Journal of
open research software

Taskfarm: A Client/Server Framework for Supporting Massive Embarrassingly Parallel Workloads

SOFTWARE
METAPAPER

MAGNUS HAGDORN
NOEL GOURMELEN
*Author affiliations can be found in the back matter of this article

]u[ubiquity press

ABSTRACT

Taskfarm is a client/server framework that can be used to keep track of massive embarrassingly parallel workloads. The system is split up into two packages: (1) a flask server that hands out new tasks via HTTP and (2) a python client that requests and updates tasks. The server stores task progress in a database. This system has been designed to manage a satellite data processing workflow with hundreds of thousands of tasks with variable compute costs. It can be used for any problem that can be solved using a task farm.

CORRESPONDING AUTHOR:
Magnus Hagdorn
School of GeoSciences, University of Edinburgh, GB
magnus.hagdorn@ed.ac.uk

KEYWORDS:
open source; high performance computing; client/server framework; task farm; embarrassingly parallel workload

TO CITE THIS ARTICLE:
Hagdorn M, Gourmelen N 2023 Taskfarm: A Client/Server Framework for Supporting Massive Embarrassingly Parallel Workloads. *Journal of Open Research Software*, 11: 1. DOI: https://doi.org/10.5334/jors.393

(1) OVERVIEW

INTRODUCTION

Broadly speaking there are three approaches to distributing work for parallel computing: 1) decomposing computations, 2) decomposing data and 3) decomposing tasks [3]. Tightly coupled problems such as matrix–matrix computations fall into the first category and are typically solved using shared memory systems. Partial differential equations arising for example in fluid dynamics are an example of the second category. They are typically solved by decomposing the domain and distributing data across the nodes of the parallel system. Processing 1000s of images to extract some quantity is an example of the third category where each image can be processed independently. Loosely coupled problems can be solved using a task farm [1] where one process, the *farmer*, hands out *tasks* to the *workers*. There is no communication between the workers and tasks can be solved in any order. The problem is solved once all tasks have been processed. Problems like this are also know as *embarrassingly parallel workloads*.

Task farms come in all sizes. Resizing a few files can be done with a shell loop sending each process into the background. If a few hundred images need to be resized a tool like GNU parallel can be used. Larger problems involving multiple nodes in a cluster need some form of coordination: a farmer process keeps track of all the tasks and hands out new tasks to workers running on different nodes. The farmer needs to be able to communicate with the workers to give them new tasks. One approach to do this is to write a parallel program using MPI. One MPI process is the farmer that sends a new task to a worker process upon request. Workers can also notify the farmer on progress of each task.

Using MPI has the advantage that it is available and well understood on scientific HPC clusters and integrates well with middleware such the grid engine or SLURM. It has the disadvantage that one process needs to be dedicated to handing out tasks. More critically, MPI programs tend to have a fixed number of processes and depending on the cluster, requesting a large MPI job might take a long time. Furthermore, there is an issue with resilience where a single failure arising for example through resource limits being exceeded or a hardware failure takes down the entire task farm. An MPI task farm does not make as much use of the embarrassingly parallel nature of the problem as it could.

Another approach to implementing a task farm on a cluster is to make use of the middleware and use array jobs. Array jobs are used to schedule the same program with a different job ID. Each job can then figure out what task it needs to run from the job ID. This approach has the advantage that the task farm can grow and shrink dynamically depending on the availability of cluster resources. It has the disadvantage that keeping track of overall progress becomes difficult when individual process are expected to occasionally fail. Also some tasks might be so small that the scheduling overhead is much larger than the task itself.

This paper describes a framework that can be used to create a task farm. It was designed to grid raw satellite data. The dataset consists of time-dependent observation of ice elevation obtained from CryoSat interferometric radar altimeter [9]. This dataset is used with success to determine change in ice mass and ice dynamic, and to determine the contribution of land-ice masses to sea level change [7, 8, 6]. CryoSat data are pre-processed using a swath processing algorithm [5] which enhanced data coverage. The data are then processed into rates of elevation change using a plane fit approach [4].

The task farm framework needs to

- handle 100 000s of tasks
- keep track of the status of each task
- make full use of the embarrassingly parallel nature of the problem
- be able to handle tasks of varying computational complexity
- be able to identify and restart individual tasks that might have failed

IMPLEMENTATION AND ARCHITECTURE

The requirements of making full use of the embarrassingly parallel nature of the problem and being able to easily track the status of individual tasks and the entire problem suggests a web application backed by a database. Web applications can scale to thousands of request per second. Each client requests a new task and updates it via HTTP requests. The clients are thus completely independent of each other. The role of the farmer is completely decoupled from the project and moved into the web application. A special client can be used to query the state of each task or the entire project by querying the database. What follows is a detailed description of the task farm server web application and the task farm python client.

