Mapping Asynchronous Iterative Applications on Heterogeneous Distributed Architectures

Raphaël Couturier David Laiymani Sébastien Miquée
Laboratoire d'Informatique de Franche-Comté (LIFC)
IUT de Belfort-Montbéliard,2 Rue Engel Gros, BP 27, 90016 Belfort, France
Email: {raphael.couturier,david.laiymani,sebastien.miquee}@univ-fcomte.fr

***Abstract*—To design parallel numerical algorithms on large scale distributed and heterogeneous platforms, the asynchronous iteration model (AIAC) may be an efficient solution. This class of algorithm is very suitable since it enables communication/computation overlapping and it suppresses all synchronizations between computation nodes. Since target architectures are composed of more than one thousand heterogeneous nodes connected through heterogeneous networks, the need for mapping algorithms is crucial. In this paper, we propose a new mapping algorithm dedicated to the AIAC model. To evaluate our mapping algorithm we implemented it in the JaceP2P programming and executing environment dedicated to AIAC applications and we conducted a set of experiments on the Grid'5000 testbed. Results are very encouraging and show that the use of our algorithm brings an important gain in term of execution time (about 40%).**

***Keywords*-Mapping algorithms; Distributed clusters; Parallel iterative asynchronous algorithms; Heterogeneous distributed architectures.**

I. Introduction

Nowadays scientists of many domains, like climatic simulation or biological research, need large and powerful architectures to compute their large applications. Distributed clusters architectures, which are part of the grid architecture, are one of the best architectures used to solve such applications with an acceptable execution time. These architectures provide a lot of heterogeneous computing nodes interconnected by a high performance network, but even with the greatest efforts of their maintainers, there are latency and computation capacity differences between clusters of each site.

In order to efficiently use this massive distributed computation power, numerous numerical algorithms have been modified. These algorithms can be broadly classified into two categories. First, *direct methods*, which give the exact solution of the problem using a finite number of operations (e.g. Cholesky, LU...). These methods cannot be applied to all kinds of numerical problems. Generally, they are not well adapted to very large problems. Then *iterative methods*, that repeat the same instructions until a desired approximation of the solution is reached – we say that the algorithm has converged. Iterative algorithms constitute the only known approach to solving some kinds of problems and are easier to parallelize than direct methods. The Jacobi or Conjugate Gradient algorithms are examples of such iterative methods.

In the rest of this paper we only focus on iterative methods. Now to parallelize this kind of algorithm, two classes of parallel iterative models can be described. In the *synchronous iteration model* after each iteration a node sends its results to its neighbors and waits for the reception of all dependency messages from them to start the next iteration. This results in large idle times and is equivalent to a global synchronization of nodes after each iteration. These synchronizations can strongly penalize the overall performance of the application particularly in case of large scale platforms with high latency network. Furthermore, if a message is lost, its receiver will wait forever for this message and the application will be blocked. In the same way, if a machine fails, all the computation will be blocked.

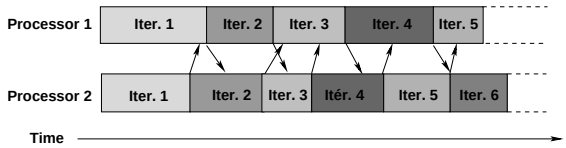

Figure 1. Two processors computing in the Asynchronous Iteration - Asynchronous Communication (AIAC) model

In the *asynchronous iteration model* a node sends its results to its neighbors and starts immediately the next iteration with the last received data. These data could be data from previous iterations, because the most recent data has not arrived in time or neighbors have not finish their current iteration. The receiving and sending mechanisms are asynchronous and nodes do not have to wait for the reception of dependency messages from their neighbors. Consequently, there is no more idle time between two iterations. Furthermore, this model

This work was supported by the European Interreg IV From-P2P project and the region of Franche-Comté

is tolerant to messages loss and even if a node dies, the remaining nodes continue the computation, with the last data the failed node sent. Unfortunately, the asynchronous iteration model generally requires more iterations than the synchronous one to converge to the solution.

