Simulating and Optimizing A Peer-to-Peer Computing Framework

Jean-Baptiste Ernst-Desmulier[1], Julien Bourgeois[1], Minh Thanh Ngo[1],
François Spies[1], and Jérome Verbeke[2]

[1]Laboratoire d'Informatique de Franche-Comté
University of Franche-Comte
1, Cours Leprince-Ringuet
25200 Montbéliard FRANCE
{ernst, bourgeoi, ngo, spies}@lifc.univ-fcomte.fr

[2]Lawrence Livermore National Laboratory
Livermore, CA 94550 USA
verbeke2@llnl.gov

Abstract

The aim of P2P computing is to build virtual computing systems dedicated to large-scale computational problems. JXTA[1] proposes an underlying infrastructure on which JNGI[2], one of the first P2P decentralized computing frameworks is built. In order to test this framework, we have built a tool named P2PPerf, which allows us to study the behavior of JNGI and to optimize it according to our simulation results.

1. Introduction

Designing, testing and tuning P2P computing frameworks are three difficult tasks and there are many reasons for this.

First of all, gathering a sufficient number of computers to test or to tune a P2P computing framework is a highly time-consuming task. Besides, even if some very useful projects like PlanetLab [15] do exist, it is very difficult to obtain a sufficient number of nodes to really study the system. For example, there are more than 2500 working nodes connected to Seti@home [2] per day as stated on their homepage.

Then, the aim of existing projects [3, 13] is to offer a testbed to developers but not a specific testbed for P2P applications. Therefore, they cannot be used to test the robustness of a P2P computing framework when nodes appear or disappear because the nodes of a testbed are already connected to a network. These environments do not exactly meet the goal of really testing P2P computing frameworks.

Finally, while the project is being developed, a developer wants to have rapid results to correct its design. The time spent to obtain the performance results is a mandatory parameter. The time required to run a complete example could be too long. The simulation must give performance results faster than the real execution. NS2 [1] has the reputation to be slow to give results when simulating a large-scale P2P network. This article will show that it can be used with very good performance results if the collected data is well chosen. Moreover, NS2 is a widely-used simulator so there is no need for the user to install a new simulator and our modules can easily be added to the NS2 base installation.

In the following section 2, we present the JNGI P2P computing framework which has been used for testing our P2PPerf framework. Section 3 discusses our model of P2P computing framework, which comprises two main modules, the network and the computational ones. Section 4 presents the tests we have conducted and section 5 concludes the article and describes the future work to be done.

2. JNGI Presentation

JNGI [18] is a peer-to-peer distributed computing framework written in Java, based on the JXTA [17] virtual network, which enables large and embarrassingly parallel applications to be executed on numerous computing peers. While the idea of achieving parallelization through performing many independent tasks is not new [2, 13], JNGI extends this approach to ad-

[1]JXTA for JuXTApose

[2]JNGI for Jerome, Neelakanth, Greg and Ilya : first names of the creators

1-4244-0054-6/06/$20.00

dress other aspects that include (1) dynamism, where nodes are added and removed during the lifetime of the jobs; (2) redundancy, so that the dynamic nature of the grid does not affect the results; (3) organization of computational resources into groups, so that inter-node communication does not occur in a one-to-all or all-to-all mode, thereby limiting the scalability of the system; and (4) heterogeneity, where a wide variety of computational platforms can take part in the computation. Heterogeneity (point 4) is achieved through the ability of Java bytecode to run on various platforms. The framework uses the JXTA open peer-to-peer communication protocols, which entails the dynamic aspect (point 1) of the grid through peer discovery, in addition to the scalability aspect (point 3) through the use of propagate pipes within groups.

