High Performance Peer-to-Peer Distributed Computing with Application to Obstacle Problem[#]

The Tung Nguyen [1,2,6], Didier El Baz [1,2,6], Pierre Spitéri [3,7], Guillaume Jourjon [4,8], Ming Chau [5,9]
[1] CNRS ; LAAS ; 7 avenue du colonel Roche, F-31077 Toulouse, France.
[2] Université de Toulouse ; UPS, INSA, INP, ISAE ; LAAS ; F-31077 Toulouse France.
[3] ENSEEIHT-IRIT, 2 rue Charles Camichel, 31071 Toulouse, France
[4] NICTA, Australian Technology Park, Eveleigh, NSW, Australia
[5] Advanced Solutions Accelerator, 199 rue de l'Oppidum, 34170 Castelnau le Lez, France
[6] {elbaz, ttnguyen}@laas.fr, [7] Pierre.Spiteri@enseeiht.fr,
[8] guillaume.jourjon@nicta.com.au, [9] mchau@advancedsolutionsaccelerator.com

Abstract—**This paper deals with high performance Peer-to-Peer computing applications. We concentrate on the solution of large scale numerical simulation problems via distributed iterative methods. We present the current version of an environment that allows direct communication between peers. This environment is based on a self-adaptive communication protocol. The protocol configures itself automatically and dynamically in function of application requirements like scheme of computation and elements of context like topology by choosing the most appropriate communication mode between peers. A first series of computational experiments is presented and analyzed for the obstacle problem.**

Keywords— peer to peer computing, high performance computing, distributed computing, task parallel model, self-adaptive communication protocol, numerical simulation, obstacle problem.

I. INTRODUCTION

Peer-to-Peer (P2P) applications have known great developments these years. These applications were originally designed for file sharing, e.g. Gnutella [1] or FreeNet [2] and are now considered to a larger scope from video streaming to system update and distributed data base. Furthermore, recent advances in microprocessors architecture and networks permit one to consider new applications like High Performance Computing (HPC). Therefore, we can identify a real stake at developing new protocols and environments for HPC since this can lead to economic and attractive solutions. Nevertheless, task parallel model and distributed iterative methods for large scale numerical simulation or optimization on P2P networks gives raise to numerous challenges like communication management, scalability, heterogeneity and peer volatility on the overlay network (see [9]). Some issues can be addressed via distributed asynchronous iterative algorithms (see [8], [13] and [14]); but in order to improve efficiency, the underlying transport protocols must be suited to the profile of the application. We note that transport protocols are not well suited to this new type of application. Indeed, existing transport protocols like TCP and UDP were originally designed to provide ordered and reliable transmission to the application and are no longer adapted to both real-time and distributed computing applications. In particular, P2P applications require a message based transport protocol whereas TCP only offers a stream-based communication. Recently, new transport protocols have been standardized such as SCTP and DCCP. Nevertheless, these protocols still do not offer a complete modularity needed to reach an optimum resolution pace in the context of HPC and P2P.

In [3] and [10], we have proposed the Peer To Peer Self Adaptive communication Protocol P2PSAP which is suited to high performance distributed computing. The P2PSAP protocol is based on the Cactus framework [4] and uses micro-protocols. P2PSAP chooses dynamically the most appropriate communication mode between any peers according to decisions made at application level like schemes of computation, e.g. synchronous or asynchronous schemes and elements of context like topology. This approach is different from MPICH Madeleine [20] in allowing the modification of internal transport protocol mechanism in addition to switch between networks.

Recently, middleware like BOINC [21] or OurGrid [22] have been developed in order to exploit the CPU cycles of computers connected to the network. Those systems are generally dedicated to applications where tasks are independent and direct communication between machines is not needed.

In this paper, we present the current version of P2PDC an environment for P2P HPC based on P2PSAP which allows direct communication between peers and facilitates programming. We display and analyze computational results obtained for the obstacle problem on the NICTA testbed. We show that the adaptation of P2PSAP allows an efficient distributed solution of the problem.

This article is structured as follows: Section II deals with P2PSAP. In Section III, we present the current version of P2PDC. The programming model is also detailed in Section III. The Section IV deals with the solution of the obstacle problem. Computational results are displayed and analyzed in Section V. Conclusion and future work is presented in Section VI.

II. SELF-ADAPTIVE COMMUNICATION PROTOCOL

In this Section, we present P2PSAP; this protocol is an extension of CTP [5], a Configurable Transport Protocol designed and implemented using the Cactus framework [4]. The Cactus framework makes use of micro-protocols to allow users to construct highly-configurable protocols for distributed system. A micro-protocol implements merely a functionality of a given protocol (e.g. congestion control and reliability). A protocol results from the composition of a given set of micro-

