Scalable Architecture for Allocation of Idle CPUs in a P2P Network

Javier Celaya and Unai Arronategui

University of Zaragoza,
Department of Computer Science and Systems Engineering
C/ María de Luna 1, Ed. Ada Byron, 50018 Zaragoza, Spain
{jcelaya, unai}@unizar.es

Abstract. In this paper we present a scalable, distributed architecture that allocates idle CPUs for task execution, where any node may request the execution of a group of tasks by other ones. A fast, scalable discovery protocol is an essential component. Also, up to date information about free nodes is efficiently managed in each node by an availability protocol. Both protocols exploit a tree-based peer-to-peer network that adds fault-tolerant capabilities. Results from experiments and simulation tests, using a simple allocation method, show discovery and allocation costs scaling logarithmically with the number of nodes, even with low communication overhead and little, bounded state in each node.

1 Introduction

One of the most recent solutions in large scale computing, primarily oriented to embarrassingly parallel and computing-intensive problems, is the use of thousands or even millions of unreliable personal computers connected to the Internet. Projects like SETI@Home [1] and distributed.net [2] established a milestone in the field of Distributed Computing by harnessing the idle cycles of personal computers donated by volunteers to solve problems that could be split into independent parts. Each computer executes the processes associated to one or more parts, receiving the input data and returning the output results. Much work around this idea has been developed covering efficiency, throughput, fairness and security aspects, but the general structure still maintains some intrinsic disadvantages: only one entity in the network generates the workload, and the rest consume it, leading to centralized management and scheduling that negatively affect the scalability and fault-tolerance of the system.

To solve this problem, we propose an architecture where any participant of the network may need idle cycles to complete its tasks. When a node does not have enough computing power to finish its work in a certain amount of time, it may divide it into n independent tasks and query the network for n more idle machines that can each execute one of them. This is not a new idea, but there is little work covering this approximation. Projects with best results have been those who use peer-to-peer (P2P) networks and distributed protocols; with them, problems concerning scalability and fault-tolerance are drastically reduced. The

use of an unstructured P2P network is simple and is based on already working ideas like Gnutella or Freenet, but does not allow the application of constraints to the idle CPU search.

For this reason we present a peer-to-peer network based on a balanced tree structure that finds the nearest free CPUs to the one that is demanding the execution of a number of tasks. At any time, any node of the network may request the execution of n tasks; this request is routed by neighbour nodes to those available ones that are closer to the originating client with a fast discovery protocol. The information about existing free nodes is dynamically managed by an availability protocol. These functionalities are obtained with little state in nodes, and low communication and CPU overhead. A simple allocation policy has been designed and implemented to evaluate the architecture behavior.

This paper takes some steps into a complete distributed computing solution, thus we will impose some restrictions to the environment: We assume that nodes execute batch tasks that do not communicate between them, so we won't be addressing the issues that arise from having dependencies. Also, we will suppose that there is low churn, that is, joins and leavings are not frequent. And finally, we will only consider a weak concept of fairness in the allocation of free nodes.

The rest of the article will be structured as follows: In Sect. 2 we will expose an overview of the system architecture and its behavior, and in Sects. 3 and 4 we will detail the protocols that allow the fast and scalable discovery of free nodes. Following, we will briefly present the hierarchical overlay topology and its management in Sect. 5. Finally, in Sect. 6 we will show the experimental results and in Sects. 7 and 8 we will explain what other work has been presented concerning distributed computing in peer-to-peer networks and the conclusions of this investigation.

2 System Architecture

The system has a three layer architecture:

- The first one defines the *connectivity protocol* that maintains the overlay links in the network. It conforms a tree-based network overlay, derived from the B-Tree [3], thus it is a balanced tree where each node can have between m and $2m$ children and the height is always a logarithm of the number of nodes N. The protocol states how nodes join and leave the network, how the tree is kept balanced, and how node failures are dealt with to rebuild the structure.
- The second layer is described by the *availability protocol*, that distributes information relative to the number of free nodes and computing power each time it changes. Every node of the network stores the global state of the branch that hangs below it, and communicates updates to its parent so it can recompute the state of its own branch. This protocol uses a number of techniques that prevent the upper levels of the tree from being flooded with update notifications, while maintaining the information accurate enough to

maximize the network use. Also, the conservative approach of notification updates yields to a more stable behavior of this protocol.

