TakTuk, Adaptive Deployment of Remote Executions

Benoit Claudel
INRIA Sardes research team
CNRS LIG Laboratory
Grenoble University, France
Benoit.Claudel@imag.fr

Guillaume Huard
INRIA Moais research team
CNRS LIG Laboratory
Grenoble University, France
Guillaume.Huard@imag.fr

Olivier Richard
INRIA Mescal research team
CNRS LIG Laboratory
Grenoble University, France
Olivier.Richard@imag.fr

ABSTRACT

This article deals with TakTuk, a middleware that deploys efficiently parallel remote executions on large scale grids (thousands of nodes). This tool is mostly intended for interactive use: distributed machines administration and parallel applications development. Thus, it has to minimize the time required to complete the whole deployment process.

To achieve this minimization, we propose and validate a remote execution deployment model inspired by the real world behavior of standard remote execution protocols (`rsh` and `ssh`). From this model and from existing works in networking, we deduce an optimal deployment algorithm for the homogeneous case. Unfortunately, this optimal algorithm does not translate directly to the heterogeneous case.

Therefore, we derive from the theoretical solution a heuristic based on dynamic work-stealing that adapts to heterogeneities (processors, links, load, ...). The underlying principle of this heuristic is the same as the principle of the optimal algorithm: to deploy nodes as soon as possible. Experiments assess TakTuk efficiency and show that TakTuk scales well to thousands of nodes. Compared to similar tools, TakTuk ranks among the best performers while offering more features and versatility. In particular, TakTuk is the only tool really suited to remote executions deployment on grids or more heterogeneous platforms.

Categories and Subject Descriptors

D.4.9 [**Operating Systems**]: System Programs and Utilities—*Command and control languages*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems

General Terms

Algorithms, Experimentation, Performance

Keywords

work-stealing, remote executions deployment, adaptivity

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
HPDC'09, June 11–13, 2009, Munich, Germany.
Copyright 2009 ACM 978-1-60558-587-1/09/06 ...$5.00.

1. INTRODUCTION

Nowadays, most of highest performance computers in the world are made of thousands of computation nodes that might themselves contain several processors or processors cores. Grids are expected to become the next major evolution of highest performing platforms in the world. Like usual desktop machines, these huge platforms have to be administrated (updates, configuration, monitoring, ...) and exploited for the development and the execution of parallel applications. These administration tasks as well as the development of parallel applications often require to perform the same operation or to execute the same program on all the computation nodes. Furthermore, the results of all these executions have to be gathered for diagnostic and analysis. For instance, a grid administrator might be interested in executing the command `uptime` on all the nodes of the platform in order to gather statistics about nodes availability. This would be the same for the developer of a master-slave distributed application who needs to execute slave processes on all computation nodes of the grid (as an example, this is what does `mpirun` for MPI applications). This task of collective remote executions is called remote executions deployment (or deployment for short). The deployment has to be highly efficient because its main uses are administration tasks and development of parallel applications which are interactive tasks. In other words, the user waits for the result of his deployment before deciding what to do next. Thus, in order to be scalable, the whole deployment time should not grow linearly with the number of machines, otherwise it would become useless for grids.

The remote executions deployment is made of two steps: the first one involves the connection to remote nodes along with a setup of a logical communication network, while the second one is made of the actual execution of the program on all the connected nodes and the redirection of their I/Os. Usually, the most difficult part of the deployment is the efficient management of remote connections initiation as this is the most time consuming step of the whole process. Among existing deployment solutions two approaches can be identified that address this problem:

- centralized approaches in which a single master node set up all the connections by itself. Using a centralized approach greatly eases the I/Os redirection of remotely executed commands to the initiating machine. Nevertheless it results in a deployment time which is linear in the number of nodes.

- distributed approaches in which the remote nodes take part of the deployment process by running a deployment engine that will eventually spawn the program and redirect I/Os. Although this approach can lead to a logarithmic deployment time, it suffers the overhead of initiating the deployment engine on all the remote nodes.

In the first case [10, 5, 3, 9, 4], the connections average initiation time is reduced by using concurrency: several connections are initiated in parallel. But, as we will demonstrate in this article, this is limited by the characteristics of the master node and is generally not scalable above several hundred of nodes. In the second case, the deployment is distributed among nodes using a tree topology to setup connections and logical communication network. To our knowledge, only `gexec` [8] uses this approach. Although this tool is very efficient, it is quite intrusive as it requires the installation of a deployment server on remote nodes. In the context of grid platforms, this tool suffers from even more serious issues: the deployment topology is fixed statically and the deployment fails if any node is down (due to failure or maintenance).

Looking back at the deployment problem from a general point of view, spreading the execution of some program on remote nodes and collecting their results reminds of usual broadcast and gather operations in a network. These two communication primitives have been extensively studied for several variants of send/receive model with 1 or k ports. One might think that the models used in this community are different than the deployment model. In this article, we show this is not the case: we conducted experiments that validate a deployment model which is similar to the *postal* model used in networking [1]. Consequently, taking advantage of existing results related to this *postal* model, we deduce an optimal solution for the deployment on homogeneous machines: as in the case of broadcast communication, the scalability of deployment will be achieved by using a tree topology for the remote connections spreading and the associated logical communication network setup. Nevertheless, when the execution platform is heterogeneous, further investigations are required in order to design an efficient deployment algorithm.

The optimal algorithm for the homogeneous case initiates connections as soon as possible. To reach this result, it distributes work among nodes and parallelizes a small quantity of connection tasks within each node. When we developed TakTuk, we kept all these ideas in mind to design an heuristic which is efficient on all kind of platforms that would converge toward the optimal on homogeneous machines. This is achieved in TakTuk by using an adaptive work-stealing ([2, 7]) algorithm to balance local parallelization and work distribution. To avoid issues that hinder existing tools in grid context, we also forced ourselves to meet the following constraints: no assumption about the network capabilities, no required installation on remote hosts, possibility to statically fix a part or all the deployment topology.

In the remaining of this article, we first present in section 2 some preliminary experiments regarding standard execution protocols and we derive from them a connection model that allows us to solve the optimal deployment problem on homogeneous machines. In section 3, we introduce our approach that extends the optimal solution to address heterogeneity among nodes and in the network. We also present the overall architecture and implementation characteristics of TakTuk. In section 4 we experimentally validate our tool. In section 5 we present in more details projects related to the deployment problem and we explain how they compare to TakTuk. We also present two projects that make use of TakTuk. Finally, in section 6 we conclude by giving a summary of our results and we present the next evolutions of TakTuk.