[#]*The present work has been funded by ANR-07-CIS7-011 grant*

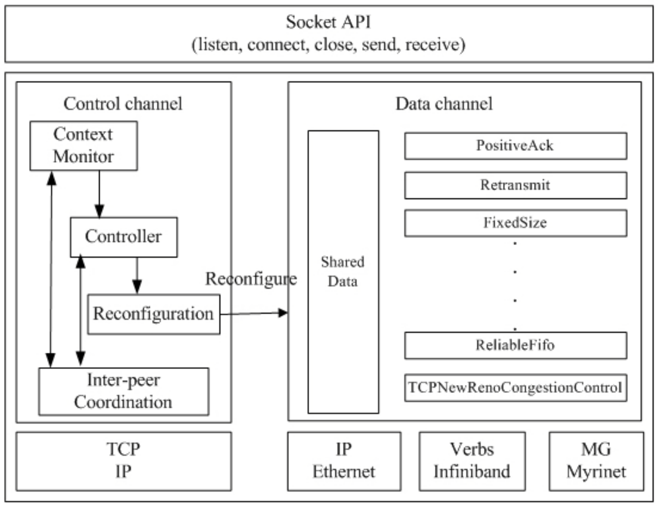

Figure 1. P2PSAP Protocol architecture

protocols. This approach permits one to reuse the code, facilitate the design of new protocols and gives the possibility to configure the protocol dynamically. Cactus is an event-based framework. Each micro-protocol is structured as a collection of event handlers, which are procedure-like segments of code and are bound to events. When an event occurs, all handlers bound to that event are executed. Cactus has two grains level. Individual protocols, the so-called composite protocols, are constructed from micro-protocols. Composite protocols are then layered on top of each other to create a protocol stack. Protocols developed using Cactus framework can reconfigure by substituting micro-protocols or composite protocols. For further details about Cactus, reference is made to [4].

In order to improve protocol performance and facilitate reconfiguration, we have introduced some modifications to the Cactus framework. Firstly, Cactus doesn't allow concurrent handler execution; this means that a handler must wait for current executed handler completion before being executed. But nowadays, almost all PCs have more than one core and concurrent handler execution is necessary in order to improve performance. So, we have modified Cactus to allow concurrent handler execution. Each thread has its own resources and its handler execution is independent of others. Secondly, we have eliminated unnecessary message copies between layers. In the Cactus framework, when a message is passed to upper or lower layers, Cactus runtime creates a new message that is sent to upper or lower layers. Hence, a significant number of CPU cycles and memories are consumed in multiple-layers systems. In our protocol, message copies occur between Socket API layer and Data channel, and within the Data channel. In order to eliminate message copies, we have modified the pack and unpack functions so that only a pointer to message is passed between layers. Therefore, no message copy is made within the stack.Finally, Cactus provides operations for unbinding handlers but it has no explicit operation for removing a micro-protocol. In order to facilitate protocol reconfiguration, we have added to Cactus API an operation for micro-protocol removing. In addition to the micro-protocol initiating function, each micro-protocol must have a remove function, which unbinds all its handlers and releases its own resources. This function will be executed when the micro-protocol is removed.

Figure 1 shows the architecture of P2PSAP; the protocol has a Socket interface and two channels: a control channel and a data channel. We present now those components.

A. Socket API

A main lack of Cactus CTP is that it has no application programming interface; application has to use an interface as though it was just another composite protocol. In order to facilitate programming, we have placed a socket-like API on the top of our protocol. Application can open and close connection, send and receive data. Furthermore, application will be able to get session state and change session behavior or architecture through socket options, which were not available in Cactus. Session management commands like listen, open, close, setsockoption and getsockoption are directed to Control channel; while data exchange commands, i.e. send and receive commands are directed to Data channel.

B. Data channel

The Cactus built data channel transfers data packets between peers. The data channel has two levels: the physical layer and the transport layer; each layer corresponds to a Cactus composite protocol. We encompass the physical layer to support communications on different networks, i.e. Ethernet, InfiniBand and Myrinet. Each communication type is carried out via a composite protocol. The data channel can be triggered between the different types of networks; one composite protocol is then substituted to another. The transport layer is constituted by a composite protocol made of several micro-protocols, which is an extension of CTP. We have added to the existing micro-protocols a remove function corresponding to the modifications we have introduced to the Cactus framework. In addition, we have designed some new micro-protocols that enable CTP to be used for sending and receiving messages in distributed computing applications as we shall see in the sequel.

Micro-protocols synchronization: CTP supports only asynchronous communication. Distributed applications may nevertheless use plural communication modes. Hence, we have implemented two micro-protocols corresponding to two communication modes: synchronous and asynchronous. These micro-protocols introduce new events, *UserSend* and *UserReceive*, that will be raised when send and receive socket commands will be called by an application. In response to messages sent from application, these micro-protocols may return the control to application immediately after message sent (asynchronous send) or wait for an acknowledgement indicating that message was received by receiver side application (synchronous send). Likely, in response to receive call from application, they may return the control to application immediately with or without message (asynchronous receive), or wait until message arrives (synchronous receive).

Micro-protocol buffer management: two buffers must be managed: a sending buffer and a receiving buffer. The sending buffer stores messages to be sent or that need to be acknowledged. The receiving buffer stores messages sent by other peers that are waiting to be delivered. This micro-protocol implements handlers for the *UserSend* and *MsgFrom-Net* events to catch messages from application and network.

Micro-protocols congestion control: CTP has several micro-protocols implementing SCP congestion control and TCP-Tahoe congestion control. We have designed and used new micro-protocols implementing the TCP New-Reno congestion control [6] and the H-TCP congestion control for high speed-latency network [7].

At this level, data channel reconfiguration is carried out by substituting or removing and adding micro-protocols. The behavior of the data channel is triggered by the control channel.

C. Control channel

The Control channel manages session opening and closure. It captures context information and (re)configures the data channel at opening or operation time. It is also responsible for coordination between peers during reconfiguration process. Note that we use the TCP/IP protocol to exchange control messages since those messages must not be lost. We describe now the main components of the control channel.

1) Context monitor: the context monitor collects context data and their changes. Protocol adaptation is based on context acquisition, data aggregation and data interpretation. Context data can be requirements imposed by the user at the application level, i.e. synchronous or asynchronous schemes of computation. Context data can also be related to peers location and machine loads. Context data are collected at specific times, periodically or by means of triggers. Data collected by the context monitor can be referenced by the controller.