- Finally, the *discovery protocol* uses the information stored by each branch to route free nodes requests up and down the tree. It tries to find those free nodes that meet a trade-off between proximity to the client and computing power by distributing the requests among the appropriate branches at each level. Therefore the search is performed in a number of network hops that depends only on the height of the tree and, consequently, on the logarithm of the number of nodes of the network.

3 Discovery of Free CPUs

As it has been said, when a node has a number of tasks to be done, it requests the network to find that number of idle machines. This service is accomplished with the discovery protocol, that works as follows. By applying heuristic rules, it will try to allocate the fastest and nearest free nodes, so that tasks execution is efficiently done. To find them, the tree structure is exploited. Each inner node stores information about its descendants; not exhaustive information, but more general information about the branch as a whole. To be concrete, it knows the number of free CPUs of the branch, the maximum computing power and the minimum number of hops to a free node. This way, the management of this information becomes scalable as it does not depend directly on the number of nodes. How it is exactly managed will be explained in Sect. 4.

A node that receives a message with n pending tasks will first check if itself is ready to execute a new one. If so, it takes one of the tasks from the message. Then it distributes the remaining tasks between its child nodes according to the number of free nodes each branch has, calculated with the availability protocol, giving priority to the branches having more computing power or less hops to a free node. If it is not enough with the children branches to cope with all the tasks, then a new message with the last tasks will be sent to the father so that it can reach more distant branches. When the message arrives at the root of the tree and it cannot be sent to another branch, it is returned to the originating node meaning that there are no free nodes left in the network.

The worst case would be that of a leaf node that needs to allocate every node of the net. The request would have to go up to the root and then down to the rest of the tree; that is the longest path a request would traverse. As discovery of idle nodes is done concurrently in every branch, that would be the same as reaching one idle leaf node in the opposite side of the tree. This is done in $O(\log_m N)$ hops, being m the minimum number of children per node in the balanced tree and N the number of nodes in the network, thus making the discovery protocol highly scalable.

This is a best effort network. That is, the intermediate nodes make its best to route the message to the most suitable free nodes, but the reception is not guaranteed. In fact, the failure of nodes is frequent in a peer-to-peer network. For this reason, both the originating node and the allocated ones must avoid

problems in the discovery phase and when sending the actual work to execute using timeout mechanisms, acknowledge messages and retransmissions.

4 CPU Availability Management

The information each node stores about its branch must be communicated to its parent so that it can efficiently route requests to the idle nodes it has under itself. Therefore, each node not only has information about its branch, but also about each of its sub-branches. The way this information propagates is critical, because it must be kept up to date without flooding the network with notification messages, specially near the root where there are less nodes per level. This propagation is performed by the availability protocol. Basically, when a node receives a notification of change from one of its child nodes, it must decide if it has to inform its parent, too. With the maximum computing power and the minimum number of hops to a free node, the process is simple. The inner node has to calculate the new maximum and minimum values, respectively, between its child nodes and itself, and if it changes, route the new information to its parent immediately. Note that when a notification goes up one level there is less probability of being the maximum or minimum of the greater number of nodes of that branch, so the traffic is self-limited and is unlikely that it reaches the top levels.

The problems arise with the number of free nodes, because when a node gets ready (busy), the number of free nodes of each of its ancestors increases (decreases) by one. If the notification were sent with every change, the root would get informed of all of them, what leads to an unacceptably high traffic in the top levels. For this reason, we have designed a technique that delays the notification of the number of free nodes at each level of the tree, reducing the traffic routed up to the root. The basis of this method lies on sending a notification when the change is "important" enough. Actually, this means that the most significant bit set to one changes; that is, the number of free nodes crosses a boundary of power of two. For example, a notification would be sent if this value changes from 7 to 8 (111b and 1000b in binary) or from 32 to 31 (100000b and 11111b), but not when it changes from 23 to 24 (10111b and 11000b). Although this yields to a precision lack, there are three main reasons for using this technique:

1. Trying to provide optimality based on exact information is senseless when we are dealing with millions of nodes that are continually and concurrently changing state.
2. The traffic of notifications in the top levels is reduced because as a notification goes up the tree it is less probable of being routed to the next level. This depends also on the number of free nodes, as a high number has also less probabilities of being routed.
3. When the number of free nodes is low, the precision of this value is better. This is most relevant as the nodes of the network get busy, because they are correctly well-spent when there last only a few free nodes.