The task farm server can support multiple *runs*. A run consists of a number of *tasks* that need to be completed to solve some problem. A UUID is automatically assigned to a new run. Each task is a single item of work that forms part of a run. A task is identified by its number starting from 0 for the first task of a run to `numTasks-1` for the last task. Each task can be in one of the following states:

waiting: the task is waiting to be scheduled,
computing: the task is being computed, and
done: the task is completed.

The task table also records the percentage completed, the start time and the time when a task was last updated.

The taskfarm server only stores the state of each task. It is up to the clients to manage any data access that they need to process the tasks.

The source code of both the server and client package is available on github. Installable packages are also available via pypi.org. The packages are called **taskfarm** and **taskfarm-worker**, respectively.

TASK FARM SERVER

The task farm server is a RESTful [2] python web application based on the flask framework. It uses the SQLAlchemy python database toolkit together with the flask-sqlalchemy module to handle database operations. SQLAlchemy supports many database backends. For testing the single file database engine SQLite is sufficient. For production runs with multiple flask workers PostgreSQL should be used. PostgreSQL supports row locking and thus multiple workers can access the database safely.

The database connection is configured by setting the `DATABASE_URL` environment variable, e.g.

export `DATABASE_URL=sqlite:///app.db`
or
export `DATABASE_URL=postgresql://user:pw@host/db`

where `user` is the database user, `pw` the associated password, `host` the host name of the machine running the postgres server and `db` the name of the database to connect to. If you are using postgres you will need to create a database first. The database is automatically created if you are using sqlite. Once the database connection has been configured the database needs to be initialised by running

adminTF –init-db

All REST API calls to the task farm server are authenticated. Task farm users are created using

adminTF -u username -p password

where `username` is the name of the user and `password` the associated password.

The flask application uses the flask-httpauth module for basic HTTP authentication. Initially, the user is authenticated using the password. On success, a token is returned which is used for all further authentication.

Table 1 shows a summary of all REST API calls. Data are exchanged using JSON objects. See the online documentation for details of the API.[1]

Flask applications come with a built-in lightweight server which is sufficient for testing. In production, a flask application should be run through a WSGI server. The flask documentation gives details for the many options available. One option is to use the Gunicorn WSGI server.

Installation of the taskfarm server requires a database and a webserver. The repository on github contains configuration files to run the web application including database and webserver as a containerised service using docker. Detailed installation instructions can be found online.[2]

TASK FARM CLIENT

Rather than calling the HTTP API directly you can use the task farm client which is also implemented in python. It provides a set of proxy classes that perform the HTTP API calls to the task farm server. It also comes with a tool that can be used to manage runs.

A new run can be created by instantiating a `TaskFarm` object with the number of tasks passed as a parameter.

Each worker process needs to instantiate a `TaskFarmWorker` object using a username and password and the UUID of the run. This registers the worker with the task farm server. The `TaskFarmWorker` object provides an iterator that is used to obtain new tasks to be processed. The server only hands out tasks that are in the `waiting` state. Once a task has been handed out it is in the `computing` state. It is up to the client to update the server with the task's progress and finally mark it as completed. Once a task has been completed it is in the `done` state.

METHOD	URL	DESCRIPTION
POST	/api/run	create a new run
GET	/api/runs	get a list of all runs
POST	/api/runs/UUID/restart	restart all tasks of a run UUID
GET	/api/runs/UUID/tasks/ID	get information of task with ID of run UUID
PUT	/api/runs/UUID/tasks/ID	update task with ID of run UUID
POST	/api/runs/UUID/task	request a task for run UUID
GET	/api/runs/UUID	get information about run UUID
DELETE	/api/runs/UUID	delete run UUID
GET	/api/token	get the authentication token
POST	/api/worker	create a worker

Table 1 Task farm server REST API URLs.

A skeleton worker might look like:

```
from taskfarm_worker import TaskFarmWorker

# connect to taskfarm
tf = TaskFarmWorker ('user','secret',
                     'da8eb1c10eac4cefb39c8889d6d7170a',
                     url_base='http://localhost:5000/api/')
print (tf.percentDone)

# loop over the tasks
for t in tf.tasks:
    print ("worker _{}_ processing task _{}"
           .format (tf.worker_uuid,t))
    # do some work
    # update the percentage done
    tf.update (50)
    # do some more work and update percentage
    tf.update (100)
    # mark task as completed
    tf.done ()
```

The `TaskFarm` class can also be used to query an existing run. Class attributes are used to provide information on how many tasks have been completed and how many tasks are being computed.

