

Once you SCOOP, no need to fork

Yannick Hold-Geoffroy

Laboratoire de vision et
systèmes numériques
Département de génie
électrique et de génie
informatique
Université Laval, Québec
(Québec), Canada G1V 0A6
yannick.hold-
geoffroy.1@ulaval.ca

Olivier Gagnon

Laboratoire de vision et
systèmes numériques
Département de génie
électrique et de génie
informatique
Université Laval, Québec
(Québec), Canada G1V 0A6
olivier.gagnon.7@ulaval.ca

Marc Parizeau

Laboratoire de vision et
systèmes numériques
Département de génie
électrique et de génie
informatique
Université Laval, Québec
(Québec), Canada G1V 0A6
marc.parizeau@gel.ulaval.ca

ABSTRACT

This paper presents SCOOP, a new Python framework for automatically distributing dynamic task hierarchies. A task hierarchy refers to tasks that can recursively spawn an arbitrary number of subtasks. The underlying computing infrastructure consists of a simple list of resources. The typical use case is to run the user's main program under the umbrella of the SCOOP module, where it becomes a root task that can spawn any number of subtasks through the standard "futures" API of Python, and where these subtasks may themselves spawn other subsubtasks, etc. The full task hierarchy is dynamic in the sense that it is unknown until the end of the last running task. SCOOP automatically distributes tasks amongst available resources using dynamic load balancing. A task is nothing more than a Python callable object in conjunction with its arguments. The user need not worry about message passing implementation details; all communications are implicit.

Categories and Subject Descriptors

D.1.3 [Programming Techniques]: Concurrent Programming; H.3.4 [Information Storage and Retrieval]: Systems and Software—*distributed systems*

General Terms

Performance

Keywords

Parallel programming, Software libraries, Distributed computing, High performance computing

1. INTRODUCTION

Over time, the difficult problems that we want to solve using computers require constantly growing computational effort. The performance of current hardware is notably limited by power envelope considerations, where higher clock frequencies is no longer an option. Current and future processors are going towards massive multicore architectures that imply a shift in programming paradigms.

Developing solutions to fully exploit parallel hardware is generally complicated and error-prone. Prototyping parallel programs requires frameworks that implicitly handle the multiple mechanisms of distributed computing such as synchronization and message passing, while keeping the code complexity as low as possible.

This paper presents Scalable Concurrent Operations in Python (SCOOP), a new parallel framework that allows programmers to exploit distributed resources with minimal overhead.

Our goal is to harness the power of parallel systems while keeping development straightforward for the user. Converting serial code to parallel code should be as simple as possible while keeping it maintainable and reusable. Future programming models must be more human-centric than the conventional focus on hardware or applications [1].

Frameworks should be flexible enough to adapt intuitively on most problems. It should also take into account the multifaceted aspect of task complexity: some are set by an amount of computation while others are bound to various I/O events.

2. RELATED WORK

The two most notable standards in parallel programming are OpenMP and MPI, both available in C/C++ and Fortran. They primarily propose two different approaches at parallelism, the first relying on the OS for launching and communication while the second offers a message passing protocol enabling distributed computations.

OpenMP [2] defines preprocessor instructions that allow programmers to easily parallelize a section of his code. These sections are usually *for* loops or functions. The key advantage of OpenMP is its simplicity: slightly modifying a serial code can potentially provide a great performance gain. It is implemented by most compilers. Under the hood, the compiler spawns threads at the beginning of a parallel section, splits the work among them and kills them once the section

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

XSEDE '14 July 13 - 18 2014, Atlanta, GA, USA

Copyright 2014 ACM 978-1-4503-2893-7/14/07 ...\$15.00
<http://dx.doi.org/10.1145/2616498.2616565>.

is over.

MPI [11] defines a communication protocol and an API used for parallel programming. The latest stable standard version is 2 with ongoing work toward a third version. Its most widely known implementations are the open source OpenMPI and MPICH2 available in C/C++ and Fortran as well as the commercial versions of HP, Intel and Microsoft. It is available in Python through the mpi4py package.

Aside from these two standards, many other frameworks exist featuring their own characteristics and views upon parallel programming. Some of them have been in existence for a long time and have a considerable knowledge base and community.

Charm++ [9] is one of these notable message-driven C++ open source library used notably for molecular dynamics. An implementation of the MPI standard¹ has been made using this library. Its usage is targeted towards tightly coupled, high-performance parallel machines. They define an executable processing unit as a *chare*. A characteristic feature of this library is its advanced object migration facilities, allowing dynamic relocation of *chares*. This provides a great versatility in load balancing, fault tolerance and scaling.

