SoftwareX 12 (2020) 100517

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

torcpy: Supporting task parallelism in Python

P.E. Hadjidoukas *, A. Bartezzaghi, F. Scheidegger, R. Istrate, C. Bekas [1], A.C.I. Malossi

IBM Research, Zurich, Switzerland

ARTICLE INFO

Article history:
Received 13 January 2020
Received in revised form 30 April 2020
Accepted 19 May 2020

Keywords:
Parallelism
Python
MPI
Multithreading

ABSTRACT

Task-based parallelism has been established as one of the main forms of code parallelization, where asynchronous tasks are launched and distributed across the processing units of a local machine, a cluster or a supercomputer. The tasks can be either completely decoupled, corresponding to a set of independent jobs, or be part of an iterative algorithm where the task results are processed and drive the next step. Typical use cases include the application of the same function to different data, parametric searches and algorithms used in numerical optimization and Bayesian uncertainty quantification. In this work, we introduce **torcpy**, a platform-agnostic adaptive load balancing library that orchestrates the asynchronous execution of tasks, expressed as callables with arguments, on both shared and distributed memory platforms. The library is implemented on top of MPI and multithreading and provides lightweight support for nested loops and map functions. Experimental results using representative applications demonstrate the flexibility and efficiency of the proposed Python package.

Code metadata

Current code version	v0.1.1
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX_2020_9
Code Ocean compute capsule	https://codeocean.com/capsule/9000816/
Legal Code License	EPL 1.0
Code versioning system used	git
Software code languages, tools, and services used	Python3, MPI
Compilation requirements, operating environments & dependencies	mpi4py with compatible MPI implementation, termcolor, coloredlogs, pytest, numpy, termcolor, cma, pillow
Link to developer documentation/manual	https://github.com/IBM/torc_py
Support email for questions	phadjido@gmail.com

1. Motivation and significance

The increasing heterogeneity of hardware and software in contemporary parallel computing platforms constitute task parallelism a natural way for exploiting their hierarchical architecture. Asynchronous tasks, either completely decoupled or part of an algorithm, can be spawned at different levels of parallelism and eventually executed by the available processing units. Due to the need for explicit communication, the Message Passing Interface (MPI) remains a dominant programming model for distributed memory systems. Meanwhile, the usage of Python constantly increases, mainly as a convenient platform for developing applications and algorithms that are connected through high-level interfaces to other languages and High-Performance Computing (HPC) libraries [1–4]. A typical application case that combines MPI and Python is distributed training of neural networks, as implemented in frameworks such as Keras [5] and PyTorch [6]. This combination, however, is not present in the available Python libraries that support task-based parallelism. Furthermore, most of these libraries lack efficient or even basic support of multilevel parallelism and do not export a unified runtime environment for sequential, shared memory and distributed memory execution.

* Corresponding author.
E-mail address: hat@zurich.ibm.com (P.E. Hadjidoukas).
[1] Current affiliation: Citadel Securities, LLC.

https://doi.org/10.1016/j.softx.2020.100517

Instead, they rely on techniques such as processor partitioning, distinction between master and worker processes and utilization of proxy servers for handling task scheduling.

Background. A typical example of task-based parallelism, in Python, can be found in the sequential execution of the same function (`work`) on a list of input data:

```
data = range(10)
results = []
for x in data:
  y = work(x)
  results.append(y)
```

Moreover, the for-loop can be replaced with the built-in `map` function:

```
data = range(10)
results = list(map(work, data))
```

Python provides two built-in solutions for expressing and executing parallelism: the `multiprocessing`[7] and `concurrent` [8] packages.

The `multiprocessing` package supports single-node parallelism by means of multiple processes, avoiding thus the effects of the Global Interpreter Lock. Task-based parallelism is supported either with the `apply_async` function or the more convenient `map` function of the Pool object, which distributes a set of multiple input values across the available processes and then applies the same function to these values in parallel.

