Stroll: A Universal Filesystem-Based Interface for Seamless Task Deployment in Grid Computing

Abdulrahman Azab and Hein Meling

Dept. of Electrical Engineering and Computer Science
University of Stavanger, Norway

Abstract. Developing applications for solving compute intensive problems is not trivial. Despite availability of a range of Grid computing platforms, domain specialists and scientists only rarely take advantage of these computing facilities. One reason for this is the complexity of Grid computing, and the need to learn a new programming environment to interact with the Grid. Typically, only a few programming languages are supported, and often scientists use special-purpose languages that are not supported by most Grid platforms. Moreover, users cannot easily deploy their compute tasks to multiple Grid platforms without rewriting their program to use different task submission interfaces.

In this paper we present Stroll, a universal filesystem-based interface for seamless task submission to one or more Grid facilities. Users interact with the Grid through simple read and write filesystem commands. Stroll allows all categories of users to submit and manage compute tasks both manually, and from within their programs, which may be written in any language. Stroll has been implemented on Windows and Linux, and we demonstrate that we can submit the same compute tasks to both Condor and Unicore clusters. Our evaluation shows the overhead of Stroll to negligible. Comparing the code complexity of a Stroll compute task with command-line clients and Grid APIs show that Stroll can eliminated up to 95% of the complexity.

1 Introduction

Large-scale distributed computation for scientific applications is on a path to become mainstream, enabling researchers to perform computations that were impossible only a decade ago. Grid computing is an enabling technology in this space, allowing scientists to leverage underutilized compute resources scattered around on the Internet. A recent study [1] shows that typical Windows classroom computers has an average CPU idleness of 97.9%. Moreover, a large number of volunteers contribute compute resources for a wide range of large-scale computing applications [2, 3]. To support scientists in gaining access to this vast collective of compute resources, Grid computing frameworks take the role of coordination and providing an access interface for submitting compute tasks.

K.M. Göschka and S. Haridi (Eds.): DAIS 2012, LNCS 7272, pp. 162–176, 2012.
© IFIP International Federation for Information Processing 2012

However despite its many benefits, the adoption of Grid computing in different scientific communities are perhaps not at the level one might expect or hope. The primary reason for this is complexity [4]. Managing and using a Grid computing framework is a challenging undertaking, and thus limits adoption to expert programmers capable of, and willing to learn how to program APIs specific to different Grid frameworks. This model is not very approachable for non-computer scientists, such as chemists, biologists, and many others, whom may use domain-specific tools and programming languages. Therefore, a major challenge faced by Grid computing vendors is to provide *a ubiquitous and easy to use interface for accessing the Grid*, enabling domain-specialist programmers to easily develop and deploy Grid computing applications.

The goal of a Grid access interface is to *hide as much complexity as possible, while retaining the necessary flexibility to implement efficient compute tasks.* State-of-the-art Grid computing access typically fall into two categories: (i) manual submission, and (ii) API-based submission. Manual submission includes web-based interfaces and command line tools that enable users to submit tasks for one-shot execution. This approach is not suitable for task deployment from within scientific applications, e.g. repeating a computation in a loop, and analyzing results in-between. The current solution to this need, is to provide a web-service, or some client-side API that can support repeated task submissions. However, a major drawback with this approach is lack of interoperability between different Grid platforms. Even if there were interoperability, most scientific programming languages lack of support for web services and client-side Grid APIs. Moreover, the goal in this paper is to make a variety Grid platforms accessible from any programming language, and to support manual submission by a simple file copy.

This paper presents Stroll, a universal interface for seamless deployment of compute tasks to a variety of Grid platforms. Stroll is based on the ubiquitous filesystem interface, and allows task submission, monitoring, and administration through simple `read()` and `write()` filesystem functions or commands. The interface is implemented as a user space *virtual filesystem* (VFS), enabling access to it from any application or programming language that can interact with a filesystem. Hence, the approach lends itself well to providing Grid access to domain-specialists whom, although not computer scientists, are expected to have general familiarity with common filesystem operations.

Stroll is implemented on Windows and Linux. The Linux version use the FUSE [5] library, while the Windows version use the Callback File System [6]. In the current implementation, we provide access to two Grid frameworks: Condor [7] and Unicore [8]. A major innovation of Stroll is that it enables users to execute compute tasks on several Grid frameworks, from the programming language of their choice, or by manual submission by means of a simple file copy.

To evaluate the simplicity of using Stroll, we analyzed the program complexity of the task submission procedures of three different Grid tasks written in three different programming languages. We compare the program complexity metrics of Stroll with native Grid submission techniques, using McCabe [9],

Halstead [10] and structural software complexity metrics [11], and find that Stroll can reduce code complexity by as much as 95%. We evaluated the performance overhead of running Grid tasks through STROLL versus native Grid access, and find the overhead to be negligible.