2. COMMUNICATION MODEL

As TakTuk is intended to be used on very large scale platforms and grids, genericity and portability have been two strong constraints during its design phase. Thus, our tool has to be built on top of standard communication methods installed on all parallel infrastructures: this will ensure it the maximal genericity. Taking this consideration into account, we built TakTuk on top of standard remote shell execution facilities (such as `rsh` or `ssh`) that are installed on all UNIX-like systems (which is the kind of operating system running on most grids). Notice though, that this does not lessen the generality of our approach as TakTuk is able to use any remote execution mechanism with I/O redirections to setup its own communication network. The only difference in using another remote execution method would be the duration of the two parts of the connector model that we present in this section.

The `rsh` and `ssh` commands open a TCP connection to a remote host and execute a command on this remote host redirecting all I/O to the initiating machine. This remote execution mechanism and the associated point-to-point communication that constitute the I/O redirection is the basis on which TakTuk is built. The duty of TakTuk is thus to parallelize and distribute among deployed hosts these remote connection initiations in order to achieve an efficient deployment. It is therefore important to precisely know the behavior of the connection commands in order to decide on the balance between local parallelization and work distribution. Any tool which performs a remote execution is based on the client/server model: the client sends a request for connection and waits for the server answer before considering the connection as operational. This means that the client machine is idle during the wait for the connection acknowledgement: it should be possible to overlap several connection initiations. This is what centralized approaches do and this explains why they are efficient on a small number of remote hosts. But to scale well as the number of involved hosts increases, we have to know the limits of the centralized approach and its local parallelization. Thus, we have conducted a batch of experiments to find out how simultaneous connection initiations overlap on the same machine.

The first experiment aims at understanding the global behavior of a group of concurrent processes performing connection initiations. It has been performed on the single cluster machine described in detail in section 4.1. In this experiment, we create a group of 100 background processes (using the `fork` system call) remotely executing the command `true` (negligible execution time) on a distinct host using either `rsh` or `ssh`. Using Linux scheduler capabilities, we have forced all the processes to be executed on the same single processor core. We have measured the completion time of each individual process involved in this experiment. These measurements are presented in figure 1 in which the processes are numbered according to their creation order. As we can notice in the figure, the completion of individual processes

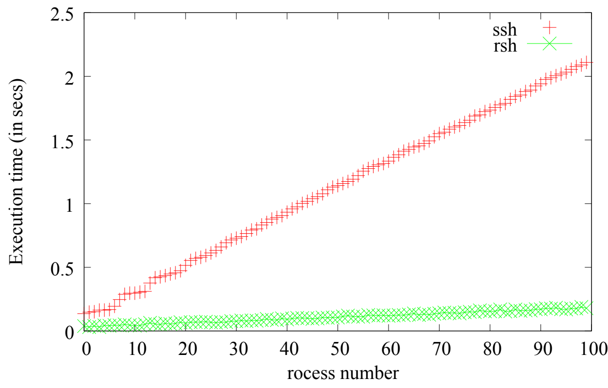

Figure 1: Behavior of concurrent connections using either `rsh` or `ssh` on a single core.

is shifted in time by a regular amount. This phenomenon is not only due to the round-robin CPU assignment in the scheduler which occurs at a finer grain. In contrary, it seems like the processes are naturally pipelined by the system due to some contention on a common resource (processor core, network interface, ...).

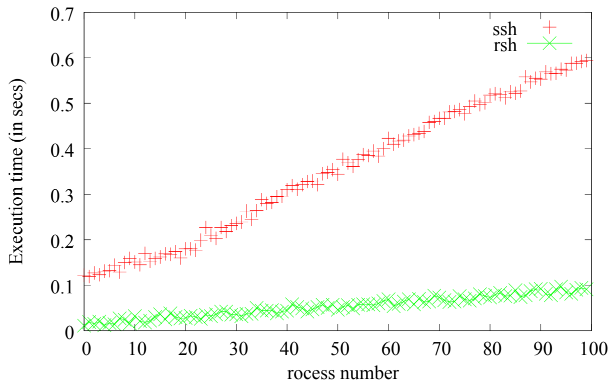

Figure 2: Behavior of concurrent connections using either `rsh` or `ssh` on multicore (4 cores).

When executing the same experiment on a multicore machine (4 cores), the contention on the processor is greatly reduced and the 100 connections complete roughly 2 times earlier for **`rsh`** and 3.5 times earlier for **`ssh`**. Although the measures are more chaotic, they keep this general linear behavior: the resources are still limited (number of cores fixed, network interface...) and some contention still occurs. This second experiment is presented in figure 2. As a consequence, it seems that a local parallelization always ends up in a near linear time progression of connections execution with some overlap. One might think that this parallelization behaves as a pipeline and could even be performed explicitly as a pipeline: it is useless to create from start more processes than the resources can handle.

To check this last hypothesis, we conducted a new multicore experiment in which we explicitly start connection processes in pipeline. More precisely, we have taken the same previous experiment (100 processes performing connections) but we have inserted a small wait delay before the creation of each process. We have measured the completion time of the last process and have repeated the experiment for various values of the delay. Figure 3 presents these experiments: the two curves report the total completion time of the group of processes depending on the inserted delay respectively for **`rsh`** and **`ssh`**. On this figure, the point for which the inserted delay is null (0 abscissa) corresponds to the completion time of the last process in the previous experiment: around 0.1s for **`rsh`** and near 0.6s for **`ssh`**.

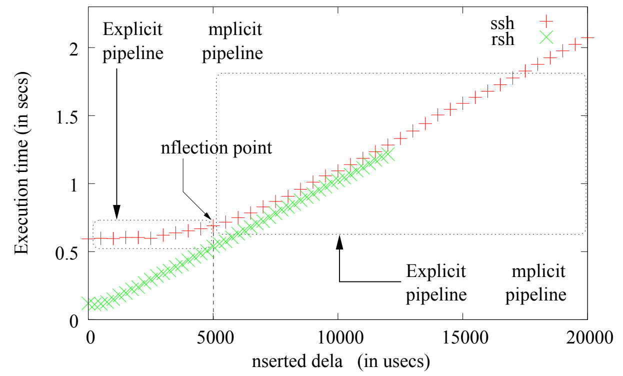

Figure 3: Comparison between fully concurrent processes and explicit pipeline on multicore (4 cores).