An example of using the `apply_async` function for task submission to a pool of 4 processes is depicted below.

```
from multiprocessing import Pool
data = range(10)
p = Pool(4)
tasks = []
for i in data:
  t = p.apply_async(work, (i,))
  tasks.append(t)
for t in tasks:
  print(t.get())
```

The use of the `map` function, provided by the Pool object, simplifies significantly the code, which becomes very similar to its sequential version shown previously.

```
from multiprocessing import Pool
data = range(10)
p = Pool(4)
results = list(p.map(work, data))
```

The `concurrent.futures` module was first introduced in Python 3.2 to provide a simple interface for asynchronous execution of tasks, according to the Python Enhancement Proposal (PEP) 3148 [9]. This interface is supported by two types of executors, the ThreadPoolExecutor and the ProcessPoolExecutor classes, which are based on threads and processes, respectively. Task parallelism can be expressed either with the `submit/wait` functions or with parallel `map` function, as shown in the following two example codes. In both cases, ProcessPoolExecutor can be replaced with ThreadPoolExecutor, allowing for execution of the tasks by multiple threads instead of processes.

```
import concurrent.futures
data = range(10)
p = concurrent.futures.ProcessPoolExecutor(max_workers=4)
tasks = []
for i in data:
  tasks.append(p.submit(work, i))
concurrent.futures.wait(tasks)
for t in tasks:
  print(t.result())
```

Using the map() function:

```
data = range(10)
p = concurrent.futures.ProcessPoolExecutor(max_workers=4)
results = list(p.map(work, data))
```

Contribution. Unfortunately, both modules target single-node multi-core systems and cannot be used in high-performance computing environments such as clusters and supercomputers. In this paper, we introduce **torcpy**, a platform-agnostic adaptive load balancing library that orchestrates the scheduling of tasks on both shared and distributed memory platforms. As an open-source tasking library, **torcpy** aims at providing a parallel computing framework that:

- offers a unified approach for expressing and executing task-based parallelism on both shared and distributed memory platforms
- takes advantage of MPI internally in a transparent to the user way but also allows the use of legacy MPI code at the application level
- provides lightweight support for parallel nested loops and map functions
- supports task stealing at all levels of parallelism
- exports the above functionalities through a simple and single Python package

torcpy exports an interface similar to that of PEP 3148. Therefore tasks (futures) can be spawned and joined with the `submit` and `wait` calls. A parallel `map` function is provided, while the exported `spmd` function allows for switching to the typical Single Program Multiple Data (SPMD) execution mode that is natively supported by MPI.

The library is implemented on top of MPI and multithreading and it can be considered as the Python implementation of the TORC C/C++ runtime library [10]. TORC has been already used extensively on small and large scale HPC environments for a range of parallel applications such as: numerical differentiation [11,12], quantification of uncertainty in simulation models [13], numerical optimization for fitting interatomic potentials [14] and multi-objective optimization of artificial swimmers [15]. The same library has been also used at the core of the Π4U uncertainty quantification and optimization package [16] and the HOMPI hybrid programming framework [17].

As part of Π4U, TORC has been used extensively on small and large scale HPC environments such as the Euler multi-core cluster of ETH Zürich and the Piz Daint GPU supercomputer of the Swiss National Supercomputer Center (CSCS). For example, the TORC-based implementation of the Transitional Markov Chain Monte Carlo (TMCMC) method [16,18] has achieved an overall parallel efficiency of more than 90% on 1024 compute nodes of Piz Daint running hybrid MPI+GPU molecular simulation codes with highly variable time-to-solution between simulations for different interaction parameters. In all these cases, the task calls an external script that receives as input the function arguments, runs a simulation and returns the result back to the task. Since the core computational work is performed outside the task-parallel algorithm, it is obvious that the overall performance is not affected by the C/C++ or the Python implementation of the algorithm and the usage of the corresponding tasking library (TORC or **torcpy**).

Related work. There is a number of Python packages and framework that enable the orchestration and execution of task-based parallelism on various computing platforms. A brief summary of these packages is depicted in Table 1.

As mentioned before, on single-node multi-core systems Python provides two native solutions: the multiprocessing and the concurrent.futures modules. The futures package of mpi4py provides an extension of futures on top of the MPI programming model. This package follows the typical master–worker execution model and is suitable for single-level tasks that can be distributed through MPI among multiple nodes. The master process does not participate in the computations and, therefore, lightweight nested

Table 1
Summary of Python frameworks for task-based parallelism.

