A Scheduling algorithm for High Performance Peer-To-Peer Platform

Nabil Abdennadher[1], Régis Boesch[1]

[1] University of Applied Sicences, 4 Rue Prairie, 1202,Geneva, Switzerland.
{nabil.abdennadher, regis.boesch}@hesge.ch

Abstract. This paper describes a scheduling algorithm used to execute parallel and distributed applications on a Global Computing (GC) environment, called XtremWeb-CH (*XWCH*). *XWCH* is an improved version of a GC tool called XtremWeb (*XW*). XWCH is an enrichment of *XW* allowing it to match P2P concepts: distributed scheduling, distributed communication and development of symmetrical models. The scheduling algorithm takes into account the heterogeneity and volatility of nodes. This paper illustrates the performance of *XWCH* in a real CPU time consuming application.

Keywords: Peer-To-Peer, High Performance Computing, Scheduling Algorithm.

1. Introduction

High Performance Computing (HPC) landscape has radically changed since the end of the last decade. Based initially on the use of parallel and vectorial computers equipped with specific development environments, computing power consumers are adopting a new approach which takes advantage of the Internet development. The idea consists on deploying High Performance applications on anonymous connected computers by using their available resources. Indeed, the challenge today is to extract, at low cost, a reasonable computing power from a widely distributed platform (by executing interactive applications) rather than extracting the maximum power from a local supercomputer (by executing batch applications). In another words, the majority of the world's computing power is no longer in supercomputer centers and institutional machine rooms. Instead, it is now distributed in a hundred of thousands of personal computers all over the world. This concept is known as Global Computing (GC).

The majority of GC projects adopted a centralized structure based on a Master/Slave Architecture: SETI@home [1], Entropia [2], United Devices [3], Parabon [4], XtremWeb [5], etc. A natural extension of the GC consists on distributing the "decisional degree" of the master in order to avoid any form of centralization. Thus, architectures such as Clients/Servers and Master/Slaves would be withdrawn. This concept, known as Peer-To-Peer (P2P), was successfully used to share and exchange files between computers connected to Internet. The most known projects are Gnutella [6] and Freenet [7]. Indeed, file sharing is well adapted to this model. However, the use of P2P in the field of HPC raises several theoretical and practical problems. Dynamic scheduling algorithms for parallel/distributed

applications can not be easily distributed. P2P Computing also goes against the policies and safety techniques largely used nowadays on Internet: Firewalls, NAT addresses, etc. The objective of these techniques is to protect resources connected to Internet from any voluntary or involuntary abusive use. Internet is then partitioned in several protected zones which are unable to cooperate mutually. Problems related to the development of a true P2P environment for HPC needs remain open.

This document describes a GC environment, called XtremWeb-CH (*XWCH*), which converges towards a P2P system. *XWCH* is an improved version of a GC tool called XtremWeb (*XW*). *XWCH* tries to enrich *XW* in order to match P2P concept: distributed scheduling, distributed communication, development of symmetrical models, etc. In P2P systems, nodes are assumed to be customers and servers at the same time. Although it is utopian, this idea is retained as guide line in the *XWCH* project.

This document is organized as follows: section 2 presents the features that should be satisfied by a GC platform in order to be considered as a real P2P system. Section 3 introduces the *XW* tool in its original version. Section 4 details the new concepts *XWCH* introduces compared to *XW*. It also describes the features of the scheduling algorithm supported by *XWCH*. Section 5 presents the experiments carried out in order to evaluate *XWCH*. Lastly, the section 6 gives some perspectives of this research.

2. What is a real Peer-To-Peer system?

A true P2P environment should satisfy three criteria:

- *Platform heterogeneity*: The system should support heterogeneous architectures (hardware) and platforms (software and operating systems). Since these resources are anonymous, the system should take into account all administration policies implemented by local administrators.
- *Natural scalability*: A P2P system should support a huge number of resources. It should be scalable by itself and not by "doping". For that purpose, the performance of the system should be provided by its distributed structure: distributed algorithms, distributed warehouses, distributed scheduling algorithms, etc. This structure should allow open access and search procedures. The search engine should take into account the dynamic nature of the network. The system should be based on a demand-driven computation model: users' queries are only processed when needed and prior results are stored in warehouses, where they can be accessed later on.
- *Symmetric view*: a node belonging to a P2P platform should be server and client at the same time.

