# A Parallel and Resilient Frontend for High Performance Validation Suites

Julien Adam[1] , Marc Pérache[2]

[1] Paratools SAS, Bruyères-le-Châtel, France
[2] CEA, DAM, DIF, F-91297, Arpajon, France

**Abstract.** In any well-structured software project, a necessary step consists in validating results relatively to functional expectations. However, in the high-performance computing (HPC) context, this process can become cumbersome due to specific constraints such as scalability and/or specific job launchers. In this paper we present an original validation front-end taking advantage of HPC resources for HPC workloads. By adding an abstraction level between users and the batch manager, our tool *JCHRONOSS*, drastically reduces test-suite running time, while taking advantage of distributed resources available to HPC developers. We will first introduce validation work-flow challenges before presenting the architecture of our tool and its contribution to HPC validation suites. Eventually, we present results from real test-cases, demonstrating effective speed-up up to 25x compared to sequential validation time – paving the way to more thorough validation of HPC applications.

**Key words:**Validation, Test-Suite, HPC, Scheduling, Fault-tolerance, Parallel, Software Quality

## 1 Introduction

In the constantly evolving landscape of parallel supercomputers, HPC applications must be updated to take advantage of the underlying architectures. In such a context, validating parallel software features can be a real challenge. Non-regression bases (NRB) can play an important role in such transitional process, constantly validating results relative to expectations – matching each features with dedicated tests. However, for larger projects, the growth of the non-regression base can become troublesome, particularly if validation system is not robust enough. A recent project with very large non-regression bases took up to several days to run and involved thousands of tests. In such a context, modifications could take up to one week to be validated, making test results more complex to analyze and impacting development reactivity. Current testing frameworks do not provide a scalable way to meet the growing validation demands of large sofware efforts.

Our goal is to simplify the continuous validation of parallel HPC applications, allowing HPC developers to constantly monitor their software quality in an efficient manner. In this paper, we present a highly modular testing framework,

called *JCHRONOSS*, that provides a convenient and consistent abstraction layer between a parallel validation suite and a given batch-manager. This tool is intended to be scalable on most HPC architecture, with dynamic scheduling and resilient execution. As we will show, JCHRONOSS has been built in a generic manner, without constraining the target execution model in order to meet the requirements of any developer, conveniently replacing the commodity test-scripts encountered in some projects. The purpose is to optimize the continuous integration process by providing a quick and reliable feedback on software quality during the development process. JCHRONOSS is built in the context of existing integration testing utilities, thereby enhancing validation work-flows in an HPC context, while allowing the user to rely on standard components.

This paper is organized as follows: Section 2 describes related work, discussing the use and limitations of non-regression bases in HPC context. Section 3.1 shortly presents JCHRONOSS's architecture. Then, Section 3 details JCHRONOSS's contributions to continuous integration in HPC context and Section 4 evaluates JCHRONOSS in different configurations relatively to a real use-case. Finally, Section 5 describes open issues and future work.

## 2 Related Work

The main focus of JCHRONOSS is to run tests in an optimized way. This process involves two main components that we have to compare with existing work: (1) schedulers and (2) test-frameworks.

*Schedulers.* Resource scheduling has been widely studied for years and a large number of tools already covers the subject particularly in HPC context. For example, a tool like YARN[8] from the Apache Hadoop framework is a powerful scheduler, able to distribute multiple applications over thousands of resources, such as those used for MapReduce[5] computations. Borg[9] from Google can distribute applications over multiple clusters, each composed of thousands of nodes, with a goal of supporting a huge number of requests per second. In the HPC context, job managers such as SLURM[10] are deployed over a cluster to efficiently manage resource allocation. Such schedulers generally need to be deployed at the system level in order to expose computing resources. On the contrary, JCHRONOSS is running in user space, processing a test-suite meta-description and generating calls to such job-manager in a more efficient manner. Indeed, running a test-suite in parallel requires more than simply submitting executions to an existing batch manager, as we will further detail.

*Test Frameworks.* As testing is a key process to ensure software quality, there are a wide range of tools and solutions. Most solutions are focused on ease of use, especially when dealing with automatic generation and configuration aspect. CMake[6] and Autotools[4] are two main project builder, able to handle the configuration and generations of test suites in a convenient way through macros. Some continuous integration platform like Jenkins[3], Travis[2] or CruiseControl[1]

are designed to create integrated test environments gathering several key components in the same interface (such as version control systems and ticket trackers) However, these solutions were not developed for HPC, as they are not able to conveniently express the execution of their workload in parallel, this burden being left to the end-user. Developers are then forced to develop their own validation script, tailored to a given test environment. JCHRONOSS proposes to avoid this redundant effort thanks to a simple XML formatted input driving a parallel execution from user-defined templates (batch-manager agnostic), without sacrificing portability. Our tool is not a job scheduler by itself, it is designed to be run by a user to generate from an XML meta-model an optimized stream of requests to an existing batch-manager (the one installed on the machine).