Framework	PEP 3148	Clusters	Nested parallelism	MPI SPMD
`multiprocessing`	No	No	No	No
`futures`	Yes	No	No	No
`mpi4py.futures`	Yes	Yes	No	No
`DTM (deap 0.9.2)`	No	Yes	Inefficiently (threads)	Yes
`SCOOP (0.7.1.1)`	Yes	Yes	Yes (coroutines)	No
`Parsl (0.9.0)`	No	Yes	Limited (partitioning)	No
`Celery (4.2.0)`	No	Yes	No	No
`Dask (1.2.2)`	No	Yes	Inefficiently (more workers)	No
`PyCOMPSs (2.6)`	No	Yes	Yes	No
`torcpy (0.1)`	Yes	Yes	Yes	Yes

parallelism and MPI SPMD code (e.g. collective MPI operations) is not supported.

DTM [19] is an obsolete MPI-based framework, part of the DEAP package [20], that supports task-based parallelism. Task spawning can be performed with the `apply_async/waitAll` or `map` calls and, in contrast to mpi4py futures, the master process of a DTM application participates in the execution of the spawned tasks. DTM also provides support of nested tasks, by means of kernel level threads. Due to thread-safety issues, MPI support was removed in the latest versions (>0.9.2) of the DEAP package and DTM was eventually discontinued. Moreover, DTM was replaced by SCOOP [21,22], which follows a more distributed-based approach without relying on MPI. Due to lightweight coroutines, SCOOP supports nested tasks efficiently. The communication is based either on sockets or a message-queue (broker), while global data are supported through containers.

Parsl [4,23] allows for scalable execution of task parallelism on a wide range of computing platforms by means of different executors tailored to low-latency, high-throughput or extreme-scale use cases. Parallelism in Parsl is expressed with Python decorators and its Extreme Scale Executor is composed of executor clients, interchange and workers. The communication between manager and workers is based on MPI while that between manager and interchange, where task queues are deployed, is performed with a message queue. MPI processes can be partitioned to multiple groups, where rank 0 becomes the manager and the rest processes serve as workers. Parsl provides limited support of nested parallelism because inner tasks can be executed only as long as workers are available, otherwise deadlock can occur. Moreover, direct support of MPI SPMD code is not supported.

Both Celery [24] and Dask [25,26] are frameworks that mainly target cloud computing environments. Both are message-queue based and follow a client–scheduler–worker approach. In Celery, client and task codes are decoupled and nested tasks are partially supported because they can be submitted for execution but there is no wait operation available. In Dask, nesting of tasks is supported only by means of additional worker threads.

Finally, PyCOMPSs [3,27] is the Python binding of COMPSs, a state-of-the-art HPC framework that supports several programming languages. The core runtime is implemented in Java and allows for tasks that include MPI code but does not support direct injection of MPI SPMD code in the task-parallel Python code.

2. Software description

A **torcpy**-based application consists of multiple MPI processes with one or multiple worker threads and a set of queues where tasks are submitted for execution according to the level of parallelism they are spawned from. Idle workers extract tasks for the queues, first by accessing their local set of queues and, if this is empty, the set of queues that reside remotely in the other MPI processes. Remote operations related to task and data management are performed with explicit, but completely transparent

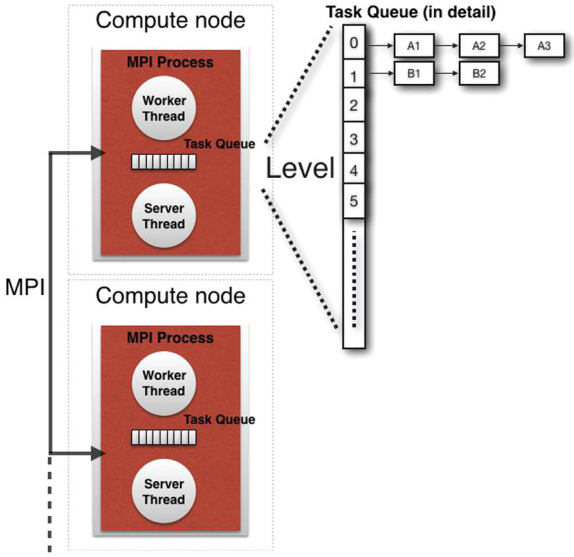

Fig. 1. Parallel architecture of the **torcpy** library.

to the user, messages to a dedicated server thread, utilized by each MPI process. If the application runs as a pure multithreaded code, i.e. with one MPI process and multiple worker threads, then all operations are performed exclusively through shared memory. The software architecture of **torcpy** is depicted in Fig. 1, assuming a single MPI process per node. In the general case, a compute node can execute multiple MPI processes and, therefore, the terms node and process can be used interchangeably.