2 Background and Related Work

Plan 9 [12] was the first operating system to introduce a filesystem-based pattern or style for accessing arbitrary IO systems, e.g. general filesystems, network interfaces, and processes. Similar ideas are provided through the FUSE framework [5], which is used in Unix-like operating systems. FUSE is an integral part of the Linux kernel and is easily installed on OS X systems. On Windows, FUSE-like functionality can be provided by the Callback File System [6].

FUSE consists of: (i) a kernel-space driver that receives control from the kernel for all filesystem commands executed on the virtual path, and (ii) an API through which a user-space program can respond to filesystem commands and take the proper actions. FUSE allow user-space applications to build their own filesystem interface to programmatically control and manipulate the execution of filesystem commands performed on some virtual path.

Attempts at simplifying the interaction with Grid computing systems have been addressed in previous works. The interaction we refer to, takes place at the *interface* between users and the Grid itself. In this work, we do not concern ourselves with the interactions that take place within the Grid infrastructure; instead we leverage existing Grid frameworks and provide an interface for users to operate on multiple Grid frameworks. Most previous efforts to provide Grid access interfaces can be classified into the following categories: (i) Grid portals, (ii) Library-based Grid APIs, and (iii) Web-service interfaces.

Grid portals [13] are web interfaces through which users can upload and submit compute tasks, and eventually download the results. It is designed for manual submission. In many compute-intensive applications, iterative computations that depend on the outcome of previous iterations is necessary, e.g. in stochastic modeling and machine learning [14]. A familiar example is Monte Carlo simulations [15]. For such applications, Grid portals are not suitable.

Library-based Grid APIs provide an interface for programmatic Grid access. These APIs support a range of language bindings, yet they cannot support an arbitrary set of languages. These APIs come in two forms: (1) APIs designed for interoperability, e.g. SAGA [16] and DRMAA [17], and (2) APIs specific to a particular Grid framework [18–21].

Such Grid APIs limits the language choice of the user. As an example, consider an R user that wants to take advantage of a Unicore cluster for solving compute-intensive statistical problems. HiLA [18], which is the main Grid API for Unicore, is based on Java. The R language [22] has no interaction interface for Java. Thus to accomplish this, a user would have to install an R package called rJava [23], and then access the HiLA API through rJava in order to submit compute tasks to Unicore, all of which becomes a tedious and complicated process. There are

independent parallel scripting languages, e.g. Swift [24] and Skywriting [25], which to some extent achieves platform independence, but not programming language independence.

To our knowledge, no existing Grid framework provides a pure filesystem interface for interacting with the scheduler and to collect results. While Genesis II [26] does provide filesystem-based access to remote files in data Grids, and can submit compute tasks, the user still has to manually prepare a JSDL [27] submission script for the compute tasks. Having to prepare a JSDL script retains the complexity sought eliminated by the filesystem interface approach. UEM [28] is a unified execution model for cloud and cluster computing which provides a filesystem interface for execution, monitoring, and control of remote processes. Each process on each node in the system can be shown as a virtual directory. This approach require the user to specify at which node a process should be allocated, making it unsuitable for Grid systems with explicit support for resource management. Parrot [29] is an interposition agent enabling integration between standard UNIX applications and remote storage systems. To use Parrot, users must specify the remote machine's location, and construct special Parrot commands which reduces transparency to the user.

Stroll is a VFS based front-end for Grid frameworks through which compute tasks and data files can be submitted to the Grid through filesystem `read()` and `write()` commands. Stroll is novel in that it can support both manual and programmatic task submission, and is accessible to any programming language with basic filesystem support. Given its programming language neutrality, there is no need for new language bindings or extensions. An early version of Stroll was presented in [30]; this paper is a significant extension. The main additions in the new version are: support for non-Windows OS and multi-Grid platforms through drivers, brokering strategy support, and enhanced submission scripting.

3 Task Structure

A Grid task is typically composed of: an executable, input files, and a configuration file, or submission script [31]. In Stroll, a task is represented as a uniquely named directory containing the appropriate executable, input and output files, and configuration files. This section describes the task structure used by Stroll.

In Stroll, a regular task contains one executable and a single configuration. A task can be executed as: (i) *One-shot*, or in (ii) *Batch* [32]. A one-shot task corresponds to a single submission and needs one CPU to run, while a batch task consist of multiple parallel submissions with the same executable, but different input files, and needs multiple CPUs to run. An example task structure of an R based [22] batch task is shown in Fig. 1. In this directory structure, we separate between *task files* and *configuration files*. Configuration files are virtual files that can be queried or updated through `read()` and `write()` calls.

To create a new task, one simply create a new directory under the Grid filesystem mount point, e.g. `mkdir /stroll/PSM`. This operation automatically creates task directory structure with virtual files containing default values.

