Computing and Software for Big Science (2021) 5:16
https://doi.org/10.1007/s41781-021-00058-y

ORIGINAL ARTICLE

Interactive analysis notebooks on DESY batch resources

Bringing Juypter to HTCondor and Maxwell at DESY

J. Reppin[1] · C. Beyer[1] · T. Hartmann[1] · F. Schluenzen[1] · M. Flemming[1] · S. Sternberger[1] · Y. Kemp[1]

Received: 21 October 2020 / Accepted: 24 April 2021 / Published online: 9 June 2021

Abstract
Batch scheduling systems are usually designed to maximise fair resource utilisation and efficiency, but are less well designed for demanding interactive processing, which requires fast access to resources while low upstart latency is only of secondary significance for high throughput of high performance computing scheduling systems. The computing clusters at DESY are intended as batch systems for end users to run massive analysis and simulation jobs enabling fast turnaround systems, in particular when processing is expected to feed back to operation of instruments in near real-time. The continuously increasing popularity of Jupyter Notebooks for interactive and online processing made an integration of this technology into the DESY batch systems indispensable. We present here our approach to utilise the HTCondor and SLURM backends to integrate Jupyter Notebook servers and the techniques involved to provide fast access. The chosen approach offers a smooth user experience allowing users to customize resource allocation tailored to their computational requirements. In addition, we outline the differences between the HPC and the HTC implementations and give an overview of the experience of running Jupyter Notebook services.

Keywords Jupyter · Notebooks · Interactive analysis · HTCondor · SLURM · Batch system

Introduction

DESY is a major research laboratory in Germany, carrying out fundamental research in particle and astroparticle physics, photon science and accelerator R&D. DESY operates several large particle accelerators, which are used by DESY scientists as well as a variety of international user communities.

Data management and analysis in all its aspects is a key component of every branch of science carried out at DESY. DESY offers several computing facilities, which we will briefly describe in the following.

✉ J. Reppin
johannes.reppin@desy.de

[1] Deutsches Elektronen-Synchrotron DESY Hamburg, Hambug, Germany

The NAF: national analysis facility

Founded in 2007 to complement the DESY Grid resources, the NAF enables particle physicists from German institutes working on LHC, as well as worldwide BELLE II users, to use interactive and fast-response compute and batch resources, with dedicated fast storage systems. For interactive access, the NAF has several work-group-servers (WGS), that can be accessed via ssh. Graphical login is possible through dedicated systems equipped with the FastX server software, allowing for fast graphical response even when accessing the NAF via low-bandwidth, high-latency network connections. A novel addition to the NAF interactive access methods is the Jupyter Notebook service, which will be described in this paper. The NAF has about 20 interactive work-group-server systems, and a batch farm (BIRD) managed by the HTCondor scheduling system [1] of about 8000 CPU cores. The sizing of the BIRD farm aims for an average utilisation of 75%, allowing for free resources for peak demands and fast turnaround times. The hardware in BIRD is partially renewed usually once per year, thus 5–7 generations and hardware setups are present in the cluster.

A small number of nodes are equipped with GPUs. Jobs are scheduled on a per core basis, multi-core jobs are supported, as long as they run in one single server. Jobs have access to the AFS global filesystem, and user Kerberos credentials are provided during the whole job run-time.

DESY Maxwell HPC system

Since 2011, DESY IT operates a high performance compute cluster system named "Maxwell". The Maxwell cluster is tightly integrated with the data acquisition systems at Petra3, FLASH and the European XFEL (Eu.XFEL) as well as the electron microscopy facility of the Centre for Structural Systems Biology (CSSB). Not surprisingly, about 80% of the 1500 active users are associated with the photon science community, but only consume a comparably small fraction of the CPU cycles. The most demanding applications originate from accelerator R&D and electron microscopy data processing pipelines. The applications supported by the cluster cover diverse scientific areas like structural biology [2], material science [3], plasma driven accelerators [4], axion dark matter fields [5] or time-resolved serial crystallography [6], to name a few examples.

In addition to classical interactive WGS, the cluster is equipped with 10 GPU accelerated front-end nodes enabling full graphical, interactive access to the cluster also supporting 3D visualisation of large data sets via FastX. The bulk compute power stems from a ~ 500 server, ~ 16,000 CPU core batch system, managed using the SLURM scheduling system [7]. Hardware in the Maxwell system usually is equipped with large RAM, powerful CPU, and InfiniBand network for data access and inter-process communication. The detailed configuration is not homogeneous as in typical HPC clusters, since it's not a monolithic system, but rather a discontinuously growing organism where groups and institutes on campus can contribute their own hardware to the Maxwell setup. Substantial investments also go into GPU-accelerated systems serving all AI-related computational tasks, and primarily the processing of data originating from e.g. electron microscopy, ptychography or X-ray computed tomography. The ownership model is reflected in SLURM via a prorogation scheme on a rather complex partition layout. The extremely heterogeneous zoo of applications, and the focus on photon science demands for the vast majority of compute jobs and exclusive access to entire nodes.

JupyterHub concepts

JupyterHub is a software package that implements a set of features to facilitate the usage of Jupyter Notebooks in multi-user environments like high-performance computing clusters or cloud computing sites [8, 9]. It is a Python process that is modular and can be adapted for various environments. The first part is the *Authenticator*. It handles user login and can be done by the host system (using PAM) or delegated to an external service, like LDAP, Oauth2 or OpenID Connect. The second part is the *Spawner*, that starts the *single-user* process. This can be a separate process on the same machine (SudoSpawner), a job submitted to a cluster via *BatchSpawner* [10]. Depending on the Cluster management software, there are different subclasses available in order to account for their individual configuration options. The singleuser process can also be spawned in a Docker Container either locally with *DockerSpawner* or in a Kubernetes Cluster through *KubeSpawner*. By using the base classes provided by the packages from the *Python Package Index* PyPI, we adapted the Spawner Class for the DESY batch systems and the Authenticator using DESY LDAP for the Maxwell Cluster and regular PAM Authentication on the BIRD cluster. After successful user login a job is then submitted to the batch system and the JupyterHub polls if the job has started and then connects to the `jupyterhub-singleuser` process via a random port and adding the route to the proxy, another part of the JupyterHub software. Once the *singleuser* process has started in its environment users can use a web browser to see the files in their home directory or other file system locations that have been integrated. This is typically a shared file system where experiment data is stored. The user can then create Jupyter Notebooks or start existing `.ipynb` files as notebooks. The notebook consists of *cells* that can either contain code or markdown. The code is executed by the notebook's *kernel* which is a new sub-process that receives and processes input. Typically this would be a Python process but other languages have added support for Jupyter Notebooks and a large ecosystem has evolved around jupyter. Using markdown gives users the possibility to add documentation to their code and makes notebooks more easily readable than regular scripts.