2.1. Software architecture

torcpy is built on top of the mpi4py package and makes use of the threading and queue modules. Tasks are instantiated as Python dictionaries, which incur less overhead than objects. The result of a task function is transparently stored in the task descriptor (future) on the MPI process that spawned the task. According to PEP 3148, the result can be then accessed as `task.result()`. Similarly, the input parameters can be accessed as `task.input()`.

All remote operations are performed asynchronously through the server thread. This thread is responsible for:

- inserting incoming tasks to the local queue of the process
- receiving the completed tasks and their results
- serving task stealing requests

An idle worker always tries first to extract work from the highest-level non-empty local queue. If no work is found, it tries to steal tasks from the remote queues. Specifically, it sends a synchronous stealing request to the server thread of the next, according to the rank, MPI process and continues until an available task has been returned or all processes have been accessed. The server thread search for work starting from the lowest-level local queue, i.e. for tasks at the innermost level of parallelism. The default task spawning policy distributes first-level tasks cyclically among the workers and submits inner-level tasks locally. Combined with task stealing, this policy favors stealing of coarse-grain tasks and local execution of deeper levels of parallelism

2.2. Software functionalities

torcpy exports an application programming interface (API) that includes task-management routines similar to those defined

by PEP 3184. In addition, it provides routines that allow the user to setup the library, control the execution mode (master–worker or SPMD), control the task stealing mechanism and query the runtime environment. Moreover, some environment variables can be used to specify the number of worker threads per process, the initial status of the task stealing mechanism and for how long idle threads release the processor. An overview of the most important parts of the API follows.

2.2.1. Library routines

- `start(f)`: initializes the library and launches function `f()` on process with rank 0 as the primary application task. When `f()` completes, it shutdowns the library. It is a collective function that must be called within `__main__`.
- `submit(f, *args, qid=-1, callback=None, **kwargs)`: submits a new task that corresponds to the asynchronous execution of function `f()` with input arguments `args`. The task is submitted to the worker with global identifier `qid`. If `qid` is equal to -1, then cyclic distribution of tasks to workers is performed. The callback function is called on the rank that spawned the task, when the task completes and its results have been returned to that node.
- `map(f, *seq, chunksize=1)`: executes function `f()` on a sequence (list) of arguments. It returns a list with the results of all tasks.
- `wait(tasks=None)`: the current task waits for *all* its child tasks to finish. The underlying worker thread is released and can execute other tasks.
- `spmd(f, *args)`: executes function `f()` on all MPI processes. It allows for dynamic switching from the master–worker to the SPMD execution mode, allowing thus legacy MPI code to be used within the function.

2.2.2. Environment variables

- `TORCPY_WORKERS (integer)`: number of worker threads used by each MPI processor. (default value: 1)
- `TORCPY_STEALING (boolean)`: determines if internode task-stealing is enabled or not. (default value: "False")
- `TORCPY_SERVER_YIELDTIME (float)`: for how many seconds an idle server thread will sleep releasing the processor. (default value: 0.01)
- `TORCPY_WORKER_YIELDTIME (float)`: for how many seconds an idle worker thread will sleep releasing the processor. (default value: 0.01)

The performance of **torcpy** was initially tuned using existing benchmark codes available in the mpi4py.futures package (such as run_julia.py and perf_primes.py). We observed that blocking MPI_Recv calls must be avoided by the server thread. Instead non-blocking calls combined with processor yielding is essential because the server thread allows the worker threads to progress faster. However, if there are multiple worker threads that execute Python bytecode, the performance will be still affected by the Global Interpreter Lock (GIL).

