Using the Volunteer Computing platform *XtremWeb-CH*: lessons and perspective

Abstract. *XtremWeb-CH* (*XWCH*) is a volunteer computing middleware that makes it easy for scientists and industrials to deploy and execute their parallel and distributed applications on a public-resource computing infrastructure. *XWCH* supports various high performance applications, including those having large storage and communication requirements.

Two high performance applications were ported and deployed on an *XWCH* platform. The first one is the *Phylip* package of programs that is employed for inferring phylogenies (evolutionary trees). It is the most widely distributed phylogeny package and has been used to build the largest number of published trees. Some modules of *Phylip* are CPU time consuming; their sequential version cannot be applied to a large number of sequences. The second application ported on *XWCH* is a medical application used to generate temporal dynamic neuronal maps. The application, named *NeuroWeb*, is used to better understand the connectivity and activity of neurons. *NeuroWeb* is a data and CPU intensive application.

This paper describes the different components of an *XWCH* platform and the lessons learned from gridifying *Phylip* and *NeuroWeb*. It also details the new features and extensions, which are being added to *XWCH* in order to support new types of applications.

1. Introduction

Since the early 90s, computing power consumers are adopting a new approach, which takes advantage of the Internet development. The idea consists of deploying high performance applications on distributed platforms instead of supercomputer centers. This concept, known as Grid Computing (GC), provides the ability to achieve higher throughput computing by taking advantage of many networked computers. The GC platforms use the resources of many separate computers connected by a network (usually the Internet) to solve large-scale computation problems. These platforms, equipped with appropriate middleware, involve organizationally owned resources: supercomputers, clusters, and PCs owned by universities, research laboratories and private companies.

Simultaneously with GC, a second alternative emerged. It consists of executing high performance applications on anonymous connected computers by using their available resources. This concept is called Volunteer Computing (VC). Most providers are individuals who own PCs and Macintoshes, connected to the Internet by cable modems or DSL. Providers are not computer experts, and participate in a project only if they are interested, or receive incentives. In the context of GC, consumers do not have control over providers.

The majority of VC and GC projects adopt a centralized structure based on a master-slave architecture: BOINC [1], Entropia [2], United Devices [3], Parabon [4], XtremWeb [5], Egee [6], NorduGrid [7], Condor [8], etc. A natural extension to the GC consists in distributing the decisional part of the master in order to avoid any form of centralization. This concept is known as Peer-To-Peer (P2P) and was successfully used to share and exchange files between computers connected to Internet and broadcast micro-news among Internet users. Architectures such as client-server and master-slave are not P2P. The most widely known P2P projects are BitTorrent [9], eDonkey [10], Kazaa [11], Gnutella [12], Freenet [13] and FeedTree [14].

The *XtremWeb-CH* project [15] aims to build an effective Peer-To-Peer system for CPU time-consuming applications. Although it may seem utopian, this idea was retained as a guideline in the *XWCH* project. Initially, *XWCH* is an upgraded version of a volunteer computing environment called *XtremWeb* (*XW*). Major improvements have been brought to it in order to obtain a reliable and efficient system. Its software architecture was completely re-designed.

The remainder of this paper is organized as follows. Section 2 presents the different components of the *XWCH* package. Section 3 details the gridification and deployment of two applications: *Phylip*, a package of programs for inferring phylogenies (evolutionary trees), and *NeuroWeb*, a medical application used to generate temporal dynamic neuronal maps. Section 4 details the lessons learned from gridifying, porting and deploying these two applications. It also describes the extensions that are currently being added to *XWCH* to support new types of applications. Finally, Section 5 gives some perspectives of this research.

2. XtremWeb-CH

The *XWCH* middleware is composed of four modules: coordinator, worker, warehouse and broker. These components are illustrated on Figure 1.

The coordinator module is the main component of *XWCH*. It is considered as the master of the *XWCH* system; it has the responsibility of managing communications between the clients (application users) and the workers (resource providers). The worker module is installed on each provider node and manages the execution of the tasks and the transfer of data to and from the worker. The warehouse is a repository used by the workers to save their output data and upload the input data they might need. A broker module is akin to a compiler in that it transforms a user request (application submission) into a set of tasks compliant with the format recognized by *XWCH*.