File sharing systems like *Gnutella* and *Freenet* satisfy all these criteria. High performance GC environments such as *XtremWeb*, *Seti@home*, *Entropia* do not satisfy any of these criteria. They are based on a non symmetric view (Master/Slaves). They are not scalable since the master is overloaded when the number of slaves

increases. The only HP oriented tool which seems to satisfy all these constraints is WOS (Web Operating System) [8]. Unfortunately, this tool remained in a purely conceptual state and no prototype was born.

3. XtremWeb

XW is a GC research project carried out at Université d'Orsay (France). Like other Large Scale Distributed Systems (LSDS), *XW* platform uses remote resources (pocket computers, PCs, workstations, servers) connected to Internet to execute a specific application (client). The aim of *XW* is to investigate how a LSDS can be turned into a High Performance Parallel Computer. *XW* belongs to the more general context of Grid research and follows the standardisation effort towards Grid Services [9]. *XW* satisfies the three main constraints imposed by any Large Scale Distributed Environment: volatility, heterogeneity and security.

Security is particularly difficult in the context of LSDS because it's impossible to trust hundreds of thousands resources. Three main security problems, linked to GC and P2P systems, are considered in the context of *XW* project:

- Data integrity/privacy: This problem could be resolved by applying the well known solutions of encryption, public/private keys, etc.
- Protection of participating resources: No aggressive application should be able to corrupt data or system of any participating resource. Sandboxing is the well known technique to resolve this problem. The idea consists on filtering the system calls which appear to be the main security holes of recent operating systems. [10] explains how does *XW* use the sandboxing to resolve the resource protection problem.
- Result certification procedure: This problem is linked to the lack of trust regarding the result provided by the remote resource. Indeed, there is no way to control precisely what happens on a participating resource. Faulty and malicious behaviour must be detected.

A typical *XW* platform is composed of one coordinator and several workers (remote resources). The coordinator is a three-tier layer allowing connection between clients and workers through a coordination service. This layer is designed so as it allows the mobility of clients and the volatility of workers.

3.1 The coordinator

The coordinator is a three-tier architecture which adds a middle tier between client and workers. There is no task direct submission/result transfer between clients and workers. The coordinator accepts task requests coming from several clients, distributes the tasks to the workers according to a scheduling policy, transfers application code to workers if necessary, supervises task execution on workers, detect worker crash/disconnection, re-launches crashed tasks on any other available worker, collects and store task results to client upon request.

The coordinator is composed of three services: the repository, the scheduler and the result server. The repository is an advertisement services. It publishes services (client applications) to make them available through standard communication ports (Java RMI, XML-RPC). These applications/services are first read from a database and inserted into the task set. The scheduler is the service factory. It instantiates applications and manages their life cycle. It starts them on workers (a task is an instantiation of service or application), stops them as expected and corrects faults (if any) by finding available workers to re-launch them. Finally the result server collects results as they are provided by workers.

3.2 Workers

The worker architecture includes four components: the task pool, the execution thread, the communication manager and the activity monitor. The activity monitor controls whether some computations could take place in the hosting machine regarding parameters such as CPU idle time and mouse/keyboard activity. The tasks pool (worker central point) is managed by a producer/consumer protocol between the communication manager and the execution thread. Each task should be in one of the three states: *ready* to be computed, *running* or *saving*. The first state concerns downloaded tasks, correctly inserted into the pool. The second state is for tasks being computed. The last state corresponds to tasks which need to upload result file to the coordinator. The communication manager ensures communication with the coordinator; it downloads task files (binaries and input data) and upload results, if any. When download completes, the task is inserted into the task pool. The execution thread extracts the first available task from the pool, recreates the task environment as provided by the client (binary code, input data, directories structure, etc.), starts computation and waits for the task to complete. When the task completes, the execution thread finally marks the task state as completed, allowing the communication manager to send results to the coordinator.

In its original version, *XW* applications are standalone modules. The system does not support any interaction between different tasks. However, developers can use asynchronous Remote Process Call called *XWRPC* in order to distribute (parallelize) their applications [11].