2.1. Structure

As shown in figure 1, two kinds of peers are present in each peer group: worker peers executing tasks, and task dispatcher peers scheduling and farming out tasks to the worker peers. The tasks to be performed as well as the completed ones are kept in a database on the task dispatcher peers called Task Repository. Redundancy (point 2) is addressed by replicating these databases onto several task dispatcher peers. To account for the volatile character of these peers, which could quit the network at any time, worker peers can dynamically migrate to become task dispatcher peers, and vice versa. To some extent, the virtual grid transparently moves over the physical network as peers drop/join JNGI. As the number of peers in the grid increases, inefficiency can arise due to overwhelming communication overheads within a peer group. For this reason, several groups exist within the framework, as shown in figure 2.

Monitor peers supervise the workloads within these groups, and are responsible for splitting them into subgroups if task dispatcher peers become overloaded. They are also responsible for directing peers, which recently joined the proper peer group.

3. The P2PPerf framework

The aim of our tool called P2Pperf is to propose generic architecture to evaluate the performance of a P2P computing framework. Thanks to P2PPerf, it will be easier to study and to evaluate the behavior of the P2P application in a simulated context without accessing existing P2P architecture. Moreover, it will be possible to simulate the same scenario as many times as necessary to generate exactly the same circumstances

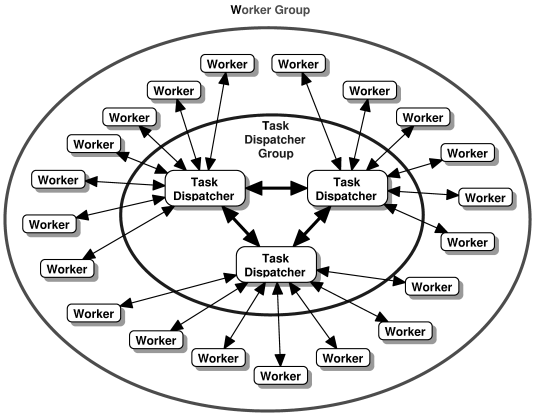

Figure 1. Communication between workers and redundant task dispatcher within a peer group.

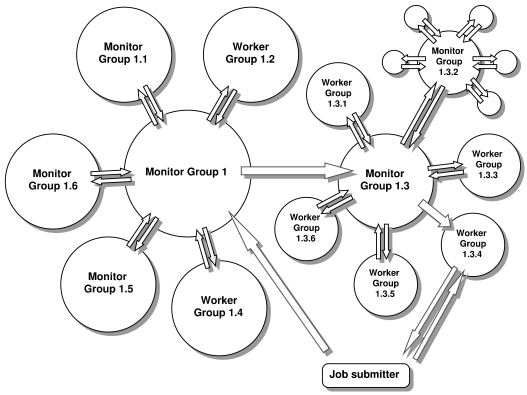

Figure 2. Peer groups hierarchy in JNGI framework.

in order to identify any types of bug or erroneous behavior and to evaluate performance in a specific situation. By studying such P2P application during a simulated stage, it will be easier and faster to extend performance in order to demonstrate the scalability of the program and the convergence of the execution time, even if specific events such as peer dropping and peer joining occur during the execution.

As can be seen in figure 3, the first input of P2PPerf is the source code in Java of the application executed on the P2P framework. P2PPerf also needs a target computer which will calibrate the performance of the workers. These inputs are sent to the CompPerf module, which micro-benchmarks [16] the target computer and evaluates the time taken by the sequential parts

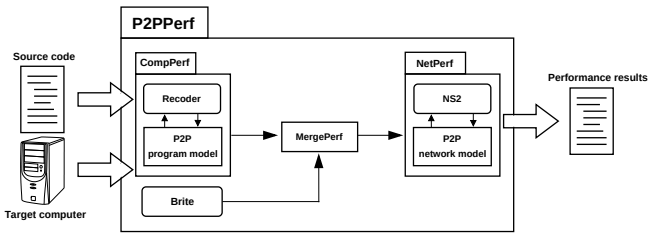

Figure 3. Design of the P2PPerf system.

of the program. The file generated by CompPerf is merged with the topology file generated by Brite [14] and then sent to the NetPerf module. NetPerf uses NS2 as a discrete-event simulation engine and just adds the modules that correspond to the P2P computing framework. The following sections will describe the CompPerf JNGI modules and NetPerf JNGI modules.

