

Fully Distributed Active and Passive Task Management for Grid Computing

Cyril Rabat Alain Bui

Olivier Flauzac

Université de Reims Champagne-Ardenne,
BP 1039, F-51687 Reims Cedex 2, France

{alain.bui,olivier.flauzac,cyril.rabat}@univ-reims.fr

Abstract

The task management is a key point in grid applications and can highly influence their efficiency. There are many solutions that we can classify according to their centralization degree: fully centralized, semi-distributed and fully distributed. A fully distributed approach leaves the nodes in charge of the scheduling: the schedule is performed thanks to the local view of the system. Such a local treatment reduces the communication cost but it must not impact on the computational power devoted to applications. This kind of solution seems to be more scalable and flexible than a centralized one, especially in a highly volatile environment like peer-to-peer networks.

In this article, we propose two fully distributed solutions for the task management based on random walks. We choose to maximize the computational power of nodes by reducing the maintenance cost of an underlying structure (a spanning structure). So, it reduces the number of control message exchanges that is a critical point in networks with a low bandwidth or in which the energy of nodes must be saved. We analyze two methods called passive and active and we present some simulation results.

1 Introduction

The grid computing aims to share computational power of interconnected sites. More generally, the authors of [12] describe the concept of grids that allows to share some different resources like storage capacities, information or scientific applications. The challenge for a grid computing middleware is to share these resources transparently for the application designers. The peer-to-peer networks are often viewed as a way to share data. But the number of nodes in such networks is very important and the available computational power can be used for global computations. The counterpart is the high volatility of nodes that involves extra-mechanisms.

In computational grids, the task management is a key point. When jobs are submitted, they have to be assigned to grid nodes following a scheduling scheme that

takes into account the global load balancing but also the local capacities of nodes (libraries needed, memory or storage requirements). Designing a task management must be achieved according to the middleware topology. We can classify these topologies based on their centralization degree: fully centralized, semi-distributed and fully distributed. For a centralized solution, it is easier to have efficient schedules because the states of resources are gathered locally like *Nimrod* ([4]). The authors of *Condor-G* [13] propose a solution that uses existing grid protocols of the *Globus* toolkit [11]. But the *Condor-G agent* must be deployed on a robust server and this condition is difficult to satisfy in a peer-to-peer network. Moreover, these solutions induce a bottleneck on this single server that can crash. In addition to security policies and network heterogeneity, it seems to be less flexible and scalable than a distributed method. So, in *CONFITT* [10], the authors deploy a fully distributed solution. No assumption is made on the node lifetime and no node is differentiated. But all nodes are set up into a ring that has to be maintained. It induces several control message exchanges each time a node fails.

Here we propose two solutions for the task management based on a peer-to-peer approach. The task parameters are distributed to all grid nodes and each one selects a task at random it can compute depending on its operating system, installed softwares or libraries, processor or computer architecture. The node properties are only known locally and no registration to specific nodes is needed. Whereas an underlying virtual structure is maintained in middleware like *CONFITT*, we propose to maximize the computational resources of nodes by using a random token circulation: no fixed structure needs to be maintained and no extra-message is induced when a node fails. This solution is fully distributed and the time for processing is shorter. We also focus on two different methods for the local assignment. Indeed, the nodes can wait for the token before selecting a task or they can select one at random that avoids the wait of the token. We give some comparisons according to grid parameters and tasks (average lengths and number of tasks).

This paper is organized as follows. In Section 2, we present the model we use to map a grid and we introduce the random walks with their properties. Then, we explain in Section 3 the two task managements we compare. In Section 4, we exhibit simulation results and comparisons between these methods. Finally, we propose some possible optimizations in the last section.

2 Preliminaries

In this section, we speak about the existing grid architectures according to their centralization degree and the associated task managements. Then, we present a model for designing grid applications on which our solutions are based [19]. Finally, we present a topology management solution presented in [2] a way in which it can be coupled with the task management solutions we propose in this article.

2.1 Grid architectures

As we mentioned previously, the task management depends on the grid topology and particularly on its centralization degree: centralized, semi-distributed (hierarchical) or fully distributed.