Parallelism is also featured in many other languages. For instance, Scala², a language based on Java, offers a collection of parallel data structures enabling task submission over multiple threads. These threads are created using either a Fork-Join or a thread pool model. Many task submission techniques are supported such as map, foreach, fold, filter and reduce. Another language geared toward parallel execution is Julia³. It is designed to consider concurrent and parallel computations and offers submit-like functionalities to execute calls remotely.

A distributed computing approach recently gaining much attention is Google's MapReduce [3]. It is composed of a task distribution phase, named the map, followed by a reduction phase. In between those two phases, a shuffle phase is performed to optimize the communication efficiency. It is currently implemented in Hadoop⁴ and aims at large datasets which cannot be contained in a single system memory.

2.1 Parallel frameworks in Python

Python has a growing community of scientific and HPC users [10]. Its general flexibility makes it a good candidate for fast prototyping and testing of ideas. This section provides an overview of the currently available Python frameworks for concurrent and parallel computing.

Some Python packages allow the use of OS threads and processes to parallelize execution. Threading can be exploited using the standard threading module. Threads in Python have a major performance limitation, however, because of the Global Interpreter Lock (GIL) that effectively limits the execution of Python bytecode to a single thread at a time. To parallelize Python code using processes, one can use the multiprocessing module⁵ that is part of the standard library since Python 2.6. It uses multiple processes, each running a distinct Python interpreter.

¹<http://charm.cs.uiuc.edu/research/ampi/>

²<http://www.scala-lang.org/>

³<http://julialang.org/>

⁴<http://hadoop.apache.org/>

⁵<http://docs.python.org/library/multiprocessing.html>

Celery⁶ is a centralized framework that executes tasks asynchronously using distributed message passing. It supports multiple backends for communication and execution. Concurrency can be handled through the multiprocessing, Eventlet or gevent modules. The communication architecture used by Celery is centralized using a broker to propagate messages through a pool of workers. The supported communication backends are either message queuing frameworks, such as RabbitMQ, or databases like Redis. Configuration, like broker URL, task serializer method and routing hints is written in Python. Tasks can be defined with a function decorator. Celery defines its own API for task creation, submission and interaction.

The Celery project is primarily used in web applications. It allows the parallel execution of HTML generation which improves web requests response time. Another common usage is the launch and polling of background tasks during and after requests.

Configuration may either be done through a configuration file or embedded in code as in the example. Tasks must be defined in a separate module to be parallelized.

Celery is an actively developed project which released its latest stable version, 3.1 in November 2013. It is supported on the latest versions of Python 2 and 3 as well as Pypy and Jython.

Another notable parallel Python framework is IPython Parallel⁷. Its target audience is scientists who analyze great amount of data. It uses ZeroMQ to communicate. One of its distinguishable features is its ability to run a parallel interactive interpreter. Its centralized architecture is composed of a single hub, one or more engines and one or more clients. The hub is the central element of the architecture: every other element is connected to it. It keeps track of engines and clients as well as all task requests and results. Engines are the processes that execute tasks while clients are the ones that generates them.

Through a client instance, IPython Parallel provides a direct handle to every available engine (often called worker in other frameworks). This means that it is possible to execute a given task on a specific engine by applying the task to its handle. It also provides a load balancing view which automatically dispatches tasks to available engines. IPython Parallel also defines an asynchronous generic data communication scheme. Since it is possible to directly address specific engines, it is also possible to send data to them. It is also the only framework allowing explicit task assignment while others coercively provide load balancing.

IPython Parallel is actively being developed and is available on the latest version of Python 2 and 3, as well as Pypy and most other Python flavors. It supports every major operating system.

Celery and IPython parallel are both out-of-the-box compatible with cloud resources such as Amazon EC2. Also, cluster schedulers such as PBS or SGE are only supported by IPython parallel and Jug.

SCOOP aims to offer a simpler alternative designed for HPC usage while staying usable on development systems. As of now, it uses ZeroMQ to communicate but there is an ongoing effort to add support of other communication backends. It also supports major cluster schedulers out of

⁶<http://www.celeryproject.org/>

⁷<http://ipython.org/>

the box. Is it developed with Python 3 while keeping in mind a support for Python 2.6 and 2.7, which are still used in HPC systems. SCOOP also simplifies the advanced usage of distributed resources; for example, launching a distributed interactive interpreter is a simple 4-line script provided as an example.