This class of algorithms is very suitable in a distributed clusters computing context because it suppresses all synchronizations between computation nodes, tolerates messages loss and enables the overlapping of communications by computations. Interested readers might consult [1] for a precise classification and comparison of parallel iterative algorithms. In this way, several experiments [1] show the relevance of the AIAC algorithms in the context of distributed clusters with high latency between clusters. These works underline the good adaptability of AIAC algorithms to network and processor heterogeneity.

As we aim to solve very large problems on heterogeneous distributed architectures, in the rest of this study we only focus on the asynchronous iteration model. In order to efficiently use such algorithms on distributed clusters architectures, it is essential to map the application's tasks to the best sub-sets of nodes of the target architecture. This mapping procedure must take into account parameters such as network heterogeneity, computing nodes heterogeneity and tasks heterogeneity in order to minimize the overall execution time of the application. To the best of our knowledge, there exits no algorithm which specifically addresses the mapping of AIAC applications on distributed architectures. The aim of this paper is to propose a new mapping algorithm dedicated to AIAC applications and to implement it into a real large scale computing platform, JaceP2P-V2. Experiments conducted on the Grid'5000 testbed with more than 400 computing cores show that this new algorithm enhances the performance of JaceP2P-V2 by about 40% for a real and typical AIAC application.

The rest of this paper is organized as follows. Section II presents the JaceP2P-V2 middleware. We focus here on one of the main drawbacks of this platform: its lack of an efficient mapping strategy. Section III presents our mapping problem and quotes existing issues to address it. Section IV describes the specificities of the AIAC model and details the main solution we propose to address the AIAC mapping problem. In section V we describe the experiments we have conducted, with their different components and results. These results were conducted on the Grid'5000 testbed with more than 400 computing cores and show an important gain (about 40%) of the overall execution time for a typical AIAC application, i.e. based on the NAS Parallel Benchmark. Finally, we give some concluding remarks and plan our future work in section VI.

II. JaceP2P-V2

JaceP2P-V2 [2] is a distributed platform implemented using the Java programming language and dedicated to developing and executing parallel iterative asynchronous applications. JaceP2P-V2 executes parallel iterative asynchronous applications with dependencies between computing nodes. In addition, JaceP2P is fault tolerant which allows it to execute parallel applications over volatile environments and even for stable environments like local clusters, it offers a safer and crash free platform. To our knowledge this is the only platform dedicated to designing and executing AIAC algorithms.

The JaceP2P-V2 architecture, is composed of three main entities:

- The first entity is the "super-node". Super-nodes form a circular network and store, in registers, the identifiers of all the computing nodes that are connected to the platform and that are not executing any application. A super-node regularly receives heartbeat messages from the computing nodes connected to it. If a super-node does not receive a heartbeat message from a computing node for a given period of time, it declares that this computing node is dead and deletes its identifier from the register.
- The second entity is the "spawner". When a user wants to execute a parallel application that requires N computing nodes, he or she launches a spawner. This one contacts a super-node to reserve the N computing nodes plus some extra nodes. When it receives the list of nodes from the super-node, it transforms the extra nodes into spawners (for fault tolerance and scalability reasons) and stores the identifiers of the rest of the nodes in its own register. Then each spawner becomes responsible for a subgroup of computing nodes, starts the tasks on the computing nodes under its command and sends a specific register to them.
- The third entity is the "daemon", or the computing node. Once launched, it connects to a super-node and waits for a task to execute. Once they begin executing an application daemons form a circular network which is only used in the failure detection mechanism. Each daemon can communicate directly with the daemons whose identifiers are in its register. At the end of a task, the daemons reconnect to a super-node.

To be able to execute asynchronous iterative applications, JaceP2P-V2 has an asynchronous messaging mechanism and to resist daemons' failures, it implements a distributed backup mechanism called the uncoordinated distributed checkpointing. For more details on the

JaceP2P-V2 platform, interested readers can refer to [2].

Benefits of mapping: In the previously described JaceP2P-V2 environment there is no effective mapping solution. Indeed, when a user wants to launch an application, the spawner emits a request to the super-node, which is in charge of available daemons. Basically, the super-node returns the amount of requested computing nodes by choosing in its own list. In this method, the super-node only cares about the amount of requested nodes and it returns in general nodes in the order of their connection to the platform – there is no specific selection. Distributed architectures such as distributed clusters, are often composed of heterogeneous clusters linked via heterogeneous networks with high latencies and bandwidths. As an example the Grid'5000 [3] testbed is composed of 23 clusters spread over 9 sites. Those clusters are heterogeneous, with computing powers starting from bi-cores at 2GHz to bi-quad-cores at 2.83GHz with 2Gb of memory for the first one to 8Gb for the second. Links relying clusters are 10Gb/s capable, but as many researchers use this platform, high latencies appear in links between sites.

