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=846563

[30] A. Casajus, R. Graciani, S. Paterson, A. Tsaregorodtsev, and t. L. D. Team, "DIRAC pilot framework and the DIRAC Workload Management System," *Journal of Physics: Conference Series*, vol. 219, no. 6, p. 062049, May 2010. [Online]. Available: http://stacks.iop.org/1742-6596/219/i=6/a=062049?key=crossref.6decb3da7fe75aa8dc6b990b94c034d8

[31] A. Luckow, L. Lacinski, and S. Jha, "SAGA BigJob: An Extensible and Interoperable Pilot-Job Abstraction for Distributed Applications and Systems," in *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, 2010, pp. 135–144. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5493486

[32] M. Rynge, S. Callaghan, E. Deelman, G. Juve, G. Mehta, K. Vahi, and P. J. Maechling, "Enabling large-scale scientific workflows on petascale resources using MPI master/worker," in *the 1st Conference of the Extreme Science and Engineering Discovery Environment*. New York, New York, USA: ACM Press, 2012, p. 1. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2335755.2335846

Appendix A
Configuration

These are the settings we used in our experiments. The following files are found in the IPython profile.

ipcluster_config.py

```
c = get_config()
c.IPClusterEngines.engine_launcher_class = \
    'MPIEngineSetLauncher'
c.MPIEngineSetLauncher.mpi_args = \
['--bynode', '--bind-to-core']
c.MPIEngineSetLauncher.mpi_cmd = ['mpirun']
```

ipengine_config.py

```
c = get_config()
c.EngineFactory.timeout = 600
```

ipcontroller_config.py

```
c = get_config()
c.HubFactory.ip = '*'
c.LocalControllerLauncher.controller_args = \
["--ip='*'"]
c.HeartMonitor.period = 30000
c.TaskScheduler.scheme_name = 'leastload'
c.TaskScheduler.hwm = 1
```

The `celeryconfig.py` file is configured at execution time using the `jinja2` template language the master node variable, which we call `node_name`.

celeryconfig.py

```
## Broker settings.
BROKER_URL = \
"amqp://guest:guest@{{node_name}}:5672//"
BROKER_CONNECTION_TIMEOUT = 120

# List of modules to import when celery starts.
CELERY_IMPORTS = ("tasks", )
CELERY_RESULT_BACKEND = "amqp"
CELERY_DISABLE_RATE_LIMITS = True
CELERYD_CONCURRENCY = 12
```

We set the following environmental variables for RabbitMQ before we launch the broker. The `directory` variable is the location of the RabbitMQ instance.

```
os.environ['RABBITMQ_LOG_BASE'] = directory
os.environ['RABBITMQ_MNESIA_BASE'] = directory
os.environ['RABBITMQ_CONFIG_FILE'] = \
os.path.join(directory,'rabbitmq')
```