3. SCOOP OVERVIEW

SCOOP is built on the premise that converting sequential code to exploit distributed resources should be as straightforward as possible. In a research context, the speed with which new ideas can be prototyped may be just as important as minimizing the runtimes necessary for testing these ideas. Of course, more efficiency means that bigger problems can be tackled, but reaching high levels of efficiency can quickly become daunting. An equilibrium is thus needed between parallel efficiency and ease of parallelization. SCOOP tries to maximize the latter while performing reasonably well on the former using a simple application programming interface (API).

The launching of a parallel application is also part of the ease of use experience. SCOOP adopts a path similar to *mpirun*, where the program is simply passed as a parameter to a launcher as in: `python -m scoop my_program.py`. This executes the launcher which, if needed, connects to remote resources and launches all of the necessary processes. These processes, called **workers**, try to import every symbol defined in the user program and then wait for tasks. Lastly, a worker which will execute the user's main program, called the root task, is launched. The main program is encapsulated in a *future* and is called the root task.

This launching mechanism allows easy integration with legacy or serial code but requires a separate top-level script environment to execute correctly. Failure to do so results in every worker executing all tasks. The main program must thus add the usual conditional statement to the main module:

```
1 from scoop import futures
2 [...]
3 if __name__ == "__main__":
4     [...]
```

and the code block of this statement will be run only by the worker executing the root task. This requirement is shared by many other parallel frameworks in python such as multiprocessing.

To operate on multiple computers, the user must specify a list of resource hostnames to the launcher. On a cluster using a scheduler like Sun Grid Engine (SGE) or Portable Batch System (PBS, Moab, etc.), SCOOP will automatically discover and use the computing nodes that were assigned to the job by the scheduler. This feature is only shared by iPython Parallel.

3.1 SCOOP API

Futures are constructs meant to synchronize concurrent algorithms. They primarily define proxy objects which contains the promise of a result. The beginning of their execution is asynchronous and undetermined at creation time. At the time of object evaluation, their result is either retrieved if available, or computed otherwise. This concept was first

described by Friedman [5], Hibbard [7] and Baker [8] in the late seventies. It is now natively supported by several programming languages, notably C++11, Java and Python 3.2 (backported to Python 2).

The *futures* are used as an API reference for SCOOP. Through the PEP-3148⁸, they specifies the smallest unit of parallel work as a task. They also define the possible task interactions such as submission and data retrieval. We believe that using a simple and standard interface will give SCOOP an edge on usage simplicity while leaving good flexibility to experiment with its inner workings.

SCOOP implements its own *futures* module that provides an explicit task submission function called `submit`:

```
1 futures.submit(func, *args)
```

This function generates a *future* that contains a callable - a function `func` - and its arguments `*args`. This future will either be kept for local execution or be serialized and sent to a remote worker, depending on the current worker load.

Three other functions are proposed to handle implicit future generation and submission: `map`, `mapReduce` and `mapScan`.

```
1 futures.map(func, *iterables)
2 futures.mapReduce(mapFunc, reductionOp, *iterables)
3 futures.mapScan(mapFunc, reductionOp, *iterables)
```

The `map` function creates and submits for execution multiple *futures* using one or more iterables. It is the most common SCOOP use case. This is a drop-in replacement for the standard Python `map` function. As such, its results are in the same order as the elements in the input iterable(s).

The `mapReduce` function automatically generates a tree-like task hierarchy, recursively splitting in two the input iterables. The reduce operator must accept two arguments as input and output a single result. The `mapScan` is the same operation, but with partial results kept to produce a scan sequence of results.

These reduce and scan features currently do not implement the whole MapReduce pattern [3]: they skip the shuffle step (often called partitioning) which is an important step of the process for efficient handling of voluminous datasets. Without the shuffle step, all of the partial results are transmitted, potentially via network, resulting in duplicated bandwidth usage that can be problematic if intermediate results are quite large.

In addition to the submission primitives, SCOOP provides two synchronization functions: `wait` and `as_completed`.

```
1 futures.wait(future_set, timeout, return_when)
2 futures.as_completed(future_set, timeout)
```

The first simply waits until the condition specified by argument `return_when` or when the specified `timeout` is attained. Three conditions are available for `return_when`: either the function returns when futures of set `future_set` are **all completed**, even if one raises an exception, or it returns as soon as the *first is completed*, including when it raises an

⁸<http://www.python.org/dev/peps/pep-3148>

exception, or it returns either after the *first exception* or after all are completed. The second function, *as_completed*, waits on the next available result; its outputs may not respect the order in which the inputs were provided.