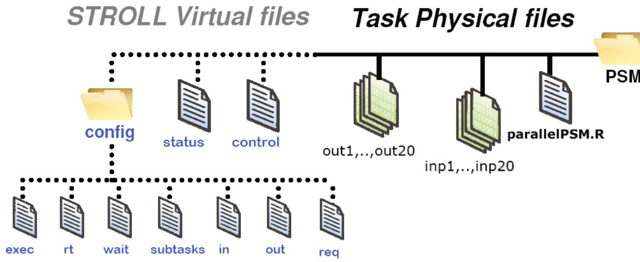


Fig. 1. Example task structure

Task files include the executable and input files supplied by user, log files and output files are created by the Grid framework during execution. The executable can be: binary, intermediate (e.g. Java or .Net assembly), or a script (e.g. R, Perl, Python). The runtime required to execute the task has to be configured before task submission. For batch processing, identically named input files must be distinguished with a unique index associated with each subtask in the batch.

Configuration files and directories are dynamically created by STROLL, allowing the user to configure, monitor, and control task execution by accessing the filesystem. The `config` virtual directory contains configuration parameters, each represented by a virtual file naming the parameter and containing the parameter's value. Listing 1.1 shows the simplicity of task submission through a shell script corresponding to the example in Fig. 1.

Table 1. Task parameters (files in the `config` directory)

CONFIG	DESCRIPTION
exec	Executable file name.
rt	Runtime environment needed to run the executable, e.g. JVM, R, or CLR.
wait	Value **true** causes the `submit` command to hold until task termination.
subtasks	Number of parallel subtasks to be executed for batch tasks.
in	Comma separated list of input files. For batch tasks, the input file names should include a zero based index, e.g. in1,in2,..,in20 for 20 subtasks.
out	Comma separated list of output files.
args	Comma separated list of task arguments.
req	Resource requirements, e.g. memory and load average. The format is a similar to Condor's ClassAd [31], only more compact: `LA0.1 && M512` corresponds to `LoadAvg<=0.1 && Memory>=512`.

Table 1 shows the main task configuration parameters/files supported by STROLL. The `status` file can be read by a user to obtain status information about the currently executing task, e.g. `cat status`. The `control` file is used to execute control commands, e.g. submission and forced termination. This can be done by executing a `write()` call on the `control` file, specifying the command as shown in Line 13 of Listing 1.1. The currently supported control commands are:

Listing 1.1. Example task submission from Unix shell

```
1 #!/bin/sh
2 # Example with the parallelPSM.R and 20 subtasks
3 cd /stroll
4 # Create PSM task directory inside the virtual path: /stroll
5 mkdir PSM
6 # Copy task files into the task directory
7 cp ~/parallelPSM.R PSM
8 cp ~/inp* PSM
9 # Set the configuration parameters:
10 echo 'parallelPSM.R' > config/exec
11 echo 'R' > config/rt
12 echo 'true' > config/wait
13 echo '20' > config/subtasks
14 echo 'inp$(i)' > config/in
15 echo 'out$(i)' > config/out
16 echo 'M128' > config/req
17 # Submit the task
18 echo "submit" > PSM/control
19 # Wait until 'echo' command returns to collect the output
20 cp PSM/out* ~/
21 # Remove the virtual task directory
22 rm -r PSM
```

Listing 1.2. Setting parameter collectively

```
1 echo "exec=parallelPSM.R;req=M128;rt=R;in=inp$(i);out=out$(i),err$(i);wait=true;subtasks=20"
2      > PSM/control
```

- `submit`: Submits the task for execution.
- `terminate`: Forces task termination.
- `restart`: Terminate the task and resubmit it.

Configuration parameters can be set or updated *individually* or *collectively*. Setting parameter values individually is shown in Lines 10-16 of Listing 1.1; e.g. Line 10 names the executable, `parallelPSM.R`, as the value of the `exec` parameter. An alternative is to set parameters collectively by issuing a single `write()` call to the `control` file as shown in Listing 1.2.

Stroll supports both *synchronous* and *asynchronous* submission. Synchronous submission causes the submit command to wait until the task has completed and returned results in the output files. Asynchronous submission returns control to the user immediately, and thus to detect completion, the user must monitor the `status` file. In Stroll, asynchronous submission is the default; setting the `wait` option to `true` in the task configuration (see Table 1) activates synchronous submission (Lines 11 of Listing 1.1). Since the execution models of different Grid systems are not identical, we are currently implementing the shared essential task management functions. Future Stroll versions should include a unified execution model for complex task management functionalities.