With such an architecture, it could be efficient to assign tasks communicating with each other on the same cluster, in order to improve communications. But, as we use very large problems, it is quite impossible to find clusters containing as many computing nodes as requested. So we have to dispatch tasks over several clusters. That implies a need to deal with heterogeneity in clusters computing power and heterogeneity in network. We should make a trade-off between both components in order to take the best part of each one to improve the overall performance.

In order to check if a tasks mapping algorithm would provide performance improvement in the JaceP2P-V2 environment, we have evaluated the contributions of a simple mapping algorithm, which is described in section V-B1. These experiments used the NPB Kernel CG application described in section V-A, with two problem sizes (the given problem sizes are the sides sizes of square matrices used) and using a distributed clusters architecture composed of 102 computing nodes, representing 320 computing cores, spread over 5 clusters in 5 sites. The results of these experiments are given in Table I.

Problem size	550,000	5,000,000
Execution Time (without mapping)	141s	129s
Execution Time (with mapping)	97s	81s
Gains	31%	37%

Table I
Effects of a simple tasks mapping algorithm on application's execution time

As can be seen in Table I, the effects of a simple tasks mapping algorithm are significant. This encouraged us to look further for better task mapping algorithms. In the next section, we describe the specificities of our model and issues which can be exploited.

III. Problem description

A. Model formalization

1) Application modeling: In high performance computing, when we want to improve the global execution time of parallel applications we have to make an efficient assignation of tasks to computing nodes. Usually, to assign tasks of parallel applications to computing nodes, scheduling algorithms are used. These algorithms often represent the application by a graph, called DAG [4], [5], [6], [7] (Directed Acyclic Graph). In this graph, each task is represented by a vertex which is relayed to others by edges, which represent dependencies and communications between tasks. This means that some tasks could not start before other ones finish their computation and send their results. As discussed in the introduction, in the AIAC model, there is no precedence between tasks.

Indeed, with the AIAC model, all tasks compute in parallel at the same time. As communications are asynchronous, there is no synchronization and no precedence. During an iteration, each task does its job and sends results to its neighbors and continues with the next iteration. If a task receives new data from its dependencies, it includes them and the computation continues with these new data. If not all dependencies data, or none, are received before starting the computation of the next iteration, old data are used instead. Tasks are not blocked on dependencies. Nevertheless regularly receiving new data allows tasks to converge more quickly. So, it appears that DAG are not appropriate to modeling AIAC applications. TIG [8], [9] (Task Interaction Graph) are more appropriate.

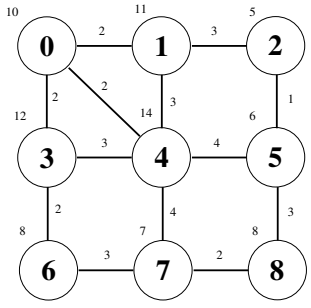

Figure 2. An example of a TIG of a nine tasks application

In the TIG model, a parallel program is represented by a graph , as can be seen in Figure 2. This graph $GT(V, E)$, where $V = \{V_1, V_2, \dots V_v\}$ is the set of $|V|$ vertices and $E \subset V \times V$ is the set of undirectional edges. The vertices represent tasks and the edges represent

the mutual communication among tasks. A function $ET : V \rightarrow R^+$ gives the computation cost of tasks and $CT : E \rightarrow R^+$ gives the communication cost for message passing on edges. We define $v = |V|$, $ET(V_i) = e_i$ and $CT(V_i, V_j) = c_{ij}$. For example, in Figure 2, $e_0 = 10$ and $c_{01} = 2$, $c_{03} = 2$ and $c_{04} = 2$. Tasks in TIG exchange information during their execution and there is no precedence relationship among tasks; each task cooperates with its neighbors. This model is used to represent applications, where tasks are considered to be executed simultaneously. Temporal dependencies in the execution of tasks are not explicitly addressed: all the tasks are considered simultaneously executable and communications can take place at any time during the computation. That is why vertices and edges are labeled with weights describing computational and communication costs.