3. Illustrative examples

Listing 1 shows a commonly used parallelization example of recursive Fibonacci, where several levels of parallelism are exploited by spawning recursively two tasks and waiting for their results. The code is executed sequentially after some depth without creating thus an excessive number of tasks. The example demonstrates the support and exploitation of recursive parallelism. As such, it does not reuse the results of function calls with the same input argument and creates a second task (task2) instead of calling the function directly.

```
import torcpy

def fib(n):
  if n == 0:
    result = 0
  elif n == 1:
    result = 1
  else:
    n_1 = n   1
    n_2 = n   2
    if n < 30:
      result1 = fib(n_1)
      result2 = fib(n_2)
      result = result1 + result2
    else:
      task1 = torcpy.submit(fib, n_1)
      task2 = torcpy.submit(fib, n_2)
      torcpy.wait()
      result = task1.result() + task2.result()

  return result

def main():
  n = 35
  result = fib(n)

  print("fib({})={}".format(n, result))

if __name__ == "__main__" :
  torcpy.start(main)
```

Listing 1: Recursive Fibonacci

The second example (Listing 2) demonstrates the usage of MPI SPMD code within a task-parallel application. The global array A is initialized by the primary application task (main) on MPI process 0. Next, the `spmd` function triggers the execution of `bcast_task` on all MPI processes, thus switching to the SPMD execution model and allowing for direct data broadcast using `Bcast` of mpi4py.

```
import numpy
from mpi4py import MPI
import torcpy

N = 3
A = numpy.zeros(N, dtype=numpy.float64)

def bcast_task(root):
  global A
  comm = MPI.COMM_WORLD
  # Broadcast A from rank 0
  comm.Bcast([A, MPI.DOUBLE], root=root)

def work():
  global A
  print("node:{}:A={}".format(torcpy.node_id(), A))

def main():
  global A

  # primary task initializes array A on rank 0
  for i in range(0, N):
    A[i] = 100*i

  # switch to SPMD
  torcpy.spmd(bcast_task, torcpy.node_id())

  # single primary task continues here

if __name__ == "__main__" :
  torcpy.start(main)
```

Listing 2: Switching to SPMD execution mode

4. Performance evaluation

A typical preprocessing stage of Deep Learning workloads includes the transformation of datasets of raw images to a single file in HDF5 format. The images are organized in subfolders, where the name of each subfolder denotes the label of the enclosed

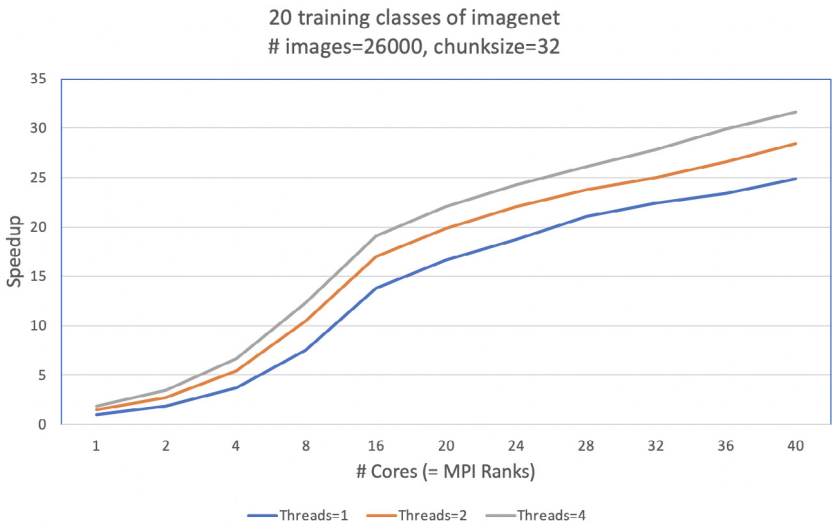

Fig. 2. Performance of parallel preprocessing on two IBM Power S822LC compute nodes.

images. For each image, the file is opened and the binary data are loaded to a buffer (numpy array). These operations include data decompression if the image is stored in JPEG format. Then, the image is resized and rescaled and additional preprocessing filters might be also applied. Finally, the result is written to an HDF5 file, that we will be used at the training phase of deep learning.

```
def process_train_image(i, target_dim):
  global reader, pipe

  # load the image and its label
  im, label = reader.get_train_image(i)

  # apply preprocessing filters
  im = pipe.filter(im)

  # resize accordingly
  dx = image_to_4d_tensor(im, target_dim)
  dy = label

  # return results
  return dx, dy
```