4 Stroll Architecture

The overall aim of any computational Grid user interface is to provide the ability to submit, monitor, control, and collect the results of compute tasks. Stroll is

providing these facilities. Fig. 2a illustrates the architecture of the entire Grid execution model driven by STROLL, and how its various components interact with the targeted Grid frameworks. In the following we describe the various components.

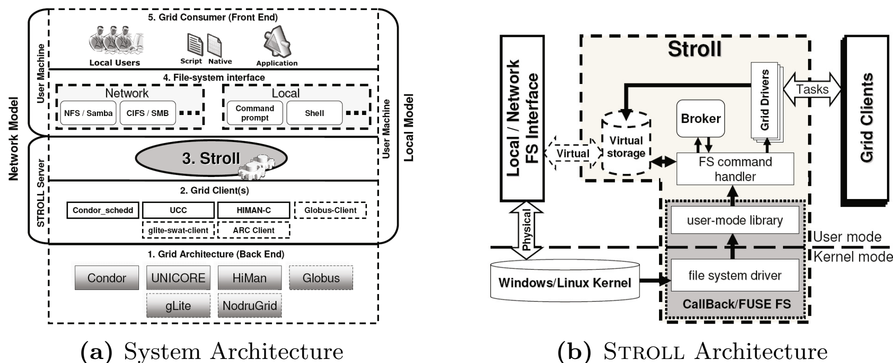
At the top, there is a Grid access front-end, through which a user can manually or programmatically submit compute tasks and collect results by performing `read()` and `write()` calls on the filesystem. The execution of these calls is done through a filesystem interface, the fourth layer, which is supported by the operating system. Thus, enabling both access to a local STROLL filesystem, or remotely to a STROLL server. Remote access can be established by having a STROLL server share its virtual access path, and have clients mount it as a network storage. This is a well-known technique and is supported in both Windows and Linux with different protocols, e.g. CIFS, SMB, and NFS. STROLL is placed as the third layer, and is designed to be accessed either through a local or network filesystem interface. STROLL simply receives Grid access commands, in the form of `read()` and `write()` calls, through the filesystem interface. These are then translated into actual Grid commands according to the target Grid architecture, and submits them to the target Grid client on the second layer. With this design, it is possible to support multiple Grid clients attached to different Grids architectures, e.g. Condor and Unicore. Grid clients can be either command line or API based, and must be granted access to Grid back-ends, the bottom layer.

In case of local filesystem access, users must install both STROLL and clients for each of the targeted Grids on their local machine. This requires that users machines to be granted access to the target Grids. Moreover, in case of network-based filesystem access, only one or a small number of machines needs to have STROLL and Grid clients installed. These machines can act as STROLL servers for other users to gain access to Grid services.

The architecture of STROLL is depicted in Fig. 2b, as the server part of layer 3 in Fig. 2a, which is a shim layer between the filesystem interface and Grid clients. The shim layer is composed of a virtual filesystem driver, a command handler, a broker, and a Grid driver. Next, we describe each of these components.

Virtual Filesystem Driver: The VFS driver, CallbackFS or FUSE, is responsible for creating and mounting a virtual storage, and directing the control of all filesystem calls on the virtual storage to the filesystem command handler. The Linux version is based on FUSE and is written in C++, while the Windows version is based on CallbackFS [6] and is written in C# with .Net framework 4.0 [33]. The VFS keeps virtual files for Grid monitoring and control management in the virtual storage, in addition to pointers to the actual task input and output files. The content of the virtual files are stored on disk at a configured location, but could also be kept in memory.

Filesystem Command Handler: This handler is responsible for capturing filesystem commands issued to the STROLL virtual storage(s) and forwarding Grid related commands to the appropriate Grid drivers. On task submission, the handler also contacts the broker for choosing a matching Grid for the task.



(a) System Architecture **(b)** Stroll Architecture

Fig. 2. Architecture

Broker: The broker is responsible for matching each submitted task with the attached Grids based on the task requirements, e.g. number of nodes and runtime environment. The broker accepts task allocation requests from the command handler, and returns a list of the drivers of the matching Grids. Generally, a Grid is considered to be a match, if it contains enough nodes that satisfy the task requirements, e.g. OS architecture and memory capacity. For deployments with a single Grid platform, the broker is not active. Grid internal brokering is handled by the Grid platform itself.

Grid Driver: Stroll requires a Grid driver for each Grid architecture that it supports. Currently Stroll can submit tasks to Unicore [8] and Condor [7] Grid frameworks. Each Grid driver is responsible for bridging the Stroll and the Grid architecture. To support multiple Grids of the same architecture, e.g. multiple Condor pools, multiple instances of the associated Grid driver is used. Each Grid driver is connected to a client of the associated Grid, e.g. Condor CLI for Condor and UCC for Unicore. In particular, the Grid driver is responsible for: 1) interpreting Grid commands received from the command handler and to translate them into the associated Grid format, e.g. Condor job ClassAd, 2) submitting the task to the Grid through the Grid client, 3) responding to task monitoring and control commands by gathering the required information and performing the required actions through the Grid client, and finally 4) collecting the task output/error files upon completion and place the files in the task directory in Stroll virtual storage. In the Linux implementation of Stroll, the grid driver is built on the top of SAGA [16], while a simple .Net based driver is built for the Windows implementation.