The fully centralized solutions like *Netsolve* ([1]) or *XtremWeb* ([7]) are organized around 3 components: the clients, the resources (the servers) and the agent. This agent is in charge both of the resource management and of the scheduling. It maintains a database that contains the resource properties and statistic uses. For each client request, it selects the appropriate servers according to a global load balancing and properties of the servers. Then, these results are sent to the client. Whereas the *Netsolve* clients communicate directly to the servers, no such communications are possible in *XtremWeb*. To improve the security, the *XtremWeb* clients receive the results through the agent or from a depository service on a dedicated server.

Semi-distributed solutions have been proposed like *DIET* ([5]) or *Globus* ([11]). The authors of *DIET* chose to build a hierarchy of servers to improve the scalability of the grid and to avoid the overload of the main server. Resources are registered to agents that are connected to upper agents and so on to the master agent. Additional protocols are used to choose the suited resources corresponding to a client request received by the master agent. In *Globus* ([11]), each node can find adapted resources through *MDS* ([8]) service. This service is distributed on several interconnected servers and it is based on *LDAP* protocol.

Finally, a fully distributed method called *CONFIT* ([10]) has been proposed. Each node has the functions of a server and a client, and it is called a *servent*. The scheduling and the task assignment are achieved locally, so the task parameters are distributed to all nodes and updated thanks to a token. When a node wants to compute a new task, it can choose it depending on its local capacities (computational power or specific components like libraries or softwares). In this way, no server is used to register the resources and for the scheduling. But a spanning structure is maintained for the token circulation: all nodes are set up into a virtual ring. So, even if the middleware is fully distributed and has a better fault tolerance, a lot of message exchanges can be generated in the event of a high volatility of nodes.

2.2 A model for grid

In [19], we propose a model for designing grid applications decomposed in five layers described in Figure 1. Each one is assigned to a specific function.

Layer 1 represents the network as a graph $G_n = (V, E)$ where E is the set of physical links (undirected) and V the set of nodes (the nodes can be computers, servers or passive components as routers or switches).

Layer 2 is the routing layer. A protocol maintains for each node i , a local set noted $Reach_i$ containing the node identities that node i can contact. We take into account the security policies and the protocols like address translation, so each link is considered as directed.

Layer 3 represents the communication layer. A protocol supplies the *send* and *receive* functions with the error and acknowledgment management. Each node can

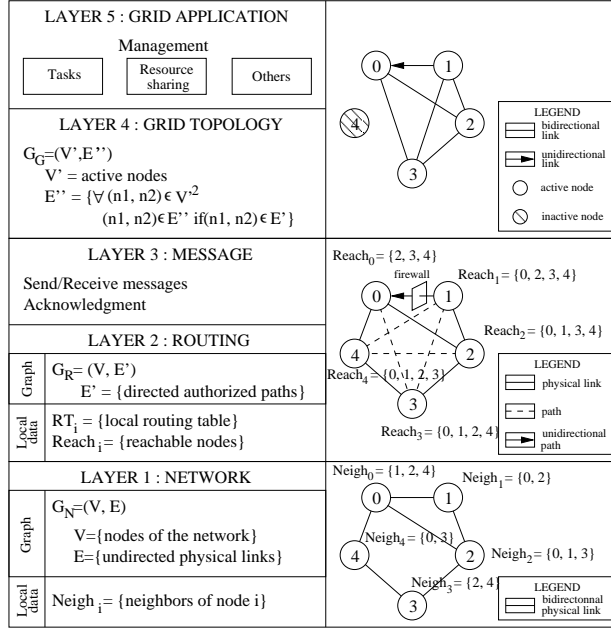


Figure 1: Grid application model.

send a message to one of the node contained in its local set *Reach*.

Layer 4 is the resource management layer of grid applications. In this layer, we distinguish two kinds of nodes: the *active* ones that use or share resources in the grid and the *inactive* ones like routers or nodes that are out of the grid. The grid is represented as a graph $G_G = (V', E'')$ where V' is the set of the active nodes and E'' is the set of the directed communication links between the active nodes. In [2], we propose a resource management based on a random walk according to our model.

Layer 5 represents all the other services of grid applications like the resource sharing and localization or a monitoring component. We need a task management component particularly for the grid computing.

This model highlights the impacts of protocols that interact under a grid. With this model, we can focus on a specific layer or a component. For instance, the resource management (layer 4) has already been focused in [19] and [2]. In the following, we present two task management solutions for layer 5.

