Multi-Criteria Scheduling Heuristics for GridRPC Systems

Yves Caniou
LIP, ENS-Lyon
France

Emmanuel Jeannot
LORIA, Université H. Poincaré – Nancy 1
France

***Abstract*— In this paper, we address the problem of dynamically scheduling independent tasks and/or application task graphs on a GridRPC environment. Resources are assumed to compute submitted jobs within the time-share model. We present a non-intrusive predictive module, the HTM, which is able to give the completion date of each task in the system. Four heuristics relying on its estimations are proposed and compared to the well-known MCT algorithm. We firstly analyze the accuracy of the HTM. Then we show with an extensive simulation study *and* with numerous scenarii of execution performed on a real world platform, that our heuristics outperform MCT on several metrics among which the makespan and the response time.**

I. Introduction

GridRPC [10] is an emerging standard promoted by the Global Grid Forum (GGF)[1]. This standard defines both an API and an architecture. A GridRPC architecture is heterogeneous and composed of three parts: a set of clients, a set of servers and an agent (also called a registry). The agent is charged with mapping client requests to servers. To ensure that a GridRPC system be efficient, the mapping function must choose a server that will fulfill several criteria. First, the choice must allow the total execution time of the client application to be as short as possible. Second, each request of every client must be served as fast as possible. Finally, the resource utilization must be optimized.

Several middlewares instantiate the GridRPC Model (NetSolve [6], Ninf [11], DIET [4], etc.). In these systems, a server executes all its assigned requests when received and never delays the start of the execution. In this case, we say that the execution is *time-shared* (in contrast to *space-shared* when a server executes at most one task at a given moment). In NetSolve, the scheduling module uses the MCT (Minimum Completion Time) [9] heuristic to schedule requests on the servers. MCT was designed for scheduling applications on space-shared servers. The goal was to minimize the makespan of a set of independent tasks. Thus, MCT chooses the server where the last request finishes the soonest. Indeed, within the user-space model, the completion date of the last task is less or equal the overall makespan. Minimizing each task's completion date implies the minimization of the overall set of tasks. But, due to interferences of the execution of two tasks, this is not true anymore in a time-share environment. Furthermore, this scheduling strategy leads to the following drawbacks:

- *Mono-criteria and mono-client*. MCT was designed to minimize the makespan of an application. It is not able to give a schedule that optimizes other criteria such as the response time of each request. Furthermore, optimizing the last task completion date does not necessarily minimize the makespan of individual clients. Indeed, in the context of GridRPC, the agent has to schedule requests from more than one client and it has no information about the application to which the task belongs.
- *Load balancing*. MCT tries to minimize the execution time of each request. This leads to overuse of the fastest servers. In a time-share environment, this implies delaying previously mapped tasks and therefore degrades the response time of the corresponding requests.

MCT requires sensors that give information on the system state. It is mandatory to know the network and server's states in order to make good scheduling decisions. However, actively monitoring the environment is intrusive and perturbs its performance. Moreover, information is passed to the agent only periodically. These data may be invalid by the time the scheduling decision is made.

In order to address these drawbacks, we propose and study four scheduling heuristics designed for GridRPC systems. Our approach is based on a prediction module that runs only on the agent. This module is called the Historical Trace Manager (HTM), and it records all scheduling decisions. Since it runs on the agent, it is not intrusive and there is no delay between the determination

[1] `http://www.ggf.org`

of the state of the system and the availability of the information. The HTM takes into account the fact that servers run under the time-share model. It is able to predict the duration of a given task on a given server as well as its impact on already mapped tasks. In this paper, we show that the HTM is accurate and is able to predict very precise task durations on moderately loaded servers. The four proposed heuristics use the HTM to schedule the tasks.

Our approach is an experimental one. We have performed both simulation experiments and real-platform tests. In order to perform the tests on a real platform, we have introduced the HTM and our heuristics into the NetSolve system and performed intensive series of tests on a real distributed platform (almost two months of continuous computation) for various experiments with several clients.