5 Evaluation

In order to evaluate Stroll, we asked ourselves the following two questions: (a) does it actually help the developer, and (b) is the performance overhead acceptable? To address the first question, we decided to compute various code

complexity metrics for three different use cases as presented next. The second question is addressed by measuring the performance overhead for the most advanced use case.

5.1 Code Complexity Evaluation

We develop three use cases for evaluating the code complexity quantitatively. The first two are embarrassingly parallel and the third is a composite structure which iteratively submits a batch task. The use cases are written in VB.Net, Java, and R. We compare the complexity metrics of our Stroll implementation with current practice using Grid APIs, command-line tools, and in the case of the R language, a language level API called Snow [34] for cluster computing. The following four complexity metrics are used in the comparisons:

1. Lines of Code (LoC).
2. System Complexity (C) [11].
3. McCabe's Cyclomatic Complexity (μ) [9].
4. Halstead's Effort metric (E) [10].

Lines of code is the simplest measure of complexity. System complexity is a high-level design metric which is a collection of two metrics: structural complexity based on the number of fanout modules, and data complexity based on the I/O variables. McCabe's cyclomatic complexity is a measure of testing difficulty based on a control flow representation of the program. Halstead's Effort metric is a measure of program understandability based on the number of operators and the number of operands. For each use case, the metrics are computed only for the part of the program submitting tasks. Thus, let P be the number of procedures involved in the submission for the different alternatives.

Parallel Matrix Multiplication (VB.Net). We implement a VB.Net parallel matrix multiplication of two matrices using a simple technique for partitioning the multiplier matrix into a set of sub-matrices. Each sub-matrix is assigned to a subtask for multiplication with the multiplicand matrix.

The code complexity of the task submission part for both synchronous and asynchronous Stroll and two command-line approaches is shown in Table 2. No Grid APIs are included here, since there are no stable .Net APIs for Condor and Unicore. The metrics show that a significant complexity reduction is obtained for Stroll over the other methods, needing only one procedure. For Condor command-line (CLI), two procedures are used for task submission, one in which a job ClassAd [31] is created, and `Main`. In the Unicore UCC case, the submission is more complex since UCC requires the preparation of a detailed configuration for each subtask. We had to create three procedures: `CreateScript` for creating the subtask scripts, the `Submit` procedure, and `Main`. Note that system complexity and cyclomatic complexity are very large for UCC; normally, μ should not exceed 10, but due to the complexity of `CreateScript`, it exceeds this value. Hence, from these results we see that Stroll can reduce the complexity of task submission by

more than 90% for the C and μ parameters compared to UCC. Moreover, Stroll has a 43% and 70% advantage over Condor CLI in the μ and C parameters, respectively.

Solving a System of Linear Equations (Java). Cramer's rule [35] is implemented in Java, and used to solve a system of linear equations represented in matrix multiplication form: $Ax = b$, by calculating matrix determinants. To solve a system of size n using Cramer's rule, determinants have to be computed for $(n + 1)$ $n \times n$ matrices including the coefficient matrix A and n matrices, one associated with each system variable. The determinants are computed in parallel such that each determinant is computed as a batch task. Since determinant computation is recursive, the parallelism can be branched into $\frac{n!}{2}$ parallel computations with $n - 2$ levels. For simplicity we use only one level. Each subtask of the batch computes a part of the determinant, and $n + 1$ batch tasks are necessary to complete the computation.

For this use case, the submission mechanism implements HTC where many parallel tasks are submitted to the same Grid. The complexity metrics are shown in Table 2. Task submission is performed through Stroll, Condor API [36], and Unicore HiLA [18]. As with Condor CLI, two procedures are used also for Condor API, in which much of the code deals with the matrix partitioning, and managing multiple submissions. Moreover, the results for HiLA shows that its complexity is very high, which is due to a very low abstraction level in HiLA. As for the previous use case, Stroll can significantly reduce complexity: 93% improvement in cyclomatic complexity and 86% for system complexity compared to HiLA, and for Condor API, the advantage of Stroll is about 54% for both.