2.3 Random walks

In [2], we propose a solution for the topology management based on a random circulation of a token in the grid. This token collects information to detect new resources or resource disconnections. We can use this random walk for the task management to limit the message exchanges. So, in this case, both the topology management and the task management can be achieved with this single token.

A random walk can be viewed as the sequence of nodes visited by a token that

starts on i and visits other nodes according to the following transition rule: if the token is on i at time t then at time $t + 1$, it will be on one of the neighbors of i that is chosen uniformly at random among all of $Reach_i$ [16]. The cover time C is the average time to visit all nodes in the graph and the hitting time $h(i, j)$ is the average time to reach node j for the first time starting from node i . They are two important values that appear in the analysis of the random walk based distributed algorithms. In his survey, the author of [16] shows that the bound of these values is in $O(n^3)$ and in [3], the authors propose methods to compute these values.

Random walks have also the following properties: (1) a random walk covers a graph in a finite time, (2) each node will be visited infinitely and (3) two random walks will meet them in a finite time.

Random walks have already proved their efficiency in the volatile or mobile networks (like peer-to-peer [17] or the ad-hoc networks [6]). They do not need any kind of virtual structure and reduce the number of control message exchanges.

2.4 Topology management based on random walks

The solution we present in [2] is based on the circulating word tool, first introduced in [15]. It is a list of the visited node identities collected by a token. The author presents a method to detect the termination of distributed algorithms by analyzing the cycles in the circulating words created by all the network nodes. In [9], the author proposes to use the circulating word in a token that circulates at random in an undirected graph to build a spanning tree. To ensure the fault tolerance of this algorithm, the token circulates infinitely in the network. So, the word content is corrected when a node disconnects or a link disappears. But the size of the word grows infinitely, so the author proposes two methods to reduce it and to ensure that the construction of a spanning tree is always possible. This solution is fault tolerant and a possible use is the construction and the maintenance of local routing tables on each node.

In [2], we propose an extension of this solution to directed networks. The reduction methods proposed in [9] delete identities in the word without taking care of the link orientation. We base our solution on the cycle detection and maintenance in the word content. When a cycle exists in the word, it is possible to build paths between any nodes contained in this cycle. So, if this cycle contains all the nodes of the visited network, it is possible to build a path between two arbitrary nodes. Our algorithm aims to maintain an image of the communication graph of the network. From this image, it is possible to build spanning structures like trees or rings and to detect all the identities of the network nodes.

This solution is fully distributed, the time for processing and the number of created messages are low. Moreover, it is possible to merge this solution to the task management by using the same token circulation.

3 Passive and active methods

The task management consists in collecting task parameters and assigning these tasks to grid nodes. This assignment takes into account the node properties and the compu-

tational power of nodes must be used equitably.

In this article, we focus on independent tasks (no result of a task is used to compute another one) and irregular tasks (we cannot predicate the execution time of a task that can be very irregular) as in *CONFITT* middleware. We define a task as a tuple $\{p, s, r\}$. p is the set of execution parameters of the task. It gathers the program used to compute it (or the address of the node that owns this program) and parameters of this program (eventually data files). s is the current state of the task. We distinguish 3 different values: *computed* if the task is already computed, *uncomputed* if the task is not computed yet and *in progress* if a grid node is computing it. r is the set of results for the task if it is already computed (or the address of the node that owns the results).

Each node i of the grid owns a local set of the task states noted S_i that is updated during the computation. A node can be in 3 different states: *waiting* if the node waits for a task to compute, *sleeping* if the node is pending (no more task to compute) and *computing* if the node is computing a task.

We define a token as a message contained a set S_j that is a task state set. This message circulates at random among the nodes of the grid and can be merged with the token of the topology management (see previous section). When it comes on node i , sets S_j and S_i are updated. For each task of these sets, we check if it exists in the other set otherwise it is added. Then, we update its state by keeping the more advanced one (we have *uncomputed* < *in progress* < *computed*).

When node i submits new tasks, their parameters are added to its local set S_i . So, when the token comes on this node, the parameters are automatically added to the token. Then, these parameters are disseminated to all the other grid nodes. So, a task submission does not produce any extra-message. In the same way, the deletion of tasks is proceeded when the results are transmitted to the submitter node.