On this figure, both curves are made of a constant part at the beginning followed by an increasing linear part. The inflection point is around 1ms for **`rsh`** and 5ms for **`ssh`**. If calculate the average shift between process in the previous experiment (figure 2), we obtain 1ms for **`rsh`** (almost 0.1s between the completion of the first process and the last one divided by 100 processes) and 5ms for **`ssh`** ((0.6s-0.1s)/100). The average shift is the same as the position of the inflection point on figure 3. Thus, as long as the inserted delay is lower than this shift amount, the overall execution time does not change. This means that doing an explicit pipeline still lets connection processes overlap nicely as they would do if they were started at the same time. Nevertheless, above this limit, the inserted delay degrades the overall execution time. This certainly indicates that the total connection initiation time is fixed and does not decrease as resources become idle.

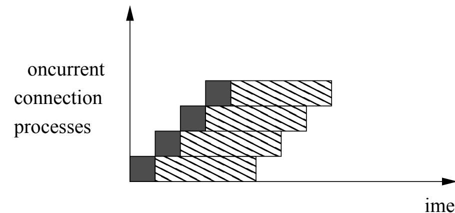

Figure 4: Behavior model for concurrent `rsh` or `ssh`.

These experiments show that we can model the execution of concurrent connection processes as a pipelined execution of fixed time tasks. The Gantt diagram in figure 4 presents this model. In this model, each connection task is composed of a part which can be overlapped and another one which cannot be overlapped. This simplified view of connections is presented in figure 5. We name T the total time it takes to initiate a connection (after this time, the remote host is ready to execute commands) and t the part which cannot be overlapped (which is the same as the shift

amount in the pipeline). For each connector command, this model expresses the possible gain that local parallelization can achieve and the limit above which additional concurrent processes are useless. This also enables to decide above which number of remote connections it is interesting to use a remote node to execute a child deployment engine and distribute the deployment process. Indeed, when the connection to N hosts is required and $Nt > T$ the deployment will be faster with some distribution. This idea is represented as Gantt diagram in figure 6. On each host, the number of simultaneous ongoing connection initiations is limited to the width of the pipeline: at most T/t processes. We call this value the concurrency index of the pipeline window.

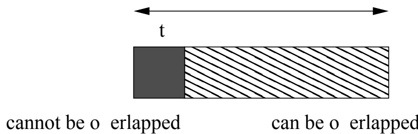

Figure 5: Overlapping model for `rsh` and `ssh`.

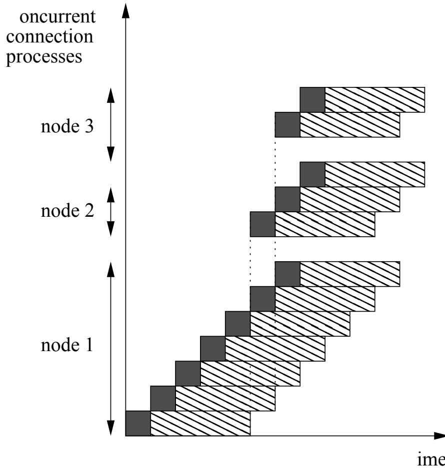

Figure 6: Optimal deployment mixing local parallelization and distribution.

Actually, this model is the same as the *postal model* presented in [1] and our deployment problem is also the same as the broadcast problem from one node to N they study. In this article, they present an optimal solution to the broadcast problem in question. This optimal solution is an “as soon as possible” (ASAP) schedule of communication tasks. This optimal schedule is straightforward to construct with a simple linear algorithm (iteration on tasks completion time along with greedy task scheduling). This optimal solution forms a deployment tree which arity depends on the respective values of t and T. The extreme cases are a flat tree when $t << T$, a chain when $t >> T$, and a binomial tree when $t = T$. Notice that the case $t >> T$ is unlikely to occur in our deployment problem as this would mean that the remote host is ready to deploy before the completion of the connection in the originating host. Outside the scope of the deployment problem, this situation could occur when transmitting large volumes of data. In such case the remote host can start forwarding chunks of data before having completely received the whole message.

3. TAKTUK, SCALABLE DEPLOYMENT

Although we presented an optimal deployment algorithm in section 2, its straightforward implementation is not really relevant as it raises several issues. First, it constructs the solution based on t and T values which depend on the deploying host performance and on the connector used. There is no obvious way to guess them for any deployment target machine. Second, it does not take into account implementation issues: distributing deployment enforces the remote execution of subdeployment engines. This implies a connection overhead and changes the values for t and T. Obviously a “‘real world” optimal would be the minimum of flat deployment without overhead and ASAP deployment with overhead. Third, it assumes that the target machine is homogeneous (same t and T on all hosts), which is not the case in general. Obviously, grids are not homogeneous, and even cluster might sometimes have aberrant nodes that are slower than usual due to unusual perturbing processes (zombies, deadlocking application, ...). Fourth, some nodes of a cluster might fail to answer to connection requests (hardware or software failure, interconnection problem, ...). In this last case, it is necessary to recompute the deployment tree to force it to adapt to actual resources availability. In summary, because each machine is unique, the deployment process should adapt dynamically to the target machine capabilities.

To achieve this adaptivity in TakTuk, we have chosen to use the well known work-stealing approach [2, 7] to distribute deployment tasks on the deployed hosts. The drawback of this approach is that we always pay the overhead for distribution of the engine on remote hosts: on small machines a flat deployment should be faster. This is not really an issue as our main target is grid platforms and large scale deployment. In our approach, each connection to a remote host will start the execution of a remote TakTuk engine instance that we will call the child instance. Then, whenever a TakTuk instance is idle, it asks for new deployment tasks to its father. Using such work-stealing approach tends to produce an ASAP schedule (assuming that the penalty for the work-stealing negotiation is negligible) which is the principle of the optimal algorithm. This approach also nicely handles nodes heterogeneity: if a node is slower than others, it will take more time to perform its deployment tasks and will ask for more work less often. Furthermore, to ensure that we avoid as much as possible ineffective nodes while lessening the overhead of the work-stealing approach, we use an amortized policy: when a TakTuk instance receives the first steal request from one of its children it gives it only one deployment task; then, whenever this child asks for more work it gives it twice the previous tasks number (limited to half the remaining tasks).