2) Architecture modeling: As TIG models the application, we have to model the targeted architecture. A distributed clusters architecture can be modeled by a three-level-graph. The levels are *architecture* (a), in our study it is the Grid'5000 grid, *cluster* (c) and computing node (n) levels. Let $GG(N, L)$ be a graph representing a distributed clusters architecture, where $N = \{N_1, N_2, \ldots N_n\}$ is the set of $|N|$ vertices and L is the set of undirectional edges. The vertices represent the computing nodes and the edges represent the links between them. An edge $L_i \in L$ is an unordered pair $(N_x, N_y) \in N$, representing a communication link between nodes N_x and N_y. Let be $|C|$ the number of clusters in the architecture containing computing nodes. A function $WN : N \rightarrow R^+$ gives the computational power of nodes and $WL : L \rightarrow R^+$ gives the communication latency of links. We define $WN(N_i) = wn_i$ and $WL(L_i, L_j) = wl_{ij}$.

An architecture with a three-level-graph is specified according as follows. All computing nodes are in the same node level. When computing nodes can communicate to one another with the same communication latency, they can be grouped into the same cluster. In addition, like in the Grid'5000 testbed, if computing nodes seemly have the same computational power with a low communication latency, a cluster of these nodes can be defined. All participating clusters, including computing nodes, are in the same architecture level and communicate through the architecture network.

3) Mapping functions: After having described the two graphs used to model the application and the architecture, this section defines our objectives.

When a parallel application *App*, represented by a graph GT, is mapped on a distributed clusters architecture, represented by a graph GG, the execution time of the application, $ET(App)$, can be defined as the execution time of the slowest task. Indeed an application ends when all the tasks have detected convergence and have reached the desired approximation of the solution, that is why the execution time of the application depends on the slowest task. We define $ET(App) = \max_{i=1\ldots v}(ET(V_i))$ where the execution time of each task i $(i = 1 \ldots v)$, $ET(V_i)$ is given by $ET(V_i) = \frac{e_i}{wn_i} + \sum_{j \in J} c_{ij} \cdot wl_{ij}$

where e_i is the computational cost of V_i, wn_i is the computational power of the node N_i on which V_i is mapped, J represents the neighbors set of V_i, c_{ij} is the amount of communications between V_i and V_j, and wl_{ij} is the link latency between the computing nodes on which are mapped V_i and V_j. We underline here that in the AIAC model, it is impossible to predict the number of iterations of a task. So it is difficult to evaluate a priori the cost e_i of a task. In the remainder, we approximate e_i by the cost of one iteration.

The mapping problem is similar to the classical graph partitioning and task assignment problem [10], and is thus NP-complete.

B. Related work

In the literature of the TIG mapping, we can find many algorithms, which can be divided into two categories. First, in the *Edge-cuts optimization* class of algorithms, the aim is to minimize the use of the penalizing links between clusters. As tasks are depending on neighbors, which are called dependencies, the goal is to choose nodes where distance, in term of network, is small to improve communications between tasks. Here we can cite Metis [11], Chaco [12] and PaGrid [13] which are libraries containing such kind of algorithms. The main drawback of edge-cuts algorithms is that they do not tackle the computing nodes heterogeneity issues. They only focus on communication overheads. Then, in the *Execution time optimization* class of algorithms the aim is to minimize the whole execution time of the application. They look for nodes which can provide the small execution time of tasks using their computational power. Here we can cite FastMap [14] and MiniMax [15] as such kind of algorithms. QM [16] is also an algorithm of this category, but it aims to find for each task the node which can provide the best execution time. QM works at the task level, whereas others work at the application level

The two classes of algorithms may fit with our goals, because in our model we have both the computational power of nodes and communication costs which may influence the applications performance. We can also cite partitioning tools like Scotch [17] which aims at privileging the load balancing of their partitioning schemes. Nevertheless, to the best of our knowledge, none of the existing algorithms take into consideration the specificities of the AIAC model (see next section).