The parallelization of the sequential for loop is performed with the map function, using a chunk size of 32 so as to reduce the number of spawned tasks.

```
seq_i = range(n_train)
seq_t = [target_dim] * n_train

# parallel map with chunksize
task_results = torcpy.map(process_train_image, chunksize, seq_i, seq_t)

# write the results to the HDF5 dataset
i = 0
for t in task_results:
  dx, dy = t
  dataset_x[i, :, :, :] = dx
  dataset_y[i] = dy
  i = i+1
```

Imagenet [28] is very large dataset of 1000 classes, each with 1300 images stored in JPEG format. We preprocess the images of the (alphabetically) first 20 training classes of the Imagenet dataset. We perform our experiments on two IBM[2] Power S822LC (8335-GTA) [29] compute nodes. Each node has two POWER8 processor sockets, which are equipped with 256GB RAM and 10 cores, each with 8 hardware threads and 8MB L3 cache. The software configuration includes Python 3.6.9, mpi4py/3.0.1 and IBM Spectrum MPI 10.3.0. The command line for executing the benchmark with `NR` MPI processes with `TORCPY_WORKERS` worker threads each, is as follows:

```
mpirun -n $NR -x TORCPY_WORKERS=$NW --bind-to core --map-by socket \
        --mca btl self,tcp  python benchmark.py
```

The measurements include the time for spawning the parallelism, executing the preprocessing in parallel and waiting for the completion of all tasks, i.e. collecting the results back. In Fig. 2, we observe that the application exhibits good scaling and achieves approximately 62% efficiency when 40 processes of one worker thread each are used. The performance does not scale linearly with the number of cores as image decompression and processing stress the memory subsystem of the node. We also observe that multithreading further improves the performance, allowing for a maximum achieved speedup of 31.6x (40 processes, 4 threads).

5. Impact

The increased popularity of Python, in combination with its extensive usage in machine learning, strongly motivated the introduction of **torcpy**, the pure Python implementation of a high-performance tasking library. This motivation was further supported by the absence of a task-parallel library that: supports the futures API of PEP 3148, allows for arbitrary nesting of tasks by decoupling them from the actual workers, takes advantage of MPI but also enables the direct integration of MPI code, provides transparent load balancing through task stealing and adapts automatically to the underlying execution environment.

6. Conclusions

torcpy is a Python library that uses multithreading and MPI to support multi-level task-based parallelism on shared and distributed memory platforms. It provides transparent data movement and load balancing hiding the details of explicit communication to the users. The same Python script can be executed with multiple threads, processes or in hybrid mode, similarly to the MPI+X programming model. The library exploits internally MPI and also extends it with lightweight task-based capabilities. Furthermore, it is orthogonal to MPI, allowing dynamic switching to the SPMD execution mode and, thus integration of legacy MPI code.

[2] IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" http://www.ibm.com/legal/copytrade.shtml.

Our current work focuses on improved compatibility of PEP 3148 and support of exception handling in the task functions. Moreover, we aim at the efficient parallel execution of small and medium-size deep learning training tasks on clusters of GPUs.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 760907, Virtual Materials Marketplace (VIMMP).