There is no way of automatically detecting a crashed worker. So some tasks might never be marked as completed. The restart method can be used to mark all tasks in the `computing` state as `waiting`.

The `manageTF` utility can be used to create new runs, list existing runs and query a particular run. It can also be used to reset a run or delete it.

QUALITY CONTROL

Both server and client come with a test suite that can be automatically run using pytest. The task farm server API is tested using the flask-testing unit testing framework. The client test suite uses the requests-mock package to mock the API calls. Both server and client documentation is generated using sphinx.

The server and client packages are supported via their github issue trackers.

The taskfarm-worker package contains an example script that configures a task farm run and launches a number of task farm workers. This script can be used to investigate the overhead of tracking tasks. The task farm server is launched using the Gunicorn WSGI server with 1, 2, 4, 8 and 16 worker processes. Gunicorn redirects incoming requests to one of the workers which in turn is directly connected to a postgres database server hosted on a different virtual machine. A number of task farm clients are used to complete a run consisting of 256 tasks. Each client requests a new task. It then sends ten progess updates to the server and finally marks it as complete. Figure 1 shows the average time each client takes until all tasks are completed and the average time to update the server with the progress of a task.

This test does not involve any computation so the runtime depends on the network and how fast the webserver can process requests and how fast the flask workers can communicate with the database server. Performance clearly depends on the webserver when only 1 or 2 Gunicorn workers are used. The overall runtime does not improve significantly when more than 4 workers are used. When more than 16 task farm clients are used (see Figure 1b) update time increase even for 16 Gunicorn workers. This suggests that the postgres database struggles to keep up with the updates from the Gunicorn workers. These tests are clearly unrealistic since no computational work is done by any of the tasks and task farm requests will happen at the same time. For more realistic examples the computational effort should be much larger than the communication effort.

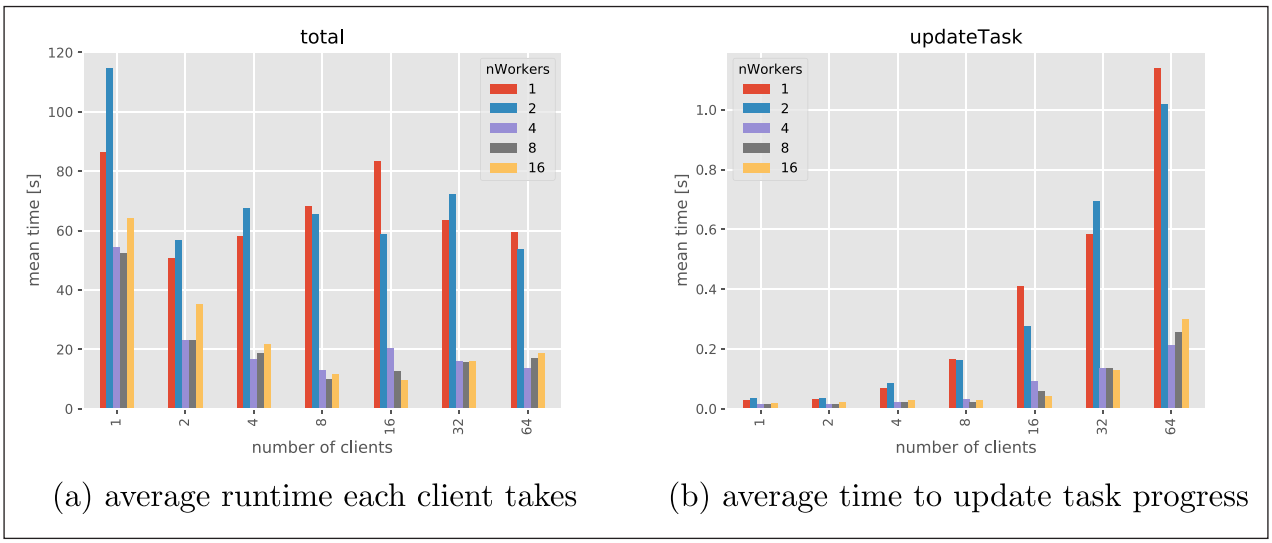

Figure 1 (a) The average time taken by each worker process to solve a total of 256 tasks when the task farm server uses 1, 2, 4, 8 or 16 worker processes. **(b)** The average time it takes to update the progress of a task.