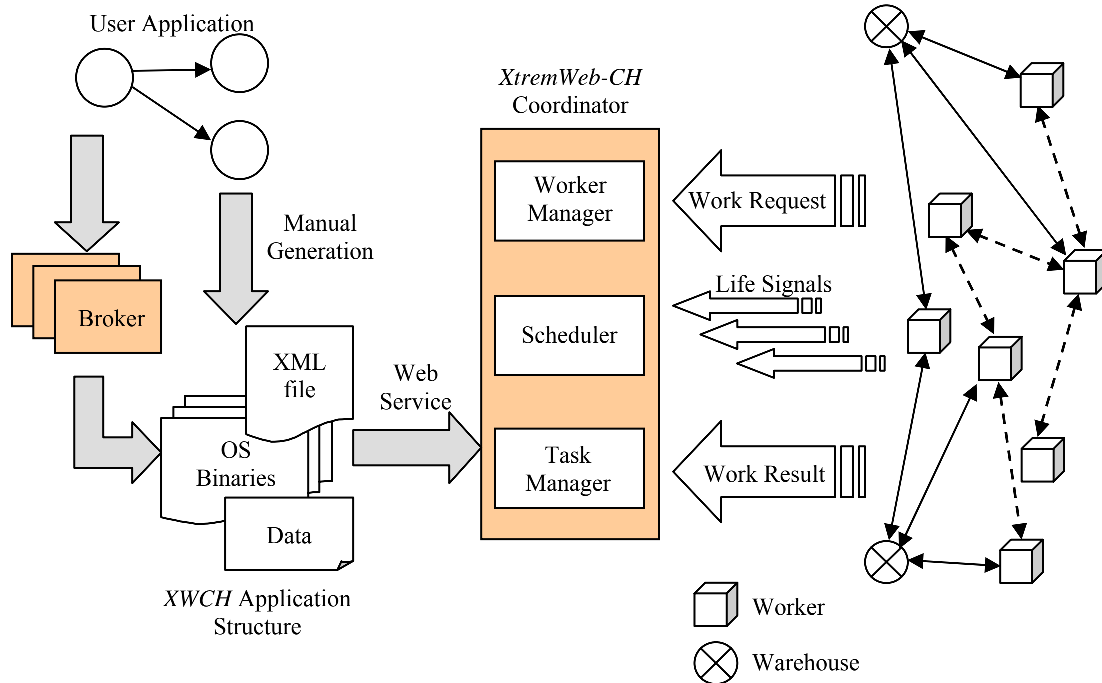

Figure 1. XWCH Architecture

2.1. Coordinator

It is a three-tier architecture that adds a middle tier between client and workers. The coordinator accepts execution requests coming from clients, assigns the tasks to the workers according to a scheduling policy and the availability of data, transfers binary codes to workers if necessary, supervises task execution on workers, detects worker crash and disconnection, and re-launches tasks on any other available worker. The coordinator is composed of three services: the worker-manager, the task-manager and the scheduler.

2.1.1. Worker-Manager

This manager maintains a list of connected workers. It receives four types of requests or signals from the workers: *Register Requests*, *Work Request*s, *Life Signal*s and *Work Result Signal*s. A worker subscribes to a nearby coordinator via a *Register Request*. When the worker-manager receives a *Work Request*, it searches for the most appropriate task to assign to the connected worker. During the execution of the task, workers send *Life Signals* to their coordinator to inform it about their status. When a worker finishes its execution, it sends a *Work Result Signal* to inform the coordinator of the location of the data it has produced.

2.1.2. Task-Manager

XWCH supports the execution of parallel and distributed applications containing communicating tasks. These applications are often modeled by a data flow graph where nodes are tasks and edges are communications between tasks. The data flow graph is represented by an XML file, which can automatically be generated by a specific broker.

A task belonging to a given application can be in one of five states: *blocked*, *ready*, *receiving*, *progressing*, and *saving*. A task is *blocked* if its input data are not yet available. The *receiving* state indicates that the

task is receiving its input data from the concerned warehouse or worker. When its input data become available, the task becomes *ready* and it is eligible for execution. The *progressing* state indicates that the task is currently under execution. The last state *saving*, corresponds to tasks which need to upload result file to the warehouse before sending a *Work Result Signal* to the coordinator.

2.1.3. Scheduler

A *Work Request* transmits the performance that can be delivered by the concerned worker. When receiving this request, the coordinator starts a scheduler module, which selects the most appropriate ready task to be allocated to that worker.

In parallel computing, the grain size (granularity) depends on the application and the number of processors in the target parallel machine. This number is generally known and fixed before execution. In this case, the granularity is determined during the development of the application. In our context, the computer is the network, and workers are free to join and leave the *XWCH* platform whenever they want. The exact number of available workers is known immediately prior to the execution and can fluctuate over time. As a consequence, the best granularity cannot be determined before execution time. This subsection describes how *XWCH* optimizes the task granularity and how these tasks are scheduled during execution.