Scheduling: batch vs. interactive

Jupyter Notebook jobs represent an interactive way of working on a compute cluster. Interactive jobs have different requirement than batch jobs, users expect them to start within a short time and typically only one job per user is running whereas a user can submit thousands of non-interactive batch jobs at once and then wait for some time for them to complete. The difference in compute requirements is strongly reflected in the resource consumption: batch jobs are typically using a very large fraction of the available cpu-power, whereas interactive jobs are typically wasting most of the cpu-cycles, as illustrated in Figs. 1 and 2.

Figure 1 shows the typical load pattern on a node in a partition of the Maxwell cluster, which is dedicated to jupyter jobs. Despite serving about a dozen concurrent jupyter servers, the usage of the 48 physical cores on that machine

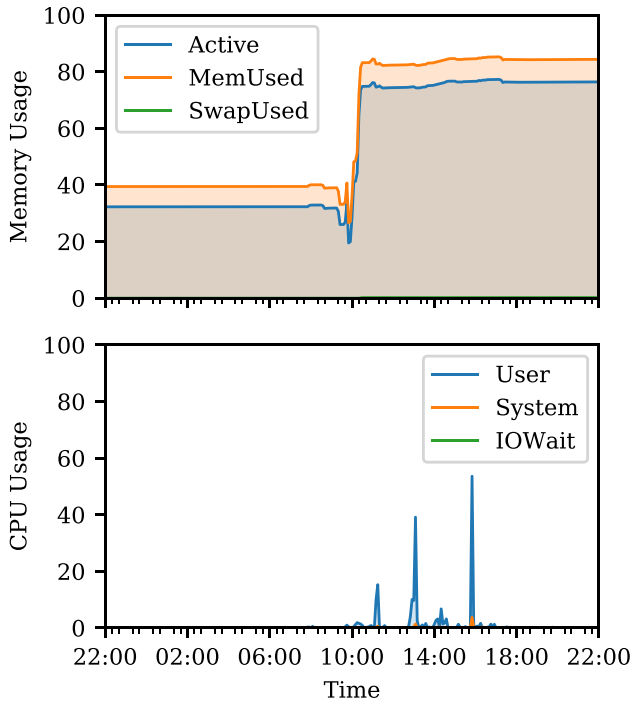

Fig. 1 Memory usage (top panel) and CPU usage (bottom panel) of one node on the Maxwell cluster that exclusively runs Jupyter Notebook jobs shown for a period of 20 h in July

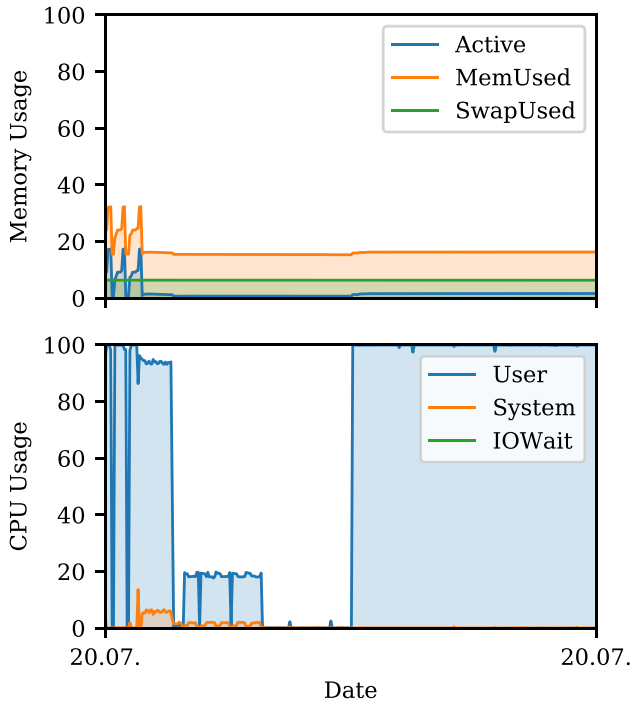

Fig. 2 Memory usage (top panel) and CPU usage (bottom panel) of one node on the Maxwell cluster that runs batch jobs, shown for the same period of time as Fig. 1

is largely negligible, showing only peaks when single users launch short multi-core computations from individual notebook cells. At the same time, the memory consumption is very significant (the jupyter compute nodes are equipped with 512 GB of memory), which is in part due to the global memory scope of cells in a notebook making garbage collection in jupyter notebooks a difficult task. The typical use of jupyter notebooks to dynamically display and update plots based on processing of (large data frames), which are usually kept entirely in memory unless explicitly de-referenced, doubtlessly has a significant contribution to the observed usage patterns.

For comparison, Fig. 2 shows a very typical load pattern on a non-dedicated batch node for the same time span. The traditional batch job usually make use of most if not all physical (and logical) cores, while only consumung a fraction of the available memory. In this particular case, the memory was 256 GB on a node with 20 physical cores, which is comparably large for conventional batch systems.

Of course, there are workflows like 3-D image reconstruction in electron microscopy or X-ray tomography, which can easily exceed memory usage of 100 GB/core, but the two figures illustrate exemplary the very different requirements from different classes of workloads on the batch clusters, and indicate that in particular multi-node compute jobs connected to jupyter notebooks might require a special treatment.

Jupyter on batch systems

The two different batch systems at DESY share quite a bit of commonalities, but differ substantially in some aspects. The network stack for example is very similar for both systems, whereas the authentication and the handling of tokens as well as the accommodation of resource requests differ substantially. We will outline below some of the commonalities and differences requiring adjustments of the JupyterHub configuration in the following sections.