(2) AVAILABILITY

OPERATING SYSTEM

Both the task farm server and client packages are pure python packages, so they should run on any operating system that supports python. The system has been tested on Scientific Linux 7 and Ubuntu 20.04 systems.

PROGRAMMING LANGUAGE

Both task farm server and client require python 3.

ADDITIONAL SYSTEM REQUIREMENTS

The task farm server needs to communicate with a database. For testing purposes SQLite is sufficient. For production runs a postgres database should be used. Other databases with SQLAlchemy support might work as well. Also for production a WSGI server together with a webserver should be used.

DEPENDENCIES

Task Farm Server

- flask
- flask-sqlalchemy
- flask-httpauth
- flask-testing
- itsdangerous
- passlib
- authlib

Task Farm Client

- requests
- requests-mock
- testtools
- fixtures

LIST OF CONTRIBUTORS

Magnus Hagdorn

SOFTWARE LOCATION

Task Farm Server

Archive

Name: zenodo
Persistent identifier: https://doi.org/10.5281/zenodo.7010122
Licence: GPL-3
Publisher: Magnus Hagdorn
Version published: 0.4.0
Date published: 19/08/2022
Name: PyPI
Persistent identifier: https://pypi.org/project/taskfarm/
Licence: GPL-3
Version published: 0.4.0
Date published: 19/08/2022

Code repository

Name: github
Persistent identifier: https://github.com/mhagdorn/taskfarm
Licence: GPL-3
Date published: 19/08/2022

Task Farm Client

Archive

Name: zenodo
Persistent identifier: https://doi.org/10.5281/zenodo.7010132
Licence: GPL-3
Publisher: Magnus Hagdorn
Version published: 0.4.0
Date published: 19/08/2022
Name: PyPI
Persistent identifier: https://pypi.org/project/taskfarm-worker/
Licence: GPL-3
Version published: 0.4.0
Date published: 19/08/2022

Code repository

Name: github
Persistent identifier: https://github.com/mhagdorn/taskfarm-worker
Licence: GPL-3
Date published: 19/08/2021

LANGUAGE

English

(3) REUSE POTENTIAL

Any embarrassingly parallel problem can be solved with a task farm approach. This project has been designed with a very large number of non-homogeneous tasks in mind. However, this approach can also be used for smaller, homogeneous problems. The task farm client communicates with the server via HTTP using a RESTful API. Although this client is implemented in python it could also be implemented in another language with a suitable HTTP client library.

NOTES

[1] https://taskfarm.readthedocs.io/.
[2] https://taskfarm.readthedocs.io/en/latest/installation.html.

ACKNOWLEDGEMENTS

The authors would like to thank Chris Hill for trying out the installation instructions and the anonymous reviewers for their very helpful comments.

FUNDING STATEMENT

This work was supported by European Space Agency contract CryoTop evolution 4000116874/16/I-NB.

COMPETING INTERESTS

The authors have no competing interests to declare.

AUTHOR AFFILIATIONS

Magnus Hagdorn
School of GeoSciences, University of Edinburgh, GB
Noel Gourmelen orcid.org/0000-0003-3346-9289
School of GeoSciences, University of Edinburgh, GB