In SCOOP, an exception raised by a child task will always find its way back to its parent task, no matter on which worker the child and parent are running on, and will escalate to the parent of the parent if not processed by the latter. This feature takes advantage of the powerful exception features to catch any voluntary (`raise` statement) or unintentional (error in code) exception that occurs on a worker. SCOOP then serializes the error message alongside useful traceback information and used the usual future return mechanism to propagate it to its parent. Just like a normal exception, it will backtrack the call stack until the exception is caught. If the exception is not caught, the program crashes after displaying the traceback related to the exception.

SCOOP also provides a mechanism to share data between workers. This allows user-controlled *broadcast* of Python data structures, variables and even callables to every worker. To circumvent synchronization issues, the data is broadcasted as constants and cannot be changed once it has been set. This ensures data coherency:

```

1 from scoop import futures, shared
2 def myFunc(parameter):
3     print(shared.getConst('myVar')[2])
4 if __name__ == "__main__":
5     # Set shared constants
6     shared.setConst(myVar={
7         1: 'First element',
8         2: 'Second element',
9         3: 'Third element',
10    })
11    shared.setConst(secondVar="Hello World!")
12    shared.setConst(myFunc=myFunc)
13
14    # Use the previously defined shared function
15    print(list(futures.map(myFunc, range(10))))
16
17    # The following line would give a TypeError
18    # because constant re-definition is not allowed
19    #shared.setConst(myVar="Re-definition")

```

The `setConst` function assigns identifiers to arbitrary python object and exports these identifiers to all workers processes. The `getConst` function acts as a barrier that will wait for the requested constant to be received before continuing execution. This feature is used primarily in two main use cases. First, to share runtime-generated data that is common to many tasks, without having to retransmit the data multiple times. Second, it enables the interactive use of SCOOP by allowing remote callable definition at runtime.

3.2 Execution and communication

There are two core elements in SCOOP: workers that executes tasks, and brokers that coordinate communications between workers. In this context, tasks are defined as collaboratively scheduled unit of work, represented by futures. They can be moved to another worker any number of times, as long as they haven't started execution. A task can compute, spawn child tasks and wait for child results before

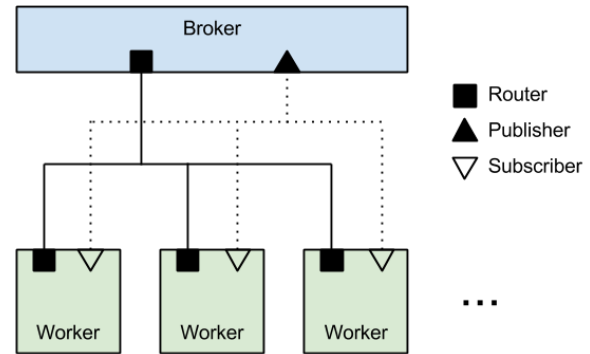


Figure 1: Broker/Worker interconnect.

continuing execution. While waiting for results, the current task is halted and another task may begin its execution. The execution of a halted task can only resume after some children results are received.

In SCOOP, a task is executed within a *greenlet*⁹ that represents a pseudo-concurrent micro-thread implementation. This module evolved from the *Stackless*¹⁰ Python version to propose lightweight coroutines. It allows the creation of coroutines with small computational and memory requirements, bypassing the overheads of operating systems.

For communications, SCOOP uses *ZeroMQ*¹¹, a lightweight library for message passing. Figure 1 illustrate the interconnections between a broker process and a number of worker processes. These processes may run on a single node multicore computer, or on a multinode cluster of multicore computers. At the time of launching a SCOOP application, the user can specify the number of worker processes per node or per core, or use the default value of one worker per available core. He can also specify the number of brokers to start, or use the default of one broker.

The broker process manages two ZeroMQ sockets: one that implements a router pattern and one that implements a publisher pattern. For p workers, the router manages p peer-to-peer sockets and handles the routing. The router serves tasks to worker processes. The publisher is used for broadcasting directives to workers, such as shutdown messages or transmission of shared constants.

Similarly, the workers possess two ZeroMQ sockets: one router for either pulling or pushing tasks, and one subscriber for listening to broker directives. The router socket of workers are also used to establish direct connections to other workers, in order to bypass the broker when returning results. A maximum number of peer-to-peer connections is fixed in order to avoid wasting worker resources. Established connections persist as long as the limit is not reached. Otherwise, least recently used connections are closed to enable new connections.