ECG-CPR Model (R). In this use case, we take a computationally intensive modeling scenario from our own research group. The project aims to model the relationship between ECG [37] characteristics and CPR [38] quality during cardiac arrest. The model has been developed in R [22] based on the Population Stochastic Modeling (PSM) [39] package. The model estimates population parameters in a mixed effects model based on stochastic differential equations [39], and is computational intensive. It takes about 500 CPU hours, or three weeks of computation for a single run of the model. We profiled the PSM estimation code, and found a function, `APL.KF`, which is called iteratively to be the most time consuming. To reduce the execution time of the model, we modified `APL.KF` to submit a batch task. As in the previous use case, task submission is performed through Stroll, Condor API, and Unicore HiLA. In addition, we also compare with an existing HPC cluster tool for R called Snow [34], by implementing another parallel version of `APL.KF` using the Snow package. In Snow, submission is simply to send the desired procedure along with input parameters to the Snow cluster. Thus, the code complexity is relatively small, as shown in Table 2, yet Stroll is still better in all complexity metrics, except C, where Snow has a 40% advantage over Stroll. In summary, Stroll is still very competitive in terms of overall complexity, even against language level frameworks like Snow.

5.2 Performance Evaluation

Next, we evaluate the performance of Stroll in terms of resource consumption and the overhead in task submission.

Testbed: Our setup includes two student labs: one with 15 Windows XP machines, each with Intel P4 2.8 GHz dual core CPU and 1 GB memory, and another lab with 20 CentOS Linux machines with identical CPU and memory configuration. The Windows machines were configured as both Condor worker and Unicore target system, and one machine is configured as Stroll server and having both Condor client and UCC installed. The Linux lab is configured as a Condor pool, and each machine has mounted a Stroll directory, through which tasks can be submitted. Submissions are performed through Stroll server to both Windows and Linux machines.

Resource Consumption: We evaluate the performance of Stroll in terms of CPU and memory consumption on Stroll server by testing the Stroll network model through recording CPU and memory usage on Stroll server during the concurrent submission of many tasks from other machines in the LAN. Similar to Unicore service orchestrator [40], Stroll carries out task submission from the server, the case in which the load on the server is directly proportional to the number and size of submissions. We submitted 10 matrix-multiplication batch tasks from the LAN through the shared Stroll network drive.

Table 2. Code complexity for different submission methods. Least complex (best) is shown in bold.

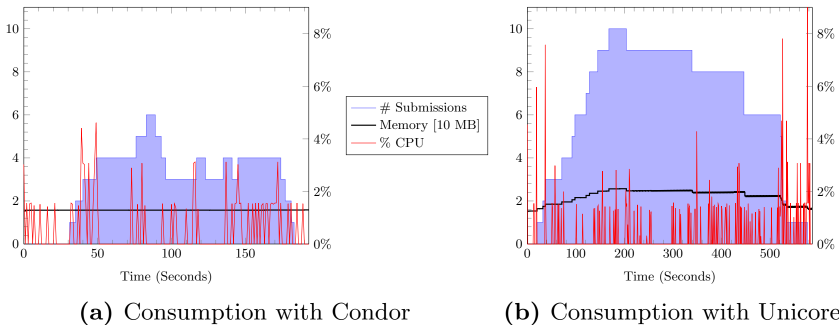
	Submission method	LoC	P	C	μ	E
Parallel Matrix × (VB.Net)	**Stroll-Sync**	**28**	**1**	**100.73**	**4**	$\mathbf{3.1 \times 10^4}$
	Stroll-Async	32	**1**	144.69	5	3.5×10^4
	Condor CLI	74	2	339.66	7	6.3×10^4
	UCC	255	3	1511.37	49	$> 10^6$
Cramer's rule (Java)	**Stroll-Sync**	**44**	**1**	**144.77**	**4**	$\mathbf{7 \times 10^4}$
	Stroll-Async	46	**1**	169.79	5	7.1×10^4
	Condor API	139	2	320.2	9	10.2×10^4
	Unicore HiLA	1043	13	998.5	56	$> 10^6$
ECG-CPR Model (R)	**Stroll-Sync**	**22**	**1**	50.37	**3**	$\mathbf{8.1 \times 10^3}$
	Stroll-Async	26	1	64.33	4	8.8×10^3
	Snow	29	2	**30.83**	5	16.3×10^3
	Condor API	47	2	209.5	7	94.1×10^3
	Unicore HiLA	950	14	1179.7	56	$> 10^6$

Fig. 3 presents two snapshots of CPU and Memory consumption by Stroll during the submission and execution of the 10 tasks to Condor and Unicore respectively. Each task included five subtasks, and the total input size of each task equals to 38.5 MB in case of Condor and 46.5 MB in case of Unicore. Fig. 3a and 3b presents the CPU consumption by only Stroll threads as a

percentage of CPU time, while memory consumption is computed as the portion of the physical memory, in 10 MBytes, which is consumed by STROLL threads. # submissions is the total number of currently running tasks. The 10 tasks are submitted in a similar sequence to both Condor and Unicore.