Network

The hosts that serve the the JupyterHub web-services on both *Maxwell* (HPC) and *BIRD* (HTC) are located in the Demilitarized Zone (DMZ), due to open ports for `http` and `https` providing the Jupyter Notebook services to users outside the campus, which is actually the case for the majority of the JupyterHub users (on the Maxwell cluster). Consequently, JupyterHub servers and execution hosts reside in different subnets separated by the firewall, so that the scheduler ports for SLURM and HTCondor have to opened, as well as broader port ranges (40,000–41,000) for the individual *singleuser* servers.

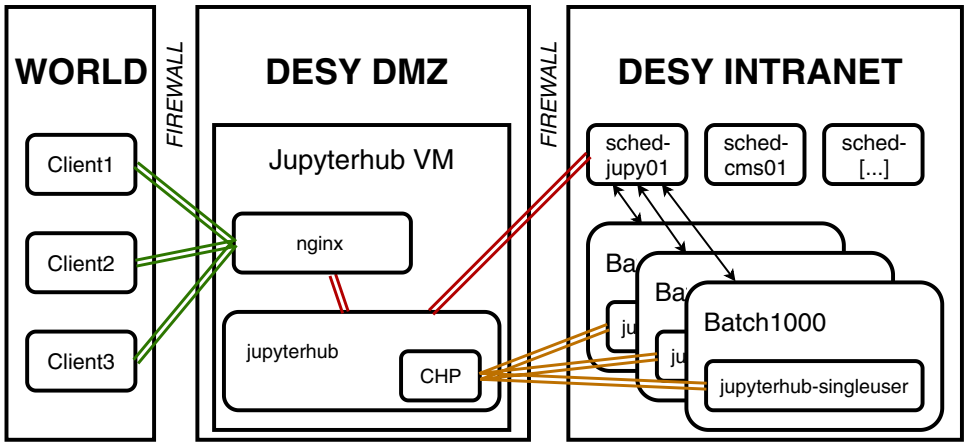

Fig. 3 Illustration of the JupyterHub setup at DESY. The JupyterHub VM runs the web application and submits HTCondor jobs. The configurable-host-proxy (CHP) establishes connections to the singleuser servers and connects them to the outside world through the nginx reverse proxy

The frontend of the JuypterHub is provided by *Nginx* which acts as a reverse proxy and SSL termination to serve the JupyterHub Python application, and is worldwide accesible (https://max-jhub.desy.de/ and https://naf-jhub.desy.de for Maxwell resp. NAF). The JupyterHub service uses a nodejs *configurable-http-proxy* which proxies to the *singleuser* processes on the batch nodes once the jobs have started (Fig. 3). The entire configuration is managed and monitored through standard service stacks provided by e.g. the DESY puppet environment.

Jupyter on HTCondor

In this section we describe the optimizations for the JupyterHub to minimize the startup time of single-user processes on the HTCondor Cluster *BIRD*.

Startup time

In order to have a service that can be used interactively, users expect it to very responsive like for any other web-service. The time beetween login on the JupyterHub and actually seeing juypters filemanager or a kernel running hence needs to be as minimal as possible. For HTCondor with the configuration used on the cluster some adjustments to the setup had to be implemented.

The concept of high throughput computing with HTCondor is based on classadds and negotiation. It does provide some configuration means to accelerate the negotiation of certain type of jobs but a dedicated *fast lane* is not part of the design and idea. The focus is far more on the overall throughput in a pool of worker nodes than on the acceleration of single jobs or jobclasses.

While in a test environment it was possible to obtain a decent time span from job submission to a running Jupyter Notebook with login, it turned out that the goal to get an acceptable *interactive work feeling* by starting a notebook in less than a minute for a user could only be realized in an overall medium busy HTCondor pool.

Supported by the HTCondor developers, a decoupled model including a dedicated scheduler, collector and negotiator for the Jupyter Notebook jobs was developed in order to be able to tune the negotiation system for very short, more frequent negotiation cycles and very fast starts of single jobs. In the following section we describe what was needed to implement these changes to HTCondor at DESY for Jupyter notebook jobs.

Obviously, first the Jupyter Notebook jobs neeeded to be tagged as such and a number of dedicated, reserved slots needed to be created. Interactive usage was not meant to be coupled to the rest of the group quota system. To ensure that at any time everyone could start interactive work inside a Jupyter Notebook, a separate accounting group was created with unlimited quota. Misuse of this is unlikely as the JupyterHub will forward a second login to a running notebook job if there is one, hence it should be impossible for any user to run more than one job using the notebook quota.

```
jobclassadd:
DESYAcctGroup = "BIRD_jupyter"
IsJupyterJob = true
Requirements = $(Requirements) &&
  (IsJupyterSlot =?= true)
slotclassadd
IsJupyterSlot = true
Start = $(START) &&
  (TARGET.IsJupyterJob =?= true)
```

This ensures that the Jupyter jobs can only run in the dedicated slots and those slots are not occupied by non-Jupyter jobs at any time, which rules out one of the more basic problems (no slots available) that could occur otherwise in a very busy pool. Usually, the maximum number of slots is equal to the number of CPU cores without Hyperthreading.

Since CPU utilization of Jupyter jobs is usually small, we add one over-committed slot to all batch nodes that run on CentOS 7. This extra slot can only be consumed by Jupyter jobs, ensuring that Jupyter Notebook servers can be *spawned* even if regular jobs get queued due to a full cluster.

Dedicated scheduler, collector and negotiator are all common condor daemons running on one physical host and decouple the negotiation of the jupyter jobs from the rest of the scheduling in the pool.