Each node owns the full task parameter set but how can the tasks be assigned to nodes? Here we focus on two fully distributed methods of task assignment based on a token. For the first one, called *passive*, the nodes wait for the token before selecting a task to compute. For the second one, called *active*, the nodes select a task at random among all of the uncomputed ones.

3.1 Passive method

As described previously, the token disseminates the task parameters to the grid nodes. But we need a global synchronization mechanism to assign the tasks to nodes and to avoid the simultaneous computation by other nodes. The new tasks to compute are selected locally by nodes at the token reception. So, the state of the tasks are immediately modified in the token and the other nodes will not select it again. The token implies a mutual exclusion on tasks and none can be replicated.

When a node finishes a task, it tags it as *computed* in its local set and enters the *waiting* mode (or the *sleeping* mode if no more tasks have to be computed). Then, it waits for the token. During this time, its computational power is wasted which we call the *wasted time*. The efficiency of this method depends directly on the evolution of this value.

3.2 Active method

For this method, the nodes do not wait for the token before selecting a new task. So, the wasted time is avoided but we need an extra-mechanism for the task assignment to avoid task replications. Instead of selecting the first task in its local set, the authors of [10] propose to choose a random task among all of the uncomputed ones. So, it reduces the probability of two nodes selecting simultaneously the same task but it implies that the number of tasks must be sufficient. If only one task remains and if several nodes have to select a new one, they will select this one and will produce replicated tasks. By simulations, we remark that about 80% of replicated tasks are detected at 95% of the execution.

4 Active versus passive

As described previously, the efficiency of the passive method depends on the wasted time. The more the nodes wait for the token, the more the efficiency decreases. In the active method, the period of waiting for the token will produce replicated tasks. In the same way, if the average execution time of tasks is too short compared to the cover time of the token, the probability for a node to select an already computed task is higher.

So, to compare the methods, we launch series of simulations with *DASOR* library [18] and we analyze independently impacts of few parameters that modify the cover time. First, we modify the grid size (the number of nodes) and the network performances (the communication time). On the other hand, we observe if the efficiency evolves depending on the number of tasks and their lengths.

4.1 Simulation parameters

In the exposed simulations, we have used a set of task lengths resulting of a real execution of the Langford combinatorial problem ([14]). The distribution of these tasks is ignored: at the beginning of simulations, each node knows this set. The execution time of each task is the same for each grid node and for each simulation: these assumptions ensure only that each simulation can be reproducible but our methods work as well if the computational power of nodes are irregular. During a simulation, no task is added in the system and the simulation ends when all tasks are considered as computed in all grid nodes (each node must be in the *sleeping* state). The task results (or the address of the node that owns them) are transmitted to all nodes.

We distinguish the time t_{first} that corresponds to the time when the first node enters in the *sleeping* mode and the time t_e when all nodes are in the *sleeping* mode (also called the *completion time* or *makespan*).

To compare the two methods, we observe the global execution time to compute all the tasks and we analyze the number of replicated tasks. We compute an efficiency e that is a comparison between the sequential execution time t_{seq} (only one node in the grid) and the distributed execution time t_{first} depending on the node number n with the following formula: $e = \frac{t_{seq}}{t_{first} * n}$.

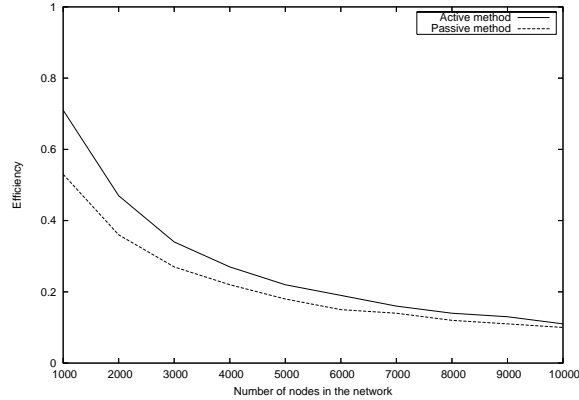


Figure 2: Grid node number.

When the efficiency is close to 1, the distributed solution is optimal. On the other hand, if it equals 0.5, it means that the distributed solution computes only during 50% of the execution time.

4.2 Impacts of grid size

The two solutions are based on a random walk of a token. As we present in Section 2.3, the bound of the random walk cover time is in $O(n^3)$. So, when the network node number increases, it means that the wasted time will increase in the passive method. When a node wants to compute a new task, it has to wait longer for the token. In the active method, the token is used to refresh the task states so the number of replicated tasks will also increase.