In [9], MCT was proposed and studied only in the context of independent task submission (called a *metatask*). In this paper our study is more general: tests contain submissions of independent tasks as well as submissions of task graphs (1D meshes and stencil). Moreover, we consider several kinds of application graphs in the parallel job scenario and independent tasks are submitted during the execution of the graphs. We do not assume any knowledge of the graphs and we schedule dynamically each request. The goal is to best serve each client request (be it a parallel task graph or a single independent task).

We have compared our heuristics against MCT implemented in NetSolve on several criteria (makespan, sumflow, meanflow, etc.). Results show that the proposed heuristics outperform MCT on a majority of these criteria with simultaneous performance gains, up to 20% for the makespan and to 60% for the average response time.

II. MCT IN GRIDRPC SYSTEMS

The GridRPC model (also called the client-agent-server model) is an extension of the client-server model where the agent dispatches client requests on already registered servers.

In this model, the agent is the critical component. It knows the state of the environment and schedules client requests on servers that are able to execute them. Servers are computational distributed resources. Each server, once launched, contacts the agent and gives its list of "*problems*" it is able to solve. Finally, a client is a program that requests computational resources. It asks the agent to find a set of the most suitable servers that are able to solve its problems.

In this approach, the agent is launched first, then the servers can register with the agent by sending the list of problems they are willing to solve as well as their peak performances and network capabilities. The client, who has a computational need, contacts the agent, which in return, gives him a ranked list of servers. The client sends the request to each server in turn using an RPC–like call until one agrees to resolve the corresponding task. The client then sends the input data and the execution of the task begins as soon as the transfer is completed. When the request has finished, the server sends back the output data to the client.

In this framework, the performance of the whole system depends greatly on the scheduling heuristic implemented in the agent and the accuracy of the information the agent has on the system.

NetSolve [6] instantiates the GridRPC model. A NetSolve agent uses Minimum Completion Time [9] (MCT) to map tasks on servers. The MCT heuristic chooses the server that will finish the task the fastest. In order to determine the completion time of a given task on a server, it assumes that the load on the server will be constant during the execution of the task. This leads to the following remarks:

1) the agent needs an evaluation of each server load;
2) the load on a given server is seldom constant. If the server is loaded by some tasks, they are likely to finish during the computation of the new mapped task;
3) the time-share model implies that mapping a task on a loaded server delays the currently executing tasks. The perturbation caused by the new task on the others may make previous decisions obsolete.

First, the load evaluation of a network or of a machine is a difficult task and is often performed by sensors. NetSolve can use NWS sensors [14] or its own. It faces two major problems: accuracy and intrusiveness.

Sensors run short processes on servers in order to evaluate the current load. It needs some CPU cycles and therefore delays other processes. In such an architecture, the delay is important only if the probe rate is high. However, relatively frequent measurements are needed in order to obtain a reasonable accuracy.

The accuracy problem is the major one. Indeed, a sensor sends to the agent, at given intervals, the load of the server it observes. Between these intervals, the load may change, and thus causing the information on the agent to become inaccurate.

Second, MCT considers that the load is constant during the execution of a task. Therefore, it can make misguided scheduling decisions. For instance, if two identical servers are equally loaded, MCT has no other information to determine on which one to schedule a new task. But, if for some reasons it is known that on one of these two servers, a task will finish very soon, a good heuristic would *a priori* choose this server.

Third, mapping a task on a server delays the other running tasks. This delay is called the *perturbation* in this paper. It is very important to take the perturbation into consideration. Indeed, delaying tasks can make former decisions obsolete ("server X has been chosen because it leads to shorter execution time than server Y"). Conversely, we need to ensure that, if we map a task to a server, there will be not too many tasks that will delay this one.

III. Historical Trace Manager

The Historical Trace Manager is an attempt to efficiently answer the three remarks exposed in the previous section. It is a prediction module that runs on the agent. It is accurate and non intrusive. It simulates the execution of the tasks on each server and therefore is able to predict load variation and to compute the perturbation of existing tasks caused by the introduction of new tasks.