The expression `NEGOTIATOR_JOB_CONSTRAINT` must be used on both negotiators to exclude the jupyter jobs from being negotiated on the main negotiator:

```
IS_JUPYTER_JOB = ifthenelse( IsJupyterJob
  =?= undefined, false, IsJupyterJob )
NEGOTIATOR_JOB_CONSTRAINT
  = $(IS_JUPYTER_JOB) =?= false
```

and to make the jupyter jobs exclusive on the dedicated scheduler:

```
NEGOTIATOR_JOB_CONSTRAINT
  = IsJupyterJob =?= True
NEGOTIATOR_SLOT_CONSTRAINT
  = IsJupyterSlot =?= True
```

On the dedicated negotiator the expression `NEGOTIATOR_ SLOT_CONSTRAINT` is used to narrow down the considered slots and collected slot classadds to the tagged jupyter slots.

With such a configuration, the workload on dedicated negotiator/scheduler is diminished to a bare minimum the reducing the scheduling and negotiation rate to a very short timespan which results in a startup of incoming jobs within a couple of seconds, no matter what the overall pool status is at that time.

```
NEGOTIATOR_INTERVAL = 5
NEGOTIATOR_CYCLE_DELAY = 5
SCHEDD_INTERVAL = 5
```

The ranking expression `NEGOTIATOR_PRE_JOB_RANK` is used to rank the possible worker nodes according to the needs of a jupyter job.

Kerberos integration

Jobs on the BIRD batch farm and any login to batch submission hosts rely on Kerberos authentication and presence of AFS-tokens to access users home-directories, which reside in the DESY-afs cell. Jupyter notebook servers spawned from the Hub therefore also need access to kerberos tickets and afs tokens to provide users with a proper environment. The kerberos integration demanded some adjustments to the JupyterHub configuration.

The login to the JupyterHub uses the default *PAMAuthenticator*, so the system authenticates the user credentials like any other DESY machine. During the login a Kerberos token is created and saved in the `/tmp` folder on the host. The Condor job is submitted by the JupyterHub process using `sudo`, with the flag `-E` that preserves the environment variable, so the variable `KRB5CCNAME` needs to be set before the job is submitted. This is done by checking all the files in `/tmp` directory that belong to the user for whom a notebook server is spawned. The following code snippet shows the idea of how this is done:

```
def get_krb5_var(spawner):
    username = spawner.user.name
    k5file = find_krb_file(username)
    spawner.environment.update(
      {'KRB5CCNAME': k5file}
      )
    return spawner

c.CondorSpawner.pre_spawn_hook = get_krb5_var
```

A function is defined that discovers the users krb5 file and sets the environment variable. The `pre_spawn _hook` executes this function before a jupyter job is spawned. For simplicity, the function `find_krb_file` is not shown here but it is just a combination of file and ownership lookups. At submit time the local Kerberos ticket (created during login on the JupyterHub) of the user is read and transfered to the credd on the scheduler. On the scheduler the ticket gets enhanced to become forwardable and renewable. The credd transfers the ticket together with the job to the workernode and a local shepherd on the workernode creates an AFS token and takes care of the ticket for the user during the runtime of the slot.

User customization

We provide support for different options and customizations for the jobs that start the Jupyter Notebook. The first is the use of GPUs, which are especially useful for *machine learning* applications. Another option is the choice between the *classical* Jupyter Notebook interface and the *JupyterLab* interface which is gaining in popularity. We also provide the option to add environment variables that are used in the HTCondor job. This can be useful since it can add to the `$PATH` so the `jupyterhub-singleuser` executable finds custom kernels

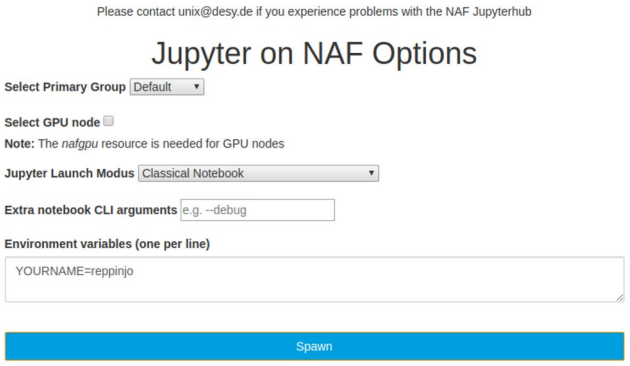

Fig. 4 User options for the Jupyter Jobs on the HTCondor compute farm *BIRD* at DESY. The most important feature is the ability to select one of the GPU node for the jupyter job

supplied by the user or the experiments. The options to configure the jupyter job can be seen in Fig. 4. These user customizations have to happen before the *singleuser* job has started but after user authentication. In order to apply these options, the job submission script is adapted to reflect the user choices, so for example in the case of a GPU requirement the line `Request_GPUs = 1` is added to the job script.

Jupyter on SLURM

In this part of the paper we explain the details of our implementation of the JupyterHub on the Maxwell Cluster available at https://max-jhub.desy.de. The main concept which includes a host virtual machine that serves the web service and talks to the batch system is the same as the HTCondor implementation. The queue and resulting job handling are different in the *Maxwell* setup because the amount of jobs and their requirements are very different for high-performance computing compared to high-throughput computing, so two setups are required. Both services are also hosted on individual virtual machines to keep the use-cases, the network setup and their individualisations separate. In general, the focus of the high performance cluster Maxwell is on fewer jobs, with each job using multiple CPU cores or compute nodes for parallel computation, and often runtimes of a few days. Compared to the high-throughput cluster BIRD that processes orders of magnitudes more jobs each using only very little resources up to a few CPU cores and running for a few hours. This means that SLURM can schedule jobs more almost instantaneously, as long as resources requested are available. There is no lengthy *negotiation* cycle like on HTCondor. The main complexity in SLURM comes not from the number of jobs it handles but from the different job queues and its different hardware capabilities and user permissions.