REFERENCES

1. **Aldinucci M, Danelutto M.** Stream parallel skeleton optimization. In *Proceedings of the 11th IASTED International Conference on Parallel and Distributed Computing and Systems, MIT*. 1999; 955–962. IASTED/ACTA press.
2. **Amundsen M, Ruby S, Richardson L.** *RESTful Web APIs*. O'Reilly; 2013.
3. **Dowd K, Severance C.** *High Performance Computing*. O'Reilly, 2nd edition; 1998.
4. **Foresta L, Gourmelen N, Pálsson F, Nienow P, Björnsson H, Shepherd A.** Surface elevation change and mass balance of icelandic ice caps derived from swath mode cryosat-2 altimetry. *Geophysical Research Letters*. 2016; 43. ISSN 19448007. DOI: https://doi.org/10.1002/2016GL071485
5. **Gourmelen N, Escorihuela MJ, Shepherd A, Foresta L, Muir A, Garcia-Mondéjar A, Roca M, Baker SG, Drinkwater MR.** Cryosat-2 swath interferometric altimetry for mapping ice elevation and elevation change. *Advances in Space Research*. 2018; 62. ISSN 18791948. DOI: https://doi.org/10.1016/j.asr.2017.11.014
6. **Jakob L, Gourmelen N, Ewart M, Plummer S.** Spatially and temporally resolved ice loss in high mountain asia and the gulf of alaska observed by cryosat-2 swath altimetry between 2010 and 2019. *Cryosphere*. 2021; 15. ISSN 19940424. DOI: https://doi.org/10.5194/tc-15-1845-2021
7. **Shepherd A, Ivins E, Rignot E, Smith B, Broeke MVD, Velicogna I, Whitehouse P, Briggs K, Joughin I, Krinner G, Nowicki S, Payne T, Scambos T, Schlegel N, Geruo A, Agosta C, Ahlstrøm A, Babonis G, Barletta V, Blazquez A, Bonin J, Csatho B, Cullather R, Felikson D, Fettweis X, Forsberg R, Gallee H, Gardner A, Gilbert L, Groh A, Gunter B, Hanna E, Harig C, Helm V, Horvath A, Horwath M, Khan S, Kjeldsen KK, Konrad H, Langen P, Lecavalier B, Loomis B, Luthcke S, McMillan M, Melini D, Mernild S, Mohajerani Y, Moore P, Mouginot J, Moyano G, Muir A, Nagler T, Nield G, Nilsson J, Noel B, Otosaka I, Pattle ME, Peltier WR, Pie N, Rietbroek R, Rott H, Sandberg-Sørensen L, Sasgen I, Save H, Scheuchl B, Schrama E, Schröder L, Seo KW, Simonsen S, Slaters T, Spada G, Sutterley T, Talpe M, Tarasov L, Berg WJVD, Wal WVD, Wessem MV, Vishwakarma BD, Wiese D, Wouters B.** Mass balance of the antarctic ice sheet from 1992 to 2017. *Nature*. 2018; 558. ISSN 14764687. DOI: https://doi.org/10.1038/s41586-018-0179-y
8. **Shepherd A, Ivins E, Rignot E, Smith B, van den Broeke M, Velicogna I, Whitehouse P, Briggs K, Joughin I, Krinner G, Nowicki S, Payne T, Scambos T, Schlegel N, Geruo A, Agosta C, Ahlstrøm A, Babonis G, Barletta VR, Bjørk AA, Blazquez A, Bonin J, Colgan W, Csatho B, Cullather R, Engdahl ME, Felikson D, Fettweis X, Forsberg R, Hogg AE, Gallee H, Gardner A, Gilbert L, Gourmelen N, Groh A, Gunter B, Hanna E, Harig C, Helm V, Horvath A, Horwath M, Khan S, Kjeldsen KK, Konrad H, Langen PL, Lecavalier B, Loomis B, Luthcke S, McMillan M, Melini D, Mernild S, Mohajerani Y, Moore P, Mottram R, Mouginot J, Moyano G, Muir A, Nagler T, Nield G, Nilsson J, Noël B, Otosaka I, Pattle ME, Peltier WR, Pie N, Rietbroek R, Rott H, Sandberg Sørensen L, Sasgen I, Save H, Scheuchl B, Schrama E, Schröder L, Seo KW, Simonsen SB, Slater T, Spada G, Sutterley T, Talpe M, Tarasov L, van de Berg WJ, van der Wal W, van Wessem M, Vishwakarma BD, Wiese D, Wilton D, Wagner T, Wouters B, Wuite J.** Mass balance of the greenland ice sheet from 1992 to 2018. *Nature*. 2020; 579. ISSN 14764687. DOI: https://doi.org/10.1038/s41586-019-1855-2
9. **Wingham DJ, Francis CR, Baker S, Bouzinac C, Brockley D, Cullen R, de Chateau-Thierry P, Laxon SW, Mallow U, Mavrocordatos C, Phalippou L, Ratier G, Rey L, Rostan F, Viau P, Wallis DW.** Cryosat: A mission to determine the fluctuations in earth's land and marine ice fields. *Advances in Space Research*. 2006; 37. ISSN 02731177. DOI: https://doi.org/10.1016/j.asr.2005.07.027

TO CITE THIS ARTICLE:
Hagdorn M, Gourmelen N 2023 Taskfarm: A Client/Server Framework for Supporting Massive Embarrassingly Parallel Workloads. *Journal of Open Research Software*, 11: 1. DOI: https://doi.org/10.5334/jors.393

Submitted: 06 September 2021 **Accepted:** 29 November 2022 **Published:** 12 January 2023