3.3 Worker queues

A worker maintains two separate queues of tasks: the pending execution and the waiting to resume queues. The

⁹<http://greenlet.readthedocs.org/en/latest>

¹⁰<http://www.stackless.com/>

¹¹<http://zeromq.org/>

tasks in the pending execution queue have either been spawn locally by a running task, or have been pulled from the broker by the worker. Algorithm 1 describes how tasks get appended to the pending execution queue. A new task gets

Algorithm 1 worker append task algorithm

```

Let pending represent the pending execution task queue.
Let task be the task being appended to the queue.
Let minimum_load be the Minimum Load parameter.
if estimated_duration(pending) > minimum_load then
    Send task to broker.
else
    Append task to pending.
end if

```

appended locally only if the estimated runtime of the pending execution queue is below a certain threshold. Otherwise, it is transferred to the broker and will eventually migrate to some other worker.

Algorithm 2 describes how tasks get popped from either the waiting to resume or pending execution queues. This

Algorithm 2 worker pop task algorithm

```

Let pending represent the pending execution queue.
Let minimum_load be the Minimum Load parameter.
if estimated_duration(pending) < minimum_load then
    Send request for tasks to broker.
end if
if a task in the waiting to resume queue then
    Pop and execute this task.
end if
if no task is available in the pending execution queue then
    Wait until a task arrives from the broker.
end if
Execute the next task in the pending execution queue.

```

algorithm is called each time a running task decides to wait for some results. This task is halted so that another task can either resume its execution or start running. It is also executed at launch on every worker aside from the one executing the root task.

Task results are sent back directly to the worker that is running the parent task. If no direct routing is possible, the results are routed through the broker.

3.4 Statistics and task categories

In order to make load balancing decisions, workers compile per function runtime statistics of the task they have executed so far. These statistics enable the prediction of the task duration and are used to estimate in seconds the total runtime of the tasks present in the pending execution queue.

Task categories are defined by the nature of the task and the size of its parameters. If the callable of two futures resolve to the same unique identifier, they are considered to be of the same nature.

Task runtimes are modeled using a log-normal distribution associated with each task category. For a log-normal of parameters μ and σ , the mode $e^{\mu-\sigma^2}$ is used as typical execution time when analyzing similar pending execution tasks.

When no category statistic is available for a given task, it is assumed to have an infinite execution time.

The minimum load parameter is the amount of work (in seconds) that the worker tries to buffer in order to stay busy at all times. It needs to be higher than the turnaround time needed to send a request and receive new work from the broker. The workers always tries to keep a safe margin of workload over this threshold.

4. CODE SAMPLES / EXAMPLES

There are several common patterns of parallelism, also called skeletons [6]. Notably, there is *data-level parallelism* which is primarily defined by the *map*, the *fork* and the *reduce* constructs. Then, there is *task-based parallelism* that encompass a higher level of flow execution with skeletons such as *farm* and *pipe*. There is an ever higher level of abstraction called *resolution* that proposes approaches to solve families of methods such as Divide and Conquer.

4.1 Map

The map skeleton is the archetypal parallel primitive. It executes multiple times a given function with different arguments each time. SCOOP offers this primitive in its futures module.

For example, a Monte Carlo computation such as the stochastic computation of π can be implemented sequentially by the following Python code:

```

1 from math import hypot
2 from random import random

3 def test(n):
4     return sum(hypot(random(), random()) < 1
5                 for _ in range(n)
6                 )

7 def calcPi(repeat, n):
8     expr = map(test, [n] * repeat)
9     return 4. * sum(expr) / float(repeat * n)

10 if __name__ == "__main__":
11     print("pi = {}".format(calcPi(3000, 5000)))

```

To parallelize this example, it suffice to import SCOOP as previously shown and to replace the standard map of line 8 by the SCOOP map:

```

8     expr = futures.map(test, [n] * repeat)

```

4.2 Fork

The *fork* skeleton consists of calling multiple functions, each with its own arguments. This behavior is achieved with SCOOP using the *submit* function which returns a future. After calling the *wait* function on the future, it is possible to get its result by calling its method *result*.

4.3 Reduce and Scan

SCOOP allows two possibilities to implement the *reduce* and *scan* primitives. Once you have mapped a function, you can reduce it serially on a single worker by calling the standard *functools.reduce* function. To implement parallel reduction, it is possible to use the *mapReduce* or *mapScan* functions of SCOOP.

4.4 Farm and Pipe

The *Farm* skeleton represents a worker bag of tasks: spawned tasks are sent to workers which execute them. Tasks can be recursively called or replicated inside a Farm to produce a task hierarchy. The bag of tasks approach is the built-in scheduling and load balancing strategy in SCOOP. Using the `map` or `submit` functionalities will effectively produce a farm or pipe pattern under the hood, depending on how the user defined its task granularity.