To tell all the truth, one question remains unresolved despite the use of our work-stealing approach: how do we implement the connection processes pipeline that should run in all the TakTuk instances ? Inserting delays before processes creation would require to guess the value of t, which is not easy (as already mentioned). In TakTuk, we have chosen instead to create a window of connection processes: at any time, the number of concurrent connection initiations is limited by the size of the window. Due to the contention

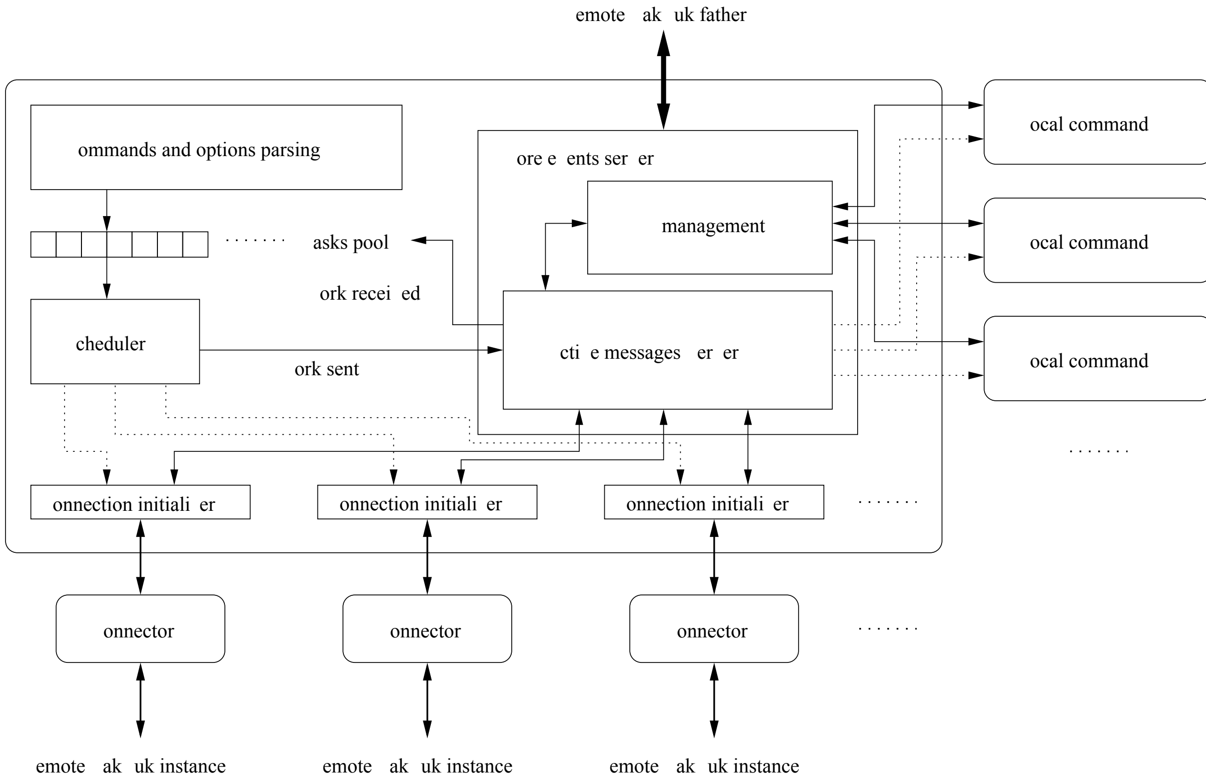

Figure 7: TakTuk Architecture.

on system resources, this results in a natural pipeline. But to know the size of this window requires, once again, to guess t and T. This is an important issue as a too small size results in missing opportunities for local parallelization and a too large size results in insufficient work distribution. Currently, we do not have a good answer to this last question. We investigated the local monitoring of system load average indicator but it is not updated sufficiently often to enable reactive window adaptation (the time scale required is in the order of tens of milliseconds). We also investigated direct measurements of already completed connections but this tend to render TakTuk overreactive (this last effort is included in TakTuk as an experimental feature). At the time of this writing, the best compromise we have found is to fix this size to 10, which is empirically suited to most platforms. For a long term solution on a fixed platform, one can perform the experiments presented in section 2 to measure t and T and fetch T/t as a window size to TakTuk.

From an architectural point of view, TakTuk is built on top of a central communication server. This communication server implements the *active messages* paradigm [21] and the TakTuk engine is mostly made of active message handlers. New connections are performed by creating new processes that execute the connector command. Once the deployment is complete, new commands are executed by the usual `fork/exec` idiom, and all their outputs are redirected to the root node. All the interactions between involved entities (local connection processes, local command processes) are triggered by I/O events and are implemented using active messages for communication. This global overview is represented in figure 7.

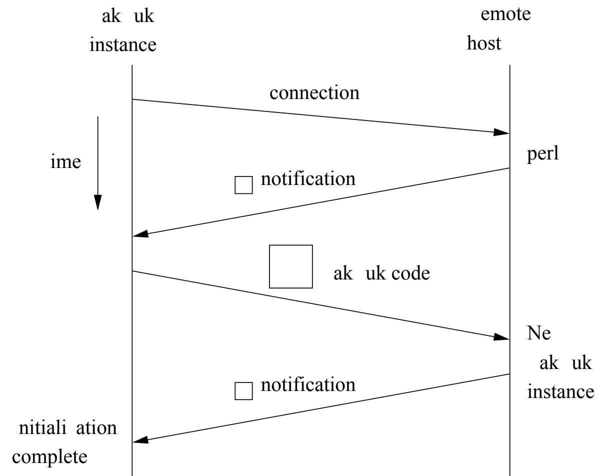

Figure 8: TakTuk self propagation mechanism.

The current version of TakTuk is implemented in Perl, which has been chosen for two main reasons. The first reason is its portability, Perl is available on most computing platforms. The second reason is its interpreted nature, this enables us to transfer the TakTuk code to remote nodes for children instances creation without requiring an installation of the tool on all the nodes of the executing platform. The drawback of using Perl is, of course, its inefficiency compared to compiled languages such as C. As of today, TakTuk has been used for the efficient deployment of real world appli-

cations on up to 4000 processor cores (see section 4) and we do not think Perl caused any major performance issue. Actually, TakTuk is mostly I/O intensive rather than computationally intensive, this is the reason why the relative inefficiency of Perl is not really hindering.