IV. AIAC mapping

A. Specificities of the AIAC mapping problem

An important point to take into consideration in the AIAC model is that we do not allow the execution of multiple tasks on the same computing node. This comes from the fact that the targeted architectures are volatile distributed environments. Assigning multiple tasks to a node provides a fall of performance when this node fails. Indeed we should redeploy all of the tasks from this node to another one, using last saves, which implies to search a new available computing node, transfer saves to it and restart the computation from this point (which could be far from this just before the failure).

Nevertheless, in order to benefit of multi-cores architectures, we use a task level parallelism by running multi-threaded sequential solver for example.

Another important point in the AIAC model is that we should take into account precisely the locality issue. This comes from the fact that in this model, the faster and more frequently a task receives its dependencies, the faster it converges. Moreover, as the JaceP2P-V2 environment is fault tolerant and tasks save checkpoints on their neighbors, it is more efficient to save on near nodes than on far ones. In the synchronous model, both heterogeneity and locality must be taken into account in a balanced way. In the asynchronous model, since no synchronizations occurs, the heterogeneity issue is less important.

B. AIAC Quick-quality Map

We present here the solution we propose, called *AIAC QM algorithm*, to address the AIAC mapping problem. We decided to improve the *Quick-quality Map* (QM) algorithm since it is one of the most accurate method to address the TIG mapping problem.

In its original version, this algorithm aims at prioritizing the computational power of nodes. Indeed, its aim is to find the more powerful node to map a task on. Moreover, a part of this algorithm is designed to map multiple tasks on the same node, in order to improve local communications. This solution can be efficient if communications between tasks are heavy and if we consider that computing nodes are stable and not volatile. This last point is in contradiction with our model, as we authorize only the execution of one task on a single node – this allows to lose only the work of a single task in case of node's fault, with a low cost on restarting mechanism. Instead assigning multiple tasks on the same computing node, our mapping algorithm tries to keep tasks locally, to improve communications, by trying to assign tasks to computing nodes in the neighborhood of which their neighbors are mapped on.

So, in this algorithm all nodes are first sorted in descending order according to their computation power, and all tasks are mapped on these nodes according to their identifier (they are also marked as "moveable"; it means that each task can be moved from a node to another). As in the original QM algorithm, AIAC QM keeps track of the *number of rounds* r ($r > 0$), that all tasks have been searched for a better node. This allows to reduce at each round the number of considered nodes. While there is at least one moveable task, it performs for each moveable task the search for a better node. It chooses a set of nodes, $\frac{f \cdot n}{r}$, where f is defined as the search factor and n is the number of nodes. r and $f \in]0, 1]$ control the portion of nodes that will be considered where more numerous the rounds are, the less the considered nodes will be. Then the algorithm estimates the execution time $ET(v)$ of the task on each node. If it is smaller than the current node on which the task is mapped on, this node becomes the new potential node for task t_i.

After having randomly searched for a new node, the AIAC QM tries to map the task on nodes that are neighbors of nodes of which the dependencies of t_i are mapped on. This is one of the major modification to the original QM algorithm. It introduces a little part of "edge-cuts" optimization. In the original version, it tries to map the task t_i on the same node of one of its dependencies. As explain in IV-A, this is not an acceptable solution in our case. Instead, the algorithm now searches to map task t_i on nodes which are near the ones its dependencies are mapped on. This search requires a parameter which indicates the maximum distance at which nodes should be from the node of dependencies of t_i.

At the end of the algorithm, if a new node is found, t_i is mapped on and its execution time is updated and t_i is set to "not moveable". The execution time of each of its dependencies is also updated, and if this new execution time is higher than the previous, the task is set to "moveable". And finally, if all tasks have been considered in this round, r is incremented.

The complexity of the AIAC QM algorithm is about $O(n^2 \cdot t \cdot ln(r))$. This complexity is the same as the original algorithm (details are given in [16], with an increase of a factor n, corresponding to the edge-cuts part).

V. Experimentation

A. The NAS Parallel Benchmark Kernel CG and the Grid'5000 platform

We used the "Kernel CG" of the NAS Parallel Benchmarks (NPB) [18] to evaluate the performance of the mapping algorithm. This benchmark is designed to be used on large architectures, because it tests communications over latency networks, by processing unstructured matrix vector multiplication. In this benchmark, a Conjugate Gradient is used to compute an approximation of the smallest eigenvalue of a large, sparse and symmetric

positive definite matrix, by the inverse power method. In our tests, the whole matrix contains nonzero values, in order to stress more communications. As the Conjugate Gradient method cannot be executed with the asynchronous iteration model we have replaced it by another method called the multisplitting method. This latter supports the asynchronous iterative model.