4. XtremWeb-CH

XtremWeb-CH (*XWCH*) is an upgraded version of *XW*. The aim of *XWCH* is to build an effective Peer-To-Peer LSDS which satisfies the three criteria detailed in section 2. *XWCH* adds four functionalities to *XW*:

1. Automatic execution of Parallel and Distributed Applications.
2. Automatic detection of the optimal granularity that can be implemented according to the number of available workers and scheduling of tasks.
3. Support of direct communication between workers.
4. *XWCH* provides a set of monitoring tools allowing users to visualize the execution of their applications.

4.1 Automatic execution of Parallel and Distributed Applications

In *XW*, jobs submitted to the system are standalone. In case of parallel/distributed applications, communicating modules are executed as separate jobs (tasks). It's the user responsibility to link manually output and input data of two communicating tasks. Contrary to this approach, *XWCH* supports the execution of a whole parallel/distributed application represented by is a set of communicating tasks. This application is modeled by a data flow graph where nodes are tasks and edges are communications inter-tasks (Fig. 1). Tasks can have the same or different codes. In Fig. 1, tasks having the same shape have the same code.

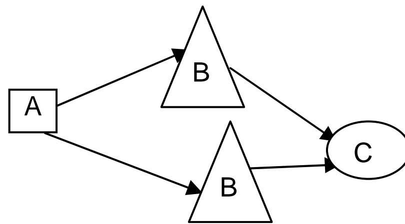

Fig. 1. Data flow graph representing a parallel/distributed application

The data flow graph is represented by an XML file whose syntax is detailed in Fig. 2.

An application is composed of several modules (*Module* element in Fig. 2). A module is represented by a source code and can have several binary versions (*Binary* element in Fig. 2). A task is an instantiation of one module. Thus, several tasks can correspond to the same module.

Precedence rules between tasks are described by *Task* elements. A task can have several inputs (*Input* element in Fig. 2) but only one output (*Output* element in Fig. 2). The element *cmdLine* indicates arguments/parameters used by the task. This field is optional.

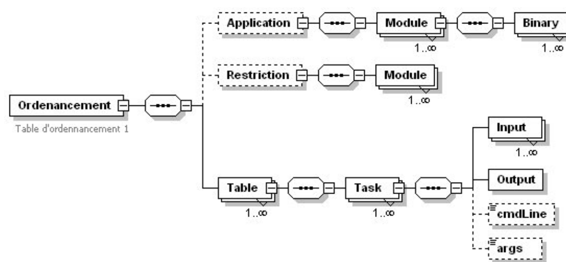

Fig. 2. XML syntax of a parallel/distributed application

A parallel/distributed application is thus, represented by:

- its XML file representing its data flow graph,
- the binary codes of its modules. Let's recall that one module can have several binary codes,

– its input data.

These files are compressed into one file.

XWCH can be perceived as a layer on *XW* that takes into account the communications between tasks belonging to the same parallel/distributed application. In this context, a task belonging to a given parallel/distributed application is considered by *XW* as a standalone application.

A client can submit his application to *XWCH* by uploading its corresponding compressed file. In addition to the three states that a task can have: *ready*, *running* and *saving*, *XWCH* adds a fourth state: *blocked*. Tasks of a given application are initially *blocked* and cannot be assigned to any worker, since their input data are not available. Only tasks whose input data are given by the user are in *ready* state and can be allocated to workers. When a task is assigned to a worker, it moves from *ready* to *running* state. Input data needed by *blocked* tasks are progressively provided by *running* tasks which finish their processing. *XWCH* detects the *blocked* tasks which can pass to ready state and can, thus, be assigned to a worker.

4.2 Granularity and scheduling

In parallel computing, the grain's size (granularity) depends on the application and the number of processors in the target parallel machine. This number is generally known and fixed before the execution. Thus, the granularity is fixed during the development of the application. In our context, the computer is the network, workers are free to join and/or leave the GC platform whenever they want. The exact number of available workers is known just before the execution and could be varied during the execution. As a consequence, the best granularity can not be fixed before execution time. This section describes how *XWCH* optimize the granularity of tasks and how these tasks are scheduled during execution.