2) Controller: the controller is the most important component of the control channel; it manages session opening and end through TCP connection opening and closure; it also combines and analyzes context information provided by the context monitor so as to choose the configuration (at session opening) or to take reconfiguration decision (during session operation) for data channel. The choice of the most appropriate configuration is determined by a set of rules that are described by a specification language such as OWL, ECA, etc. These rules specify new configuration and actions needed to realize it. The (re)configuration command along with necessary information is sent to component Reconfiguration and to other communication end point.

3) Reconfiguration: reconfiguration actions are made by the reconfiguration component via the dedicated Cactus functions. Reconfiguration is mainly made at the transport layer by substituting or removing and adding micro-protocols that support communication mode.

4) Inter-peer coordination: the coordination component is responsible of context information exchange and coordination process related to peers reconfiguration so as to ensure proper working of the protocol.

D. Self-adaptation mechanism

Similar machines connected via a local network with small latency, high bandwidth and reliable data transfer can be gathered in a cluster.

During solution, the transport protocol is configured according to the following context data: schemes of computation (i.e. synchronous, asynchronous or hybrid iterative schemes) and topology parameters like type of connection (i.e. intra or inter cluster). Decision rules are summarized in Table 1. In the sequel, we explain those rules.

TABLE I. COMMUNICATION ADAPTATION RULES

Scheme / **Connection**	**Synchronous**	**Asynchronous**	**Hybrid**
Intra-cluster	Synchronous Reliable Com.	Asynchronous Reliable Com.	Synchronous Reliable Com.
Inter-cluster	Synchronous Reliable Com.	Asynchronous Unreliable Com.	Asynchronous Unreliable Com.

Sometimes, communication mode must fit a computational scheme requirement (e.g. a special requirement related to the convergence of the implemented numerical method) as in the case where synchronous computational schemes are imposed. Then, synchronous communications are imposed in both intra-cluster and inter-cluster data exchanges. In this case, micro-protocols used for the data channel will be the Synchronous micro-protocol with some reliability and order micro-protocols. To explore the high-speed long distance network, the data channel can use H-TCP congestion control micro-protocol for inter-cluster communication instead of TCP New-Reno congestion control micro-protocol which works well only in low latency network.

Likely, in the case where asynchronous schemes of computation are required by user, asynchronous communication must be preferably implemented in both intra-cluster and inter-cluster data exchanges. We note that asynchronous schemes of computation are fault tolerant in some sense since they allow messages losses. However, messages losses may lead to some extra relaxations. For this reason, in intra-cluster communication with low latency, it may be better to add some reliability micro protocols to the data channel along with the Asynchronous micro-protocol. While in inter-cluster communication with high latency and message losses recovery time may be comparable with updating time, thus those messages can become obsolete. Hence, reliability micro protocols are not needed in this case.

There are also some situations where a given problem can be solved by using any combination of computational schemes. In this latter case, the user can leave the system to freely choose communication mode according to elements of context like inter-cluster or extra cluster connection. As a consequence, the self adaptive communication protocol will choose the most appropriate communication mode according to topology parameters. This corresponds to the so-called *Hybrid* scheme of computation. In this case, if computational loads are well balanced, then synchronous communication between peers are appropriate. On the other hand, synchronization may be an obstacle to efficiency and robustness in inter-cluster data exchanges situations where there may be some heterogeneity, i.e. processors, OS, bandwidth, and communications may be unreliable and have high latency. Thus, asynchronous communication seems more appropriate in this latter case.

III. ENVIRONMENT P2PDC

In this Section, we display the global architecture of P2PDC and present the current version with simplified and centralized functions.

A. *Environment global architecture*

Figure 2 illustrates the architecture of our environment. We describe now its main components.

1) User daemon is the interaction interface between the application and the environment. It allows users to submit their tasks and retrieve final results.

2) Topology manager organizes connected peers into clusters and maintains links between clusters and peers.

3) Task manager is responsible for sub-tasks distribution and results collection.

4) Task execution executes sub-tasks and exchanges intermediate results.

5) Load balancing estimates peer workload and migrates a part of work from overloaded peer to non-loaded peer.

6) Fault tolerance ensures the integrity of the calculation in case of peer or link failure.

7) Communication provides support for data exchange between peers using protocol P2PSAP.

B. *Programming model*

We have proposed a programming model that allows all programmers to develop their own application easily.

Communication operations: The set of communication operations is reduced. There are only a send and a receive operations (*P2P_Send* and *P2P_Receive*). The idea is to facilitate programming of large scale P2P applications and hide complexity of communication management as much as possible. Contrarily to MPI communication library where communication mode is fixed by the semantics of communication operations, the communication mode of a given communication operation which is called repetitively can vary with P2PDC according to the context; e.g. the same *P2P_Send* from peer A to peer B, which is implemented repetitively, can be first synchronous and then become asynchronous. As a consequence, the programmer does not fix directly the communication mode; he rather selects the type of scheme of computation he wants to be implemented, e.g. synchronous or asynchronous iterative scheme or let the protocol free by choosing a hybrid scheme. When the system is set free, the choice of communication mode will depend only on elements of context like topology change and thus be dynamic.