We launch series of simulations on random networks. We modify the number of nodes (1.000 to 10.000) and we fix the task set and the communication time. We obtain the results in Figure 2.

We remark that the active method is more clearly efficient than the passive method even if some tasks are replicated. When the number of nodes increases, this difference is reduced and their efficiency is nearly the same in both methods. This can be explained by the cover time that becomes too high.

4.3 Network performances

The efficiency of a grid depends on the number of nodes and their computational power but also on the network that interconnects these resources. A low-bandwidth network and also long distances between nodes (i.e. transversal of number of passive nodes) imply slower communications.

We choose to simulate the influence of the network by modifying the communication time (the average time a token spends to circulate between two nodes) and we obtain the results in Figure 3. When the grid is interconnected with a high speed network, we observe that the two methods have the same efficiency (near 1) and the difference

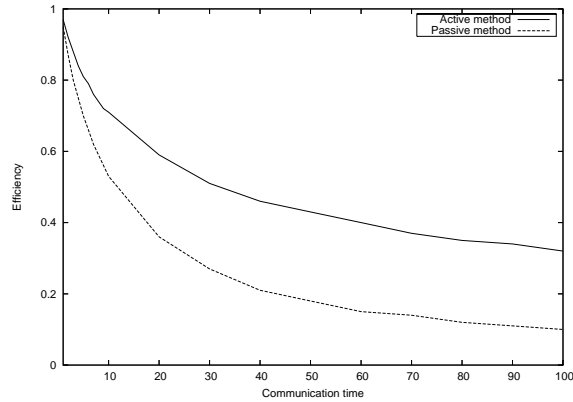


Figure 3: Communication time.

between the first node execution times is very small. In these conditions, the token circulates very fast and the cover time decreases. On the contrary, when the communication time is high, the active method becomes clearly faster than the passive one. The more the token is waited for, the more time is wasted.

4.4 Task lengths and task number

In this section, we observe the impacts of the task number or the average task length. First, we increase the number of tasks: we use the same task set but we compute each task several times (1 to 10). We obtain the results described in Figure 5 on 1000 node networks. For a small set of tasks (about 1 per node), the mutual exclusion on each task avoids a large number of replicated tasks. So, the efficiency of the passive method is better than the active one. This efficiency seems to be constant when the number of tasks increases. With the active method, as we show in Section 3.2, the replicated tasks appear when the ratio of the node number and the number of tasks is small. So, when the number of tasks increases, the efficiency also increases (almost twice as much as the passive method).

Then, we study the impacts of the average task execution time. We use the initial task set but we multiply each length by a factor 0.1 to 20. We launch series of simulations on random 1.000 node networks and we observe that both methods have the same behavior: for short tasks, the efficiencies are worse than for long tasks. The ratio of the token cover time and the average execution time is higher.

5 Fault tolerance

As we specified previously, our solutions have been proposed for volatile networks. In this section, we study the influence of node disconnections on the task management. On the other hand, the two methods are based on a token circulation. So, we need to prevent the loss of the token to stop the nodes waiting infinitely with the passive

method and the increasing of task replication with the active method. We also need to manage when a task is begun by a node that disconnects from the network before finishing it.

5.1 Node disconnections or corruption of a task state

When a node disconnects from the grid, we have the following cases to consider. 1), the node is waiting for the token to select a task (only with the passive method). 2), the node has started a task but the token has not yet visited it¹. 3), the node owns the token and crashes before transmitting it to one of its neighbors. 4), the node has started a task and the token has already visited it.

For cases 1) and 2), there is no impact on the task management and there is only wasted time. In case 3), we need an extra-mechanism described in Section 5.2.

If a node begins a task and crashes before its end, the task can be viewed as *in progress* by the other nodes. So, no other node can select this task again and it will never be computed (we call this task a *zombie task*). When no other task in a node is considered as uncomputed, the node enters the sleeping mode. The authors of *CONFIT* propose the voluntary replication. It implies that the node does not enter the sleeping mode and selects a task tagged as *in progress*. It induces a replicated task but as we explained previously, it allows the computation of the *zombie tasks*.