3.1. CompPerf JNGI modules

The aim of this module is to evaluate the time taken by the sequential parts of a program. It is based on our experience in performance prediction [6]. To do this, we use a static analysis because it decreases the slowdown of the performance prediction and it generates a parametric representation of the performance model. With this approach, no more modelization process is needed to create a new model: only the parameters would be adapted. To evaluate the sequential parts, it is necessary to know the structure of the program. Then, a two-part analysis of the program is necessary. The first part counts the basic instructions i.e. solves the unknown parameters of the program and the second one deduces the execution times of these instructions. These steps are explained below.

3.1.1 Description of a JNGI application

A typical JNGI application is a class that extends the *Runnable* class and the *Serializable* class. It is composed of three important parts: the first part is the constructor of the class. In the constructor, there are six important steps: splitting the job into tasks, initializing the RemoteThread object, starting the job, waiting until the job is completed to retrieve the results, removing the job from the code repository and finally stopping the RemoteThread. The second and most important part is the *run()* function, which is the core of the computation. This function contains the code executed by a worker peer to complete its associated task. The last part is the *postprocess()* function, which represents the postprocess mechanism used to aggregate the different results sent by the workers to obtain the result of the job. Note that in this version, the post-process mechanism is executed on the peer which submits the job to the framework. It is needed to evaluate the execution and communication times of these three parts. To do this, a static analysis of the JNGI application should be done.

3.1.2 Static analysis

It is necessary to define the terms that describe a static identification properly. The static identification of unknowns defines a method where identification is performed without executing the source code, just by parsing it. Unknowns in a program are the number of iterations in loops and the branch rate of conditionnal instructions. Thus, the complete static identification of a program means that the identification of unknowns is performed in a static way, just like the execution time approximation.

The tool chosen to traverse the application source code, to construct and to analyze the syntax-tree is RECODER [5]. RECODER provides a parsing tool and other services to analyze or modify Java programs. It is written in Java. The source code of the three parts of a JNGI application is parsed and a syntax-tree is built.

3.1.3 Solving the unknowns

The static identification of unknowns implies facing two problems. The first one is the approximation of conditional expressions found in structures such as the *if...else* structure or the *switch...case* structure. The second one consists in knowing how to count the number of iterations of loops.

Conditional expressions The approximation of conditional expressions in a static way in all the configurations is a problem. It is difficult to solve this problem without resorting to the program designer's knowledge. Some conditions on scalar variables can be solved automatically. However, in most cases, the conditions are not easily computable. The solution adopted is to turn to the designer. Thus, the conditions should be preceded by the reservated function named *CompPerf_Proba(x)* to be evaluated more precisely. The addition of the *CompPerf_Proba(x)* function is the only example of the user's intervention.

Loops and nested loops The nested loops are an essential aspect of an application because they often represent most of the computing time. A loop is a *for*-type or a *while*-type structure. To simplify the syntax

of the following examples, only the management of the *for* loops will be described, but the *while* loops are included in the same way in our model because it is easy to transform a *while* loop into a *for* loop. The static identification of the number of iterations can be applied to three categories of nested loops: nested loops with independent numerical bounds, nested loops with independent and constant scalar bounds and interdependent nested loops with constant scalar bounds.

As long as the bounds are independent, the static identification of the nested loop iteration number comes down to solve every loop separately.

When one or several bounds of the loop are defined by a constant variable or when the increment is a constant variable, the number of iterations is expressed literally. When the bounds of a loop depend on another loop, they are called interdependent nested loops. In this case, finding the number of iterations is to solve a system of inequations and to count the integer solutions. This system of inequations defines a convex polytope [12] in a n dimension space where n is the number of nested loops.

The method used to determine the integer points that are inside the polytope is based on Ehrhart polynomials [9]. This method and the resolution algorithm were developed at the ICPS by Philippe Clauss and Vincent Loechner [7].