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 effective performance of worker w_j; p_j can be expressed in terms of CPU performance, main memory size, network bandwidth, etc.

Configuration step: Assuming that $|W| = n$, this step dispatches the quantity of data to process among n tasks. A task t_k, supposed to be executed by worker w_j (with performance p_j), is assigned a quantity of data q_k that depends of p_j. q_k is called the workload of t_k. The more powerful the worker is, the bigger q_k is. At this point, the system behaves as if the n workers are fully monitored by the coordinator. In other words, parallelization granularity and load balancing are set according to the number and performance of available workers.

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

The XML file describing the application is automatically generated at the end of this step by the broker module.

Execution step: The configuration step assumes that the available worker set W is 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 sent by the workers and received by the coordinator.
- The arrivals of *Work Requests* are unpredictable.
- A *Work Request* sent by a worker indicates its current performance level p.
- One or several workers selected during the discovery step can disappear during the execution step.
- One or several new workers can register and start to send *Work Requests* after the 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 the configuration step (or with new workers with higher performance). Since workers are volatile, a *Work Request* received by the coordinator is not necessarily sent by one of the workers selected during the configuration step. For that reason, the scheduling policy of *XWCH* is the following: when receiving a *Work Request* from worker w having performance p, the task t allocated to w is the one having a workload q that is closest to p. Thus, the scheduler of *XWCH* allocates task t of T to w if

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

The scheduling algorithm is executed when the coordinator receives a *Work Request*. According to this algorithm, a given task is not executed unless an appropriate worker sends a *Work Request*. This means that a task could stay indefinitely in a ready state and never be assigned to a worker. In order to avoid this situation, a deadline is assigned to each task of the application: if a task stays in its ready state for a time beyond its deadline, it is automatically allocated to the first free worker. A small value for the deadline means that the user prefers to allocate tasks to workers as soon as possible. In this case, tasks could be assigned to an inappropriate worker. A high value for the deadline suggests that the user prefers to wait and allocate tasks to the most appropriate worker. But in this case, the task could be blocked indefinitely.

2.2. Workers

The worker module includes two components: the activity monitor and the execution thread. The activity monitor controls whether some computations are taking place in the host-machine and is concerned with parameters such as CPU idle time. The execution thread extracts the assigned task, starts the computation and waits for the task to complete.

2.3. Warehouses

XWCH supports direct communication between workers executing two communicating tasks. Direct communication can only take place when workers can see each other. When one of the workers is protected by a firewall or by a NAT address, communication under this condition is impossible. To take this case under consideration, it is necessary to go through an intermediary node: the *XWCH* coordinator for example. However, to avoid overloading the coordinator, one possible solution consists of installing warehouse nodes, which can act as intermediaries. Workers use these nodes to download input data that are needed to execute their allocated task and to upload output data produced by the tasks. A warehouse node acts as a repository or file server. It must be reachable by all workers contributing to the execution of a given application.

The protocol is the following:

- When a worker registers to a nearby coordinator (*Register Request*), it received the list of available warehouses;
- When a worker finishes the execution of a task it uploads its result in a one of the known warehouses (selected randomly). Thus, the result is stored in the worker and in the warehouse;
- The worker sends a *Work Result Signal* to the coordinator with the two locations (IP address and path) of the result produced by the given task;
- When a worker sends a *Work Request* to execute a new task, it receives as a reply, the binary code of the allocated task and the two locations of its input data.

2.4. Brokers

XtremWeb-CH optimizes the granularity of the application according to the state of the platform. The broker splits the user application into a set of tasks according to the state of the platform. In other words, the broker module *compiles* the user request (application submission) and generates the optimal number of tasks and the best workload (quantity of data to be processed) of each task according to the number of the available workers and their performance. The broker module can be installed on the client node (computer from which the user launches its application). The broker module depends on the application itself. An API is provided within the *XWCH* package that allows the user develop his own broker. This API is not detailed in this paper. The *XWCH* broker can be compared to the Globus broker who is responsible of transforming a high level RSL (Request Specification Language) request into a low level RSL request [16]. During execution, the broker does not interfere with the *XWCH* platform.

3. Applications

Gridification is the process of parallelizing and porting a high performance application on a Grid platform. The gridification should take into account several constraints linked to the targeted Grid and/or VC platform: volatility and heterogeneity of the nodes, limited bandwidth of the network, etc.