References

[1] Vincent P, Witherden F, Vermeire B, Park JS, Iyer A. Towards green aviation with python at petascale. In: SC'16: Proceedings of the international conference for high performance computing, networking, storage and analysis. IEEE; 2016, p. 1–11.
[2] Conejero J, Ramon-Cortes C, Serradell K, Badia RM. Boosting atmospheric dust forecast with pycompss. In: 2018 IEEE 14th international conference on E-science (E-Science). 2018, p. 464–74.
[3] Tejedor E, Becerra Y, Alomar G, Queralt A, Badia RM, Torres J, Cortes T, Labarta J. PyCOMPSs: Parallel computational workflows in Python. Int J High Perform Comput Appl 2017;31(1):66–82.
[4] Babuji Y, Woodard A, Li Z, Katz DS, Clifford B, Kumar R, Lacinski L, Chard R, Wozniak JM, Foster I, Wilde M, Chard K. Parsl: Pervasive parallel programming in Python. In: 28th ACM international symposium on high-performance parallel and distributed computing (HPDC). HPDC '19, 2019, p. 25–36.
[5] Keras: The Python Deep Learning library. https://keras.io/.
[6] PyTorch: From research to production. https://pytorch.org/.
[7] Python standard library, Python multiprocessing. https://docs.python.org/3/library/multiprocessing.html.
[8] Python standard library, Python concurrent.futures. https://docs.python.org/3/library/concurrent.futures.html.
[9] PEP 3148 – futures - execute computations asynchronously. https://www.python.org/dev/peps/pep-3148/.
[10] Hadjidoukas PE, Lappas E, Dimakopoulos VV. A runtime library for platform-independent task parallelism. In: 20th international conference on parallel, distributed and network-based processing (PDP). Euromicro; 2012, p. 229–36.
[11] Voglis C, Hadjidoukas PE, Lagaris IE, Papageorgiou DG. A numerical differentiation library exploiting parallel architectures. Comput Phys Comm 2009;180(8):1404–15.
[12] Hadjidoukas PE, Angelikopoulos P, Voglis C, Papageorgiou DG, Lagaris IE. NDL-v2.0: A new version of the numerical differentiation library for parallel architectures. Comput Phys Comm 2014;185(7):2217–9.
[13] Hadjidoukas PE, Angelikopoulos P, Rossinelli D, Alexeev C, Papadimitriou C, Koumoutsakos P. Bayesian uncertainty quantification and propagation for discrete element simulations of granular materials. Comput Methods Appl Mech Engrg 2014;282:218–38.
[14] Voglis C, Hadjidoukas PE, Papageorgiou DG, Lagaris IE. A parallel hybrid optimization algorithm for fitting interatomic potentials. Appl Soft Comput 2013;13(12):4481–92.
[15] Verma S, Hadjidoukas PE, Wirth P, Koumoutsakos P. Multi-objective optimization of artificial swimmers. In: 2017 IEEE congress on evolutionary computation, CEC 2017, Donostia, San SebastiáN, Spain. 2017, p. 1037–46.
[16] Hadjidoukas PE, Angelikopoulos P, Papadimitriou C, Koumoutsakos P. Π4U: A high performance computing framework for Bayesian uncertainty quantification of complex models. J Comput Phys 2015;284:1–21.
[17] Dimakopoulos VV, Hadjidoukas PE. HOMPI: A hybrid programming framework for expressing and deploying task-based parallelism. In: Euro-Par 2011 parallel processing. 2011, p. 14–26.
[18] J. Y. Ching J, Chen YC. Transitional Markov chain Monte Carlo method for Bayesian model updating, model class selection, and model averaging. J Eng Mech 2007;133(7):816–32.
[19] DTM, http://deap.gel.ulaval.ca/doc/0.9/api/dtm.html.
[20] Fortin F-A, De Rainville F-M, Gardner M-A, Parizeau M, Gagné C. DEAP: Evolutionary algorithms made easy. J Mach Learn Res 2012;13:2171–5.
[21] SCOOP, https://github.com/soravux/scoop/.
[22] Hold-Geoffroy Y, Gagnon O, Parizeau M. Once you SCOOP, no need to fork. In: 2014 annual conference on extreme science and engineering discovery environment. ACM; 2014, p. 60.
[23] Parsl: Productive parallel programming in Python. https://github.com/Parsl/parsl.
[24] Celery: Distributed Tasks Queue. https://github.com/celery/celery.
[25] Dask, https://github.com/dask/dask.
[26] Rocklin M. Dask: Parallel computation with blocked algorithms and task scheduling. In: 14th python in science conference. 2015, p. 130–6.
[27] PyCOMPSs, https://pypi.org/project/pycompss/.
[28] Deng J, Dong W, Socher R, Li L-J, Li K, Fei-Fei L. ImageNet: A large-scale hierarchical image database. In: CVPR09. 2009.
[29] Caldeira A, E. KM, Saverimuthu G, Vearner KC. IBM power systems S822LC technical overview and introduction. IBM Redbooks; 2015.