Application programming model: Figure 3 shows the steps that a parallel application must follow in order to be deployed. We want the environment to carry out those activities automatically. Hence we propose a programming model based on this diagram. Only activities with solid line boundary are taken into account by the programmers. Activities with broken line boundary are taken into account by the environment and are transparent to programmers. Thus, in order to develop an application, programmers have to write code for only three functions corresponding to the following three activities: *Problem_Definition()*, *Calculate()* and *Results_Aggregation()*. In the *Problem_Definition()* function, programmers define the problem in indicating the number of sub-tasks and sub-task data. The computational scheme and number of peers necessary can also be set in this function but they can be overridden at start time in command line. In the *Calculate()* function, programmers write sub-tasks code; they can use *P2P_Send()* and *P2P_receive()* to send or receive updates at each relaxation. Programmers define how sub-tasks results are aggregated and the type of output, i.e. a console or a file, in the *Results_Aggregation()* function.

We note that with this programming model all difficult tasks like load balancing and fault tolerance are managed by the environment; this reduces the work of programmers. Moreover, it allows P2PDC to implement some automatic functionality that are not implemented with MPI e.g. automatic load balancing in function of peer characteristics and load at start and run time.

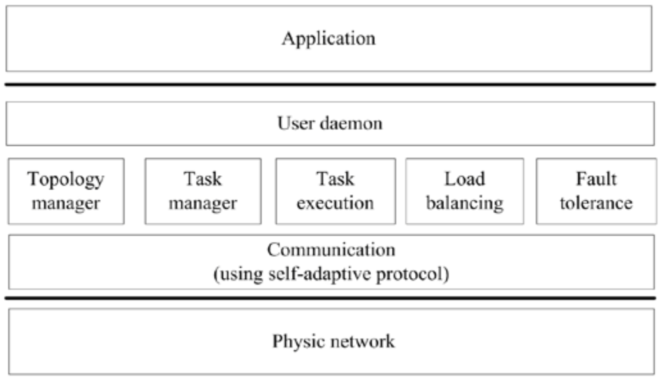

Figure 2. Environment architecture

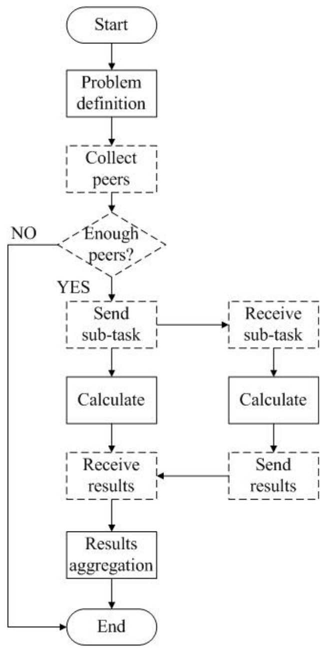

Figure 3. Activity diagram of a distributed application

C. Implementation

The current implementation of P2PDC is presented in the sequel.

1) User daemon

The User daemon component constitutes for the moment the interface between user and environment. We outline here some principal commands:

- *run*: run an application. Parameters are application name and application owner parameters that will be passed to *Problem_Definition()* function.
- *stat*: return actual state of node.
- *exit*: quit the environment.

2) Topology manager

The topology manager component is currently centralized. We use a server in order to store information about all nodes in the network. When a node joins the network, it sends to the server a message. The server adds the new node to peer list and sends to the node an acknowledgement message. Peers must send ping messages periodically to server to inform it that they are alive. If the server does not receive ping message from a peer after 3 ping periods, the server considers that this peer is disconnected and removes it from the peer list.

The topology manager is also responsible of peer collections. When Task manager requests peers to execute a new application, it sends a request to the server with number of peers needed; the server checks its peer list and returns free peers to the task manager of the request peer.

3) Task manager

Task manager is the main component that calls functions of the application. When an user starts an application using the *run* command, this component finds the corresponding application via application name and calls the *Problem_Definition()* function. It requests peers from Topology manager on the basis of number of peers needed by application and sends sub-tasks with their data to collected peers. When all peers have sent the results, Task manager calls the *Results_Aggregation()* function.

4) Task execution

When a peer receives a sub-task, it finds the corresponding application via application name and calls the *Calculate()* function.

Load balancing and Fault tolerance components have not yet been developed.

IV. APPLICATION TO OBSTACLE PROBLEM

The application we consider, i.e. the obstacle problem, belongs to a large class of numerical simulation problems (see [8] and [12]). The obstacle problem occurs in many domains like mechanics and financial mathematics, e.g. options pricing.

A. Fixed point problem and projected Richardson method

The discretization of the obstacle problem leads to the following large scale fixed point problem whose solution via distributed iterative algorithms (i.e. successive approximation methods) presents many interests.

$$\begin{cases} Find\ u^* \in V\ such\ that \\ u^* = F(u^*), \end{cases} \quad (1)$$

where V is an Hilbert space and the mapping $F: v \mapsto F(v)$ is a fixed point mapping from V into V. Let α be a positive integer, for all $v \in V$, we consider the following block-decomposition of v and the associated block-decomposition of the mapping F for distributed implementation purpose:

$$v = (v_1, \dots, v_\alpha),$$
$$F(v) = \big(F_1(v), \dots, F_\alpha(v)\big).$$

We have $V = \prod_{i=1}^{\alpha} V_i$, where V_i are Hilbert spaces; we denote by $\langle .,. \rangle_i$ the scalar product on V_i and $|.|_i$ the associated norm, $i \in \{1, \dots, \alpha\}$; for all $u, v \in V$, we denote by $\langle u, v \rangle = \sum_{i=1}^{\alpha} \langle u_i, v_i \rangle_i$, the scalar product on V and $\|.\|$ the associated norm on V. In the sequel, we shall denote by A a linear continuous operator from V onto V, associated such that $A.v = (A_1.v, \dots, A_\alpha.v)$ and which satisfies:

$$\forall i \in \{1, \dots, \alpha\}, \forall v \in V, \langle A_i.v, v_i \rangle_i \geq \sum_{j=1}^{\alpha} n_{i,j} |v_i|_i |v_j|_j, \quad (2)$$

where

$$N = (n_{i,j})_{1 \leq i,j \leq \alpha}\ is\ an\ M-matrix\ of\ size\ \alpha \times \alpha. \quad (3)$$