This section details the gridification and deployment of two applications: *Phylip* and *NeuroWeb*. The deployment of these two executions were carried out on a platform with one coordinator (Linux OS), 300 heterogeneous Windows and Linux workers ranging from Pentium I to Pentium IV, and 2 warehouse nodes. The workers are geographically located at three different places: the Engineering Schools of Geneva and Yverdon-les-Bains (Switzerland) and Franche Comté University (France). During the executions, students also use the 300 workers; they are often switched off or disconnected.

3.1. Phylip

3.1.1. Background

It is commonly accepted that contemporary genes, genomes, and organisms evolved from ancestors under the influence of natural selection. Consequently, the knowledge of the evolutionary tree behind their origin is crucial for understanding these entities. Knowledge about the relationships within gene families plays an

important role in understanding, for example, the origins of biochemical pathways, regulatory mechanisms in cells as well as the development of complex systems. For example, knowing relationships between viruses is central for understanding their ways of infection and pathogenicity.

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 to the reaction of viral strains to medical treatments.

Computer applications dealing with the reconstruction of evolutionary relationships of organisms, genes, or gene families have become basic tools in many fields of research [17-20]. These applications reconstruct 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 the relationship among species. The optimal tree is the one that is supposed to be the most realistic one.

Phylip (the PHYLogeny Inference Package) is a package of programs for inferring phylogenies (evolutionary trees). Developed in the 1980s, *Phylip* is one of the most widely distributed phylogeny packages. It has been used to build the largest number of published trees. *Phylip* has over 15 000 registered users. The package is freely available over the Internet and written to work on as many different kinds of computer systems as possible. Binary and source code (in C) are distributed. In particular, already-compiled executables are available for Windows, MacOS and Linux systems. The parallel version of *Phylip* is fully operational and experimental results are detailed in [21].

3.1.2. Gridification

This section describes the gridification on the *XWCH* middleware of the five modules composing *Phylip*: *SeqBoot*, *DnaDist*, *Fitch*, *NJ* and *Consensus*. Communications between tasks are based on file transfers. Input data are nucleotide sequence data (DNA and RNA) coded with an alphabet of the four nucleotides *Adenine*, *Guanine*, *Cytosine*, and *Thymine*. Nucleotide are denoted by their first letters: *A*, *G*, *C* and *T*. Every nucleotide sequence belonging to the input data is a leaf node of the evolutionary tree to be constructed.

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 generates a table of similarity between the sequences. For each pair of sequences, the distance estimates the total branch length between both sequences and represents the divergence time between these two sequences.

Fitch-Margoliash (*Fitch*) and *Neighbor-Joining* (*NJ*) are two modules that generate the evolutionary tree for a given data set. The *Fitch* method is a time consuming method and its sequential version cannot be applied to a large number of sequences.

Finally, the *Consensus* module constructs the consensus tree from the collection of intermediate trees generated from bootstrapped data sets.

The 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. An evolutionary tree is then gradually built by sticking the pairs of sequences having the smallest distance between them into the same branch. Even if the concept is simple, the algorithm is CPU time consuming. This complexity is due to two factors:

1. The methods used to group sequences into branches are complex. As an example, the *Fitch* module, one of the most used methods, takes two hours to execute on a Pentium 4 (3 GHz) with 120 sequences.
2. The application constructs not only one tree from the original 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 original data set. The final (or consensus) tree is obtained by retaining groups that occur as often as possible. If a group occurs in more than a given fraction of all the input trees, it will definitely appear in the consensus tree.

The application, as developed, has two parameters (user input):

1. The set of nucleotide sequences from species (or viruses) under investigation. In the reminder of this paper, the number of sequences is noted by *s*.
2. The number of replications (*r*), which used to produce multiple data sets from original DNA sequences by bootstrap re-sampling. Better results are obtained with higher *r* values.

The structure of the obtained parallel or distributed application is shown in Figure 2.

The *SeqBoot* task generates multiple data sets. Each of these data is used by a *DnaDist* task to generate one distance matrix. This matrix is then used by a *Fitch* (or *NJ*) task to generate an intermediate evolutionary tree. Finally, the *Consensus* task constructs the evolutionary tree from the intermediate trees. The *Fitch* module is very time consuming ($O(n^5)$). This is not the case of *SeqBoot*, *DnaDist*, *NJ* and *consensus* modules.