5.2 Loss of the token

If a node disconnects from the network after receiving the token and before transmitting it to one of its neighbors, the token is lost. It is also possible that the network loses its connectivity² after several node failures. In that case, the token cannot reach again all nodes of each part of the network.

So, we add a timeout on each node reseted at each token reception. If a timeout ends, the associated node creates a new token based on the local view of the node task state set. To avoid having several tokens in the network, we tag the token with an identifier composed of a message identifier and the identity of the node that creates the token. At each reception of the token, the nodes compare the token identifier with the trace of the last token identifier. For instance, if the node has the trace (*OldNodeId*, *OldMessageId*) and the message has the identifier (*NodeId*, *MessageId*), we have the following cases. *OldMessageId* < *MessageId*: the token has been created more recently than the last token, so it is valid and the trace on the node is updated. *OldMessageId* > *MessageId*: the token is an old one, it can be deleted. *OldMessageId* = *MessageId*: we need to compare the identity of the node that creates the two tokens. If *OldNodeId* > *NodeId*, the token can be deleted or the token is valid and the trace on the node is updated.

At the connexion of a node, its trace is initialized with *OldNodeId*=0 and *OldMessageId*=0. When a new token is created, *NodeId* equals the identity of the node that

¹We have the similar impacts if the node has selected and finished the task and crashes before the token visit.

²Here we suppose that the network will be strongly connected after a given time.

creates it and $MessageId=OldMessageId+1$ ($OldNodeId$ and $OldMessageId$ are also updated with these new values).

The timeout must be reseted with a value that depends on the token cover time: the average communication time and the number of nodes in the network. For the simulations we exhibit in Section 5.3, we reset the timeout with a random value chosen in the interval $[2 \times t \times n, 3 \times t \times n]$ where t is the average communication time and n is the number of nodes in the network.

The timeout mechanism can have an influence on the efficiency of both methods. Indeed, in the passive method, the token is created with the local view of the node and this local view can be incorrect. So, it could induce replicated tasks. Moreover, if the timeout is reseted with a value that is too high, the wasted time will increase.

In the active method, the creation of new tokens can also induce replicated tasks until the new token is fully updated. But several tokens could increase the efficiency of this method. Indeed, depending on the meeting time of the two tokens, they will circulate in the network and faster update the local task state sets of the nodes (in Section 6, we present a multi-token solution for both the methods).

5.3 Simulations

With the previously described mechanisms, we launch series of simulations for both methods on 1.000 node networks on which we apply a fault model. Each time, only a fixed number of nodes are awake. The method efficiency is now computed following the number of awake nodes: for a network of 1.000 nodes, if only 500 nodes are awake, we can consider that this network is a 500 node network.

We decrease the number of awake nodes from 1.000 to 500 and we obtain results described in Figure 4. The efficiency of both methods decreases when the number of faults increases. They are two main explanations. First, a lot of nodes begin a task and fail before the end of the task. Therefore, this time is wasted and it involves a decreasing of the efficiency. Then, when a node starts a task, this one is tagged as *in progress*. So, at the end of the computation, there are a lot of zombie tasks. Thus it involves a lot of replicated tasks.

6 Optimizations

To improve the efficiency of these methods, we propose two optimizations: a multi-token solution and an hybrid method that switches between the two methods depending on the number of nodes and the number of tasks.

A multi-token solution can increase the efficiency in the passive method. If we add new tokens, the nodes will wait less and will quickly select a new task. But to avoid the replication, we need to divide the whole task set into several parts: for instance, one per token. This division can be achieved by dividing the whole task set into equal shares. Another solution can be the gathering of task submitter or according to the library needed to compute the tasks. During the computation, a synchronization between the tokens can be achieved by moving dynamically tasks from a token to another one in order to balance them.

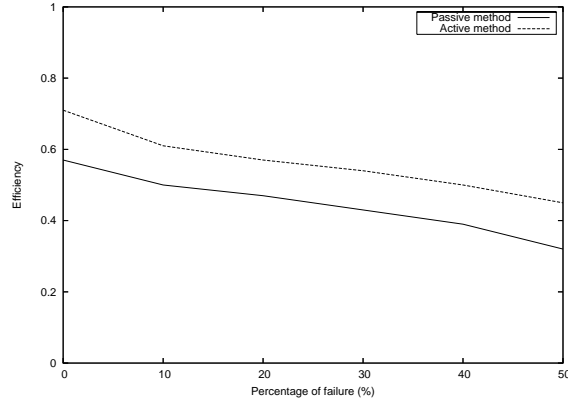


Figure 4: Percentage of failure.