In Fig. 3a and 3b, it is clear that STROLL introduces low CPU and memory consumption throughout the concurrent executions. This small overhead is due to the interaction with the Grid clients, condor CLI and UCC, together with the created monitoring threads. It is also noticed that STROLL introduces more CPU consumption in case of submitting to Unicore, which was expected due the many threads usually created by the UCC. It is also noticed that task execution is faster in Condor than in Unicore, which is a pure performance difference between the two frameworks and irrelevant to STROLL.

Task Submission Overhead: To evaluate the time overhead of STROLL, we compared the time consumed in task submission in case of submission through STROLL with that in case of submission through Grid APIs. In case of Grid APIs, Task submission time (TST) is the time taken by the API procedures to construct and submit the task to the Grid using the local privileges. In case of STROLL, TST is the time interval starting from detecting the submission filesystem command through task construction to task submission through the Grid client to the local broker.



(a) Consumption with Condor **(b)** Consumption with Unicore

Fig. 3. STROLL resource consumption under concurrent submissions

TST is more effective for evaluating the time overhead of STROLL other than the total execution time since task execution is entirely managed by the Grid broker. Fig. 4 presents the task submission time of five internal batch tasks, from the ECG-CPR model,with different sizes: 5, 10, 20, 30, and 40. Each task has been submitted five times using the following configurations: Condor API (through rJava), HiLA (through rJava), and synchronous STROLL submitting to Condor and Unicore. The presented values are the median values of the five runs.

It is noticed that the submission time increases with the task size, which can explained as follows: for a batch task submission, each subtask has to be submitted individually to the broker queue in addition to creating a separate

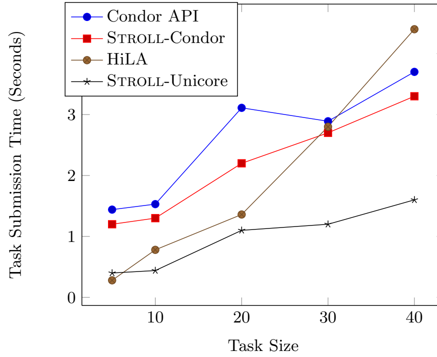


Fig. 4. Submission time for batch tasks of different sizes

monitoring process/object at the client. It can be seen that Stroll-Condor actually submits faster than Condor API. The reason is that Condor API, being based on Java, turns out to be slower in task submission than Stroll, which interacts directly with Condor CLI. In the Unicore experiment, Stroll is slightly faster than HiLA due to the overhead caused by HiLA for creating a separate object for each subtask, while Stroll uses UCC for submission. Overall, time consumed in task submission through Stroll is relatively small compared to that in case of submission through Grid APIs.

6 Conclusion

Given the complexity of Grid computing middleware, providing an easy to use interface for task submission is a significant challenge. In this paper, we proposed Stroll, a universal filesystem-based interface for seamless task submission to one or more Grid computing facilities. Stroll provides drivers for both Condor and Unicore, enabling non-expert users to submit compute tasks to the Grid both manually and programmatically. Our evaluation indicate that the overhead imposed by Stroll over native Grid clients is negligible, nevertheless concurrent submission of batch tasks through one Stroll server can be resource consuming. We have also demonstrated that the complexity of the compute task, written using Stroll can reduced the program complexity significantly.

We are planning to expand the testing scope to include the evaluation of Stroll server delays in case of a considerable number of concurrent submissions using a powerful Stroll server. In addition, we are currently building a composite task model to allow building workflow tasks as directory structures together with the support for inter-task communication. Also to include drivers for other Grid frameworks, e.g. Globus and gLite. And finally, we are planning to build a security model in which Grid user privileges are mapped to filesystem privileges.