## 3 Contribution

In this section, we present the three main contributions of our tool. First, we detail JCHRONOSS's architecture and its main components. Then, we explain how tests are scheduled over a supercomputer. Eventually, we describe the fault-tolerance mechanism. These contributions allow JCHRONOSS to use a surface-based scheduler with resiliency to run tests in parallel and optimize validation time.

### 3.1 Global Model

JCHRONOSS is designed for ease of use and interoperability. It loads a standard validated XML input and produces a standard JUnit formated output compliant with common continuous integration platforms. As depicted in Figure 1, the master-worker architecture is based on two independent layers doing mostly the same processing. In order to keep resources as busy as possible, layers share the same algorithm following a "greedy" approach. Jobs are scattered in sub-pools assigned to *workers*.

Workers are responsible for executing individual sub-pools. Sub-pool resources are subtracted from a global resource allocation counter. Then, when there are no resources left, the master stops creating workers. Upon completion, results are merged in a *post-run* list gathering completed tests' results – process repeated until test-suite completion. The only difference between master and worker is their scope. The master is responsible for the global validation system whereas a worker manages a subset of tests, effectively running them over the system.

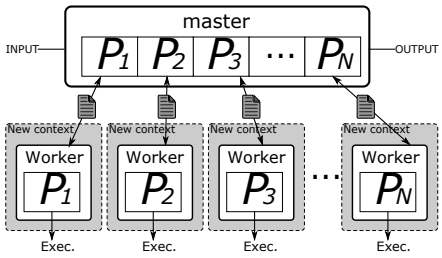


**Fig. 1.** Master/Worker Architecture

### 3.2 Job Ordering

Making requests to the job manager is as important as the scheduling itself. In the context of overloaded supercomputers, the more requests are made by a user, the harder it is for the job manager to satisfy them. Generally, allocation grants are based on multiple criteria. This is why requesting 2 nodes twice is not always equivalent to a 4 node request. Allocation rate depends on current cluster load, past requests, quotas, and the number of resources. Given these constraints, the most basic test runner would make a request for individual tests. This can seriously degrade user priority, making future allocation attempts longer.

JCHRONOSS offers a way to gather jobs depending on deterministic criteria, such as number of resources. This way, if a test requests four nodes to run, JCHRONOSS will attempt to create a worker with multiple jobs requesting the same number of nodes, allocating the node configuration only once. This follows a very simple principle: if the allocation is created according to type and number of required resources, then jobs sharing similar requirements can be dependent on the same allocation. By gathering jobs with the same requirements in the same allocation, this policy tries to limit the number of resource requests, leading to larger workers (more jobs per allocation) and lowering global allocation overhead. However, as such contexts ask for more resources, the batch-manager can be a little longer to fulfill the request. But, if the batch-manager does not penalize allocation following a linear allocation time formula like $f(x) = ax$ (which is generally the case), this algorithm will always be preferable for this kind of configuration. This approach is less stressful, and best suited for homogeneous validation suites. Indeed, with imbalanced job pools, one worker will have to process more tests than the others, eventually leading to a parallelism loss.

Another approach can be considered to take advantage of a higher level of parallelism. Another solution consists in running validation suite depending on available resources instead of test requirements. The strategy evenly divides resources among workers. Then, jobs are scheduled using a two-dimensional heuristic over both resources and time, the purpose being to fill each parallel subset as much as possible. Jobs are first sorted by resource requirements and then by decreasing estimated time. Thanks to this ordering, larger jobs are scheduled first, using a classical greedy scheduling heuristic. This way, JCHRONOSS can guarantee an efficient use of available resources at any time. Ideally, efficient scheduling requires a prior knowledge of individual test duration in order to correctly apply the "surface" scheduling heuristic. However, if not provided, or at least bounded by individual job timeout, JCHRONOSS approximates job duration as the mean of previous duration.

This algorithm is the most efficient for non-homogeneous test-suites in terms of job manager requests as it allocates large subset and tries to fill them – maximizing resource efficiency. However, if the batch-manager policy is resource-based, allocating large buckets can lead to very long allocation time, leading to poor performance. Nonetheless, we observed that in most cases, the best-fit policy is a good trade-off between efficiency and execution time.