With the multisplitting algorithm, the A matrix is split into horizontal rectangle parts. Each of these parts is assigned to a processor – so the size of data depends on the matrix size but also on the number of participating nodes. In this way, a processor is in charge of computing its $XSub$ part by solving the following subsystem: $ASub \times XSub = BSub - DepLeft \times XLeft - DepRight \times XRight$.

After solving $XSub$, the result must be sent to other processors which depend on it.

For more details about this method, interested readers are invited to see [1]. In our benchmark, the sequential solver part of the multisplitting method is the Conjugate Gradient, using the MTJ [19] library. Its implementation is multi-threaded, to benefit from multi-core processors.

We point out here that this benchmark is a typical AIAC application. In our study, we consider that the computational costs of tasks are approximately the same and that the communications costs are also the same (this comes from the difficulty to evaluate real costs in the AIAC model). For our experiments the bandwidth of matrices has been reduced in order to limit the dependencies and we fixed it to $35,000$. This bandwidth size generates, according to the problem's size, between 10 and 25 neighbors per tasks.

The platform used for our tests, called Grid'5000 [3], is a French nationwide experimental set of clusters which provides a configurable and controllable instrument. We can find many clusters with different kinds of computers with various specifications and software. Clusters are spread over 9 sites, and the computing power represents more than 5000 computing cores interconnected by the "Renater" network. This network is the national network for research and education; it provides a large bandwidth with high latency. Intra-clusters networks present small bandwidth and low latencies.

B. Other mapping algorithms

1) A Simple Mapping algorithm: The *Simple Mapping algorithm* (SMa) was designed to show the benefits of a mapping algorithm in the JaceP2P-V2 platform.

The algorithm puts each node in a cluster entity. Then it sorts clusters by their size, from the higher to the lower. Finally, all tasks are mapped in order on the sorted clusters; each task is assigned to a particular computing node of the chosen cluster.

2) Edge-cuts optimization: As explained in section III, the asynchronous iteration model is so specific and unpredictable that we would like to evaluate the second kind of mapping algorithm, which aims to optimize the "edge-cuts". We choose the Farhat's algorithm [20], which has the ability to divide the graph into any number of partitions, thereby avoiding recursive bisection.

This algorithm aims to do a "clusterization" of the tasks. First, it groups computing nodes in clusters, which are sorted according to their number of nodes, from the higher to the lower. Tasks are ordered following their dependency degree, starting from the higher to the lower. Tasks in the top of the list have a higher priority to be mapped. Next, the algorithm tries to map on each cluster the maximum number of tasks. To map a task on a cluster, the algorithm evaluates if there is enough space to map the task and some of its dependencies. This amount of dependencies is fixed by a factor δ, which is a parameter of the algorithm. In the positive case, the task is mapped on the current cluster and its dependencies become priority tasks to be mapped. This allows to keep the focus on the communicating tasks locality.

C. Experiments

After having described the different components of the experiments, we now present the impacts of the AIAC QM mapping on applications running with JaceP2P-V2 on a heterogeneous distributed clusters architecture. In the following, we note "heterogeneity degree" the degree of heterogeneity of distributed clusters; it is the ratio between the average and the standard deviation of the computing nodes power. This heterogeneity degree may vary from 0, nodes are homogeneous, to 10, nodes are totally heterogeneous. In these experiments, we consider that there is no computing nodes failing during applications execution.

The application used to realize these experiments is the KernelCG of the NAS parallel benchmark, in the multi-splitting version. Two problem sizes were used: one using a matrix of size $550,000$ (named "class E") using 64 computing nodes and the other using a matrix of size $5,000,000$ (named "class F") using 128 nodes.

Our experiments concern the study of the impact of the heterogeneity of the computing nodes on the mapping results. Heterogeneity is an important factor in high performance computing in the grid all the more so when using the asynchronous iteration model.