A. Time-share Model

At a given moment it is possible that a server has to run more than one job. This happens, for instance, when the system is heavily loaded or when the set of servers is heterogeneous (in that case, for performance reasons, the agent may often select the fastest servers). This is true even if servers are dedicated to the grid middleware.

We use the following model to simulate time-shared resources. When n tasks are using the same resource (CPU, network, etc.), each one uses $1/n$ of its peak power.

B. The HTM Algorithm

The GridRPC standard imposes that each request is divided in three parts: first, the transmission of the input data, then the computation on the server and finally, the transmission of the output data. The HTM performs a discrete event simulation for these three parts. In order to do that, it needs several types of information: server and network peak performances, the size of input and output data and the number of operations of each task. These information are all static and can be computed off-line (note that MCT also needs this information). Therefore, the HTM answers the three remarks exposed in section II:

1) the HTM is not intrusive because it uses only static information. Furthermore, since it runs on the agent, information is immediately available for scheduling heuristics. The accuracy of the information given by the HTM is very high and will be experimentally demonstrated in the results section;
2) the HTM is able to compute the completion date of each task in the system, in particular of a new request, by simulating its execution. Therefore, the load is not assumed to be constant. For instance, if two identical servers are equally loaded, it knows the remaining duration on each request on each server and therefore is able to optimize the choice of the server to map a new request;
3) the HTM can simulate the mapping of a task on any server. Hence, it can compute the perturbation of this task on all currently running tasks. The perturbation is different for each server and can therefore be used for making scheduling decisions, as we will see in the next section.

C. Notations

We use the following notations: a_i is the arrival date of task i. T'_i is the simulated finishing date in the current system state and C_i is the real one (*post-mortem*). The HTM can simulate the execution of a new task n and give the new simulated completion dates T_i of all tasks $i, i \leq n$. We define for all $k \leq n, \delta_k = T_k - T'_k$, the *perturbation* the task n produces on each running ones (Fig 1). We also define for all $k \leq n, D_k = T_k - a_n$, the *remaining* duration of the task k before completion. $p(i)$ is the server where the task i is mapped and d_i is its duration on the unloaded server. Note that these notations are related to a given server j. However, since in the following the server number is always known we deliberately ignore subscript j.

D. Usefulness of the HTM

Here follows an example that shows how the Historical Trace Manager can help in making good scheduling decisions.

Let us suppose that the set of servers is made up of two identical servers (same network capabilities, same CPU speed peak, same set of problems, etc.). At time 0, the client sends to these servers two tasks 1 and 2, whose durations on each server is respectively 100 and 1000 seconds, with no input data. Let the agent schedule task 1 on the server 1 and task 2 on the server 2. At time 80, let a client request that the agent schedule a task 3 whose duration is 100 seconds.

Without the Historical Trace Manager, the agent knows only that server 1 and server 2 have the same load and therefore is not able to decide which is the best server to schedule task 3 (in practice, as there is dynamic information and as the evolution of the load