It is important to notice that in some case, it is impossible to resolve the unknowns in a static way due to the complexity of the problem. In these cases, the solution is to trace the execution of the program on a target computer and to analyse the trace file generated. This trace module is still in development. It has to be noticed that trace file is only used to solve the unknowns. Trace files does not contain execution time informations.

3.1.4 Execution times

After solving the unknowns, the second step is to deduce the execution times. There are two methods to calculate the execution time of a program. The first technique consists in benchmarking the execution time of the program. The drawback is that the program has to be completely executed. The second approach is to analyze the basic instructions of a program and to measure their execution times. The advantage of this method is that it separates the analysis step from the simulation step. The set of instructions of an application is detailed below.

Basic instructions The choice of a pertinent set of instructions is very important to the simulation. The set of instructions of CompPerf is composed of 313 instructions divided into three classes: operations on data types, control structure and mathematical functions.

All these instructions describe a large set of applications. Note that all these instructions are basic Java standard instructions. The micro-benchmarking technique is used to identify the instruction execution time.

Micro-benchmarking The aim of the micro-benchmarking technique is to measure very short time events on a computer. All the instructions in the instruction set are micro-benchmarked on a reference computer. But, in a large-scale peer-to-peer network, peers are strongly heterogeneous and this heterogeneity should be taken into account when receiving the execution times. To build a realistic peer model, adaptive coefficients are used to modify the execution times. Two kinds of modifications are allowed:

- Speed modification. This modification is done by multiplying every instruction time by a coefficient to increase the global speed of the computer.
- Characteristic modification. It is possible to modify one or several characteristics of the peer model by applying an adaptive coefficient to the execution time of one or more specified instructions, to increase the speed of the floating point unit or of the memory access.

Table 1 presents the results of defining an adaptive coefficient of speed variation. A mixed benchmark composed of various instructions is executed on four Athlon XP (1700+ to 2000+) with 512 MB memory running Linux Debian with the same version of Java Runtime Environnement. In this test, only the CPU frequency varies. The fourth line of the table presents the ratio between frequency and execution time. The absolute value of the last line presents the speed coefficient due to the linear aspect of the ratio. Other specific tests have been executed in order to determine other adaptive coefficients.

These modifications allow the execution times to be as close as possible to real peer-to-peer architecture. Note that the static expression that describes the execution time of an application could be relatively complex and difficult to understand. A simplification module is used to simplify most of the expressions. This module is JEP which is a Java API for parsing and evaluating mathematical expressions. With this library, it is possible to enter an arbitrary formula as a string, and instantly simplify it and evaluate it. We use this library to gather up equivalent expressions and simplify computation time equations.

computer	1700+	1800+	1900+	2000+
frequency	1466	1533	1600	1666
bench (ms)	7832.69	7199.56	6560.45	5951.47
ratio	3572.31	4100.28	4696.39	5342.90
coefficient	-	-9,449	-9,448	-9,452

Table 1. Determining the CPU speed adaptive coefficient.

3.2. NetPerf JNGI modules

The aim of these modules is to simulate the execution of a JNGI application. These modules use an OTCl file generated by the MergePerf module using the result file of the CompPerf Module merge with a BRITE topology. Simulations involving JNGI applications cannot be run using the standard NS2 package, NetPerf JNGI modules have to be included in NS2. One of the major difficulties in creating a new application in NS2 is to define the way that the user's data is transmitted on application-level. NS2 provides a structure to pass data among applications and to pass data from application to transport agents as shown in figure 4.

3.2.1 Designing and implementing the JNGI modules

The JNGI modules are implemented as child classes of "Application". The matching OTcl hierarchy name is "Application/JNGI". The task dispatcher, worker and job submitter behavior of the application is implemented respectively in the JNGITaskDisp, JNGIworker and JNGIJsub classes. These classes are implemented as child classes of "Application/JNGI". The major additions and modifications are explained below.