4.5 Divide and Conquer

The *Divide and Conquer* skeleton is a problem solving approach that generates a tree-like task hierarchy. This is a perfect match for SCOOP's API. A simple example is the following function that recurses 12 levels down to compute the number of tree leaves:

```
1 from scoop import futures
2 DEPTH = 12
3 def recursiveFunc(level):
4     if level == 0:
5         return 1
6     else:
7         args = [level-1] * 2
8         s = sum(futures.map(recursiveFunc, args))
9         return s
10 if __name__ == "__main__":
11     result = recursiveFunc(DEPTH)
12     print("2^{DEPTH} = {result}".format(**locals()))
```

This code will spawn two tasks which will themselves spawn two subtasks each, etc.. This will continue until a depth of DEPTH is attained. This approach can be used to divide a given dataset or problem down to its base case.

4.6 Evolutionary Algorithm

Evolutionary Algorithms (EA) are population-based optimization algorithms that consist of individuals being evolved over generations. Every individual represents a potential solution to a problem which can be mutated or mixed with some other individuals of the same generation.

DEAP [4] is a Python framework allowing rapid prototyping of evolutionary algorithms. Once you have a working serial DEAP program, parallelizing it using SCOOP is done as follows:

```
1 from scoop import futures
2 toolbox.register("map", futures.map)
```

Every map done by DEAP's toolbox will then be parallel. Since DEAP's built-in `eaSimple` algorithm uses the toolbox's `map` function to evaluate the fitnesses of all individuals, our program automatically became parallel.

A common EA paradigm is the island model where multiple populations evolve separately and exchange individuals on a regular basis. Each island can be mapped to a task which itself generates tasks for the evaluation of individuals. This task hierarchy with a depth of two can be implemented in DEAP using SCOOP as such:

```
1 toolbox.register("algorithm", algorithms.eaSimple,
2                 toolbox=toolbox, cxpb=0.5,
3                 mutpb=0.2, ngen=FREQ)
4 islds = [
5     toolbox.population(n=300) for _ in range(5)
6 ]
7 for i in range(0, NGEN, FREQ):
8     results = toolbox.map(toolbox.algorithm, islds)
```

This code first registers the already parallelized `eaSimple` to the name `algorithm`. It then generates the initial islands and begins its evolution loop, which maps the islands to the algorithm. As previously mentioned, the `toolbox.map` function is an alias to SCOOP's `futures.map`. This will create one SCOOP future per island. The `eaSimple` algorithm will create a future for each individual in its island when mapping them to their evaluation function.

4.7 Working hosts

The list of working hosts can either be passed as an argument to the launcher (the `--hosts` argument) or be in an *hostlist* along the number of workers to spawn on it. Hosts can be defined by their IP address or their DNS names. Here is an example of host list:

1	desktop1	8
2	desktop2	8
3	10.10.1.7	4

Using this feature allows for easy use of fixed computer grids. It also can be used to get the resource list allocated from a custom scheduler.

5. RESULTS

The goal of parallel programming is to engage multiple resources simultaneously on the same job in order to perform it faster on the whole. SCOOP strives to offer this while simplifying prototyping and software development. To keep the framework flexible, it does not require the prior registration of tasks; in fact, the task hierarchy should be defined at runtime. Furthermore, computing resources can be added to the working pool while the computation takes place. The developer does not need to specify communication between resources: they are handled automatically by the framework. All these features induce unknown variables in parallelization algorithms. Interpreting these unknowns suboptimally generates undesired overheads.

We first use the previous Monte Carlo sampling method for computing the value of π to evaluate the performance of SCOOP on an embarrassingly parallel problems. The number of futures (tasks) created is set to 1800. Each of these futures performs 4 million tries, accounting for roughly 2 seconds on our test hardware. Scaling results are presented in figure 2. They are represented in terms of efficiency which is defined by:

$$E_p = \frac{T_1}{p \cdot T_p} \quad (1)$$

where E_p is the efficiency on p computing resources. T_1 and T_p represent respectively the serial and parallel execution times. The light degradation in performance when approaching 256 workers can be explained by the low total execution time, around 14 seconds for 256 workers. Since

the execution time of a future is around 2 seconds, the imbalance of a single future amounts for almost 15% of the execution time.

The execution timeline is shown at figure 3. The time where most workers are not effectively computing is at the end of the job, where a discrepancy of roughly 2 seconds is visible. The load imbalance thus represents the work time of a single task or less. The graphic shows a minimal amount of idle workers everywhere besides the job end.



Figure 2: SCOOP efficiency for computing π using the Monte Carlo sampling method.