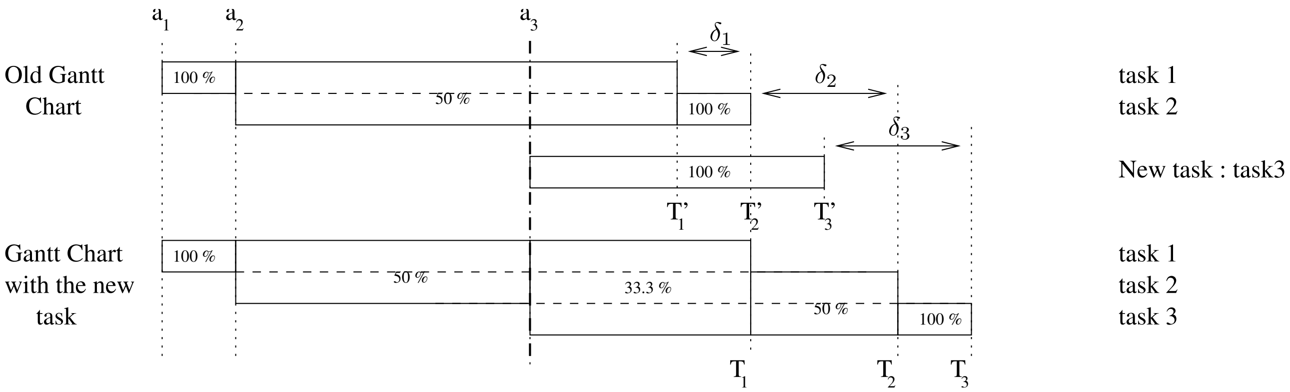

Fig. 1. Notations for the Historical Trace Manager (Above: task 3 is submitted to the server at time a_3 while it is running two tasks. Bottom: once scheduled on the server this leads to perturbations δ_i). Percentages show the amount of CPU available for each task.

average is not necessarily exactly the same on the two machines, the decision is blurred).

However, the HTM simulates the execution of the tasks on each server and the agent knows that, at time 80, the remaining duration of task T_1 is 20 seconds while the remaining duration of task T_2 is 920 seconds. Therefore it knows that scheduling T_3 on server 1 will lead to a shorter completion time than scheduling T_3 on server 2.

IV. Heuristics

We introduce here four new heuristics, HMCT, MP, MSF and ML, which are compared to MCT in next sections. The HTM simulates the new task on each server which is able to solve the problem and gives resultant information to the scheduler. Therefore, the heuristic considers the perturbation the new task will induce on each running one, and computes the "best" server accordingly.

A. Historical Minimum Completion Time

HMCT is the Minimum Completion Time algorithm relying on the HTM. For each new arriving task, the HTM simulates the mapping and the execution of the task on each server. Therefore, we have an estimation of the finishing date of this task on each server. The agent then maps the task to the server that minimizes its finishing date (Fig 2).

For each new task t
  For each server j
    Ask the HTM to simulate t on server j
    Affect to $T_{t,j}$ the termination date of task t
  Map task t to server j_0 such as $T_{t,j_0} = \min_j T_{t,j}$
  Tell the HTM that task t is allocated to server j_0

Fig. 2. *HMCT Algorithm*

B. Minimum Perturbation

Scheduled by MP, the new task is mapped to the server j that minimizes the sum of all the perturbations that the new task induces on the environment. In the case of equality, for instance when the first task is submitted to the agent, the server that minimizes the completion date of the last incoming task is chosen (Fig 3).

For each new task t
  For each server j
    Ask the HTM to simulate t on server j
    Compute $P_j = \sum_i \delta_i$
  If all P_j are equal
    Map task to server j_0 that minimizes $T_{t,j}$
  Else Map t to server j_0 such as $P_{j_0} = \min_j P_j$
  Tell the HTM that task t is allocated to server j_0

Fig. 3. *MP Algorithm*

C. Minimum Sum Flow

The heuristic uses the HTM to compute the sum of all the flows when assigning the last task t to each server. Hence, the heuristic returns the identity of server j_0 that minimizes the system sumflow, e.g. $\min_j(\sum_i(T_i - a_i))$, if assigning the task to that server. But as the difference between two values is only due to perturbations and to the new simulated task duration, the heuristic only needs to compute $\sum_i \delta_i + T_t - a_t$ for each server j, that is to say the perturbation of the last task on the server plus the manager estimated length of the new task (Fig 4). This heuristic is equivalent to *MTI* (Minimize

Total Interference) proposed by Weissman in [13].

For each new task t
  For each server j
    Ask the HTM to simulate task t on server j
    Compute $P_j = \sum_i \delta_i + T_t - a_t$
  Map t to server j_0 such as $P_{j_0} = \min_j P_j$
  Tell the HTM that t is allocated to server j_0

Fig. 4. *MSF Algorithm*

D. Minimum Length

For each new arriving task, ML requests that the HTM simulate the execution of the request on each server. After each simulation, ML computes the quantity $\sum_i D_i$. Then, it chooses the server that minimizes all these values, e.g. the server on which the sum of the remaining duration of each task, including the new one (which is its actual flow), is the minimum (Fig 5).