Similarly, we denote by K_i, a closed convex set such that $K_i \subset V_i, \forall i \in \{1, \dots, \alpha\}$, we denote by K, the closed convex set such that $K = \prod_{i=1}^{\alpha} K_i$ and b, a vector of V that can be written as: $b = (b_1, \dots, b_\alpha)$. For all $v \in V$, let $P_K(v)$ be the projection of v on K such that $P_K(v) = \left(P_{K_1}(v_1), \dots, P_{K_\alpha}(v_\alpha)\right)$, where P_{K_i} denotes the mapping that projects elements of V_i onto $K_i, \forall i \in \{1, \dots, \alpha\}$. For any $\delta \in R, \delta > 0$, we define the fixed point mapping F_δ as follows (see[8]).

$$\forall v \in V, F_\delta(v) = P_K(v - \delta(A.v - b)), \quad (4)$$

The mapping F_δ can also be written as follows.

$$F_\delta(v) = \left(F_{1,\delta}(v), \dots, F_{\alpha,\delta}(v)\right)\ with$$

$$F_{i,\delta}(v) = P_{K_i}\big(v_i - \delta(A_i.v - b_i)\big), \forall v \in V, \forall i \in \{1, \dots, \alpha\}.$$

B. Parallel projected Richardson method

We consider the distributed solution of fixed point problem (1) via projected Richardson method combined with several schemes of computation, i.e. a Jacobi like synchronous scheme: $u^{p+1} = F_\delta(u^p), \forall p \in N$ or asynchronous schemes of computation that can be defined as follows (see [8]).

$$\begin{cases} u_i^{p+1} = F_{i,\delta}\left(u_1^{\rho_1(p)}, \dots, u_j^{\rho_j(p)}, \dots, u_\alpha^{\rho_\alpha(p)}\right)\ if\ i \in s(p), \\ u_i^{p+1} = u_i^p\ if\ i \notin s(p), \end{cases} \quad (5)$$

where

$$\begin{cases} s(p) \subset \{1, \dots, \alpha\}, s(p) \neq \emptyset, \forall p \in N, \\ \{p \epsilon N | i \in s(p)\}, is\ infinite,\ \forall i \in \{1, \dots, \alpha\}, \end{cases} \quad (6)$$

and

$$\begin{cases} \rho_j(p) \in N, 0 \leq \rho_j(p) \leq p, \forall j \in \{1, \dots, \alpha\}, \forall p \in N, \\ lim_{p \to \infty} \rho_j(p) = +\infty, \forall j \in \{1, \dots, \alpha\}. \end{cases} \quad (7)$$

The above asynchronous iterative scheme can model computations that are carried out in parallel without order nor synchronization. In particular, it permits one to consider distributed computations whereby peers go at their own pace according to their intrinsic characteristics and computational load (see [8]). Finally, we note that the use of delayed components in (5) and (7) permits one to model nondeterministic behaviour and does not imply innefficiency of the considered distributed scheme of computation. The convergence of asynchronous projected Richardson method has been established in [8] (see also [15] to [17]).

The choice of scheme of computation, i.e. synchronous, asynchronous or any combination of both schemes will have important consequences on the efficiency of distributed solution as we shall see in the next Section. We have shown the interest of asynchronous iterations for high performance computing in various contexts including optimization and boundary value problems, e.g. see [8], [13], [14] and [18].

V. COMPUTATIONAL EXPERIMENTS

We present now and analyze a set of computational experiments for the obstacle problem.

A. *NICTA testbed*

Computational experiments have been carried out on the NICTA testbed [11]. This testbed is constituted of 38 machines having the same configuration, i.e. processor speed 1GHz, memory 1GB based on Voyage Linux distribution. Those machines are connected via 100MBits Ethernet network.

NICTA testbed uses OMF (cOntrol and Management Framework) to facilitate the control and management of the testbed [19]. Furthermore, we use OML (Orbit Measurement Library) to orchestrate the measurement during the experiment. OMF provides a set of tools to describe and instrument an experiment, execute it and collect its results; OMF provides also a set of services to efficiently manage and operate the testbed resources (e.g. resetting nodes, retrieving their status information, installing new OS image). In order to perform our experimentations, we have written plural descriptions files, using OMF's Experiment Description Language (OEDL), corresponding to different scenarios. Each description file contains: configuration of the network topology, i.e. peer’s IP address assignment so that they are in the desired cluster; network parameters, i.e. communication latency and path to application with appropriate parameters.

B. *Implementation*

In our experiments, the computation scheme (synchronous, asynchronous or combination of both schemes) is chosen at the beginning of the resolution whereas the communication mode is decided at runtime by the adaptive transport protocol.

For simplicity of presentation and without loss of generality, we have displayed in Figure 4 the basic computational procedure at node k with $k \neq 1, k \neq \alpha$. The k-th node updates the sub-blocks of components of the iterate vector denoted by $U_{f(k)}, U_{f(k)+1}, \dots, U_{l(k)}$, where $U_{f(k)}$ stands for the first sub-block of the k-th node and $U_{l(k)}$ stands for the last sub-block of the k-th node. We note that the transmission of $U_{f(k)}$ to node $k-1$ is delayed so as to reduce the waiting time in the synchronous case.