As we just mentioned, TakTuk does not require any installation on remote nodes on which it deploys itself. We call this feature "self propagation". It relies on the interpreted nature of Perl (no compilation required) and the ability of the interpreter to fetch code from the standard input. This self propagation mechanism works in three steps: first it uses a connector command to remotely send a notification and execute a perl interpreter, second it fetches the TakTuk code to the remote interpreter, third, upon reception of TakTuk initialization notification, it adds the new remote instance to the TakTuk network and continues the deployment. This mechanism is summarized in figure 8. Of course this self propagation mechanism slows down the remote TakTuk instance initiation and the overall deployment time. On stable production sites, it is possible to install TakTuk on all the nodes of the execution platform and to use simple remote TakTuk execution instead of self propagation.

Overall, TakTuk has been designed with versatility in mind. The deployment tree it constructs might be completely static and given at launch time, dynamically determined during deployment, or a mix of both. Its outputs are customizable using very general templates. Its options (connector command, self propagation, connections window, tree topology,...) can be changed in any part of the deployment tree. Thanks to all these features, TakTuk is completely suited to applications deployment on highly heterogeneous platforms (Grids, P2P networks, ...). It is distributed under the terms of the GPL license and is available in the Debian GNU/Linux distribution or downloadable at [20].

4. EXPERIMENTS

In this section, we evaluate the real wold performance of `TakTuk`. It is made of the following parts: the presentation of the experimental platform we use, the evaluation of the TakTuk engine overhead, the comparison between TakTuk and flat deployment tools, the comparison between TakTuk and distributed deployment tools, and finally the presentation of some experiments on grid platforms.

4.1 Experimental setup

In the remaining of this section, we use two distributed platforms to analyze the behavior of TakTuk and other deployment tools. The first platform, the single cluster platform, is constituted of 100 nodes of the *grelon* cluster which is part of the Grid5000 platform [14]. This cluster is made of bi-processors nodes that contain two Intel Xeon 5110 (dual-core 2.4 Ghz) and 2GB of RAM. Nodes are interconnected using a switched Gigabit Ethernet network. The second platform, the multi cluster platform, is constituted by 484 real nodes of Grid5000 (respectively 112, 47, 48, 70, 17, 25, 27, 60, 57, 7 and 14 nodes of the *grelon, grillon, capricorne, sagittaire, chti, chicon, pastel, paravent, paraquad, paramount and bordereau clusters*). These nodes are made of either one or two dual-core processors (clocked in the range of 2.0 GHz to 2.6 GHz) and at least 2 GB of RAM. Distinct cluster sites are connected by a 10Gb/s backbone provided by RENATER (the French national network for research and education). To scale to 2000 nodes, we virtualize these real nodes: we use them several time as remote host in a round-robin manner.

In all the experiments, nodes are configured with a clean Debian GNU/Linux OS installation (unstable branch, Linux kernel 2.6.24) that does not make use of any external server (such as NFS, LDAP, DNS, etc.). This ensure that no contention can occur due to external source. To measure deployment time, we remotely execute the command `true` which has a negligible execution time. All the scripts written to run experiments and all the raw data we obtained can be downloaded from the svn repository of the TakTuk developers website, see [20].

4.2 Evaluation of the TakTuk engine overhead

In this section we compare the cost of contacting remote hosts using TakTuk to the raw cost of the connectors used (`rsh` or `ssh`). The overhead induced by TakTuk is due to: the remote execution of a TakTuk instance, the internal mechanics for communication and I/Os redirections, and the possible "self propagation" (when in use). This overhead can be deduced from the comparison of a flat deployment using TakTuk (work-stealing disabled) with the raw parallel execution of connection commands (as presented in section 2). The figure 9 presents the time to complete the execution of the command `true` on remote hosts with TakTuk depending on the connector used (`rsh` or `ssh`) and the use of self propagation or not. This experiment has been performed on the single cluster platform presented in section 4.1.

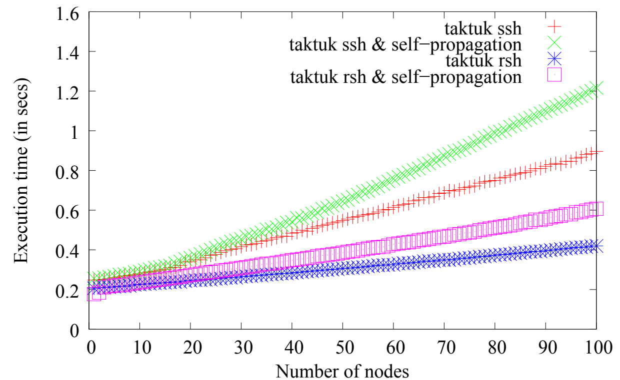

Figure 9: TakTuk flat deployment cost depending on connector and use of self propagation.

Compared to `rsh`, the total connection time T (calculated as in section 2) is increased from roughly 10ms to almost 210ms and the part t, which cannot be overlapped, from less than 1ms to 2ms. As T is a fixed cost in the deployment, only t is relevant for asymptotic comparison. Thus, TakTuk induces an overhead of roughly 100% with the `rsh` connector. When using self propagation, T remains roughly the same and t increases to 4ms, a 300% overhead compared to raw `rsh`. Overall, because `rsh` is very light and efficient, TakTuk overhead is comparatively significant.

Compared to `ssh`, the total connection time T is increased from roughly 120ms to almost 250ms. For the part which cannot be overlapped, we can notice that the beginning of the curve is almost flat: the computations related to the connections is hidden, overlapped with TakTuk execution. For a fair evaluation of t, we should only consider a number of connections above 20. This results in an increase of t

from 5ms to almost 7ms: a 40% overhead. When using self propagation, T remains roughly the same and t increases to more than 10ms, a 100% overhead compared to raw `ssh`. Overall, the relative overhead induced by TakTuk is less significant for computationally intensive connectors.

Regarding self propagation, the overhead of its use is very reasonable considering that it enables distributed deployment without any installation of a TakTuk executable or daemon on the remote nodes. In all the following experiments, we do not use self propagation.

4.3 Comparison with flat deployment

In this section, we compare TakTuk with `pdsh` [3] on the multi cluster platform. `pdsh` is a free clone of the `dsh` command included in the software suite of IBM clusters. `pdsh` is a highly efficient tool that can perform flat deployment of remote executions. It is written in C and uses a window of parallel connection threads. As a flat deployment engine, it requires neither the remote launch of subdeployment engines nor any installation on the remote nodes. Connectors used by `pdsh` are developed as plugin modules and developing a new one require little effort. `rsh` and `ssh` connectors are included in the `pdsh` distribution. `pdsh` is available in the Debian GNU/Linux distribution and is widely used in clusters as an administration tool.