For each new task t
  For each server j
    Ask the HTM to simulate task t on server j
    Compute $P_j = \sum_i D_i$
  Map t to server j_0 such as $P_{j_0} = \min_j P_j$
  Tell the HTM that t is allocated to server j_0

Fig. 5. *ML Algorithm*

V. Criteria

In this section, we present metrics that have been observed when comparing our heuristics against a modelization of MCT. In GridRPC middlewares, it is important to improve the completion of the application but also the resource utilization and the quality of service for each request. This is why we do not only observe the makespan metric. Thus, we study a number of metrics from a number of fields including system environments and continuous stream. Here are the criteria we observe for each heuristic:

- the makespan: it is the completion time of the last finished task, $\max_i C_i$. The makespan is much more an application metric, for it is its completion date. So, although it is widely utilized (basically with the MCT heuristic in Legion [8], NetSolve [6]), we do not think it is the appropriate metric to use when considering GridRPC. The agent serves more than one user, so the agent does not necessarily deal with a single application, and must do its best for each of them;
- the sumflow [1]: this is the amount of time that the completion of all tasks has taken on all the resources, $\sum_i (C_i - a_i)$. Executing tasks on servers has a cost proportional to the time it takes. We can therefore consider it as a system and economics metric, for it estimates the utilization of the resources and the profit realized by using a given heuristic when the cost of each resource is the same;
- the meanflow, also called the response time: in our context, it is the average duration of a task in the system, given by the value $\sum_i (C_i - a_i)/n$ if n is the number of tasks submitted in the system. This metric is mostly considered for independent tasks and focuses on the quality of service that the system is able to deliver to each request;
- the maxflow [2]: this is the maximum time a task has spent in the system, $\max_i (C_i - a_i)$. In a loaded system, a task will generally execute longer than expected. This is even more true if it is allocated on a fast server (which is generally more solicited). This value can account for high resource contention and promote better load balance in heterogeneous platforms;
- the maxstretch [2]: this metric denotes the maximum factor, $\max_i \left((C_i - a_i)/d_i\right)$, by which a query has been slowed down relative to the time it takes on the same but unloaded server. A client can have an approximation of the minimum time its task will take on a server, but a task can require much more time than it would due to contention with previously allocated tasks and with hypothetical arriving ones. This value gives the worst case of slowdown for a task among all the ones submitted to the agent;
- the number of tasks that finish sooner: whereas this is not a metric, this value gives, in correlation to the previous metrics, a relevant idea of a quality of service given to each task when comparing two heuristics. For instance, comparing the heuristics H_1 with MCT (on the *same* set of tasks $\{t_1 \ldots t_n\}$ and *same* environment), it is $\mid \{t_i | C_{t_i\,H_1} < C_{t_i\,MCT}\} \mid$

A typical user is generally not interested in minimizing the completion time of the final task in the system (optimizing the makespan) but rather that his own tasks (a subset of all client requests) finish as fast as possible. Therefore, if we can provide a heuristic where most of the tasks finish sooner than if scheduled by MCT without delaying too much other task completion dates (that can be verified with the meanflow for example), we can claim that this heuristic, from the user point of view, outperforms MCT.

VI. Validity of the Time-Share Model and Accuracy of the HTM

In this section, we compare real tasks duration against their estimation made by simulation in the HTM.

We tackle two objectives. First, we determine the validity of the time-share model and therefore the limits of simulation. Second, we evaluate the accuracy of the HTM.

We first have experimented using the time-share model on LINUX and Solaris systems when tasks are matrix multiplications and have the same priorities (two examples, where tasks are ranked by their completion date, are given in Table I). The percentage error is defined by 100 multiplied by the absolute value of the difference divided by the real duration of the task). We have seen small variations between the simulated and real completion dates (a mean of less than 3% with regard to the duration). One can see that in this test, at most five tasks are running at the same time on the server (tasks 3, 5, 6, 7 and 8 between time 181.45 and time 204.84). Task 3 is the most impacted task as all the task run during its execution (it started while task 1 had not yet finished and ended after the start of task 9).