C. *Problems and results*

We have considered several 3-Dimensional obstacle problems. Let n^3 denote the number of discretization points,

```
send $U_{l(k)}$ to node $k+1$
do until convergence
    $i \leftarrow f(k)$
    receive $U_{i-1}$ from node $k-1$
    $U_i \leftarrow F_{i,\delta}(U_{i-1}, U_i, U_{i+1})$

    do $i = f(k)+1, l(k)-1$
        $U_i \leftarrow F_{i,\delta}(U_{i-1}, U_i, U_{i+1})$
    end

    send $U_{f(k)}$ to node $k-1$
    $i \leftarrow l(k)$
    receive $U_{i+1}$ from node $k+1$
    $U_i \leftarrow F_{i,\delta}(U_{i-1}, U_i, U_{i+1})$
    send $U_i$ to node $k+1$
end
```

Figure 4. Basic computational procedure at node k.

the iterate vector is decomposed into n sub-blocks of n^2 points. The sub-blocks are assigned to α nodes with $\alpha \leq n$. The sub-blocks are computed sequentially at each node..

In this paper, we present a set of computational experiments obtained with $n = 96$ and $n = 144$. Experiments have been carried out on 1, 2, 4, 8, 16 and 24 machines.

In the distributed context, i.e. for several machines, we have considered the case where machines either belong to a single cluster or are divided into 2 clusters connected via Internet. We used the Netem tool to simulate the Internet context; the latency between 2 clusters is set to 100ms. We have carried out experiments with different schemes of computation, i.e. synchronous, asynchronous and hybrid.

Figures 5 and 6, respectively, show the time, number of relaxations, speedup and efficiency of the different parallel schemes of computation in the case where $n = 96$ and $n = 144$, respectively. For the application and topologies considered, we note that asynchronous schemes of computation have performed better than the synchronous ones.

The efficiency of asynchronous schemes of computation decreases slowly with the number of processors; while the efficiency of synchronous schemes of computation deteriorates greatly when the number of processors increases (this is particularly true in the case of 2 clusters); this is mainly due to synchronization overhead and waiting time.

The speedup of synchronous schemes of computation is very small for 24 nodes. This can be explained as follow: when 24 nodes are used, each node calculates only a small number of sub-blocks; since exchanged messages and sub-blocks have the same size, communication overhead and waiting time then reach a significant proportion.

When we compare the computational results with 1 and 2 clusters, we can see that there is not much difference with regard to the asynchronous schemes; while in the synchronous cases, 1 cluster results are better than 2 clusters results. This is due to the fact that communication latency between 2 clusters (100ms) increases the waiting time due to synchronization; this means that synchronous communication is sensible to latency increase and not appropriate for the communication between clusters.