Data flow graph representing an application comprises generally a set of stages $\{S_i\}$. A stage S_i is represented by a set of tasks having the same source code (module in the XML file) and can be executed in parallel on different workers. The precedence rules between two stages S_i and S_{i+1} depends on the application. Tasks belonging to the same stage have no precedence rules. They are fed with different data and are executed according to the Single Program Multiple Data (SPMD) model. Thus, every stage is responsible of processing a "quantity" of data noted Q_i. The number of tasks belonging to stage S_i depends also on application but could be fixed according to the number of workers.

Fig. 3 and 4 show two kinds of parallel/distributed applications experimented on *XWCH*.

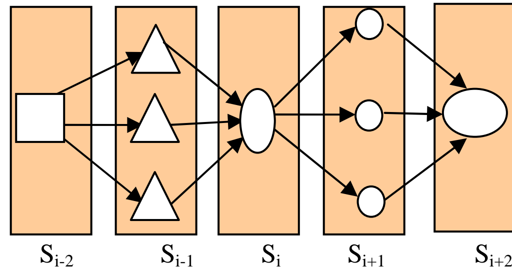

Fig. 3. Phylogenetic applications

In Fig. 3, odd stages contain one task while even stages contain a variable number of tasks. This means that odd stages concentrate results of even stages before sending them to the next stage.

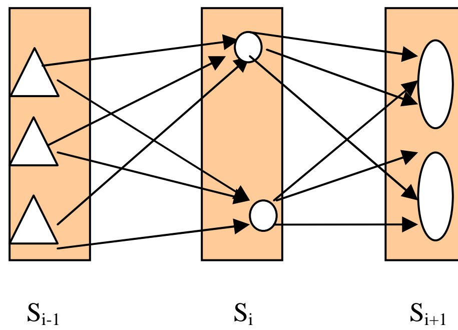

Fig. 4. Numerical application

In Fig. 4, every task of a stage S_i sends its result to all tasks of stage S_{i+1} (multicast operation).

To deploy an application on *XWCH*, three steps are required:

Discovery step: This step consists of searching for a set of available workers W to execute the application (or one stage of the application). The output of this step is a set of workers $W = \{(w_j, p_j)\}$ where p_j is the performance of w_j. p_j can be expressed in term of CPU performance, main memory size, network bandwidth, etc.

Configuration step: Assuming that $|W| = n$, this step dispatches the quantity of data to process by a stage S_i (Q) among the n tasks which compose the given stage. A task t_k, supposed to be executed by worker w_j (with performance p_j), is assigned a quantity of data q_k function of p_j. q_k is called the workload of t_k. The more the worker is powerful, the bigger is q_k. At this point, the system behaves as if the n workers are fully monitored by the coordinator. In another term, granularity of the parallelization and load balancing are fixed according to the number of available workers and the state of the targeted P2P platform.

The output of the configuration step for a given stage S of a given application is a set of couples $\{(q_k, p_j)\}$ where p_j is the performance of the worker that will process the task having the workload q_k.

The XML file, describing the application, is automatically generated at the end of this step.

Execution step: Configuration step assumes that available workers W are fixed and controlled by the coordinator. However, during execution, tasks allocation is not totally controlled by the coordinator. Indeed, tasks are allocated to workers when the coordinator receives work requests from workers. At this point, it is worth going into some details:

- A work request is a remote procedure called by the workers and executed by the coordinator.
- A work request, called by a worker, indicates its current performance p.
- One or several workers selected during discovery step can disappear during execution step.
- One or several new workers can connect and start to send work requests after discovery step.

During execution, the coordinator manages a set of tasks $T = \{t_k\}$ belonging to different applications. Every task t_k has its workload q_k.

Ideally, tasks belonging to a given stage of a given task are executed in parallel on workers selected during configuration step (or new workers with higher performance). Since workers are volatiles, a work request received by the coordinator is not necessarily sent by one of the workers selected during the configuration step. Moreover, arrivals of work requests are unpredictable. For that reasons, the scheduling policy of XWCH is the following: when receiving a work request from a worker w having performance p, the task t allocated to w is the one whose workload q is closer to p. Thus, the scheduler of XWCH allocates task t of T to w if:

$|q - p| = \min(|q_k - p_w|)$ for all t_k belonging to T.