### 3.3 Fault Tolerance

Depending on code coverage, validation suites can take a lot of time, ranging from a few minutes up to several days. However, HPC environments are not fully reliable with, for example, failing nodes, batch-manager and timeouts – possibly impacting running jobs. JCHRONOSS has been designed to be fault tolerant. It supposes that any layer can crash. If a worker is interrupted, the master considers all jobs as not run and reschedules them, making our approach completely resilient to failing workers. Indeed, a new worker will be created to replace the failing one and the tests will be rescheduled. Therefore, losing a worker has no effect on validation's coverage. The case where the master instance is interrupted is more problematic as job results are only merged at the end of the test-suite. Consequently, a crash prior to this point would lead to a complete loss of master's state. In order to circumvent this limitation, we implemented an asynchronous check-pointing mechanism which consists in storing current job states in a file as the workers are running. Thanks to this approach, a validation can be restarted from the last coherent checkpoint, even if the master instance failed, providing a complete fault-tolerance support.

*Checkpoint Time.* A checkpoint is initiated when the master expects a worker to end to maximize the overlapping. It consists in storing current jobs' state and their configuration. Workers do not need to be checkpointed, they will be recreated upon restart, scheduling remaining jobs. Our backup consists of a single JSON formatted file stored in JCHRONOSS's build directory, alongside other temporary files. JSON format is flexible and easy to manipulate inside JCHRONOSS, however, for now, the JSON file is not compressed and can lead to both IO and parsing overhead depending on validation suite size. We are considering the use of a binary JSON (BSON) to optimize this process.

*Restart Time.* After an interruption, JCHRONOSS can be restarted from the backup file. To do so, current configuration is ignored and previous one is reloaded. Then, job manager's state is restored from the backup JSON file. Finally, validation can restart seamlessly. In order to save disk space, following backup files replace previous ones. Therefore, the most recent backup is always kept and calling the same command line over again in case of failure allows the completion of an incomplete test-suite thanks to our fault-tolerance mechanism.

*Overhead.* We plan to make a deep evaluation of fault-tolerance mechanism overhead. For now, our experiments show that it takes 1 second per worker to back up 10,000 tests and the global overhead does not exceed 1.2%. Clearly, the number of tests can be different and the number of workers can noticeably vary depending on the user's configuration. By trying to checkpoint only validation state and not JCHRONOSS itself, we significantly decrease implied backup overhead. It is important to say that the major part of this overhead is recovered by workers instance currently running. However, this mechanism can become really costly with an important number of workers, this increasing checkpoint time, not completely recovered by shorter workers.

## 4 Experimental Results

JCHRONOSS has been developed for and is being used on a daily basis as MPC[7] validation system to manage a test base of forty thousand jobs, test-suite likely to be executed on several supercomputers, involving different environments for portability tests. JCHRONOSS's goal is to speedup validation processes without sacrificing their portability between machines. In this purpose, the important variability between HPC machines had to be taken into account. Indeed, as aforementioned several parameters affect scheduling such as current user priority and specific latency due to cluster load. Moreover, as the machine load is highly variable, we cannot predict allocation overhead. Then, two successive JCHRONOSS runs, with similar parameters might not lead to the same result. We were careful to present tests with similar configurations while mitigating these random effects. These benchmarks were performed on two different supercomputers. First the Curie supercomputer, operated in the TGCC , which is heavily loaded by multiple users, leading to long waiting queues. The batch manager, based on SLURM is configured with user priorities. Second supercomputer is a 111 nodes $\times$ 8 cores prototype, with fewer users and a flexible batch manager. Comparisons will be made between these two environments, respectively with and without priority based algorithms applied at batch manager level. The NRB used here is a suite of 39,366 jobs with fixed execution times to allow policies comparison over multiple runs while minimizing measurement noise. These configurations have been run with the same subset of available resources, allowed to perform tests on 48 nodes. We compare policies in terms of elapsed time on each of these supercomputer. These comparisons will be made alongside CTest performance with the same set of tests. The Figure 2 depicts these results.

Complete validation suites were run on each of these machines with different policies in order to compare batch manager configuration effects. The fixed number of resources is set to 4. Vertical axis represents the number of hours elapsed in the run. CTest results have been run sequentially (`ctest -j4`) to be able to compare with JCHRONOSS. Indeed, the `-j` option allows tests to be run in parallel without discrimination, impling job oversubmissions and causing