User data transmissions over TCP are emulated in the same way as TcpApp. The sender uses a buffer for application data, then the bytes received by the receiver are counted. When the receiver has got all the bytes of the current data transmission, it receives the data directly from the sender. Overhead time is added to the TCP communication time according to [4] to reflect the JXTA pipe communication slowdown.

The worker model The worker uses a timer so that the next REGISTER message transmission is scheduled for the task dispatcher to ask for a new task. If there is no task at the task dispatcher, the worker will receive a SLEEP message; it has to wait a few seconds

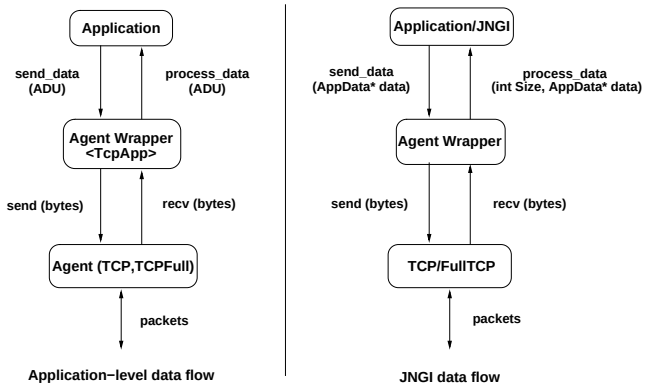

Figure 4. Structure of a JNGI agent.

before sending a new REGISTER message. Once tasks are available at the task dispatcher level, another timer is used by the JNGI worker to simulate the time that a JNGI worker needs to perform a distributed task. When this timer expires, a RESULT message is sent to the task dispatcher which has distributed the task. The worker will ask a new task to perform. A new worker has to send a GETROLE message to the task dispatcher to join the group.

The task dispatcher model Initially, the task dispatcher is in the Wait state; it passively waits for the messages from the workers in its group or from the job submitter. The job name, the number of tasks distributed and the number of finished tasks of each task dispatcher nodes are stocked in a temporary file. The task dispatcher does not keep track of which workers are performing which tasks or which job submitters are submitting which jobs.

The job submitter model The job submitter does not only send the job to the task dispatcher, but also the number of individual tasks to be distributed to the workers' group. A SUBMITCODE message is sent to the task dispatcher for this purpose. Once the task dispatcher has received all the jobs, the job submitter sends a SUBMITJOBID message periodically to know if the job is finished. Once a job is completed, the results are sent back to the job submitter, and a REMOVEJOBID message is sent by the latter in order to remove the job from the task dispatcher's stock.

4. Case study

P2PPerf can be used either to help tune a P2P computing framework or develop new features in a P2P

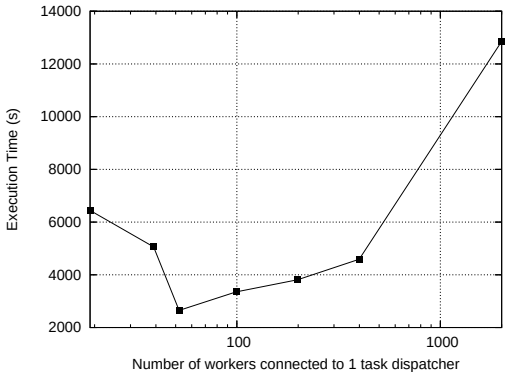

Figure 5. Prime number execution with varying task dispatcher number.

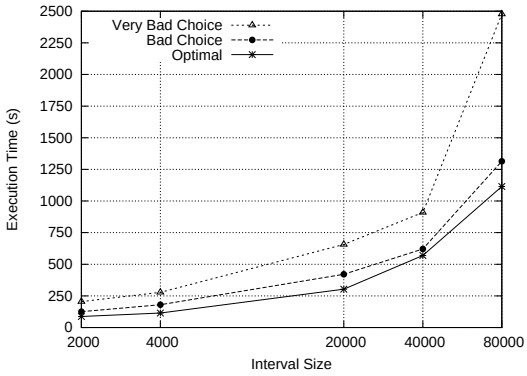

Figure 6. Task dispatcher bottleneck.

computing framework. The following tests represent these three possibilities.