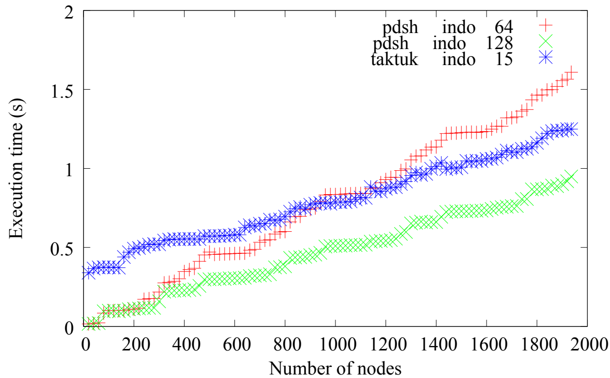

Figure 10: TakTuk compared to pdsh using rsh.

When using a fast connector such as `rsh`, the optimal deployment is likely to be very close to a flat tree, and the overhead of deploying the engine in TakTuk will be more hindering than helpful. This is confirmed by the experiment presented in figure 10, which reports the respective execution time of TakTuk and `pdsh` (using two different sizes for the window) depending on the number of nodes. The performance irregularities in this experiment are due to the heterogeneous nature of the multi cluster platform: the almost flat parts match the addition of new nodes of the same cluster, while the discontinuities match the addition of nodes in a different cluster (contacted using the Grid5000 backbone). The same pattern is then repeated as we use the same 480 nodes sequence for the virtualization.

We can notice that the performance of `pdsh` is highly dependent on the window size, 128 is the best we have found. This is less relevant for TakTuk, for which we observe around 10% of degradation for less appropriate window size (for this slightly heterogeneous platform, a window size of 15 was better than the default). As expected in this context, TakTuk is outperformed by `pdsh` by roughly 350ms provided that we use the correct window size for `pdsh`. We expect that for a sufficiently large number of nodes, the asymptotically logarithmic execution time of TakTuk will eventually give better results than `pdsh`. Notice also that choosing an inappropriate window size for `pdsh` makes TakTuk become more efficient much quicker.

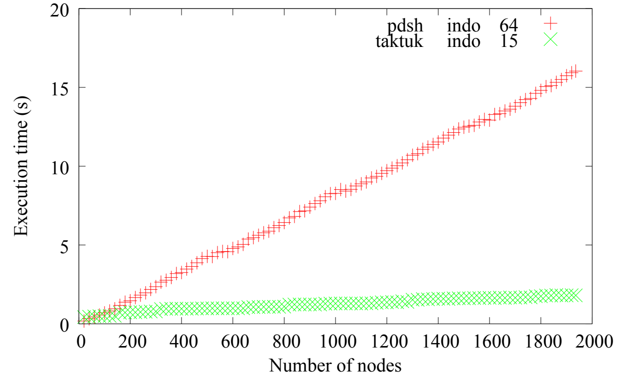

Figure 11: TakTuk compared to pdsh using ssh.

But the real targets of TakTuk are grids and heterogeneous platforms for which the available connection methods often reduce to `ssh` (this is the case in Grid5000 which enforces either `ssh` or its own variant `oarsh`). In this context, the connector requires significant computational power and the optimal solution is far from a flat tree. This is confirmed by the experiment presented in figure 11: in this experiment, the performance of `pdsh` (with a window size of 64, the best we have found) is badly hurt by its linear nature. Above 100 nodes, TakTuk is clearly faster, and the gap linearly increases with the number of nodes.

4.4 Comparison with distributed deployment

In this section, we compare TakTuk with the only tool we know which uses distributed deployment, `gexec` [8], on the multi cluster platform. This tool requires the installation of a daemon, `gexecd`, on all the remote nodes with administrative privileges. `gexec` uses its own connection protocol that performs `ssl` authentication but does not encrypt data. Thus, this is a protocol in between `rsh` and `ssh`. `gexec` is part of the Ganglia toolbox for clusters [17].

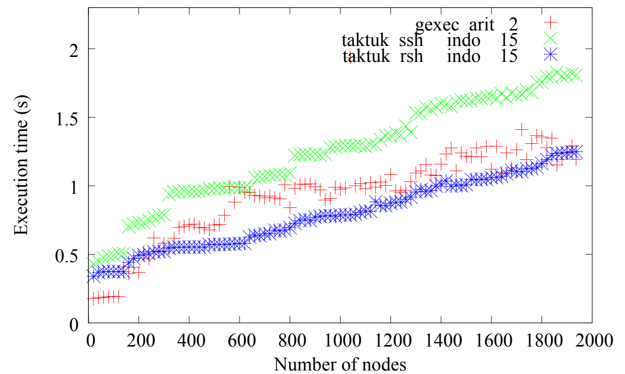

Figure 12: TakTuk compared to gexec.

The deployment in `gexec` uses an n-ary tree determined at launch from the list of remote hosts to contact. Knowing

in addition that gexec is written in C, it should outperform TakTuk on homogeneous machines. The experiment presented in figure 12 reports the comparison between gexec and TakTuk using rsh and ssh connectors. As expected, gexec clearly outperforms TakTuk on the first 150 nodes which belong to the same cluster. Above this number, the platform starts being slightly heterogeneous and the performance of gexec degrades: it ranges between the two TakTuk variants (rsh and ssh). This constitutes a fairly good result knowing that TakTuk is written in Perl and does not rely on remote TakTuk daemons to perform its deployment. We should also mention that, contrary to TakTuk, gexec fails on a large number of virtualized nodes (4000 and more). We do not know if the problem comes from gexec or from the virtualization, but the lack of issue with TakTuk suggests the former.

4.5 Deployment with failing nodes

Node failures often occur on very large platforms such as grids. This is due to hardware and software failure rates which translate into a non null number of crashed machines for a sufficiently large number of nodes. It is therefore important, for a deployment tool targeted at grids, to handle nicely nodes failure. This is the case for pdsh which reports unresponsive nodes as connection timeouts occur. This is not the case for gexec which cancels the whole deployment whenever any connection fails, making it even more inappropriate for grids.