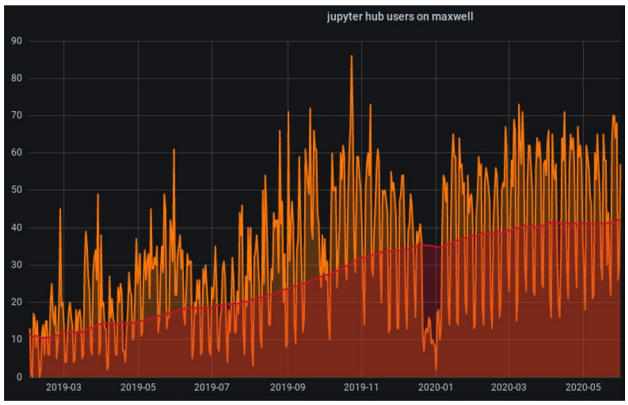

Fig. 5 Development of individual users on the Maxwell JupyterHub per day. The pattern is typical for interactive processes showing reduced activity outside working hours

Development of the jupyter service

Deployment of the jupyter services started end of 2018 on demand by a few users of the HPC cluster. Originally, the scope of jupyter notebooks was mainly to provide user-friendly hands-on tutorials, to provide reference workflows or develop (mostly) python code. These type of applications did not require a high-availability, or dedicated compute resources. So initially, the jupyter *single-user* could be spawned only on a dedicated slurm partition (named jhub) equipped with three already decommisioned AMD-Opteron nodes, which were more than sufficient to handle the jupyter jobs.

The usage of the JupyterHub was constantly raising once available in production. We count meanwhile up to 100 different users per day (Fig. 5), and regularly more than 300 per month, which is quite a substantial fraction of users on the cluster.

Not surprisingly, the scope of the JupyterHub quickly expanded to multi-node jobs, online-processing along running experiments (e.g. [11, 12]), the wide spectrum of ML training and inference up to quantum computing simulations, which naturally altered restraints on implementations and operation as outlined below.

SLURM partitions

The Maxwell cluster follows a co-operative model. It's core consists of storage systems, infiniband-infrastructure and compute nodes (*maxwell partition*) provided by DESYÍT department. The SLURM *maxwell partition* serves all users on campus with typical HPC applications. The Maxwell core does not satisfy all needs, particularly not of groups operating instruments at one the large user facilities, namely FLASH, PETRA3 or the European XFEL, which require for

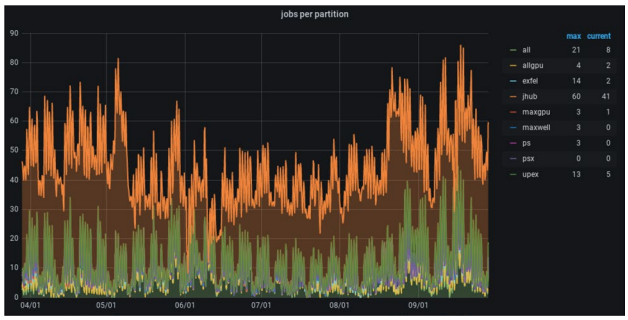

Fig. 6 Distributions of concurrent jupyter jobs over the various partitions. The stacked plot shows, that the majority of jobs are running in the jhub partition (shown in orange). The upex partition (shown in green) serving users of the Eur. XFEL is the other partition with a quite substantial number of concurrent jobs, accounting for about 35% of the JupyterHub processes

example different or specialized hardware, guaranteed availability during experiments or different storage backends. The groups can contribute and integrate their own resources to the maxwell cluster available in restricted SLURM partitions with configurations tailored to their needs. The groups largely benefit from central infrastructures, like from the fast Infiniband network connections and the highly scalable shared file systems BeeGFS and IBM Spectrum Scale (GPFS). In return, all contributed resources become available to all users of the cluster in a specialized partition (*all* partition). The *all* partition is generously configured allowing a single user to allocate literally all resources in the cluster for up to 14 days—as long as resources are not requested by users in any of the other partitions, in which case competing jobs in the *all* partition get terminated (*preempted*) and nodes freed with a maximum delay of 300 s. Jobs incapable of capturing the textitsigint signal sent by SLURM to preempted jobs, and that's the majority of jobs, will terminate after 30 s.

This concept offers quite a bit of flexibility, but leads to a very complex partitioning scheme, and to quite some inhomogenity in available hardware depending on the partition and users privileges. To serve the needs of the users also from within Jupyter, the job submission process must reflect at least part of these complexities.

The majority of users or Jupyter jobs can however be satisfied by the dedicated *jhub* partition (Fig. 6).

Dedicated partition for jupyter jobs

Most of the partitions in the cluster provide entire nodes to user jobs, only exceptions are the *jhub* partitions and partitions used for reservation which allow concurrent usage by users of specific groups, for example associated to a particular experiment. Jupyter jobs are to a larger interactive, and are hardly able to consume resources a significant fraction of cpu cycles, but are most of the time running idly. To avoid a very poor resource utilization, we configured a light-weight partition named *jhub* exclusively to Jupyter jobs. The partition comprises just three AMD EPYC nodes, each with 48 cores and 256 GB of memory. The partition is configured for 40-fold oversubscription, allowing for a total of 120 concurrent jobs, which so far was fully sufficient (Fig. 6 shows that it is rare to run more than 50 concurrent notebook jobs).

Very similar to the HTCondor configuration, this allows close to immediate startup of jupyter sessions, regardless of the load on the cluster, providing a smooth user experience. If there are no slots in the *jhub* partition, which can happen if one or more nodes are in *drain* mode, users see an error in the jupyter starting page that says that a server could not be started in the given time. They can then go back and select a different startup option (usually a dedicated node) and try again. The Hub catches failed starts without displaying HTTP error codes. We monitor the amount of parallel sessions on the *jhub* partition and if the user base increases and demand more resources, this partition can be adjusted in size, with more HPC nodes for a larger amount of user sessions in parallel. The runtime for a jupyter job on the dedicated *jhub* partition is pre-defined by the hub to 7 days. Particularly for code development, this is quite convenient without wasting too many resources. Even after a logout a session persists and users can come back to running notebooks and kernels with variables in memory. A manual jupyter server shutdown is possible through the *Hub Control Panel*.

Intentionally, we do not impose cgroups to limit memory or CPU resources. Due to the peculiar memory consumption of jupyter jobs (Fig. 1) cgroups might become disruptive to notebook jobs, and being able to consume multiple cores for example to develop distributed tasks is an intended application. It bears of course the risk of excessive resource cosumption by a single user, affecting up to 39 competing jupyter jobs on that node, but such incidences are (also illustrated by Fig. 1) extremely rare. In our experience, this simplistic approach is a quite feasible way to accommodate jupyter workflows efficiently while giving users a reasonable amount of flexibility.