Our tests have been conducted with a standard application included in the JNGI package. This application finds all the prime numbers in a given interval by the well-known method of the Eratosthen's Sieve. The task dispatcher gives each worker asking for a job a number to test. The first worker that asks for a job receives the first number of the interval, the second worker the second number, etc.

4.1. Tuning and testing JNGI

4.1.1 Ratio between worker and task dispatcher

The JNGI task dispatchers are special peers that are responsible for peers scheduling and farming out tasks to the worker peers. Task dispatchers are a possible bottleneck in JNGI. Indeed, if a task dispatcher has too many peers to manage, worker peers will wait for him and will remain idle instead of working.

The prime number application has been used with the following parameters. The searching interval is $[2 \times 10^7, 2.002 \times 10^7]$. The average computing time for calculating a prime number included in this interval is 10 seconds, running on a Pentium 4 2.8 GHz.

The network topology was generated by Brite with 10000 nodes. A peer can only be a leaf while the other nodes represent the network components of internet. 2000 peers join the JNGI architecture either as task dispatchers or as workers. The number of task dispatchers increase from 1 to 1000. Therefore, at the beginning of the simulation, one task dispatcher manages 1999 workers whereas at the end, one task dispatcher manages only one worker.

Figure 5 presents the results of the simulations. The curve shows that the optimal ratio between the number of task dispatchers and the number of workers is around one task dispatcher per 50 workers. It can be seen that if this ratio is not respected, the performance of JNGI is greatly altered. Indeed, for a ratio of 1/39, the execution is 71% longer because the task dispatchers are overloaded.

4.1.2 Choosing the right task dispatcher

The second experiment determine the importance of electing a new task dispatcher in the JNGI framework (see 2.1). In figure 6, the impact of choosing a worker with poor network characteristics as a task dispatcher is shown. For this experiment, the brite topology is the same as in the first simulation with a fixed ratio of 50 workers for each of the 50 task dispatchers. The searching interval varies from $[4.5 \times 10^7, 4.50002 \times 10^7]$ to $[4.5 \times 10^7, 4.5008 \times 10^7]$. The average computing time for calculating a prime number included in this interval is 25 seconds, running on a Pentium 4 2.8 GHz. The plain curve shows the execution time of the optimal solution i.e. choosing a worker with high bandwidth and low latency to become a task dispatcher. The dotted dashed curve shows the execution time of the same experiment using a task dispatcher with 50% less bandwidth and 50% more latency than the optimal task dispatcher. The second dashed curve (with a triangle) shows the execution time using a task dispatcher linked to the network with a 56Kb modem. It is important to notice that if the task dispatcher is not well chosen, the performance of the framework decreases by about 46% in the first case and about 130% in the second case. So, it is very important to choose task dispatchers peers carefully.

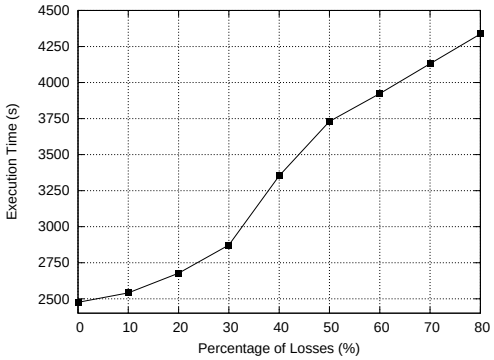

Figure 7. Fault-tolerance of the framework.

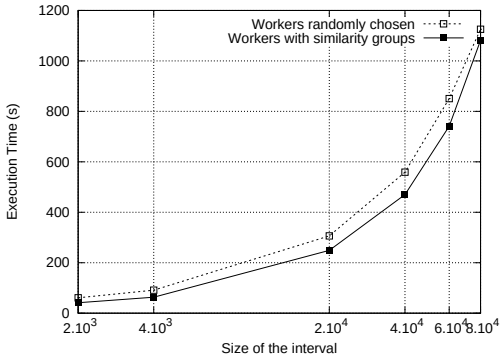

Figure 8. Prime numbers execution with or without similarity groups.