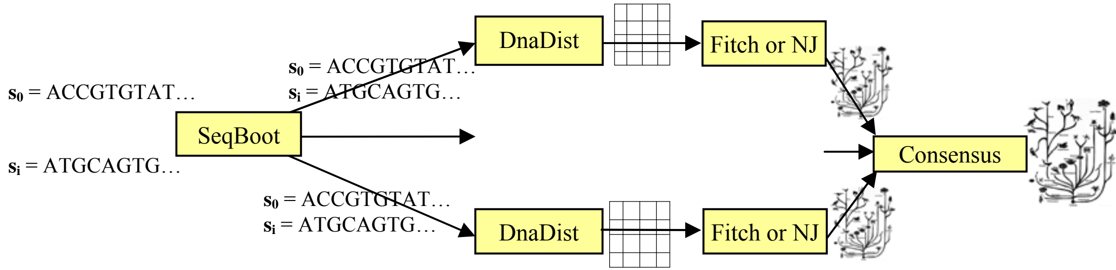

Figure 2. Data flow graph of *SeqBoot*, *DnaDist*, *Fitch*/*NJ* and *Consensus* modules

In order to apply the Fitch module to a large number of sequences, a parallel version of this package was designed and ported on *XWCH*. The data flow graph of the parallel implementation of the Fitch module is given in Figure 3. Each *Fitch* node in Figure 2 is thus replaced by the graph of Figure 3.

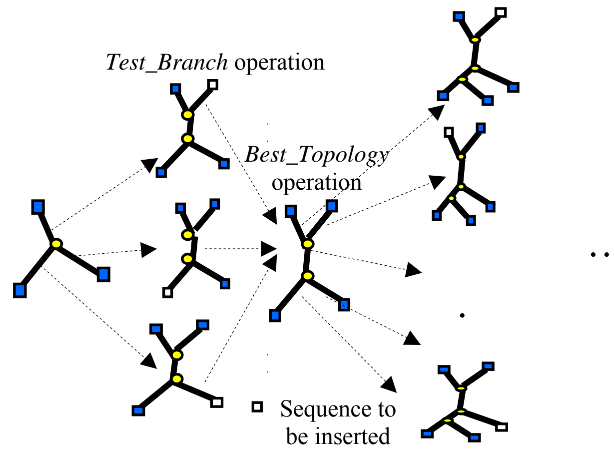

Figure 3. Data flow graph of a parallel *Fitch* task

The evolutionary tree is a non-rooted tree represented by two sets of nodes. The external (or leaf) nodes (square nodes in Figure 3) represent the sequences under investigation. An external node is always linked to one internal node. When the evolutionary tree is completely constructed, the number of external nodes is equal to *s*. The second set of nodes consists of internal nodes (round nodes in Figure 3). These nodes are virtual and do not represent sequences. Each internal node is linked to exactly 3 other nodes (internal or external). When the evolutionary tree is completely constructed, the number of internal nodes is equal to $s - 2$.

The evolutionary tree is generated progressively. The *Fitch* algorithm starts by creating a tree with one internal and three external nodes. At each step, the method inserts one sequence (external node) in every possible branch of the already constructed tree, and evaluates an objective function (*Test_Branch* operation in Figure 3). The selected branch is the one that minimizes a predefined criterion (*Best_Topology* operation in Figure 3). In addition to the external node inserted at each step, an internal node is also created and inserted in the same step. This process is repeated until all the sequences are inserted. The last step contains $2s - 5$ *Test_Branch* operations.

Thus, the number of *Test_Branch* operations for one parallel *Fitch* is $O(s^2)$, where *s* is the number of sequences. Since the number of *Fitch* tasks is equal to the number of replications, *r*, the maximum number of *Test_Branch* operations is $O(r\, s^2)$. The maximum number of parallel *Test_Branch* operations that could be executed at the same time is equal to $r\,(2s - 5)$. The execution time of a *Test_Branch* operation increases with the size of the evolutionary tree.

The parallelized version of *Phylip* is used to generate evolutionary tree related to HIV sequences. In this context, one needs to keep in mind that the number of sequences *s* can vary from 100 to 300 while the number of replications *r* varies from 100 to 1000. A specific broker (web service) was developed to allow a dy-

namic configuration of *Phylip* (number of *Fitch* tasks and number of trees generated by each *Fitch* task) as well as a visualization of the current state (number and performance of the workers) of the platform.

3.2. NeuroWeb

3.2.1. Background

Machine learning is the science of extracting information from data based on a model in order to understand, explain, test, detect or predict a phenomenon of interest such as brain functions. With the resolution increase of current Positron Emission Tomography (PET) and Brain Electromagnetic Tomography (BET), more and more data are being collected that need processing. For instance, PETs now attain a high spatial resolution on the order of a millimeter, and BETs can now achieve a high temporal resolution on the order of a microsecond. These resolutions are still growing. Consequently, yesterday's tools are no longer optimal or appropriate today. The goals of signal and image processing are also becoming more challenging, asking for instance for the detection of locally spatial or temporal brain functions. In brain imaging, today's challenges are the non-invasive reconstruction of a temporal sequence of three-dimensional objects inside the head, like reconstructing the generators of the brain's electromagnetic activity from BET measurements around the head at the scalp.