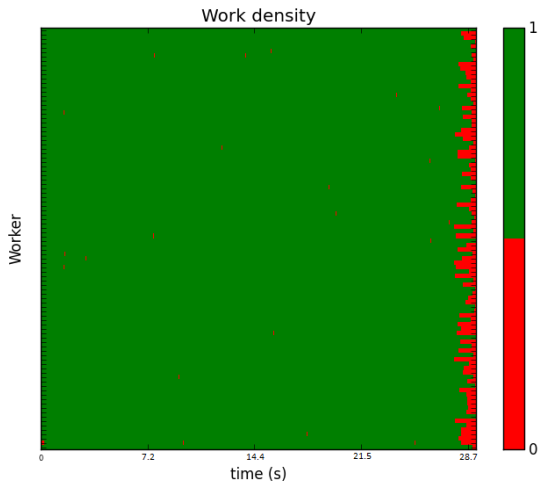


Figure 3: SCOOP's work density map for computing π on 96 workers using the Monte Carlo sampling method.

The second benchmarked program contains a tree shaped task hierarchy. Each task may submit up to four other tasks up to a depth of 10. The tree contains 2317 nodes, of which 1415 are leaves. Each of these tasks have a different execution time as shown on figure 4. The efficiency of SCOOP on this problem is shown at figure 5. The hierarchy depth map (figure 6) and the time and task distribution (figure 7) show the load balancing performance of SCOOP on this difficult job. It is worth nothing that the maximum theoretical parallel efficiency of this problem decreases when the number of workers increases because of hierarchichal constraints.

The load balancing is made to optimize the execution time, which explains the large variance in task distribution and relatively small execution time variance. The hierarchy depth map represents the number of tasks executing or awaiting for results at any given time. It gives insights on load distribution over the pool. A worker keeping a depth of one means that it only worked on the tree leaves. The higher the hierarchy depth is for a worker, the more it spawned tasks.

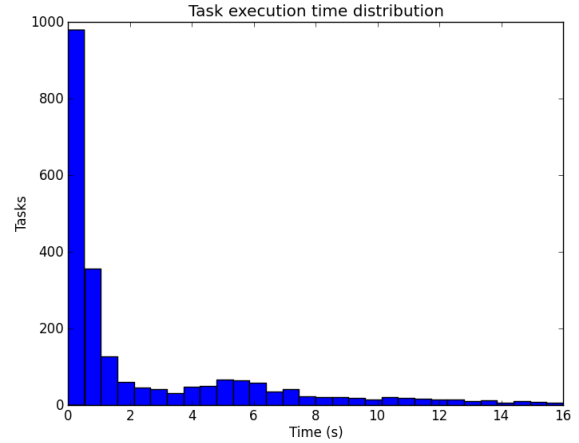


Figure 4: Task execution time distribution.

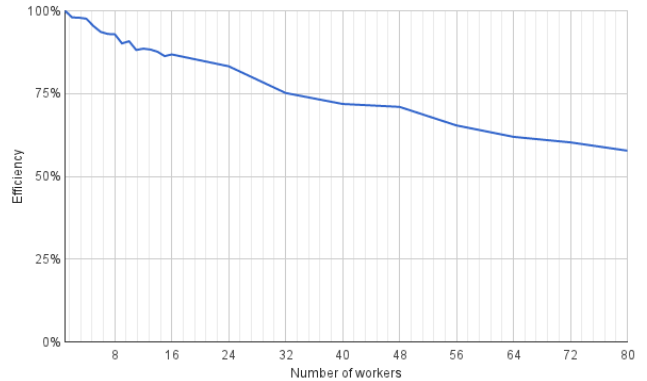


Figure 5: SCOOP's Efficiency on the tree program.

Figure 8 shows results from three different setups. The first one, called *local*, represents a single worker that keeps all of its tasks locally. It will not serialize them nor send them to the broker. The setup called *one node* represents a single machine on which two worker processes are running. The first process runs the root task that spawns all other tasks that are forced to run on the second worker. This stresses the loopback interface of the machine. For the *two node* setup, we again have two workers, but this time running on two distinct machines that communicate across a network. For both figures, tasks are spawn with a variable length argument in bytes, but the task function itself does nothing

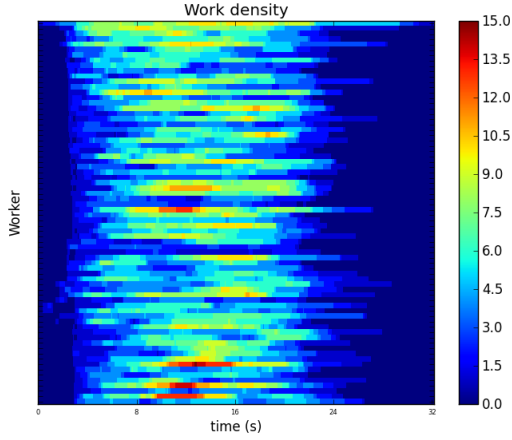


Figure 6: Worker hierarchy depth map of the tree program on 72 workers.