User customization

As mentioned, Jupyter is increasingly being used for online processing while experiments are running at one of the facilities. Results are continously being visualized in notebooks and demanding compute jobs are distributed across several cores or nodes. This requires the ability for users to select *their* partition or even slurm reservation. We hence had to interface the batchspawner of the JupyterHub with SLURM (utilizing pyslurm), to present all allowed—and only allowed—partitions to the user (Fig. 7). The user can

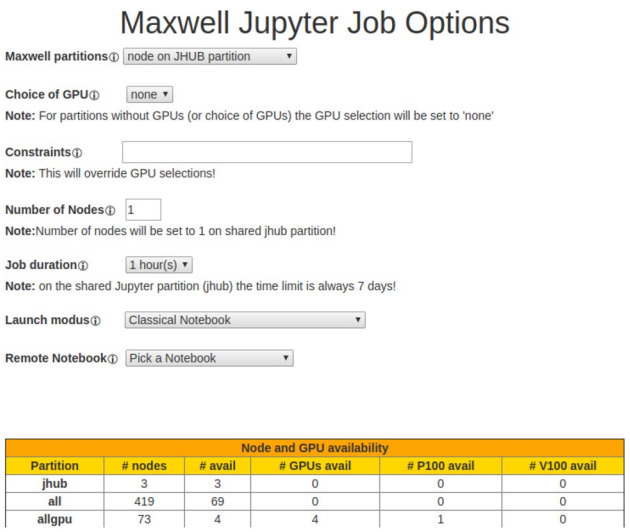

Fig. 7 User options for Jupyter Jobs on the Maxwell Cluster. Note the different *partitions* that can be selected which is different from BIRD

select a partition of his choice, but is generally limited to a runtime of 8 hours, to avoid massive wasting of resources. For long running experiments this might turn out to be rather inconvenient. For such cases we implemented a rather simple web-based reservation services, which allows administrative staff to create reservations on specific partitions through a REST API, without giving administrative rights on any other part in SLURM. Users or groups entitled to use the reservation, can also select that from the JupyterHub startup page, which automatically configures the runtime to the remaining lifetime of the reservation. This way resources can be fine-grained assigned to user groups in an automatic manner.

As mentioned, the Maxwell cluster is fairly untypical for a conventional HPC platform due to it's heterogenity. The continous extension of the cluster unavoidably comes with many generations of CPUs and GPUs, different hardware features like memory and core counts. The JupyterHub frontend was customized to allow for selection of specific features, and a free-text declaration of any constraint definition (Fig. 7). The choice of specific partitions or reservations, or selection of constraints can depend on resources currently available, in particular when the cluster is heavily oversubscribed. The login page of the Jupyterhub presents a small table to the user, showing currently available resources to hint at potential delays when selecting specific resources, so that users are not badly surprised when notebook requests run into timeouts, but it's currently not more than a hint lacking dynamic updates.

Other options allow the pre-selection between a classical jupyter notebook and a jupyterlab instance, instantiation of preconfigured notebooks, or the on-demand creation of memory-resident conda environments from github repositories. Some of the options turned out to be not particularly useful or are better suited for kubernetes based deployment.

The number of different choices to launch a single-user server can make the configuration by the user a bit tedious. Assuming that users usually choose a rather similar setup every time, the last configuration is stored in a simple sqlite database, which is being used to provide users with the configuration last used upon on the JupyterHub.

Operational experience and current work

The JupyterHub instances for both HPC and HTC clusters are in operation for more than a year. Efforts to run and maintain the services are quite moderate and some aspects are discussed in the following sections. Most of the efforts actually go into user driven problems, like user installed, incompatible python packages or conda environments. The JupyterHub services are still being developed aiming to improve the operation and overall user experience of the JupyterHub, which will also be described in the following.

Jupyter kernels

Starting the `jupyterhub-singleuser` process within the SLURM or HTCondor job gives the web UI that displays the files in der users' home directory and gives the ability to create new Notebooks or edit and run existing notebooks. A notebook connects to a jupyter kernel which typically is a Python kernel, but there are multiple options available, both for Python version and other programming languages that make use of the Jupyter Notebook interface. In the following part we describe how the choice and installation is realized on Maxwell and for Belle II users on BIRD.

Maxwell conda environments

On Maxwell we use the *Anaconda* Python distribution which is installed on a shared filesystem available on all nodes. The Anaconda distribution is not centrally managed by Puppet, updates to package versions require a login by an admin. Anaconda uses so called environments to separate individual Python versions and packages. Maxwell comes with multiple environments preinstalled, for example for the data analysis tools pytorch, pyFAI, tomopy, or special environments for GPU optimized machine learning, like tensorflow-GPU. The Python module *nb_conda_kernels* makes conda environments available as kernels on Maxwell, and environments created by users will also be added to the list if the *ipykernel* module is installed. Kernels for other software that is installed on the Maxwell Cluster can be added to the jupyterhub. The most

prominent examples for this are Matlab and Mathematica, but a wide range of kernels is available for users to install, if they need anything beyond the default software.

Software on NAF and Belle2 kernels in CVMFS

The worker nodes in BIRD are managed by Puppet and only CentOS 7 nodes can run Jupyter Notebook jobs. This means that the most straightforward way of installing Python is the package from the EPEL repository, which is currently at version 3.6 We also curate a list of packages from the Python package index *PyPI*, which are installed by using `pip3 install`. This list contains the most commonly used packages but users can then add further modules by installing them into their home directory. The Belle2 experiment has developed software that builds on Python but adds specific routines for their own data analysis pipelines. As nowadays common in high energy physics experiments, the software is distributed globally via the CernVM filesystem (CVMFS), and accessible via a fuse mount from the worker node. Since their users wanted to integrate theses software packages into the JupyterHub custom kernels were created by the Belle2 experiment. Those kernels are *json* files that specify the format of input and locations of executable, in this case the belle2 Python software binary. By adding the location of the json files to the `JUPYTER_PATH` users that specify *belle2* as their group automatically get the option of using the belle2 software in the JupyterHub. This method can be extended and used by other groups in the future, if they wish to create own Python kernels and environments that need to be made available in the JupyterHub interface.