There are two policies deciding what availability value to take as reference for a branch when a child node notifies a change to its father: optimistic and conservative. An optimistic policy would use the same value sent by the child. On one hand, it has the advantage of having better precision in the information each node stores about its branch, but on the other hand the real number of free nodes of a branch could decrease and be less than the number its father is using, making top level nodes route requests to branches that cannot cope with them. A conservative policy would store a lower value, for example the higher power of two less than or equal to the notified value. With such a policy, the system has a better behavior against situations when there last very few free nodes, as it delays too big requests, although it does not make the most of the network.

We have decided to adopt a conservative approach. It forces a stabilization mechanism in the value of free CPUs each node contains, providing a gradual convergence in the occupation of the network.

5 B-Tree Based Topology

The overlay network topology is a hierarchy where every node of the network is mapped to a node of the tree. In our approximation we use a B-Tree [3] variant; it maintains the balance in every join and departure and allows more than two child nodes, thus reducing the tree height. But the main objective of the tree is grouping nearby nodes in the same branch. However, the concept of locality usually depends on many variables, so it is actually an approximation. We have decided to use the simple yet effective way of organizing the nodes in the tree by their physical address, their IP address actually. Based on the subnet partitioning of the IP address space and the studies on geographic locality of IP addresses [4], this method allows a fast and easy decision of where to insert a node in the tree when it joins the network, while maintaining good metrics between nodes of the same branch, specially near the leaves.

Our tree has some differences with the original B-Tree model. Each node holds only one value (its IP address) and forms part of a group of siblings; therefore, every node of the network participates in the management of the tree. There exist a constant m so that every node not being the root of the hierarchy always has between m and $2m$ siblings. If these limits are exceeded, the tree must be rebalanced by splitting or joining groups. Also, every node that is not a leaf has a pair of values that represent the interval of addresses of its descendants, including itself. These intervals are used to route messages along the tree, mainly in the operations of insertion and deletion of nodes.

Concerning *fault-tolerance*, every node knows the address of the k predecessors and k successors at the same level (they can be "brothers" or "cousins"). When a node fails, the tree structure can be repaired by its neighbours using these references, because they allow the communication between a node and the brother of its dead father. The value of k is an trade-off between fault-tolerance and an overload in the management of the tree.

5.1 Joining and Leaving the Network

The connectivity protocol consists in two operations that affect the structure of the network: joining and leaving. Joining is usually easier: when a node requests an insertion, the request message is routed up the hierarchy looking for a node whose interval contains the address of the new node, and then it goes down until it reaches the node with the nearest address to the new node's address. Finally they become brothers and the new node updates its references to its neighbours. Then the father node is notified, and it may request a group split to re-balance the tree if the number of its child nodes is greater than $2m$. When a node is added to the group of the root and it already has $2k$ nodes, a new root node will be created.

By leaving the network we assume, usually, a voluntary action, so the leaving node will supply its neighbours with the necessary information to maintain the network connectivity. First of all, a leaving node must check if it has any child. If so, it looks for a leaf node that becomes the new father of all of them, similarly to the creation of a new root node. Once done, or if it had no child nodes, it notifies its siblings and its father that it is going to leave and then they update their reference lists. Similarly to the joining, when the father node is notified of the node leaving, it must check if the number of child nodes is less than m. In that case, it will ask its predecessor or successor to send it child nodes, or to join into only one branch. One special case is when the father is the only one node in the root group. Then it will check if it has less than $2k$ child nodes, and if it has so, it will insert itself at the leafs, leaving its children as the new root group. In some cases nodes can fail and leave the network without notification. In this case, the fault-tolerance strategy presented above is applied.