References

1. Domingues, P., Marques, P., Silva, L.: Resource usage of windows computer laboratories. In: ICPP Workshops 2005, pp. 469–476 (2005)
2. Anderson, D.P.: Public Computing: Reconnecting People to Science. In: Shared Knowledge and the Web, Madrid, Spain (November 2003)
3. McConnell, B.: Beyond Contact: A Guide to SETI and Communicating with Alien Civilizations. O'Reilly (2001)
4. Bijsterbosch, M., et al.: DRIVER, Digital Repository Infrastructure Vision for European Research II. Technology Watch Report (December 2008)
5. Filesystem in userspace, http://fuse.sourceforge.net/
6. Callback File System (2011), http://www.eldos.com/cbfs/
7. Litzkow, M., Livny, M., Mutka, M.: Condor - a hunter of idle workstations. In: ICDCS (June 1988)
8. Erwin, D.W., Snelling, D.F.: Unicore: A grid computing environment. In: ECPP, pp. 825–834 (2001)
9. McCabe, T.J.: A complexity measure. IEEE Trans. Softw. Eng. (1976)
10. Halstead, M.H.: Elements of Software Science (Operating and programming systems series). Elsevier Science Inc., NY (1977)
11. Card, D.N., Agresti, W.W.: Measuring software design complexity. The Journal of Systems And Software 3(8) (June 1988)
12. Pike, R., et al.: The use of name spaces in Plan 9. SIGOPS Oper. Syst. Rev. 27(2), 72–76 (1993)
13. Wang, X.D., Yang, X., Allan, R.: Top ten questions to design a successful grid portal. In: SKG, pp. 18–24 (2006)
14. Saad, Y.: Iterative Methods for Sparse Linear Systems, 2nd edn. Society for Industrial and Applied Mathematics, Philadelphia (2003)
15. Abramson, D., Giddy, J., Kotler, L.: High performance parametric modeling with nimrod/g: Killer application for the global grid? (2000)
16. Goodale, T., et al.: Saga: A simple api for grid applications. high-level application programming on the grid. In: Comput. Methods in Science and Tech. (2006)
17. Herrera, J., Huedo, E., Montero, R.S., Llorente, I.M.: Developing grid-aware applications with drmaa on globus-based grids (2004)
18. Hagemeier, B., Menday, R., Schuller, B., Streit, A.: A universal api for grids. In: Cracow Grid Workshop (July 2007)
19. Chapman, C., et al.: Condor birdbath: Web service interfaces to condor. In: UK e-Science All Hands Meeting, Nottingham, UK (2005)
20. Grid ASCII Helper Protocol, http://www.cs.wisc.edu/condor/gahp/
21. Wegener, D., et al.: GridR: An R-based tool for scientific data analysis in grid environments. Future Gener. Comput. Syst. 25, 481–488 (2009)
22. R Development Core Team, R: A Language and Environment for Statistical Computing, R Foundation for Statistical Computing, Vienna, Austria (2011)
23. Urbanek, S.: rJava: Low-Level R to Java Interface (2009), http://cran.r-project.org/package=rJava
24. Wilde, M., et al.: Swift: A language for distributed parallel scripting. Parallel Computing 37(9), 633–652 (2011)
25. Murray, D.G., Hand, S.: Scripting the cloud with skywriting (2010)
26. Morgan, M.M., Grimshaw, A.S.: Genesis ii - standards based grid computing. In: CCGRID, pp. 611–618 (2007)
27. Anjomshoaa, A., et al.: Job Submission Description Language Specification (2005)

28. van Hensbergen, E., Evans, N.P., Stanley-Marbell, P.: A unified execution model for cloud computing. In: LADIS (October 2009)
29. Thain, D., Livny, M.: Parrot: Transparent user-level middleware for data-intensive computing. Scalable Computing: Practice and Experience 6(3), 9–18 (2005)
30. Azab, A., Meling, H.: A Virtual File System Interface for Computational Grids. In: Aagesen, F.A., Knapskog, S.J. (eds.) EUNICE 2010. LNCS, vol. 6164, pp. 87–96. Springer, Heidelberg (2010)
31. Condor submit description file, `http://www.cs.wisc.edu/condor/manual/v7.6/condor_submit.html`
32. Batch processing, `http://www.hpcx.ac.uk/support/documentation/#userguide/hpcxuser/batch_processing.html` (retrieved 6, 2011)
33. Microsoft.Net (2011), `http://www.microsoft.com/net/`
34. Tierney, L., Rossini, A.J., Li, N., Sevcikova, H.: Snow: Simple Network of Workstations (November 2011)
35. Boyer, C.B.: A History of Mathematics, 2nd edn., p. 431. Wiley (1968)
36. Condor Java API, `http://staff.aist.go.jp/hide-nakada/condor_java_api/`
37. Bowbrick, S., Borg, A.: ECG complete. Churchill Livingstone (2006)
38. Safar, P.: History of cardiopulmonary-cerebral resuscitation. In: Cardiopulmonary Resuscitation, New York, pp. 1–53 (1989)
39. Klim, S., et al.: Population stochastic modelling (psm)-an r package for mixed-effects models based on stochastic differential equations. Comput. Methods Prog. Biomed. 94, 279–289 (2009)
40. Schuller, B., Demuth, B., Mix, H., Rasch, K., Romberg, M., Sild, S., Maran, U., Bała, P., del Grosso, E., Casalegno, M., Piclin, N., Pintore, M., Sudholt, W., Baldridge, K.K.: Chemomentum - UNICORE 6 Based Infrastructure for Complex Applications in Science and Technology. In: Bougé, L., Forsell, M., Träff, J.L., Streit, A., Ziegler, W., Alexander, M., Childs, S. (eds.) Euro-Par Workshops 2007. LNCS, vol. 4854, pp. 82–93. Springer, Heidelberg (2008)