The scheduling algorithm is executed inside the work request call. According to this algorithm, a given task is not executed unless an appropriate worker calls a work request. This means that a task could stay indefinitely in a *ready* state and never assigned to a worker, the application is blocked. In order to avoid this situation, a deadline is affected to each stage of the application: if a task spends in a ready state a time higher than its deadline, it is automatically allocated to the first free worker. A small value of the deadline, means that the user prefers allocate tasks to workers as soon as possible. In this case, tasks could be assigned to a non appropriate worker. A high value of the deadline means that the user prefers wait and allocate tasks to the best appropriate worker. In this case, the task could be blocked indefinitely.

4.3 Direct communication

Two versions of *XWCH* were developed. The first, called *XWCH-sMs*, manages inter-tasks communications in a centralized way. The second version, called *XWCH-p2p*, allows a direct communication between workers without passing by the coordinator

In the *XWCH-sMs* (slave-Master-slave) version, workers cannot directly communicate, they cannot "see" each other. Any communications between tasks take place through the coordinator. This architecture overloads the coordinator and could affect the application performances.

In order to cure the gaps of the *XWCH-sMs* version, it is necessary to have direct worker-to-worker communications. In other term, the worker executing module *A* (called *worker A* in Fig. 5) must be able to directly send its results to *workers B* and *C*.

The *XWCH* coordinator can, thus, allocate tasks *B* and *C* to two available workers. Every worker receives the binary code of the module it will execute and the necessary information relating to its input file (IP address, path and name of the input file). Data transfer between workers *A* and *B* (*resp.* C) can thus take place on the initiative of the receiver.

This version called *XWCH-p2p* has two main advantages:

1. it discharges the coordinator from data routing.
2. it avoids the duplication of communications.

In this context, the coordinator keeps only the responsibility of tasks scheduling. *XWCH-p2p* tends towards the Peer-To-Peer concept which one of its principles is to avoid any centralized control.

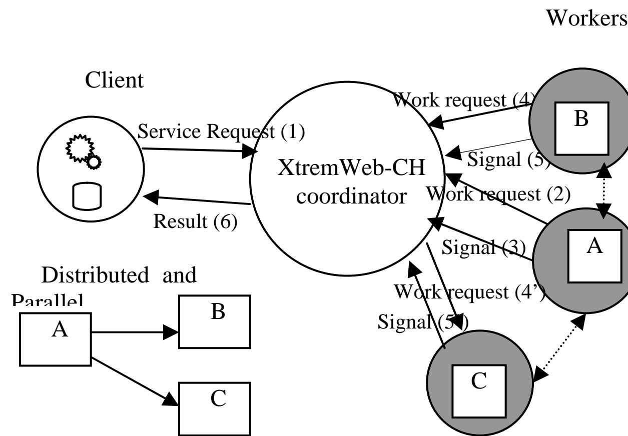

Fig. 5. Execution of an application on a *XWCH-p2p* platform

Direct communication can only take place when the workers can “see” each other. Otherwise (one of the two workers is protected by a firewall or by a NAT address), direct communication is impossible. In this case, it is necessary to pass by an intermediary (*XWCH* coordinator for example). This scenario is similar to *XWCH-sMs* version. However, to avoid overloading the coordinator, one possible solution consists on installing a relay machine, called "data collector" which acts as an intermediary. This machine is used by worker *A* (in our example) to store its results and by workers *B* and *C* to seek their data. “Data collector” machine is chosen by the user when launching the application. This machine must be reachable by all workers contributing to the execution of the concerned application.

4.4 Monitoring tools

XWCH proposes a package of tools allowing the user to debug and/or visualize the progression of the execution of their applications:

- Tasks allocation: The user can “spy” the execution of his application. He can follow the allocation of tasks (which worker is executing which task)
- Progression of tasks execution: When executing, every task can send progression report to its worker informing it about its state. Currently, this progression report is expressed in term of percentage of execution.
- Step by step execution: It’s a debugging mode. When activated, every task sends messages to the worker. These messages are inserted in the source code by the developer.