6 Experimental Results

This architecture has been implemented as a simulated system with the OMNeT++ simulation framework. The allocation policy used in the tests has been a simple one, where as soon as nodes are discovered they are allocated.

Tests have been done aimed to measure free nodes discovery time, control messages traffic and CPU load. Every test has been issued with variations in the number of nodes, N, and the B-Tree parameter, m, to study the impact of the size and structure of the network in the performance of the protocols. The simulations have been performed with up to 50000 nodes and values of m from 6 to 10. Variations on the duration of the tasks and the size of the data have also been applied to recreate more realistic situations. There are three constants, though: the latency of the network connections has been established to 200 ms, the mean continental value for Internet, to simulate a very wide area network; 1 Mbps has been taken for the bandwidth, a conservative value for a home Internet connection; and the mean computing power of the nodes has been set to 2000 MIPS.

Time tests show that both the number of nodes and the number of child nodes per parent affect the discovery of free nodes. Just as expected, the last

free node of the n requested is reached in $O(\log_m n)$ hops. For this reason, a network with a higher value of m performs better, while an increasing value of n is hardly appreciated. The results of the free nodes discovery time tests can be seen in Fig. 1 as a logarithmic growth. We can extrapolate the results to higher values of n. For example, we calculate that, for the test network, requesting the execution of 100,000 tasks would discover 100,000 free nodes in 2 seconds, 1,000,000 in 2.4 seconds, 10,000,000 in 2.8 seconds, and so on.

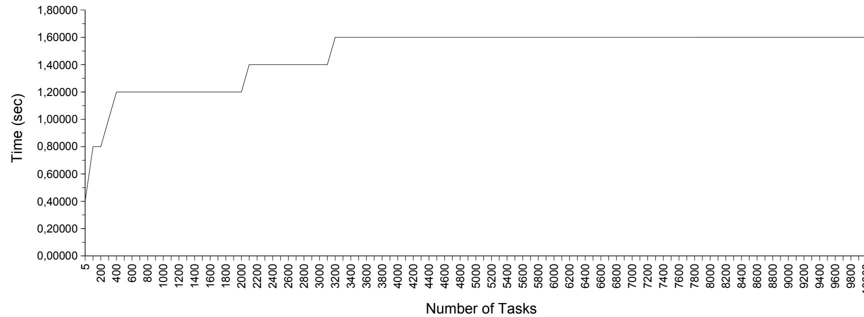

Fig. 1. Discovery time for as many free nodes as requested tasks. The test network has 50000 nodes and $m = 10$. One network hop is 200 ms.

Control traffic (traffic of non-data messages) and CPU load tests have been done under two situations: participants have a normal and high activity. Normal activity means that there are frequent requests from randomly chosen nodes, but the network does not get completely busy. On the other hand, under high activity, every node is busy and continuously receiving new requests, so we expect this to be the worst case. Traffic has been measured in bytes per second. CPU load is more difficult to measure in a simulation, but as every message is managed in constant time we have decided to express it in terms of messages per second. Tables 1 and 2 show the results of the normal and high-activity behavior. They present the value of m, the tree height and the mean and maximum values of CPU load and control traffic for the root and leaves of a network of 50000 nodes.

While the discovery protocol was positively affected by the value of m, the overall system load suffers when the tree is lower, thus a trade-off is needed between them. Looking at each network variant it can be seen that, by using the availability protocol, under normal behavior both control traffic and CPU load is heavier at the leaves than at the root nodes. Also, control traffic hardly reaches 1KBps, what represents less than 1% of the total bandwidth. However, under high activity rate the root suffers waves of very high CPU load and control traffic.

Results are promising. As we can see in Tables 1 and 2, control overhead is very low. Under normal activity, the control traffic is only 1485 Bps and the CPU

Table 1. CPU load and control traffic under normal activity. The net has 50000 nodes, with $m = 10$ and a bandwidth of 1Mbps.