As mapping algorithms take in parameter a factor of research (for AIAC QM) and the amount of local dependencies (for F-EC), we fixed both to 50%. That means for AIAC QM that at each round the amount of considering nodes would be divided by two, and for F-EC that each task requires half of its dependencies on the same local cluster.

Four experiments were done using four architectures having different heterogeneity degrees – in two architectures computing nodes are more heterogeneous than in the others. In these experiments, we did not affect the networks heterogeneity, because of the difficulty to disturb and control network on Grid'5000; by default, networks are already quite heterogeneous. We needed more than 200 computing nodes to execute our application because of the small capacity of some clusters to execute the largest problems (there is not enough memory). The nodes used have more than 2 GB of RAM and both execute a Linux 64 bits distribution.

The first architecture, Arc1.1, was composed of 113 computing nodes representing 440 computing cores, spread over 5 clusters in 4 geographically distant sites. In Arc1.1 we used bi-cores (2 clusters), quad-cores (2 clusters) and bi-quad-cores (1 cluster) machines. Its heterogeneity degree value is 6.43. This architecture was used to run class E of the CG application using 64 computing nodes. The second architecture, Arc1.2, used to execute class F of the CG application, using 128 computing nodes, was composed of 213 computing nodes representing 840 computing cores, with a heterogeneity degree of 6.49. This architecture was spread on the same clusters and sites as Arc1.1. The results of the experiments on Arc1.1 and Arc1.2 are given in Table II and Table III, which give the gains in execution time obtained in comparison to the version without mapping.

Algorithm	None	SMa	AIAC QM	F-EC
Execution time	150s	110s	101s	90s
Gains	–	27%	33%	40%

Table II
Gains in time of the execution of the class E of the CG application on Arc1.1 using 64 nodes

Algorithm	None	SMa	AIAC QM	F-EC
Execution time	403s	265s	250s	218s
Gains	–	34%	38%	46%

Table III
Gains in time of the execution of the class F of the CG application on Arc1.2 using 128 nodes

At first, we can see that the Simple Mapping algorithm, though it is simple, provides a significant improvement of application execution time. This highlights that JaceP2P-V2 really needs a mapping algorithm in order to be more efficient. Then, we can see that the F-EC and the AIAC QM algorithms provide a better mapping than the Simple Mapping algorithms. We can see a significant difference between both algorithms. This comes from the homogeneity of clusters. In this case, the F-EC algorithm is more efficient since the minimization of the communications becomes more important than the tackle of the computational power heterogeneity problem. The effect is that tasks do less iterations as they receive more frequently updated data from their neighbors. In addition, as tasks and their dependencies are on the same cluster, communications are improved, but also as computations take approximately the same time, the amount of iterations is reduced and the algorithm can converge more quickly.

The third architecture, Arc2.1, was composed of 112 computing nodes, representing 394 computing cores, spread over 5 clusters in 5 sites. In this architecture we used bi-cores (3 clusters), quad-cores (1 cluster) and bi-quad-cores (1 cluster) machines. Its heterogeneity degree's value is 8.41. This architecture was used to run class E of the CG application, using 64 computing nodes. The fourth architecture, Arc2.2, used to execute class F of the CG application, using 128 computing nodes, was composed of 212 computing nodes representing 754 computing cores, with a degree of heterogeneity of 8.44. This architecture was spread on the same clusters and sites as Arc2.1. The results of the experiments on Arc2.1 and Arc2.2 are given in Table IV and Table V, which give the gains in execution time obtained in comparison to the version without mapping.

Algorithm	None	SMa	AIAC QM	F-EC
Execution time	498s	341s	273s	385s
Gains	–	32%	45%	23%

Table IV
Gains in time of the execution of the class E of the CG application on Arc2.1 using 64 nodes

Algorithm	None	SMa	AIAC QM	F-EC
Execution time	943s	594s	453s	660s
Gains	–	37%	52%	30%

Table V
Gains in time of the execution of the class F of the CG application on Arc2.2 using 128 nodes

To begin with, these experiments confirm that a mapping algorithm is needed and that improvements are always scalable. Then, we can see that the F-EC algorithm falls in performance and AIAC QM is improved. What is surprising is that the Simple Mapping algorithm is better than F-EC. This can be explained by the fact that as computing nodes are quite heterogeneous, computations are not the same, so it is not significant to map dependencies close to tasks. In this case, the most important is the power of computing nodes. So, in this kind of architecture, it is more efficient to choose

the best computing nodes to compute iterations more quickly and to improve the convergence detection.