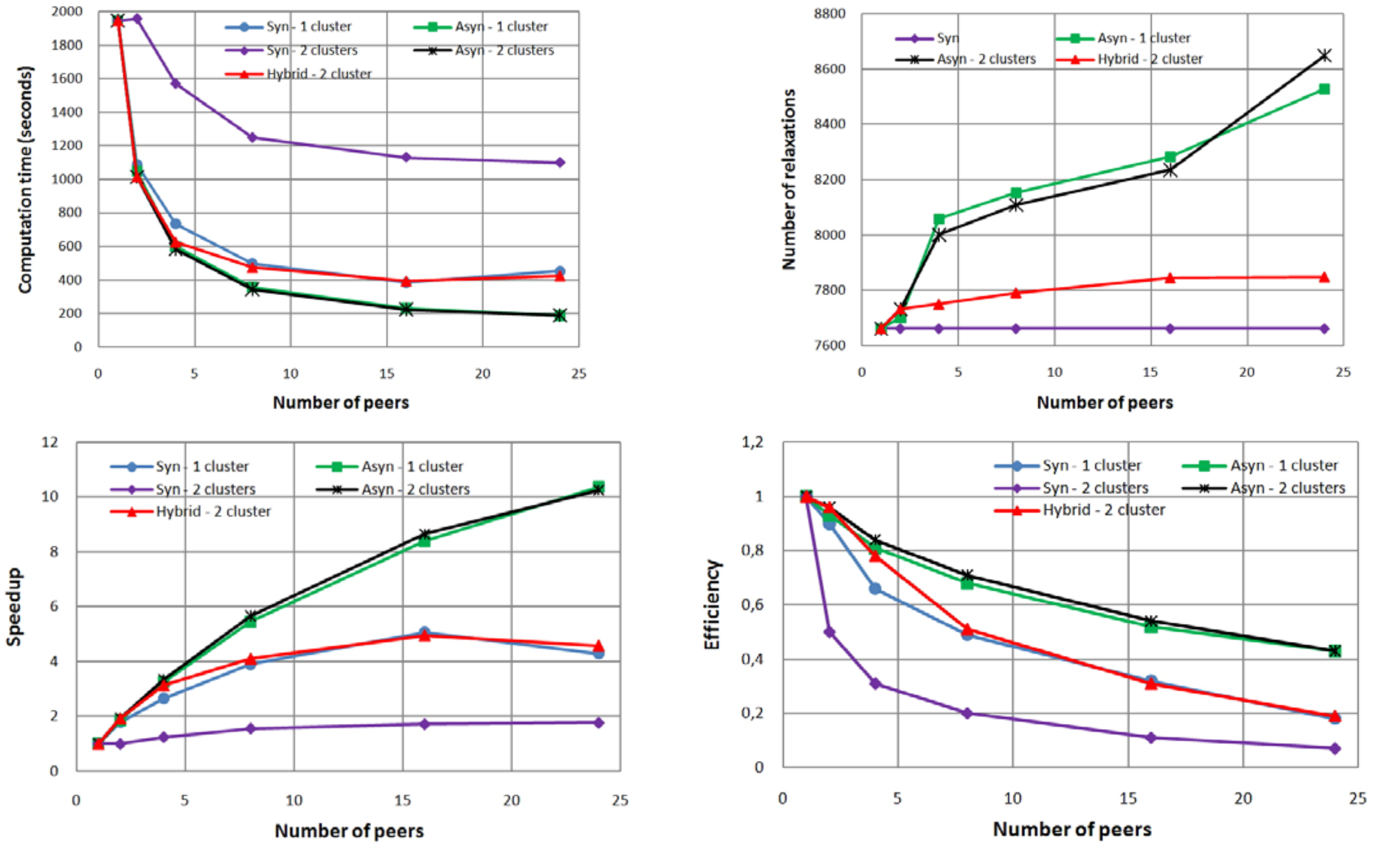

Figure 5. Computational results in the case of problem size 96x96x96

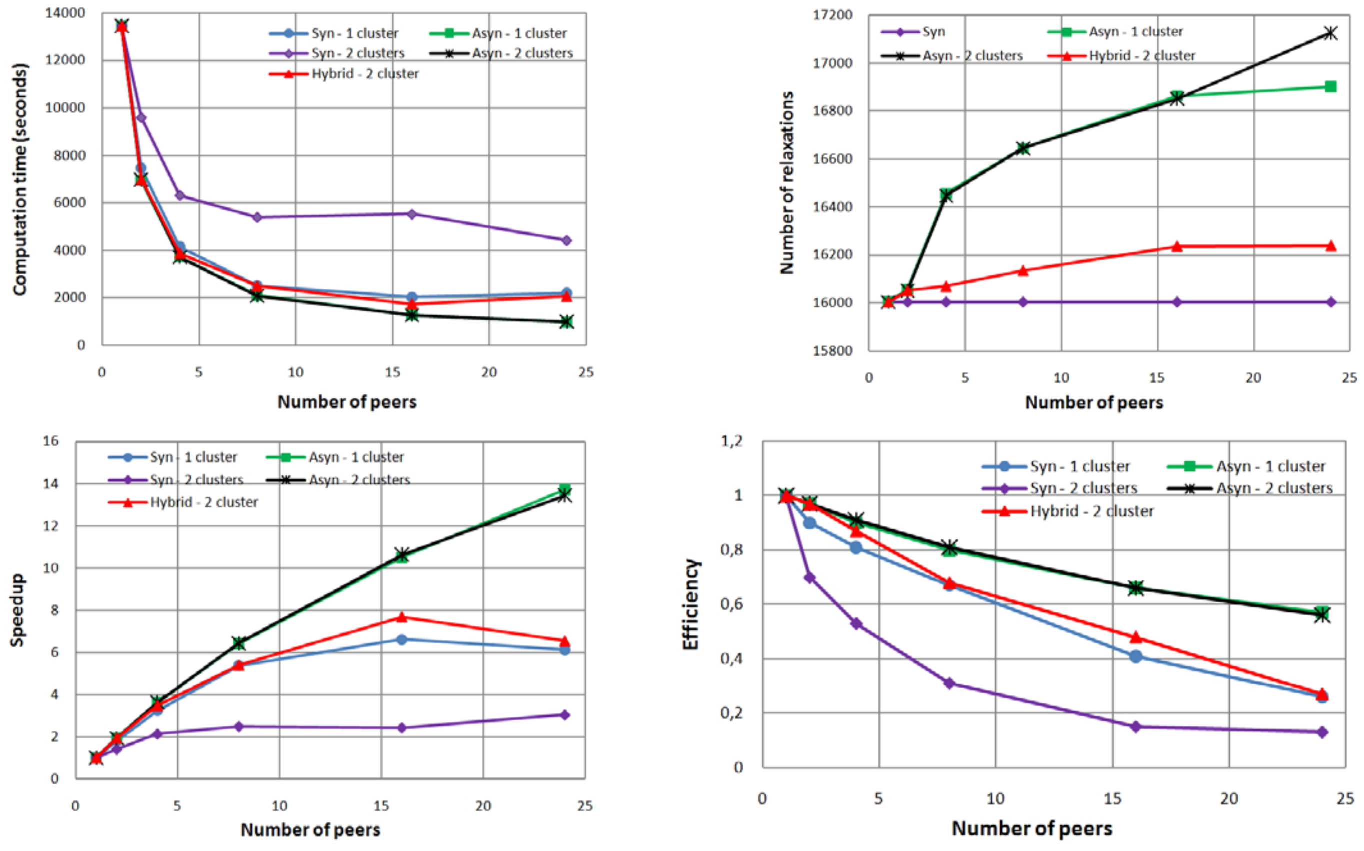

Figure 6. Computational results in the case of problem size 144x144x144

When the problem size increases from $n = 96$ to $n = 144$, the efficiency of distributed methods increases since granularity increases.

The number of relaxations performed by synchronous schemes remains constant although the sub-block processing order is changed by the distribution of computation.

In the case of asynchronous schemes of computation, some nodes may iterate faster than others; this is particularly true when nodes have fewer neighbors than others, like nodes 1 and α that have only one neighbor. Then, the average number of relaxations increases with the numbers of machines, as depicted in Figure 5 and 6.