		Root				Leaves			
Tree		Load (msg/s)		Traffic (Bps)		Load (msg/s)		Traffic (Bps)	
m	height	mean	max	mean	max	mean	max	mean	max
4	7	0.08	2.87	24.10	519.98	3.56	4.21	1235.89	1343.51
6	6	0.09	5.64	24.98	1005.69	3.71	4.33	1293.18	1405.28
8	5	0.09	5.64	25.28	1005.35	3.85	4.56	1330.41	1436.15
10	5	0.09	5.62	25.43	999.48	4.12	5.77	1435.08	1485.60

Table 2. CPU load and control traffic under high activity. The net has also 50000 nodes, with $m = 10$ and a bandwidth of 1Mbps.

		Root				Leaves			
Tree		Load (msg/s)		Traffic (Bps)		Load (msg/s)		Traffic (Bps)	
m	height	mean	max	mean	max	mean	max	mean	max
4	7	0.13	28.20	38.19	4980.56	39.68	41.62	14359.74	16031.69
6	6	0.13	28.46	39.53	5021.05	44.20	50.55	15198.24	16205.83
8	5	0.14	27.65	39.75	4526.82	54.15	59.85	17417.23	19256.67
10	5	0.14	28.44	40.05	5018.49	63.72	65.29	19566.90	21947.28

load only reaches 5.77 messages per second, in the worst case. And under heavy activity, the control traffic is 21947 Bps and the CPU load is 65.29 messages per second.

7 Related Work

As it has been pointed out in the introduction, the main approximation until now to a highly scalable distributed computing environment is one entity harnessing the idle cycles of personal computers donated by volunteers, as in SETI@Home project, the BOINC generic framework [5] and distributed.net. Those projects use the traditional client/server paradigm to schedule tasks and return results, what soon leads to scalability problems. For that reason, more elaborated network structures and distributed algorithms have been adopted. One example is Javelin++ [6], which extends the concepts of Javelin [7] replacing the broker that scheduled the tasks with a network of brokers. Recently, more strict peer-to-peer networks have been used to select the nodes which would execute the tasks. BOINC and similar projects adopt an application-driven perspective, in which the existence of an element that is generating all the workload determines the structure of the network and the management algorithms. Following a more general view, another family of projects, in which this paper is included, have proposed an architecture where every participant can generate the workload, which is better suited for this peer-to-peer philosophy, as every node is equal to the other ones.

CompuP2P [8] is one of the first works to use a decentralized peer-to-peer network to manage processor cycles as a shared resource. It arranges all the nodes in a Chord [9] ring and organizes them into 'compute markets', where idle cycles are traded with. However, it presents a scalability problem because it has no mechanism to limit the number of nodes in a market or to balance load between markets. G2-P2P [10] uses an object-oriented approach. It uses Pastry [11] to create a Distributed Hash Table (DHT) where computation objects are stored. Each object is assigned a random ID and stored in the Pastry node closer to that number. Using an uniform hashing function they claim to achieve a good load-balancing property, but there is no other criterion to select the most appropriate free node. In [12] the Pastry DHT is also used, but exploiting its locality awareness to discover near idle nodes. It then announces availability with controlled message floodings, what leads to inefficiency as a node surrounded by busy neighbours won't find a free node which is more distant than the maximum number of hops that a request is allowed to make. On the other hand, in [13] it is proposed the use of an unstructured overlay network, as it is easier to manage, and traverse it with random walks. Even though, that is also inefficient because there is no way of knowing if the next node of the walk is free or not.

For the overlay topology, other authors have proposed the use of a virtual tree on top of a DHT, where each node store only part of a tree index, mainly oriented to range queries. Examples of this are P-Tree [14], P-Grid [15] and VBI-Tree [16]. However, they rely on a uniform distribution of the shared resource; for example, using a uniform hashing function for the DHT. For that reason, BATON [17] uses a balanced binary tree where each node of the network maintains one node of the tree. This type of organization is better suited for a non-uniform resource distribution because the tree gets balanced automatically when the insertions or deletions occur within the same zone. We adopt these ideas but with more than two children per node.