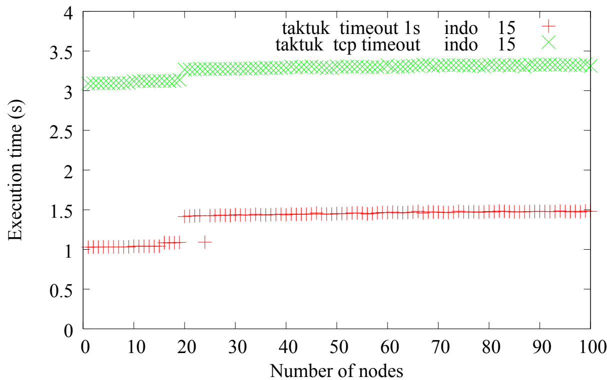

Figure 13: TakTuk in presence of faulty nodes.

Unresponsive nodes are simply excluded from deployment by TakTuk which naturally adapts its deployment tree to reactive nodes. By default, a node is considered faulty upon termination of the connector before TakTuk initialization (caused by built-in TCP timeout for rsh and ssh). Nevertheless this default behavior can be overridden by using TakTuk internal timers to cancel connection to unresponsive nodes earlier. This is summarized by figure 13 which reports TakTuk execution on a group of nodes including 10% faulty nodes (rounded up uniformly distributed). The deployment time is mostly dominated by the timeout value: the responsive nodes perform the deployment flawlessly and the failing nodes are the last ones that TakTuk waits in order to complete its execution.

4.6 Large deployment on a hierarchical grid

The GRIDS@WORK plugtest [19] is an annual event organized by ETSI in collaboration with other international Grid initiatives and several sponsors. The objective of this event is to gather international teams competing for the resolution of a large computationally intensive problem. We have been participating to this event for three consecutive years (since 2006), using TakTuk to deploy a KAAPI [12] parallel application. Each of our participation ended up with a victory of our team thanks to KAAPI/TakTuk efficiency.

During the fifth edition of this event that occurred in 2008, the challenge consisted of the computation of a financial math computation problem. This problem is embarrassingly parallel and the difficulty lies in deployment, nodes performance heterogeneity, and network topology. The execution platform consisted of Grid5000 and Intrigger [16] (a Japanese hierarchical grid made of a cluster of clusters). Connections to computing nodes had to be performed using ssh on InTrigger and oarsh on Grid5000. Direct communication between nodes of InTrigger and nodes of Grid5000 was impossible (deployment on the whole platform had to be subdivided into two subdeployments). During this event, TakTuk successfully deployed a single KAAPI application on more than 3600 distinct processor cores belonging to both platforms. Of course the deployment were not hindered by faulty nodes (several hundred of nodes were down due to maintenance) and the application terminated successfully despite two node failures during its one hour time slot. This experiment assesses TakTuk scalability despite connectors heterogeneity and partial network connectivity.

4.7 Features comparison

Our target, grids and large scale heterogeneous platforms in general, is often constituted of several clusters with distinct administrative domains. Thus, installing executables or daemons on remote nodes is not always possible. The connection method to remote nodes should not be limited to rsh or ssh but rather be easily extensible. Furthermore, different parts of the target platform might require different connection methods: a deployment tool for grids should be able to mix several connectors in the same deployment. Because of their large scale, target platforms always include unresponsive nodes, this should not hinder the deployment tool which is still expected to deploy on the subset of functional nodes. Also, because of this large scale, the deployment should be distributed: this is the only way to scale to thousands of nodes whatever the connection method in use. All these important characteristics are compiled in table 1 for main deployment tools.

5. RELATED WORKS

In this section, we reference tools that also perform deployment but are less relevant to this article and project that use TakTuk to solve specific problems.

5.1 Comparable tools

Beside the tools specifically designed for remote executions deployment mentioned in the introduction (section 1), there exists some systems, developed for other purposes, that include deployment as a part of their functionalities. This is the case of Slurm [23] which is a resources management system that is also able to perform deployment. As we can expect, the deployment part in this tool does not implement a very elaborated algorithm: this is a locally parallelized deployment that connects to slurm daemons installed on all the remote nodes. As in the case of gexec,

	No remote installation required	New connector plugin	Can mix several connectors	Insensitive to nodes failure	Distributed deployment	Compiled engine
TakTuk [20]	Yes	Immediate	Yes	Yes	Yes	No
pdsh [3]/dsh	Yes	Simple	Yes	Yes	No	Yes
gexec [8]	No	No	No	No	Yes	Yes

Table 1: Features comparison table for several deployment tools.

we do not believe that such system are suited to grids and heterogeneous platforms. Furthermore, its performance is certainly similar to the performance of `pdsh`. The same arguments hold for `mpirun`, the deployment command of MPI applications [22].

Some deployment system alleviate the overhead of connection process initiation by integrating it deeply into the kernel code. This the case of GLUNIX [13] or Bproc [15]. These systems remind of single system image cluster operating systems, they integrate the management of remote processes directly into the kernel code. The deployment is still linear, but the overhead is reduced to a bare minimum. Nevertheless, these systems are so intrusive that we consider them completely unsuited to grids and heterogeneous machines. For this reason, we have chosen to compare TakTuk to `pdsh` which is less performant but also less intrusive. There even exists tools that rely on hardware capabilities to solve the issues related to deployment. This is the case of the Storm project [11] which rely on Quadrics broadcast facilities [18] to deploy in constant time. Because of this hardware requirement, this project is obviously out of the scope of this article.

5.2 Projects using TakTuk

TakTuk is used in several projects regarding distributed platforms and parallel programming. In this section, we present briefly the most relevant of these projects and the problem TakTuk solves for them.

- KAAPI [12] is a parallel programming environment. It proposes a programming model that abstracts parallel tasks and the data they share. Using this model, KAAPI then automatizes the creation and scheduling of parallel tasks as well as the communications between them. TakTuk is used as the basis of the KAAPI application launcher: `karun`.

- OAR [6] is a batch scheduler for clusters. It is able to manage "best effort" jobs as well as precise resources cleaning and it performs resources matching using a very elaborated selection scheme. TakTuk is used in OAR as a monitoring tool: it is launched periodically to check nodes availability.

6. CONCLUSION

In this article, we have presented the remote executions deployment problem which is of utter importance in the context of large distributed platforms administration and exploitation. We proposed a realistic communication model for remote connections that we validated with experiments. Thanks to this model and to well known results from the network communications community, we derived an optimal algorithm for optimal deployment on homogeneous machines. Unfortunately, this algorithm does not adapt to heterogeneity and requires informations about the platform that are difficult to compute.