Figure 4. Magnetoencephalography (MEG)

Figure 5. Inverse problem

The reconstruction of dynamic neural maps requires the solution of a spatiotemporal inverse problem (Figure 5), which is an active field of research of great clinical importance. This technique can, for example, localize an epileptogenic region for a remote chirurgical operation. The green cylinders portrayed on Figure 5 represent the BET measurements of the internal cerebral activity around the scalp that are obtained for example from a MEG (Figure 4). The small yellow cylinders represent neuronal activities, called generators, and create electromagnetic fields that are detected by a BET. These fields are extremely noisy and distorted by the cell tissue (represented by the yellow triangles) lying between the generators and the BET. In essence, we have a lot of generators but very few BET spatial measurements. The difficulty, and hence the computational requirement effort, is to construct a robust and efficient estimator for a neuronal map.

An epileptic seizure is an example of a local feature, because high discharges of electric currents are generated over short periods of time and in small areas of the brain. In Switzerland, around 70 000 patients suffer from epilepsy and 20% are pharmaco-resistant. For these 20%, surgical removal of the epileptogenic region is recommended, which requires accurate imaging tools to locate the brain region that is to be removed.

In brain imaging for instance, physiological considerations tells us that the brain is made of organized tissues, so that the reconstructed object must be spatially coherent or smooth. Imposing appropriate smoothness will lead to better extraction of information from the data, with possible early detection of cancerous tissues, better prediction of epileptic seizures, or better understanding of neural connectivity. The current approaches that are often linear do not allow clear detection of local features. Instead, we propose to use a prior distribution based on ℓ_1 Markov random field. Hence the estimated images of brain activity will be a solution to a high-dimensional, non-differentiable optimization problem. The size of the solution space is extraordinarily large, about 10^7 and requires more than one day of intensive calculations. Finding the solution to this mathematical problem can be done efficiently using parallel computing.

The neuronal map extracted from brain measurements and calculated with the NeuroWeb grid software will be integrated into the NeoBrain software to visualize the spatial and temporal brain activity in 3D stereo world [22]. NeoBrain allows a smooth and interactive visualization, and enables a fine, precise and

convenient analysis of this massive reconstruction bank of spatial and temporal images, previously grid calculated. We call them neuronal maps. Practitioners are then able to virtually cut the cortex and identify sources of brain activity for epileptic patients, work that was previously restricted by the available technology.

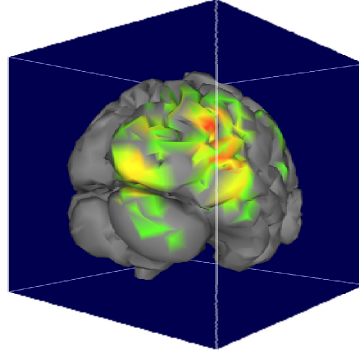

Figure 6. Typical neuronal map after extraction from *NeoBrain*

3.2.2. Gridification

The data flow graph of the *NeuroWeb* application is portrayed on Figure 7. The application produces the dynamic neural map (DNM) in which each row represents the electromagnetic activity of one neuron. With 60000 neurons and 1500 measures (one measure every 1 millisecond), the DNM has a size that is equal to 230 MB. Moreover, other input data are required for the computation of the resulting DNM. These are the functional and anatomic data that are provided from the EMG and IRM scanners, and their sizes are estimated at 16 MB. The application starts by choosing a random map called DNM_0. At each step i, a new matrix DNM_i is constructed based upon DNM_{i-1}. The iterations stop when, the DNM_i matrices can no longer be improved, i.e., when convergence is attained.