4.1.3 Fault-tolerance

Figure 7 presents the results of experimentation to test the fault-tolerance of the framework. The network structure and the application are the same as in 4.1.1 with 40 task dispatchers. At a random time, a random worker is removed from the framework until threshold is reached. There are 3 parts in the graph. The first one between 0% and 30% shows an increase of 16% in the execution time, the second one between 30% and 50% shows an increase of 29% and the last one shows an increase of 16% between 50% and 80%. In the 3 parts, the curves are almost linear. It can be seen that the architecture can easily bear losses of workers, for example in a very bad case where 80% of the workers disconnect, the execution time is less than twice the execution time without disconnection.

4.2. Optimizing the performance of an application

The prime number application has performance characteristics which depend on the size of the numbers included in the interval. For example, determining a prime number is faster for a number as 10,007 than for 20,000,000,003. The consequence is the following: if the numbers can be calculated quickly, communication times will prevail over computation times and vice versa.

A task dispatcher can then choose the most adapted workers for the computation. That is to say, either workers with good CPU characteristics, (if the numbers are big) or workers with good network characteristics (if the numbers are small). This approach to group nodes according to their attributes is called similarity groups and it has been proposed in previous works [11, 10] but we were not able to test them with large numbers of peers.

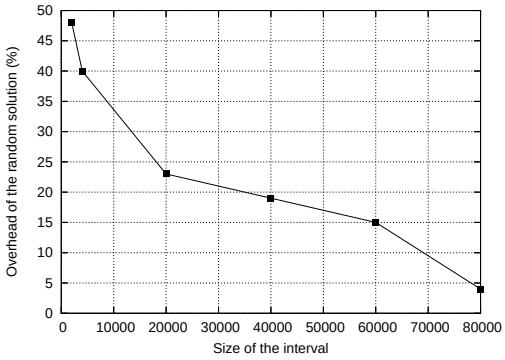

Figure 9. Impact of similarity groups: overhead of the random solution.

Figure 8 presents the result of the simulations when the 2000 workers depending on 40 task dispatchers are chosen randomly (dashed curve) or chosen according to their network characterisitics (plain curve). The parameters of the simulations are the same as in 4.1.2. As the size of the interval is growing, the execution time becomes higher. Grouping nodes according to their network caracteristics is a good choice as can be seen on figure 9. The difference between the random solution and the similarity groups solution is 48% when the size of the interval is 2000. But, this difference decreases by 4% when the size of the interval reaches 80000. Indeed, when the size of the interval is short, communication times prevail over computation times. When the size of the interval reaches 80000, the similarity groups have to be changed: workers must be grouped according to their CPU characteristics and not anymore to their network characteristics for this application. This test has shown that the similarity groups must be dynamic.

NS2 is well adapted to all these simulations because huge traces are not generated. The only recorded event of the simulation is the execution time of the application. A simulation lasts approximately from half an hour to two hours on a Pentium 4 2.8GHz according to the parameters whereas a real execution of the tests lasts from 50 minutes to 3.6 hours.

5. Conclusion

Tests have shown that P2PPerf can be successfully used to tune a P2P computing framework. The performance of JNGI can be greatly enhanced when choosing the right number of workers and choosing workers with good network characteristics to become task dispatcher. The JNGI framework has proved to be fault-tolerant and similarity groups have also proved to be efficient even with great numbers of peers.

Some important features have to be added to P2PPerf. First, measuring the impact of the dynamicity when adding new peers during a computation. Second, including not only the cost of JXTA in the communication times, but also in the management of the peers. This will be done in collaboration with the University of Darmstadt [8].

Some new features have also to be studied for JNGI. The first step is to integrate the similarity groups in P2PPerf completely in order to be able to validate our approach. Besides, the tests have shown that similarity groups must be dynamical in order to be efficient. The second one is to test different strategies of direct communications between peers when applications have data dependency problems.