5. Experimental measures

The purpose of this section is to assess the performances of *XWCH* in a real case of a CPU time consuming application. *XWCH* was evaluated in the case of a phylogenetic application: *PHYLIP* (the *PHYL*ogeny *I*nference *P*ackage) package [12]. The parallelized version of *PHYLIP* is used by the Laboratory of virology at the Geneva Hospital in order to generate phylogenetic tree related to HIV virus.

Phylogenetic is the science which deals with the relationships that could exist between living organisms. It reconstructs the pattern of events that have led to “the distribution and diversity of life”. These relationships are extracted from comparing Desoxyribo Nucleic Acid (DNA) sequences of species. An evolutionary tree, termed life tree, is then built to show relationship among species. This tree shows the chronological succession of new species (and/or new characters) appearances.

In a medical context, the generation of a life tree for a family of microbes is particularly useful to trace the changes accumulated in their genomes. These changes are due, inter-alia, to the "reaction" of the virus to the treatments.

A multitude of applications aiming at building evolutionary trees are used by the scientific community [13] [14] [15] [16]. These applications are known to be CPU time consuming, their complexity is exponential (*NP-difficult* problem). Approximate and heuristic methods do not solve the problem since their complexity remains polynomial with an order greater than 5: $O(n^m)$ with $m > 5$. Parallelization of these methods could be useful in order to reduce the response time of these applications.

PHYLIP is a package of programs for inferring phylogenies (evolutionary trees). It is the most widely-distributed phylogeny package. *PHYLIP* has been used to build the largest number of published trees. It has been in distribution since 1980, and has over 15,000 registered users. *PHYLIP* was ported on *XWCH* platform.

An evolutionary tree is composed of several branches. Each branch is composed of sub-branches and/or leaf nodes (sequences). Two sequences belonging to the same branch are supposed to have the same ancestors. To construct the tree, the application defines a “distance” between all pairs of sequences. Evolutionary tree is then gradually built by sticking to the same branch, the pairs of sequences having the

smallest distance between them. Even if the concept is simple, *PHYLIP* is a CPU time consuming application. This complexity is due to two factors:

1. Methods used to group sequences into branches are complex. As an example, the *Fitch* program, one of the most used methods, takes two hours to execute on a Pentium 4 (3 GHz) with 100 sequences.
2. The application constructs not only one tree from the origin data set, but a set of trees generated from a large number of bootstrapped data sets (somewhere between 100 and 1000 is usually adequate). These data are randomly generated from origin data. The final (or consensus) tree is obtained by retaining groups that occur as often as possible. If a group occurs in more than a fraction *l* of all the input trees it will definitely appear in the consensus tree.

The application, as adapted to *XWCH*, is composed of 5 programs: Seqboot, Dnadist, Fitch-Margoliash, Neighbor-Joining and Consensus.

- *Seqboot* is a general bootstrapping and data set translation tool. It is intended to generate multiple data sets that are re-sampled versions of the input data set. It involves creating a new data set by sampling *N* characters randomly with replacement, so that the resulting data set has the same size as the original, but some characters have been left out and others are duplicated.
- *Dnadist* uses sequences to compute a distance matrix. It computes a table of similarity between the sequences. The distance, for each pair of species, estimates the total branch length between the two species. Each distance that is calculated is an estimate, from that particular pair of species, of the divergence time between those two species.
- *Fitch-Margoliash (FITCH)* and *Neighbor-Joining (NJ)*: These two programs generate the evolutionary tree for a given data set. *FITCH* method is a time consuming method and can not be applied to a large number of sequences.
- *Consensus*: This program constructs the consensus tree from the set of trees generated from bootstrapped data sets.

The structure of the obtained parallel/distributed application is shown in Fig. 3. The application, as developed, has two parameters (fed by the user):

- Set of DNA Sequences from species under investigation.
- Number of evolutionary tree to generate: This parameter represents the quantity of data: Q. It's used to produce multiple data sets from original DNA sequences by bootstrap re-sampling. The higher is Q, the finest is the result.

Two versions of PHYLIP were deployed on *XWCH*:

- The first version (Version 1 in Fig.6) is composed of *Q* tasks in the stage corresponding to the *FITCH* module. Each task processes one data (one tree)
- In the second version (Version 2 in Fig.6), the number of tasks and their workload are processed as explained in paragraph 4.2.