Next, we performed tests to measure the accuracy of the model as the number of concurrent processes changes. In order to perform these tests, we have integrated our HTM into the code of the agent in NetSolve. We have performed 100 experiments where 500 non identical jobs have been requested. Indeed, a job can require 20, 35 or 50 seconds on the unloaded server. The real and the HTM estimated duration of each task during each experiment have been recorded. In figures 6, 7 and 8, we observe the following information:

- in dark dots: the ratio for each task of the HTM estimated completion date divided by the real post-mortem one, indexed by the submission date on the x-axis. Hence, the closer to 1 is the ratio, the most accurate the prediction is;
- in light dots: the number of tasks that have interfered during the task execution.

We have found three different behaviors.

In Figure 6 we see that the HTM predicts accurate completion dates of previously assigned and still running tasks, taking into account interferences tasks have on each other.

In Figure 7, estimations are degrading with the load of the server. Indeed, until 2800 seconds, estimations are more than 96% accurate on the average and still 93%

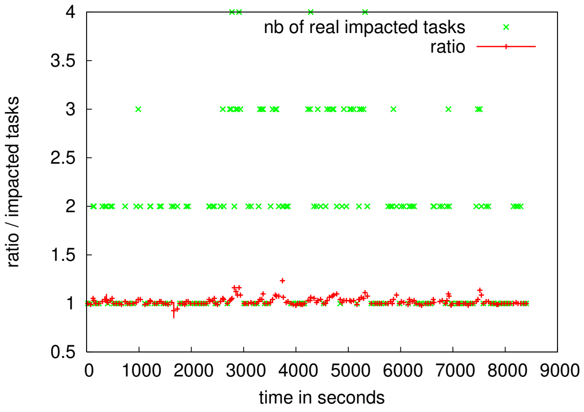

Fig. 6. Independent Tasks Submission with MP

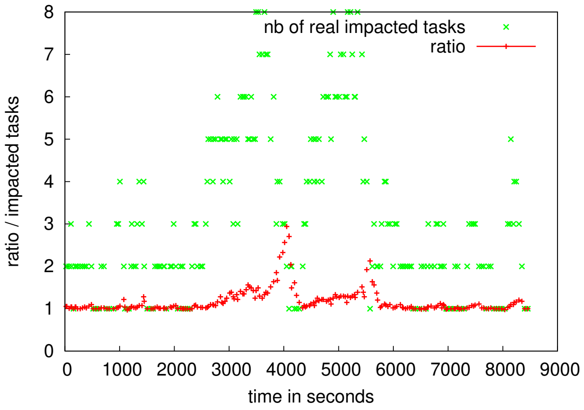

Fig. 7. Independent Tasks Submission with MSF

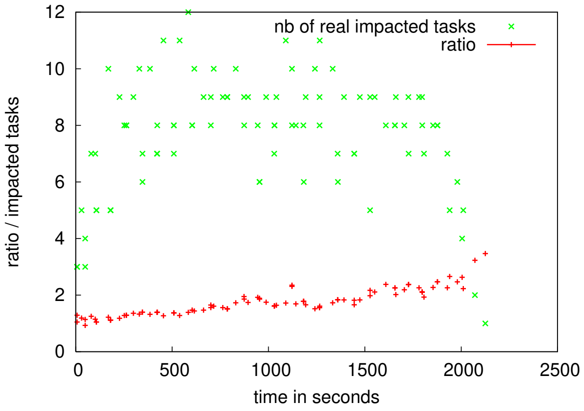

Fig. 8. Task Graph Submission with HMCT

accurate until 3156 seconds. Then, more than 6 tasks are simultaneously executed on the server, thus increasing