Contrary to the passive method, a multi-token solution can be easier to adapt to the active method. Each token can own a full copy of the whole task set, but we can also choose to divide it into the tokens. A synchronization between them is useless because their circulation will induce an automatic update of task states.

Each token added in the grid increases the number of exchanged messages independently of the grid properties and tasks. Therefore, gains of multi-token solutions seem to be less significant than the one with an hybrid solution.

From the simulations, we observe that the efficiency of the two methods depends on the grid properties (the number of nodes and the network bandwidth) and also on tasks to compute (number and average length). To obtain better results, it is possible to use an hybrid solution. The algorithm begins in the active mode and when the ratio of number of tasks and number of nodes becomes too small, it turns into the passive mode: the nodes wait for the token before selecting a new task. The lengths of tasks cannot be taken into account because, as specified in Section 3, the tasks are irregular and their lengths cannot be predicated.

As shown in Figure 5, the passive method is better than the active method with a ratio lower than 5. Thus, we launch series of simulations of the hybrid method with this ratio value and we obtain the results in Figure 5 and Figure 6. The hybrid solution gives better results whatever the number of nodes. Its efficiency is nearly the same as the active method when the ratio is upper than 5 but the number of replicated tasks is widely reduced. It can be explained by the percentage of node computational power use. As shown by Figure 6, a large part of the node computational power is saved with the hybrid method and can be used for another computation or for a local use (the grid applications share the computational power with other local applications).

7 Conclusion

In this article, we compare two methods of task management for grid applications: active and passive methods. These methods use only one random walk of a token

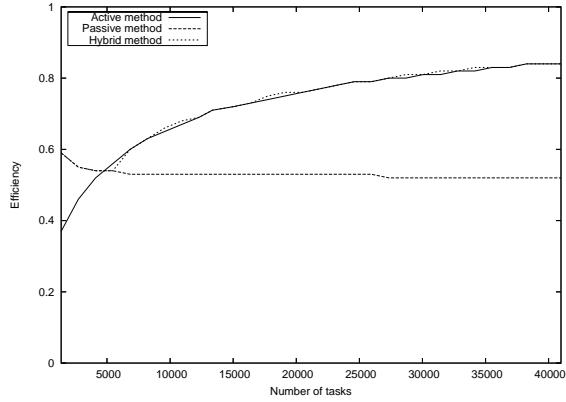


Figure 5: Task to compute.

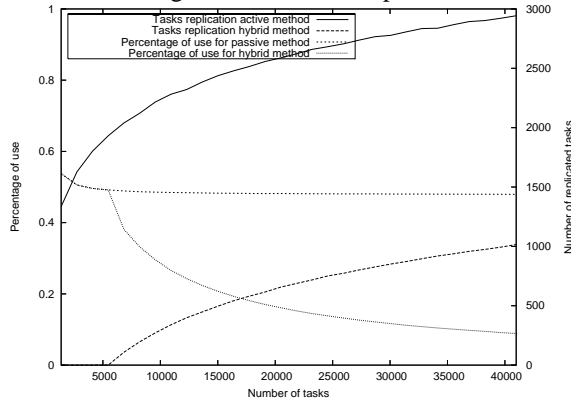


Figure 6: Use percentage and task replication.

and they do not need any kind of centralization. Their efficiency depends on several parameters like the grid physical properties and the average task length or the task number. We present an hybrid solution that switches between the active and passive solutions by comparing the grid node number and the task number. We observe that it gives better results and saves computational power. We may improve this solution if we also take care of the network properties. Moreover, we observe that when the network is too large, both passive and active solutions give a reduced efficiency. By dividing the network into several parts, we will decrease the cover time of the token and improve the efficiency.

In further works, we will focus on a larger study by combining these parameters. Particularly, we plan to integrate several metrics into each layer of our model in order to give more detailed comparisons between grid strategies. Indeed, the time to compute all tasks is not the main concern, the saved computational power is also important.

Acknowledgments

This work was partly supported by “Romeo”³, the high performance computing center of the University of Reims Champagne-Ardenne, France.