The efficiency of hybrid schemes of computation is situated in between efficiencies of synchronous and asynchronous schemes.

It follows from the computational experiments that the choice of communication mode has important consequences on the efficiency of the distributed methods. The ability for the protocol P2PSAP to choose the best communication mode in function of network topology and context appears as a crucial property. We note also that the choice of communication mode has important consequences on the reliability of the distributed method and everlastingness of the high performance computing application. With regards to these topics, we note that asynchronous communications are more appropriate in the case of communications between clusters.

VI. CONCLUSION

In this paper, we have presented P2PSAP, a self adaptive communication protocol. We have also detailed the current version of P2PDC, an environment for high performance peer to peer distributed computing that allows direct communication between peers. We have displayed and analyzed computational results on the NICTA platform with up to 24 machines for numerical simulation problem, i.e. the obstacle problem.

The computational results show that P2PSAP permits one to obtain good efficiency, in particular, when using asynchronous communications or a combination of synchronous and asynchronous communications.

In the future, we plan to study a specification language for controller decision rules description. We shall also develop decentralized functions of P2PDC. This type of environment will permit one to use all the specificities offered by the P2P concept to high performance computing. Self organization of peers for efficiency purpose or for ensuring everlastingness of applications in hazardous situations or in the presence of faults will also be studied. Finally, we plan to consider other applications e.g. process engineering application with many more machines. The different applications considered will permit us to validate experimentally our protocol and decentralized environment in different high performance computing contexts.

REFERENCES

[1] Gnutella Protocol Development. *http://rfc-gnutella.sourceforge.net.*

[2] The FreeNet Network Projet. *http://freenet.sourceforge.net.*

[3] D. El Baz, T. T. Nguyen et al, “CIP - Calcul intensif pair à pair”, Poster, session, *Ter@tec2009*, Gif-sur-Yvette, France, June 30 - July 1, 2009.

[4] Matti A. Hiltunen, “The Cactus Approach to Building Configurable Middleware Services”, in *DSMGC2000*, Nuremberg, Germany, 2000.

[5] G.T Wong, M.A Hiltunen, R.D Schlichting, “A configurable and extensible transport protocol,” in *Proceedings of IEEE INFOCOM ’01*, Anchorage, Alaska (2001), pp. 319–328.

[6] S. Floyd, T. Henderson, “The New-Reno Modification to TCP’s Fast Recovery Algorithm,” *RFC 2582*, Apr 1999.

[7] D. Leith and R. Shorten, “H-TCP protocol for high-speed long distance networks,” in *PFLDnet*, Feb. 2004.

[8] P. Spitéri, M. Chau, “Parallel asynchronous Richardson method for the solution of obstacle problem” in *Proceedings of the 16th Annual International Symposium on High Performance Computing Systems and Applications*, 2002, pp. 133-138.

[9] D. El Baz, G. Jourjon, “Some solutions for Peer to Peer Global Computing,” in *13th Euromicro conference on Parallel, Distributed and Network-Base Processing*, 2005, pp. 49-58.

[10] D. El Baz, T.T. Nguyen, “A self-adaptive communication protocol with application to high performance peer to peer distributed computing”, in *The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, 2010, Pisa.

[11] NICTA testbed. http://www.nicta.com.au.

[12] J.L. Lions, “Quelques méthodes de résolution des problèmes aux limites non linéaires”, *Dunod* 1969.

[13] D. El Baz, “M-functions and parallel asynchronous algorithms”, *SIAM Journal on Numerical Analysis*, Vol. 27, N° 1, pp. 136-140, 1990.

[14] D.P. Bertsekas, D. El Baz, “Distributed asynchronous relaxation methods for convex network flow problems”, *SIAM Journal on Control and Optimization*, Vol. 25, N° 1, pp. 74-85, 1987.

[15] J. Miellou, P. Spiteri, “Two criteria for the convergence of asynchronous iterations”, in *Computers and computing*, P. Chenin et al. ed., Wiley Masson, Paris, pp. 91-95, 1985.

[16] L. Giraud, P. Spiteri, “Résolution parallèle de problems aux limites non linéaires”, *M2AN*, vol. 25, pp. 597-606, 1991.

[17] J. Miellou, P. Spiteri, “Un critère de convergence pour des methodes generales de point fixe”, *M2AN*, vol. 19, pp. 645-669, 1985.

[18] D. El Baz, “Nonlinear systems of equations and parallel asynchronous iterative algorithms”, in *Advance in Parallel Computing,vol. 9, Parallel Computing Trends and Applications*, North-Holland, pp. 89-96, 1994.

[19] T. Rakotoarivelo, M. Ott, I. Seskar, and G. Jourjon, “OMF: a control and management framework for networking testbeds,” in *SOSP Workshop on Real Overlays and Distributed Systems* (ROADS ’09).

[20] O. Aumage, G. Mercier, "MPICH/Madeleine: a True Multi-Protocol MPI for High Performance Networks," *15th International Parallel and Distributed Processing Symposium* (IPDPS'01), 2001.

[21] David P. Anderson, “BOINC: A System for Public-Resource Computing and Storage,” *5th IEEE/ACM International Workshop on Grid Computing*. November 8, 2004, Pittsburgh, USA.

[22] N. Andrade, W. Cirne, F. Brasileiro, P. Roisenberg, “OurGrid: An approach to easily assemble grids with equitable resource sharing”, in *Proceedings of the 9th Workshop on Job Scheduling Strategies for Parallel Processing*, pp. 61-86, June 2003.