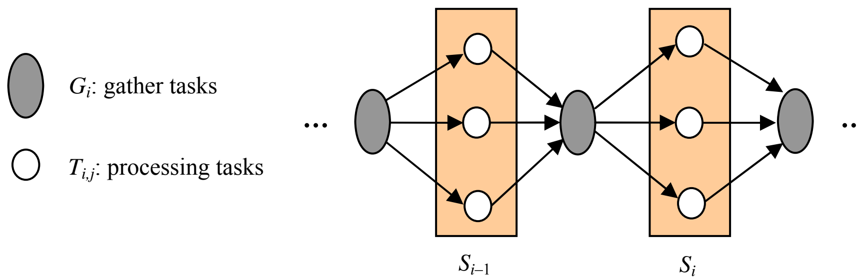

Figure 7. Neuroweb data flow graph

A given task $T_{i,j}$ (belonging to a stage S_i) processes a set of columns that will yield DNM_i. Tasks $T_{0,i}$, $T_{1,i}$, $T_{2,i}$ are supposed to process the same set of columns of the DNM matrices at each iteration. At each iteration, each task $T_{i,j}$ computes the same set of columns. To process column $C_{i,k}$ of matrix DNM_i, columns $C_{i,k-1}$ and $C_{i,k+1}$ are required. This means that tasks $T_{i,j}$ and $T_{i,j-1}$ need to exchange a common column, and so do tasks $T_{i,j}$ and $T_{i,j+1}$. For the sake of clarity, these communications are not shown on Figure 7. Tasks G_i merge the sub-matrices coming from $T_{i,j}$ to build the matrix DNM_i and decide whether the next stage $i + 1$ should be started or not.

It could thus be deduced from what is presented above that:

- The number of stages cannot be fixed before the execution. Hence, the XML file (introduced in Section 2.1.2) representing the data flow graph cannot be generated before deployment.
- Some tasks have to be executed on the same worker in order to avoid transferring huge quantities of data from one node to another.
- To minimize the bottleneck between main memory and the hard disk, some data have to remain in main memory even after the end of the process task. This would allow the next task to work on these data without fetching them from the hard disk or from a remote node (worker or warehouse).

4. Lessons and Perspectives

The extensions we outline are application driven. They are deduced from experiments carried out and lessons learned during the gridification and the deployment of applications *Phylip* and *NeuroWeb*.

4.1. Task Management

Currently, the number of tasks and the precedence rules among tasks are manually determined by the developer or generated automatically by dedicated brokers according to the state of the platform: number of workers, effective performance of workers, bandwidth of the network, etc. In the current version of *XWCH*, the data flow graph associated to a particular application cannot be modified during the execution. This constraint prevents developers and users from deploying applications where the number of tasks cannot be fixed before the execution, *NeuroWeb* application for example.

In order to correct this drawback, we aim at creating tasks on the fly during execution. Running tasks can decide to create new tasks according to their needs. To achieve this endeavor, an API should be developed that would enable the creation, monitoring, and the halting of new tasks. This API is similar to the one proposed by BOINC [1] which allows programmers to create tasks during the execution.

4.2. Task Allocation

For some classes of applications, particular tasks have to be executed on dedicated workers. This constraint can be imposed by the semantic of the application or be required by the user.

Semantic constraints: Some tasks require particular resources, which are not available to all workers. For this case, the system should specify, before the execution, the worker to which the task must be assigned. For some other applications, the system should be able to assign a set of tasks *T* to the same worker: the worker is not determined before the execution; it is identified by the first task belonging to the set *T*. This feature is tightly related to data persistence (Section 4.3).

Security constraints: Due to security concerns, the user may require that particular tasks execute on dedicated workers (trusted machines). The users can effectively require executing gather-tasks (which collect final output data) on known workers. These trusted workers are known before the execution.

4.3. Data Persistence

Due to performance concerns, in many cases, data should remain in the main memory of a worker even when the task ends its execution. The next task will then not fetch data from a remote machine (worker or warehouse) or local hard disk, but from the main memory of the worker.

The proposed solution consists of launching a first task to create a *daemon* process. This process is assumed to provide a set of services that can be invoked by future tasks and keeps data in main memory of the worker. The system should guarantee that the last task kills the daemon process and keeps the worker machine in a clean status.

The data persistence feature is particularly useful in the context of applications processing a huge quantity of data (satellite and medical images, etc.). The processing is here carried out in several stages represented by several tasks.

4.4. Data Communication

In *XWCH*, data communication is assured by exchanging files betweens two nodes (worker, warehouse or coordinator). A given task cannot be executed before the end of its precedent tasks. This model, named *task precedence model*, has two main drawbacks:

- When a given task broadcasts a file to several tasks, the operation overloads the network and the node (worker or warehouse) where the file to broadcast is located.
- When several tasks simultaneously fetch the same file from one node (worker or warehouse), the resulting gather operation can easily overload of the network and the crashing of the system.