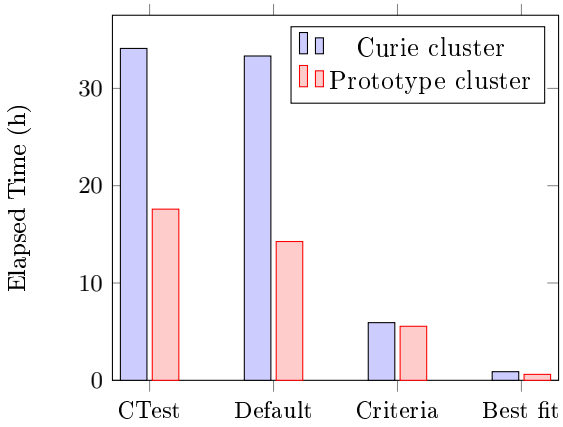


**Fig. 2.** Policies efficiency comparison between two supercomputer job managers.

the user to violate the QoS policy and account to be blocked if the value is too high. Each job keeps the same execution time in each execution. We consider that machine load variation did not impact test-suite duration between policies, Curie being loaded and our test cluster almost empty. These results illustrate the need to carefully choose scheduling policies according to cluster, allocation overhead being highly depending on batch manager. Default sequential policy clearly shows its limits, providing no performance gains on the test-cluster and leading to an important penalty on Curie. More importantly, allocation overhead even led to poor performances relatively to the aggregated reference. Default policy created around 40,000 new allocation requests, each of them associated with a resource allocation, explaining the overhead observed on the loaded cluster. This policy roughly applies the same methodology than other test-runner tools, as depicted by the sequential CTest performance results.

Our criteria-based policy shows a non-negligible time reduction with a 2.5 speedup. Packing jobs relatively to resources seems to be a good alternative to sequential execution. Indeed, considering of $N$ jobs, this solution can save up to $N-1$ new allocations if they are all using the same number of resources.

Eventually, best fit algorithm shows the best speed-up of 25, independently from the underlying batch manager. The optimizations made by this policy, spreading jobs among resources in order to save time have proven to be effective. More importantly, this approach seems to be less sensitive to batch-manager policy, making it more suitable for portability. Best fit is then both the fastest and the most portable policy – reason why it is the default one in JCHRONOSS.

## 5 Future Work

*Optimize time to result.* Currently, all tests defined by the user must be performed before publishing the results. Therefore, it can happen that the whole test suite has to be completed before the user is able to consult the results, including intermediate ones. In order to make time to result shorter, a deamon server, provided as a JCHRONOSS plugin and running globally on interactive nodes, could interact directly with worker instances, periodically collecting job results and making data accessible from a client browser. To reduce the number of deamons, a single server would handle multiple JCHRONOSS instances.

*Becoming a complete end-to-end validation tool.* For now, existing validation processes would have to be rewritten in order to generate a suitable input for JCHRONOSS. We suggest making our tool compliant with upstream and downstream tools, avoiding test specifications rewriting. JCHRONOSS should include a job generator module, which could take data from existing build systems like CMake or Autotools. Dealing with the output, JCHRONOSS generates it in standard JUnit XML format. However, some other formats could be more suitable for post-processing. A generic output generation module would bring more flexibility to the end-user. Our idea is to gather in one single tool all validation steps from the build system to the result mining platform, leading to an end-to-end validation tool.

## 6 Conclusion

JCHRONOSS is a parallel and resilient frontend for high-performance validation suites that run distributed tests in parallel in order to reduce time to result. Beyond just taking advantage of parallel computing resources, JCHRONOSS looks for optimal trade-offs between efficiency and duration. Its multiple scheduling policies are suitable for most use cases, allowing JCHRONOSS to be an innovative agile tool designed for HPC workloads. JCHRONOSS can be adapted to various execution environments and is compatible with existing validation tools such as Jenkins and BuildBot. We demonstrated validation speedup up to $25\times$ on an actual use case of $\approx$ 40,000 tests, clearly showing the advantage of our approach. JCHRONOSS is then a convenient building block for developers willing to apply continuous integration methods to their HPC project without developing their own launch scripts to speedup validation. As validation system reactivity is a critical point, the important duration associated with large NRB can be a possible explanation of why some projects are not validated regularly. The purpose of our work is to make HPC project validation suites more efficient in terms of both computational costs and execution time. Indeed, a faster validation system simplifying continuous testing opens the way for better programming practices and transitively enhances code quality.

## References


1. Cruisecontrol website. `http://cruisecontrol.sourceforge.net/`.
2. TravisCI website. `https://travis-ci.org/`.
3. A. Berg. *Jenkins Continuous Integration Cookbook*. Packt Publishing Ltd, 2012.
4. J. Calcote. *Autotools: A Practitioner's Guide to GNU Autoconf, Automake, and Libtool*. No Starch Press, 2010.
5. J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
6. B. Hoffman, D. Cole, and J. Vines. Software process for rapid development of hpc software using cmake. In *DoD High Performance Computing Modernization Program Users Group Conference (HPCMP-UGC), 2009*, pages 378–382. IEEE, 2009.
7. M. Pérache, H. Jourdren, and R. Namyst. Mpc: A unified parallel runtime for clusters of numa machines. In *Euro-Par 2008–Parallel Processing*, pages 78–88. Springer, 2008.
8. V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.
9. A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, page 18. ACM, 2015.
10. A. Yoo, M. Jette, and M. Grondona. Slurm: Simple linux utility for resource management. In D. Feitelson, L. Rudolph, and U. Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 2862 of *Lecture Notes in Computer Science*, pages 44–60. Springer Berlin Heidelberg, 2003.