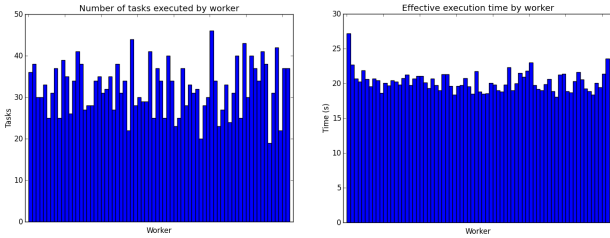


Figure 7: Task (left) and execution time (right) distribution of the tree program on 72 workers.

but return its argument to the caller. Execution time is hence negligible; only communication overheads count.

The figure is done by sending many tasks at once and dividing the task quantity by the time it took to execute them.

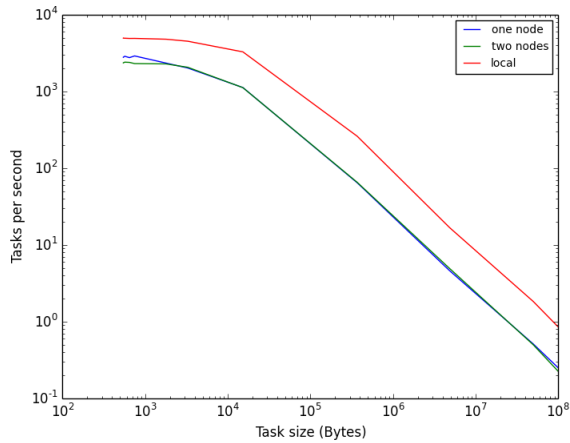


Figure 8: Task execution throughput

All of the tests were performed using Python 3.3.0, pyzmq 14.0.2-dev and zeromq 4.0.3 on 64 bits Linux machine(s),

each equipped with two Xeon X5560 2.8 Ghz processors and Infiniband QDR networking. This setup provides 8 real computing cores per node.

6. CONCLUSION

To lessen the overhead of converting a standard program to its parallel version, SCOOP allows the execution of parallel tasks defined in a serial module. In order to do so, it enforces a top-level script environment delimiter which differentiate between the root task code and object definitions needed on all workers. Some other frameworks require parallel tasks to be defined in separate modules to circumvent this requirement. This increases the adaptation overheads needed to parallelize standard algorithms.

Parallel programming is an active subject on which multiple tools are currently being developed. This paper introduced SCOOP, a parallel framework in Python striving for usage simplicity. This framework is still a work in progress with ongoing development, but it's already stable enough with a growing community. While sharing a few common features with other parallel frameworks, it proposes new ideas such as time-based statistics. Multiple improvements are forecast such as out-of-the-box support for cloud services, fault tolerance and performance through multiple brokers, network performance awareness and enhancements of the reduce functionality.

7. REFERENCES

- [1] K. Asanovic, R. Bodik, and B. Catanzaro. The landscape of parallel computing research: A view from Berkeley. *Vol. 2. Technical Report UCB/EECS-2006-183*, 2006.
- [2] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering*, pages 46–55, 1998.
- [3] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, pages 1–13, 2008.
- [4] F.-A. Fortin, F.-M. De Rainville, M.-A. Gardner, M. Parizeau, and C. Gagné. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13:2171–2175, 2012.
- [5] D. Friedman and D. Wise. *The impact of applicative programming on multiprocessing*. Indiana University, 1976.
- [6] H. González-Vélez and M. Leyton. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Software: Practice and Experience*, pages 1–26, 2010.
- [7] P. Hibbard. Parallel processing facilities. *New Directions in Algorithmic Languages*, 1976.
- [8] H. B. Jr and C. Hewitt. The incremental garbage collection of processes. *ACM SIGART Bulletin*, 1977.
- [9] L. Kale and S. Krishnan. *CHARM++: a portable concurrent object oriented system based on C++*. ACM, 1993.
- [10] T. E. Oliphant. Python for Scientific Computing. *Computing in Science & Engineering*, 9(3):10–20, 2007.
- [11] M. Snir, S. Otto, and D. Walker. *MPI: the complete reference*. MIT press, 1995.