VI. Conclusion and future works

In this paper we have presented a specific mapping algorithm for the AIAC model, called AIAC QM. This algorithm is based on the execution time optimization but it also includes a small degree of edge-cuts optimization. Experiments show that the AIAC QM mapping algorithm is efficient on architectures with a high heterogeneity degree. This can be explained by the fact that all iteration computations are quite different, for our example, and the convergence is more quickly detected as the more powerful computing nodes progress in the computation.

In our future work we plan to take into consideration the fault tolerance problem. In this study we have realized our experiments without computing node fault, which is not the real case. We have to take into account the AIAC QM algorithm about this important parameter. First we have to efficiently choose new nodes to replace failed ones. Secondly, as we do checkpointing to save tasks' states, we have to efficiently choose backup nodes not to fail in case a whole cluster fails, as we save on neighbors (which are in general on the same cluster for communication efficiency reasons), an important part of the application is lost and we cannot restart this part; so the whole application fails. A trade-off should be done by having some saving nodes in external clusters.

References

[1] J. Bahi, S. Contassot-Vivier, and R. Couturier. *Parallel Iterative Algorithms: from Sequential to Grid Computing*, volume 1 of *Numerical Analysis & Scientific Computating*, chapter Asynchronous Iterations, pages 124–131. Chapman & Hall/CRC, 2007.

[2] J.-C. Charr, R. Couturier, and D. Laiymani. Jacep2p-v2: A fully decentralized and fault tolerant environment for executing parallel iterative asynchronous applications on volatile distributed architectures. In *GPC*, pages 446–458, 2009.

[3] Grid'5000. http://www.grid5000.fr.

[4] T. Yang and A. Gerasoulis. Dsc: Scheduling parallel tasks on an unbounded number of processors. *IEEE Trans. Parallel Distrib. Syst.*, 5(9):951–967, 1994.

[5] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* MIT Press, Cambridge, MA, USA, 1989.

[6] Y.-K. Kwok and I. Ahmad. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7(5):506–521, 1996.

[7] H. Topcuouglu, S. Hariri, and M. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distrib. Syst.*, 13(3):260–274, 2002.

[8] D. L. Long and L. A. Clarke. Task interaction graphs for concurrency analysis. In *ICSE*, pages 44–52, 1989.

[9] D. L. Long L. A. and Clarke. Task interaction graph: An intermediate representation for concurrency. Technical report, University of Massachusetts, Amherst, MA, USA, 1988.

[10] M. Garey and D. Johnson. *Computer and Intractability : a guide to the Theory of NP-Completness.* W.H. Freeman & Co, 1979.

[11] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[12] B. Hendrickson and R. W. Leland. *The Chaco User's Guide.* Sandia National Laboratory, Albuquerque, 1995.

[13] S. Huang, E. E. Aubanel, and V. C. Bhavsar. Pagrid: A mesh partitioner for computational grids. *J. Grid Comput.*, 4(1):71–88, 2006.

[14] S. Sanyal, A. Jain, S. K. Das, and Rupak Biswas. A hierarchical and distributed approach for mapping large applications to heterogeneous grids using genetic algorithms. In *CLUSTER*, pages 496–499, 2003.

[15] S. Kumar, S. K. Das, and Rupak Biswas. Graph partitioning for parallel applications in heterogeneous grid environments. In *IPDPS*, 2002.

[16] P. Phinjaroenphan. *An Efficient, Pratical, Portable Mapping Technique on Computational Grids.* PhD thesis, School of Computer Science and Information technology Science, Engineering and Technology Portfolio, RMIT University, 2006.

[17] C. Chevalier and F. Pellegrini. Pt-scotch: a tool for efficient parallel graph ordering. *Parallel Computing*, 6-8(34):338–331, 2008.

[18] The NAS Parallel Benchmarks. http://www.nas.nasa.gov/Resources/Software/npb.html.

[19] Matrix Toolkit Java. http://code.google.com/p/matrix-toolkits-java/.

[20] C. Farhat. A simple and efficient automatic fem domain decomposer. *Computers & Structures*, 28(5):579 – 602, 1988.