DASK Jobqueue on Maxwell

On Maxwell it is possible to submit jobs to SLURM from another job. If the Jupyter Notebook runs as a SLURM job one can offload memory intensive or CPU heavy computations by submit more jobs to SLURM and doing the work in those jobs. This process can be done from the Notebook, either by using the `pyslurm` package, slurm magic or using the SLURM commands directly without Python. Dask [13] has become a popular package in data analysis since it makes it very easy to parallelize large Numpy array operations using multiple workers. One can do this on SLURM with the `dask_jobqueue` packages, which allows creating workers on the fly. Figure 8 shows the widget which can be used to scale up the cluster and add the workers to the scheduler. It submits SLURM jobs in the background which then execute the Python dask commands.

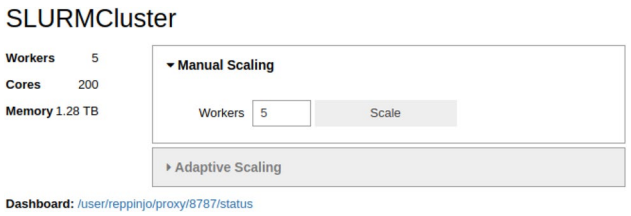

Fig. 8 Jupyter Notebook widget from the *dask_jobqueue* package that allows for manual or adaptive scaling of dask workers. In this example 5 SLURM jobs were started on 40 core nodes allowing for large scale data analysis with a total of 1.28 TB or memory

Blackbox testing for monitoring

Starting a single job and expecting it to start in a small timeframe of less than 60 s is an unusual way of using HTCondor. It can be quite error prone since HTCondor is built to process a large number of jobs most efficiently and reschedule a job if it fails. Some issues can arise for example if a node seems healthy to the scheduler and accepts job starts but then does not execute the `jupyterhub-singleuser` fast enough. The JupyterHub will then wait until a timeout has been reached and then show the error page and displays a message that the spawn was unsuccessful. Or if an issue with the network filesystem exists on the executing compute node it will lead to an error and the job start will not be successful. In order to check if starting the singleuser notebook server works correctly, we implemented a scheduled pipeline in our Gitlab instance. The Gitlab runner starts a pod on Kubernetes that runs a Python script. This script uses the *selenium* module in order to emulate the login process and to start a notebook server. Selenium, which usually is used for browser testing purposes, is used here because the login process creates a Kerberos ticket which is necessary for the HTCondor job. If we used the JupyterHub's REST API to create a singleuser server this would not be the case and a spawn would not succeed so the workaround of browser emulation is needed for the C/I pipeline that automates user startup monitoring. The pipeline is scheduled to run daily and notifies the JupyterHub admins if it fails. This makes it easier to find issues and fix them before users start experiencing any problems.

Hold jobs

The current *Batchspawner* does not account for jobs in HTCondor that go into a *hold* state. This can happen for various reasons, for example if the user selects a primary group for the job to run but does not have the required resources in our LDAP server. Another reason for jobs not starting is if there is an issue with the kerberos token and HTCondor can't open the output file in the AFS directory. In those

cases the job scheduler put the jobs into a *hold* state but the *batchspawner* only checks if the job is either pending or running. If the job is in *hold state*, the JupyterHub waits for the singleuser process to start. When this does not happen after a timeout, an error message is shown. We are working on adding this additional *hold state* to the batchspawner, so it recognises this fact that there is a problem either with the user configuration or with the batch system. This should then give users enough feedback so they can take appropriate actions or simply try again and start another job.

Condor jobs from the notebook

In order to utilise the power of the high-throughput cluster BIRD and the Jupyter Notebook, it is necessary to be able to submit jobs from within the Jupyter Notebook. HTCondor has Python bindings the wrap the submission in a very simple way using *HTmap* or `condorpy`. HTmap is especially interesting since one can replace the Python *map* function with the *htmap* function and do work on the whole htcondor cluster instead of locally on the node that runs the notebook. The implementation of HTCondor at DESY uses kerberos tickets to authenticate users' job submissions, though. This mechanism is not integrated in the Python bindings of HTCondor yet, so if a user wants to submit the submission will fail. Work is ongoing[1] to integrate the Kerberos credentials into the Python bindings but as long as this has not happened the users are bound to using the resources that are provided by HTCondor for the Jupyter Notebook job.

Outlook

While we have reached a production state of out Jupyter Notebook services on the *BIRD* and *Maxwell* clusters, research is still ongoing here of how to move into an era of cloud computing at DESY. In this section we will present the current state of these projects and how they relate to interactive data analysis via Jupyter Notebooks.

Cloud computing

DESY operates on-premise compute cloud resources using OpenStack. Differnt methods are possible to enable scientific computing use cases, in the context of application deployment, Kubernetes is a common approach. We use Rancher to set up clusters, provisioning Openstack VMs in the background and installing the kubernetes runtimes via docker images. This allows us to scale clusters in a flexible manner and use Openstack features to administer the cloud infrastructure in the background and use its features like software defined networks (Neutron), block storage (through Cinder), and others.

Jupyter on Kubernetes

One of the first deployments on our Kubernetes cluster was a JupyterHub. Within the JupyterHub development community a subset of developers has specialized on kubernetes with a project which is called *zero to JupyterHub*. This project contains a *Helm Chart* which consists of templates for Kubernetes deployment files. By setting values for the templates one can install the deployment onto a cluster and the JupyterHub web service starts up. Using the kubernets allows for specification of users' resources in a fine-grained manner using CPU and memory limits. One major research area is the treatment of users' data. While the distribution of jupyter notebooks can be done from within a running jupyter session via `git` or http requests, even with authentication, this syncing process is not a viable option for experiment data, which is why the data location is usually mounted via network storage protocols, like NFS, GPFS or others. In this case though, the mount must also be accessible in the Docker container, and the authentication and authorisation must match, and be secure along the whole set of layers. While it is possible, to start the Jupyter Pod in Kubernetes with a UID that can be obtained from a REST API endpoint, handling the file system and the ACLs through the OpenStack layer has not been solved, and is topic of investigation.