References

[1] The network simulator - *NS2*. http://www.isi.edu/nsnam/ns/.

[2] D. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. Seti@home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002.

[3] N. Andrade, W. Cirne, F. Brasileiro, and P. Roisenberg. Ourgrid: An approach to easily assemble grids with equitable ressource sharing. In *Proc. of the 9th Workshop of on Job Scheduling Strategies For Parallel Processing*, June 2003.

[4] G. Antoniu, P. Hatcher, M. Jan, and D. A. Noblet. Performance evaluation of jxta communication layers. In *Workshop on Global and Peer-to-Peer Computing (GP2PC 2005)*, May 2005.

[5] U. Assmann and A. Ludwig. Introducing connections into classes with static metaprogramming. In *Coordination 1999*, volume 1594 of *LNCS*. Springer, Apr 1999.

[6] J. Bourgeois and F. Spies. Performance prediction of an NAS benchmark program with ChronosMix environment. In *6th Int. Euro-Par Conference (EuroPar'2000)*, pages 208–216, Munich, Allemagne, Sept. 2000.

[7] P. Clauss and V. Loechner. Parametric analysis of polyhedral iteration space. *Journal of VLSI Signal Processing, Kluwer Academic Pub.*, 19(2):179–194, July 1998.

[8] V. Darlagiannis, A. Mauthe, and R. Steinmetz. Overlay design mechanisms for heterogeneous, large scale, dynamic p2p systems. *Journal of Network and Systems Management, Special Issue on Distributed Management*, 12(3):371–395, September 2004.

[9] E. Ehrhart. Polynômes arithmétiques et méthode des polyèdres en combinatoire. *International Series of Numerical Mathematics*, 35, 1977.

[10] J.-B. Ernst-Desmulier, J. Bourgeois, F. Spies, and J. Verbeke. Using similarity groups to increase performance of P2P computing. In *10th Int. Euro-Par Conference (Europar'04)*, volume 3149 of *LNCS*, pages 1056–1059, Pisa, Italy, Aug. 2004. Springer.

[11] J.-B. Ernst-Desmulier, J. Bourgeois, F. Spies, and J. Verbeke. Adding new features in a Peer-to-Peer distributed computing framework. In *13th Euromicro Conf. on Parallel Distributed and Network Based Processing (PDP'05)*, pages 34–41, Lugano, Switzerland, Feb. 2005. IEEE computer society press.

[12] T. Fahringer and B. Scholz. Symbolic evaluation for parallelizing compilers. In *Proc. of the 11th ACM International Conference on Supercompting*, Vienna, Austria, July 1997.

[13] G. Fedak, C. Germain, V. Néri, and F. Cappello. Xtremweb : A generic global computing system. In *CCGRID2001, Workshop on Global Computing on Personal Devices*. IEEE Press, May 2001.

[14] A. Medina, A. Lakhina, I. Matta, and J. Byers. Brite: An approach to universal topology generation. In *International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS)*, August 2001. Cincinnati, Ohio.

[15] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proceedings of HotNets–I*, Princeton, New Jersey, October 2002.

[16] R. H. Saavedra-Barrera, A. J. Smith, and E. Miya. Machine characterization based on an abstract high-level language machine. *IEEE Trans. Comput.*, 38(12):1659–1679, December 1989.

[17] B. Traversat, A. Arora, M. Abdelaziz, M. Duigou, C. Haywood, J. Hugly, E. Pouyoul, and B. Yeager. *Project JXTA 2.0 Super-Peer Virtual Network*. Sun Microsystems Inc., May 2003.

[18] J. Verbeke, N. Nadgir, G. Ruetsch, and I. Sharapov. Framework for peer-to-peer distributed computing in a heterogeneous, decentralized environment. In *Proc. of GRID 2002, Baltimore*, Sun Microsystems, Inc.,Palo Alto, CA 94303, USA, January 2002.