References

- [1] D. Arnold, S. Agrawal, S. Blackford, J. Dongarra, M. Miller, K. Seymour, K. Sagi, Z. Shi, and S. Vadhiyar. Users’ Guide to NetSolve V1.4.1. Innovative Computing Dept. Technical Report ICL-UT-02-05, University of Tennessee, Knoxville, TN, June 2002.
- [2] T. Bernard, A. Bui, O. Flauzac, and C. Rabat. Decentralized Resources Management for Grid. In *RDDS’06 International Workshop on Reliability in Decentralized Distributed systems*, volume 4278 of *LNCS*, pages 1530–1539. Springer-Verlag, 2006.
- [3] A. Bui and D. Sohier. On Time Analysis of Random Walk Based Token Circulation Algorithms. In *ISSADS’05 International School and Symposium on Advanced Distributed Systems*, volume 3563 of *Lecture Notes in Computer Science*, pages 63–71. Springer, 2005.
- [4] R. Buyya, D. Abramson, and J. Giddy. Nimrod/G: An Architecture of a Resource Management and Scheduling System in a Global Computational Grid. *CoRR*, cs.DC/0009021, 2000.
- [5] E. Caron, F. Desprez, F. Lombard, J.-M. Nicod, M. Quinson, and F. Suter. A Scalable Approach to Network Enabled Servers. In *Proceedings of the 8th International EuroPar Conference*, volume 2400 of *LNCS*, pages 907–910. Springer-Verlag, 2002.
- [6] W. Choi, S. Das, J. Cao, and A. Datta. Randomized dynamic route maintenance for adaptive routing multihop mobile ad hoc networks. *Journal of Parallel and Distributed Computing*, 65:107–123, 2005.
- [7] G. Fedak, C. Germain, V. Neri, and F. Cappello. XtremWeb: A Generic Global Computing System. In *IEEE/ACM - CCGRID’2001 Special Session Global Computing on Personal Devices*. IEEE Press, May 2001.
- [8] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke. A Directory Service for Configuring High-Performance Distributed Computations. In *Proc. 6th IEEE Symp. on High Performance Distributed Computing*, pages 365–375. IEEE Computer Society Press, 1997.
- [9] O. Flauzac. Random Circulating Word Information Management for Tree Construction and a Shortest Path Routing Tables Computation. In R. G. Cardenas, editor, *OPODIS*, Studia Informatica Universalis, pages 17–32. Suger, Saint-Denis, rue Catulienne, France, 2001.
- [10] O. Flauzac, M. Krajecki, and J. Fugère. CONFIT : a middleware for peer to peer computing. In *The 2003 International Conference on Computational Science and its Applications (ICCSA 2003)*, volume 2669 (III) of *LNCS*, pages 69–78, Montréal, Québec, 2003. Springer-Verlag.
- [11] I. Foster and C. Kesselman. Globus : a metacomputing infrastructure toolkit. In I. Press, editor, *Supercomputer Applications*, volume 11 (2), pages 115–128, 1997.
- [12] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a Future Computing Infrastructure*. Morgan Kaufmann, September 1999.
- [13] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. *Cluster Computing*, 5(3):237–246, 2002.

³<http://www.univ-reims.fr/Calculateur>

- [14] C. Jaillet and M. Krajecki. Solving the Langford Problem in Parallel. In *Proceedings of the 3rd International Symposium on Parallel and Distributed Computing (ISPDC 2004)*, pages 83–90, Cork, Ireland, July 5-7 2004. IEEE Computer Society.
- [15] I. Lavallée. Contribution à l’algorithmique parallèle et distribuée, application à l’optimisation combinatoire, thèse d’état, 1986. Université de Paris XI, Orsay.
- [16] L. Lovász. Random walks on graphs : A Survey. In *Combinatorics : Paul Erdos is Eighty*, volume 2, pages 353–398. Janos Bolyai Mathematical Society, 1993.
- [17] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and replication in unstructured peer-to-peer networks. In *ICS '02: Proceedings of the 16th international conference on Supercomputing*, pages 84–95. ACM Press, 2002.
- [18] C. Rabat. DASOR Home Page. <http://cosy.univ-reims.fr/~crabat/DASOR/>.
- [19] C. Rabat, A. Bui, and O. Flauzac. A random walk topology management solution for grid. In *I2CS*, volume 3908 of *LNCS*, pages 91–104. Springer, 2006.