Conclusions

In this work we described the setup and process of making Jupyter Notebooks available to users the high performance cluster *Maxwell* as well as on the high throughput cluster *BIRD*. At first the computing environment at DESY was presented with a brief summary of the two main compute clusters, *Maxwell* and *BIRD*, highlighting the differences in usage by the photon science community that utilises high performance computing resources and the high energy physics (HEP) community that does most of its data analysis on high throughput computing resource where many low CPU core jobs are run. The general architecture of the JupyterHub and its concepts was presented and we explained how these concepts were implemented at DESY, including the details of how the spawning of a Jupyter Notebook server works and what the network architecture of the JupyterHub and its spawned notebook servers looks like. We then presented the work we did in order to achieve a reasonable startup time on the HTC cluster, which is critical for the user experience since the JupyterHub is meant as an interactive web service it needs to be highly responsive and with out setup we have

[1] https://htcondor-wiki.cs.wisc.edu/index.cgi/tktview?tn=6734.

starting times of 10–20 s. On the HPC cluster the time in which a job starts was not the issue but many Jupyter jobs share few HPC nodes which is not the case for regular batch jobs. We then explained other customisations of the JupyterHub, allowing people to use different computing resources like GPUs or their groups' specific computing jobs queues. Finally, we showed what work is still ongoing, describing job submission from a Jupyter Notebook job and how that will also be achieved from HTCondor.

The Jupyter Notebook service has been well received by the users, it has become a regular part of the workflow for many scientists. There have been many requests, both from long term users and new users and communities. The Jupyter Notebook service that is now available at DESY is also well suited for schools and workshops where new computing concepts can be presented together with documentations in the form of notebooks. This service opens the door to many more novel developments in high-performance and high-throughput computing and might give a completely new view on scheduling and batch systems. We plan to build on this service in the future and integrate new developments and establish the Jupyter Notebooks as an integral part of accessing computing resources at DESY.

Funding Open Access funding enabled and organized by Projekt DEAL.

References

1. Thain D, Tannenbaum T, Livny M (2005) Distributed computing in practice: the Condor experience. Concurren Pract Exp 17:323–356
2. Goessweiner-Mohr N, Kotov V, Brunner Vadim MJ, Mayr J, Wald J, Kuhlen L, Miletic S, Vesper O, Lugmayr W, Wagner S, Di Maio F-, Lea S, Marlovits TC (2019) Structural control for the coordinated assembly into functional pathogenic type-3 secretion systems. bioRxiv. https://doi.org/10.1101/714097
3. Abuin M, Kim YY, Runge H, Maier S, Dzhigaev D, Lazarev S, Gelisio L, Seitz C, Richard M, Zhou T, Vonk V, Keller TF, Vartanyants IA, Stierle A (2019) Coherent X-ray imaging of CO-adsorption-induced structural changes in Pt nanoparticles: implications for catalysis. ACS Appl Nano Mater 2:4818–4824
4. Pousa AF, de la Ossa AM, Brinkmann R, Assmann RW (2019) Compact multistage plasma-based accelerator design for correlated energy spread compensation. Phys. Rev. Lett. 123:054801
5. Knirck S, Schütte-Engel J, Millar A, Redondo J, Reimann O, Ringwald A, Steffen F (2019) A first look on 3D effects in open axion haloscopes. J Cosmol Astropart Phys 2019:026. https://doi.org/10.1088/1475-7516/2019/08/026
6. Pandey S, Bean R, Sato T, Poudyal I, Bielecki J, Cruz Villarreal J, Yefanov O, Mariani V, White TA, Kupitz C, Hunter M, Abdellatif MH, Bajt S, Bondar V, Echelmeier A, Doppler D, Emons M, Frank M, Fromme R, Gevorkov Y, Giovanetti G, Jiang M, Kim D, Kim Y, Kirkwood H, Klimovskaia A, Knoska J, Koua FHM, Letrun R, Lisova S, Maia L, Mazalova V, Meza D, Michelat T, Ourmazd A, Palmer G, Ramilli M, Schubert R, Schwander P, Silenzi A, Sztuk-Dambietz J, Tolstikova A, Chapman HN, Ros A, Barty A, Fromme P, Mancuso AP, Schmidt M (2020) Time-resolved serial femtosecond crystallography at the European XFEL. Nat Methods 17:73–78
7. Jette MA, Yoo AB, Grondona M (2003) SLURM: simple linux utility for resource management. Lecture notes in computer science: proceedings of job scheduling strategies for parallel processing (JSSPP), pp 44–60
8. Pérez F, Granger BE (2007) IPython: a system for interactive scientific computing. Comput Sci Eng 9:21–29
9. Kluyver T, Ragan-Kelley B, Pérez F, Granger B, Bussonnier M, Frederic J, Kelley K, Hamrick J, Grout J, Corlay S, Ivanov P, Avila D, Abdalla S, Willing C (2016) Jupyter notebooks: a publishing format for reproducible computational workflows. In: Loizides F, Schmidt B (eds) Positioning and power in academic publishing: players, agents and agendas. IOS Press, Amsterdam, pp 87–90
10. Milligan MB (2018) Jupyter as common technology platform for interactive HPC services, PEARC '18: Proceedings of the practice and experience on advanced research computing. Association for Computing Machinery, NY, 17, pp 1–6
11. Hafner AJCE, Kluyver T, Bertelsen M, Upadhyay KM, Lecz Z, Nourbakhsh S, Mancuso AP, Fortmann-Grote C (2020) VINYL: the VIrtual Neutron and X-ray laboratory and its applications. Adv Comput Methods X Ray Opt 5:114930Z
12. Bücker R, Hogan-Lamarre P, Mehrabi P, Schulz EC, Bultema LA, Gevorkov Y, Brehm W, Yefanov O, Oberthür D, Kassier G, Miller RJD (2020) Serial protein crystallography in an electron microscope. Nat Commun 11:996
13. Dask Development Team, Dask: Library for dynamic task scheduling (2016). https://docs.dask.org/en/latest/cite.html. Accessed Aug 2020