Then, we presented TakTuk: a tool for large scale remote executions deployment. This tool has been designed from start to scale to grids and huge clusters. It relies on standard point-to-point connection tools available on any high performance computing platform, which makes it very portable. It implements a dynamic adaptation mechanism based on work-stealing that balances deployment tasks between local parallelization and remote distribution. In principle, on homogeneous platforms, this algorithm should have the same behavior as the optimal algorithm. Thanks to this dynamic adaptation mechanism and to the resulting efficiency and scalability, TakTuk is perfectly suited to interactive parallel tasks (such as cluster administration, or parallel application debugging and tuning) on both homogeneous and heterogeneous machines.

TakTuk is currently integrated in the software stack of several project related to experimental grid (Grid5000 [14]) or parallel programing environments (KAAPI [12]). Compared to other tools performing deployment tasks, TakTuk exhibits a performance which is among the best. Furhtermore, it is the only one to combine local parallelization and distribution using an adaptive algorithm, which makes it insensitive to platform heterogeneities or nodes failures. The closest tool we have found, `gexec` [8] uses a fixed n-ary tree. But because its tree is fixed, `gexec` is unable to cope with heterogeneity and missing nodes. It also requires an installation on all the remote nodes and is restricted to its own connection method. For all these characteristics, it is unsuited to deployment on grids.

Nevertheless, as we mentioned in section 3, some investigation are still required regarding the adaptation mechanism used in TakTuk. To reach the perfect balance between local parallelization and remote distribution, TakTuk needs to evaluate the available computing power of its execution machine at a rate which is in the milliseconds timescale. In the absence of such precise estimation, the adaptation mechanism only results in a coarse approximation of the optimal algorithm.

Another issue we noticed when using TakTuk is its behavior regarding centralized services. When performing large scale deployment, TakTuk might badly load centralized servers such as ldap or nfs. Such servers are not necessarily designed to handle massive requests bursts. For this issue, we foresee two solutions: avoiding using the service (this is easy with nfs for instance) or taking this external contention into account. Future work will also focus on this last option.

6.1 Acknowledgements

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, an initiative from

the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS and RENATER and other contributing partners (see https://www.grid5000.fr).

Special acknowledgements to Cyrille Martin who laid the basis of adaptive deployment, implemented former (and now obsolete) TakTuk engine and contributed to several ideas presented in this article.

7. REFERENCES

[1] A. Bar-Nov and S. Kipnis. Designing broadcasting algorithms in the postal model for message passing systems. In *ACM Symposium on Parallelism in Algorithms and Architectures*, pages 13–22, 1992.

[2] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Symposium on Foundations of Computer Science*, pages 356–368, 1994.

[3] R. L. Braby, J. E. Garlick, and R. J. Goldstone. Achieving Order through CHAOS: the LLNL HPC Linux Cluster Experience. Technical Report UCRL-JC-153559, Lawrence Livermore National Laboratory, 2003.

[4] R. Brightwell and L. A. Fisk. Scalable Parallel Application Launch on Cplant. In *Proceedings of the IEEE/ACM International Conference on Supercomputing*, pages 40–40, 2001.

[5] M. Brim, R. Flanery, A. Geist, B. Luethke, and S. Scott. Cluster Command and Control (C3) Tool Suite. In *Proceedings of 3rd Austrian-Hungarian Workshop on Distributed and Parallel Systems in conjunction with EuroPVM/MPI*, 2000.

[6] N. Capit, G. Da-Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounier, P. Neyron, and O. Richard. A batch scheduler with high level components. In *Cluster Computing and Grid 2005 Proceedings*, 2005.

[7] T. Casavant and J. Khul. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, 14:141–154, 1988.

[8] B. N. Chun. GEXEC, Scalable Cluster Remote Execution System. http://www.theether.org/gexec.

[9] B. N. Chun. PSSH. http://www.theether.org/pssh.

[10] B. N. Chun and D. E. Culler. REXEC: A Decentralized, Secure Remote Execution Environment for Clusters. In *Proceedings of 4th Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing*, 2000.

[11] E. Frachtenberg, F. Petrini, J. Fernandez, S. Pakin, and S. Coll. STORM: Lightning-Fast Resource Management. In *Proceedings of the IEEE/ACM International Conference on Supercomputing*, pages 1–26, 2002.

[12] T. Gautier, X. Besseron, and L. Pigeon. KAAPI: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In *Proceedings of the 2007 international workshop on Parallel symbolic computation*, pages 15–23, 2007.

[13] D. P. Ghormley, D. Petrou, S. H. Rodrigues, A. M. Vahdat, and T. E. Anderson. GLUnix: A Global Layer Unix for a Network of Workstations. *Software Practice and Experience*, 28(9):929–961, 1998.

[14] Grid'5000, An infrastructure distributed in 9 sites around France, for research in large-scale parallel and distributed systems. https://www.grid5000.fr.

[15] E. Hendriks. BProc: the Beowulf distributed process space. In *Proceedings of the IEEE/ACM International Conference on Supercomputing*, pages 129–136, 2002.

[16] InTrigger, a distributed platform for information technology research for Information Explosion Era. https://www.intrigger.jp/wiki/index.php/InTrigger.

[17] M. L. Massie, B. N. Chun, and D. E. Culler. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. *Parallel Computing*, 30(7):817–840, 2004.

[18] F. Petrini, S. Coll, E. Frachtenberg, and A. Hoisie. Performance Evaluation of the Quadrics Interconnection Network. *Journal of Cluster Computing*, 6(2):125–142, 2003.

[19] Fifth GRIDS@WORK. http://www.etsi.org/plugtests/GRID2008/GRID.htm.

[20] TakTuk, Large Scale Remote Executions Deployment. general - http://taktuk.gforge.inria.fr, developpers - https://gforge.inria.fr/projects/taktuk.

[21] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *19th International Symposium on Computer Architecture*, pages 256–266, 1992.

[22] J. W. W. Yu and D. K. Panda. Scalable Startup of Parallel Programs over InfiniBand. In *International Conference on High Performance Computing*, 2004.

[23] A. B. Yoo, M. A. Jette, and M. Grondona. *SLURM: Simple Linux Utility for Resource Management*, pages 44–60. Springer Berlin / Heidelberg, 2003.