In order to address these weaknesses, two alternatives can be considered. The first consists of adopting a storage model similar to the one used by file sharing peer-to-peer (P2P) tools – the file is stored on several nodes. When executing a broadcast operation, a given task can load its data from different warehouse nodes. This solution only solves the problem related to the broadcast operation. However, it does not provide a solution to the second problem. Its implementation is quite easy and does not require any modification in the *XWCH* kernel.

The second alternative seems more complicated and needs deeper consideration. In the reminder of this section, the task that generates a file is termed producer. The task requiring this file for its processing is termed consumer. The rationale behind the second scheme is to transfer the file from the producer to the consumer as soon as it becomes available. This means that the consumer task must be allocated to a worker when at least one of its input files is available. It is worth recalling that this task cannot be executed since all of its input data are not completely available. However, its inputs files can be downloaded as they are produced. This data precedence model guarantees the load balancing of the network and the nodes. Nonetheless, the system must decide when to it allocate the consumer task to a worker.

It is worth noting that allocating a consumer to a worker when all input files are available is equivalent to the task precedence model. On the other hand, allocating a consumer to a worker as soon as its first input file becomes available can increase the risk of volatility since the worker can disappear during the download of the input files or the end of the execution.

5. Conclusion

This paper presented the volunteer computing environment *XtremWeb-CH* (*XWCH*) used for the execution of high performance applications on a highly heterogeneous distributed environment. *XWCH* can support direct communications between workers without passing by a coordinator. A scheduling policy is proposed in order to minimize synchronizations between the coordinator and its workers and optimize load balancing of workers. The porting of *PHYLIP* and *NeuroWeb* on *XWCH* has demonstrated the feasibility of our solution. Several lessons have been learned from these deployments and are currently being investigated in order to implement useful extensions to *XWCH*.

The current version of *XWCH* allows the decentralization of communications between workers. The next step consists in designing a distributed scheduler. This approach offers a strong basis for the development of a distributed and dynamic scheduler and could confirm and reinforce the tendency detailed in the introduction.

6. References

[1] D.P. Anderson, *BOINC: A System for Public-Resource Computing and Storage*, 5th IEEE/ACM International Workshop on Grid Computing ((GRID'04)), Pittsburgh, USA, pp. 4-10, November 2004

[2] Entropia, http://www.entropia.com/

[3] United Devices, http://www.ud.com/home.htm

[4] Parabon Computation Inc, The Frontier Application, Programming Interface, Version 1.5.2. 2004

[5] G. Fedak, C. Germain, V. Neri, F. Cappello, *XtremWeb: A Generic Global Computing System*, Proc of the First IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID2001), Brisbane, Australia, pp 582-587, May 2001

[6] Egee: Enabling Grids for E-sciencE, http://www.eu-egee.org/

[7] Nordugrid, http://www.nordugrid.org/

[8] Condor: High Throughput Computing, http://www.cs.wisc.edu/condor/

[9] B. Cohen, *Incentives Build Robustness in BitTorrent*, Workshop Econ. Peer-to-Peer Syst, Berkeley, CA, June 2003

[10] eDonkey2000, http://www.edonkey2000.com/

[11] Sharman Networks: Kazaa, http://www.kazaa.com/us/index.htm

[12] G. Kan, *Gnutella*, Book chapter of Peer-to-Peer: Harnessing the Power of Disruptive Technologies, O'Reilly, March 2001

[13] I. Clarke: *A Distributed Decentralised Information Storage and Retrieval System*, Master's thesis, Division of Informatics. Univ. of Edinburgh, 1999

[14] FeedTree, http://feedtree.net/

[15] XtremWeb-CH: Towards a True P2P Computing Plateform, http://www.xtremwebch.net

[16] Globus Alliance, http://www.globus.org

[17] Tree-Puzzle on Biowulf: http://biowulf.nih.gov/apps/puzzle/index.html

[18] Tree-Puzzle 5.2: Maximum Likelihood Analysis for Nucleotide, Amino Acid, and Two-State Data, http://www.tree-puzzle.de/

[19] Tree-Puzzle: http://www.dkfz.de/tbi/tree-puzzle

[20] H.A. Schmidt, Phylogenetic Trees from Large Datasets, Ph.D. Thesis, Computer Science, Heinrich–Heine–Universität, Düsseldorf, Germany, 2003

[21] N. Abdennadher, R. Boesch, Deploying PHYLIP Phylogenetic Package on a Large Scale Distributed System, Proc. of the 7th IEEE International Symposium on Cluster Computing and the Grid (CCGRID'07), Rio De Janeiro, Brazil, May 2007

[22] NeoBrain Project, http://www.he-arc.ch/hearc/fr/isic/Projets/Projets/ProjetsIT/NeoBrain.html