8 Conclusions and Future Work

In this paper, we have presented a network architecture that discovers the presence of idle machines with a scalable ($O(log_m N)$) and fast (1.8 seconds for 10000 requested tasks) method. It organizes the nodes in a balanced tree structure to efficiently distribute information about free nodes in a per-branch basis, that is eventually used to route the request from a client to the appropriate idle CPUs. The connectivity protocol, discovery protocol and availability protocol are all three designed in a totally distributed way, that provide high scalability and fault-tolerance. Moreover, the experimental simulation results show low overhead in the control traffic and CPU load.

We envision to validate these results with a real prototype to be implemented over the PlanetLab testbed. We believe this can be a valuable step to develop system support for high performance computing applications.

References

1. Anderson, D.P., Cobb, J., Korpela, E., Lebofsky, M., Werthimer, D.: Seti@home: an experiment in public-resource computing. Commun. ACM **45**(11) (2002) 56–61
2. Distributed.net: http://www.distributed.net (2000)
3. Bayer, R., McCreight, E.M.: Organization and maintenance of large ordered indexes. In: Record of the 1970 ACM SIGFIDET Workshop on Data Description and Access, November 15-16, 1970, Rice University, Houston, Texas, USA (Second Edition with an Appendix), ACM (1970) 107–141
4. Freedman, M., Vutukuru, M., Feamster, N., Balakrishnan, H.: Geographic Locality of IP Prefixes. In: Internet Measurement Conference (IMC) 2005, Berkeley, CA (2005)
5. Anderson, D.P.: Boinc: A system for public-resource computing and storage. In: GRID. (2004) 4–10
6. Neary, M.O., Brydon, S.P., Kmiec, P., Rollins, S., Cappello, P.: Javelin++: scalability issues in global computing. Concurrency: Practice and Experience **12**(8) (2000) 727–753
7. Christiansen, B.O., Cappello, P.R., Ionescu, M.F., Neary, M.O., Schauser, K.E., Wu, D.: Javelin: Internet-based parallel computing using java. Concurrency - Practice and Experience **9**(11) (1997) 1139–1160
8. Gupta, R., Somani, A.K.: Compup2p: An architecture for sharing of computing resources in peer-to-peer networks with selfish nodes. In: Online Proceedings of Second Workshop on the Economics of Peer-to-Peer Systems, Harvard University (2004)
9. Stoica, I., Morris, R., Liben-Nowell, D., Karger, D.R., Kaashoek, M.F., Dabek, F., Balakrishnan, H.: Chord: a scalable peer-to-peer lookup protocol for internet applications. IEEE/ACM Trans. Netw. **11**(1) (2003) 17–32
10. Mason, R., Kelly, W.: G2-p2p: A fully decentralised fault-tolerant cycle-stealing framework. In: ACSW Frontiers. (2005) 33–39
11. Rowstron, A.I.T., Druschel, P.: Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: Middleware. (2001) 329–350
12. Butt, A.R., Fang, X., Hu, Y.C., Midkiff, S.P.: Java, peer-to-peer, and accountability: Building blocks for distributed cycle sharing. In: Virtual Machine Research and Technology Symposium. (2004) 163–176
13. Awan, A., Ferreira, R.A., Jagannathan, S., Grama, A.: Unstructured peer-to-peer networks for sharing processor cycles. Journal Parallel Computing (PARCO) **32**(2) (2006) 115–135
14. Crainiceanu, A., Linga, P., Gehrke, J., Shanmugasundaram, J.: Querying peer-to-peer networks using p-trees. In: WebDB '04: Proceedings of the 7th International Workshop on the Web and Databases, New York, NY, USA, ACM Press (2004) 25–30
15. Aberer, K.: P-grid: A self-organizing access structure for p2p information systems. In: CooplS '01: Proceedings of the 9th International Conference on Cooperative Information Systems, London, UK, Springer-Verlag (2001) 179–194
16. Jagadish, H.V., Ooi, B., Vu, Q., Zhang, R., Zhou, A.: Vbi-tree: A peer-to-peer framework for supporting multi-dimensional indexing schemes. In: 22nd IEEE International Conference on Data Engineering (ICDE), 2006 (to appear). (2006)
17. Jagadish, H.V., Ooi, B.C., Vu, Q.H.: Baton: A balanced tree structure for peer-to-peer networks. In: VLDB. (2005) 661–672