Execution times consumed by the two versions are shown in Fig. 6. *PHYLIP* was executed on an *XWCH* platform composed of more than 100 heterogeneous PC (Pentium 2, 3, 4) with Windows and Linux operating systems.

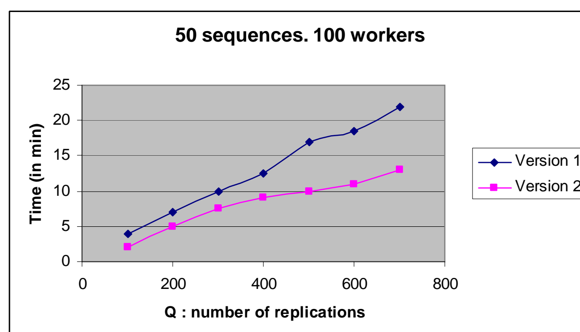

Fig. 6. Execution times of PHYLIP

For both versions, *XWCH* insures that executing codes are transferred from coordinator to workers only at the start of the execution: if the same task is re-executed on the same worker, its code is not downloaded again. The difference of execution times in Fig. 6 is due to the synchronization between the coordinator and workers: When a worker ends the execution of one task it stores the results locally and on the relay, generates a work request call to ask for a new job, and finally generates a data request call to receive input data it needs.

6. Conclusion

This paper presents a new GC environment (*XtremWeb-CH*), used for the execution of high performance applications on a highly heterogeneous distributed environment. *XWCH* can support direct communications between workers, without passing by the coordinator. A scheduling policy is proposed in order to minimize synchronization between coordinator and workers and optimize load balancing of workers. The porting of *PHYLIP* on *XWCH* has demonstrated the feasibility of our solution. Other experiments are in progress to evaluate *XWCH* in other High Performance applications cases.

The current version of *XWCH* allows the decentralization of communications between workers. The next step consists on designing a distributed scheduler. This scheduler shall avoid allocating communicating tasks to workers that can not reach each other. This approach offers a strong basis for the development of distributed and dynamic scheduler and could confirm and reinforce the tendency detailed in section 2.

7. References

1. http://setiathome.berkeley.edu/
2. http://www.entropia.com/
3. http://www.ud.com/home.htm
4. Parabon Computation, Inc: The Frontier Application. Programming Interface, Version 1.5.2. 2004 (www.parabon.com)
5. Gilles Fedak et al. XtremWeb : A Generic Global Computing System. CCGRID2001, workshop on Global Computing on Personal Devices. Brisbane, Australia. May 2001. http://xtremweb.net
6. KAN G., Peer-to-Peer: harnessing the power of disruptive technologies, Chapter Gnutella, O'Reilly, Mars 2001.
7. Ian Clarke. A Distributed Decentralised Information Storage and Retrieval System. Division of Informatics. Univ. of Edinburgh. 1999. http://freenet.sourceforge.net/
8. Babin, G; P. Kropf; and H. Unger. A two-level communication protocol for a Web Operating System: WOS. Vasteras, Sweden, Aug 1998. In IEEE Euromicro Workshop on Network Computing, 939–944.
9. I. Foster, C. Kesselman, J. Nick, and S. Tuecke. Grid Services for Distributed System Integration. IEEE Computer, pages 37-46, June 2002.
10. Franck Cappello et al. Computing on Large Scale Distributed Systems: XtremWeb Architecture, Programming Models, Security, Tests and Convergence with Grid. In Future Generation Computer Science (FGCS), 2004.
11. Samir Djilali. P2P-RPC: Programming Scientific Applications on Peer-to-Peer Systems with Remote Procedure Call. GP2PC2003 colocated with IEEE/ACM CCGRID2003. Tokyo Japan, May 2003.
12. http://www.phylip.com/
13. http://biowulf.nih.gov/apps/puzzle/tree-puzzle-doc.html
14. http://www.tree-puzzle.de/
15. http://www.dkfz.de/tbi/tree-puzzle/
16. Heiko A. Schmidt, Phylogenetic Trees from Large Datasets, 'Ph.D.' in Computer Science, Düsseldorf, Germany, 2003.