Task	Arrival Date	Size of the Matrix	Real Completion Date	Simulated Completion Date	Difference	Percentage of Error
1	33.00	1500	80.79	79.99	0.8	1.7
2	59.92	1200	92.08	93.19	-1.11	3.4
3	73.92	1800	142.79	142.50	0.29	0.4
1	29.41	1500	76.69	76.29	0.4	0.8
2	56.43	1200	89.15	89.50	-0.35	1
4	96.41	1200	136.97	139.40	-2.43	5.9
6	140.41	1200	204.84	204.85	-0.01	0.02
3	70.42	1800	210.61	195.74	14.87	10.6
5	121.43	1500	235.38	232.92	2.46	2.2
8	181.45	1200	248.02	248.56	-0.54	0.8
9	206.41	1200	259.91	261.63	-1.72	3.2
7	166.42	1800	289.08	288.91	0.17	0.1

TABLE I
TWO TIME–SHARE MODEL EXPERIMENTS

prediction errors. The HTM regains a high accuracy when the load decreases. The desynchronization and resynchronization of the HTM occurs either when the rate is much too fast for the environment or when the heuristic tends to overload some servers. We should also note that the HTM always predicts tasks flow greater than in reality.

The results in Figure 8 correpond to an execution of numerous task graphs on the environment. We observe that HTM estimations are always higher than real durations. A high number of tasks, more than 6 tasks on the average, are executed by the server at the same time. This induces high delays for the tasks scheduled on this server and creates errors. Due to precedence relationships, errors are cumulative, causing the accuracy of the predictions to degrade with time. This happens only for some specific scheduling heuristics that overload servers with independent tasks.

Because the HTM is an environment simulator, results induce that independent tasks simulations are relevant to foresee what can be expected in reality. Indeed, the HTM and the Simgrid tool [5] for example use similar simulation mechanisms. Nonetheless, there are some limitations due to the arrival rate, to the heterogeneity (tasks and servers) and to the heuristic: accuracy is obtained if less than 5 tasks are executed simultaneously on a server.

In conclusion, concerning simulating tasks in the time-share model (as performed by the Simgrid tool [5] for instance), we see that the accuracy of the model depends on the number of simultaneously running tasks. When too many tasks are running at the same time, the time-share model over-estimates the task duration. Therefore, the HTM, which is based on this model is highly accurate when the number of simultaneous running jobs is lower than 5. When this number is greater than 6 the accuracy may degrade. For independent tasks submission, the HTM is able to remain highly accurate and its accuracy does not degrade later on. In the case of some task graphs and for some specific heuristics the error is cumulative. Despite this, we will see in the next section that heuristics based on the HTM are able to give good results.

VII. INDEPENDENT TASKS SUBMISSION: SIMULATION RESULTS

In the previous section, we have shown that when submitting a set of independent tasks, the time share model is reasonably accurate to allow simulations.

A. *The Simulated Environment*

A Problem Solver Environment is simulated with the Simgrid tool [5]. The dynamic mapping heuristics are evaluated using some parameters that characterize heterogeneous servers and each task of the set of independent tasks. We explain in this section the different models that we used for client-agent-server mechanisms, heterogeneous entities and independent tasks. We used the Gnu Standard Library [7] for all the probabilistic distributions used thereafter.

We assume the agent has perfect knowledge of the following information:

- current server load;
- current network load;
- peak CPU and network performances;
- number of operations of any tasks;
- size of the input and output data of any tasks.

MCT needs all these information while the Historical Trace Manager (and consequently, our heuristics) needs only the static ones (last three items). Static information is easier to compute accurately as compared to dynamic (for example, tasks durations can be obtained from benchmarks or from means of executions performed on each server [12]). Therefore, in our simulations, the performance that MCT will obtain using these information will be better than in any real GridRPC middleware.

1) Platform Model Characterization: We assume that the client and the agent are able to reach each server. Experiments are conducted on the basis that servers are dedicated to the environment (for instance a set of clusters where reservation is possible). Moreover, we suppose that all problems can be solved on any server. We do not consider any arrival/withdrawal of any server in the system.

2) Application Model Characterization: We consider independent task submissions (tasks that have no precedence relation), requested by one or more users. A task is not preemptible: it can not be suspended nor be removed to be scheduled on another server.

A task begins to be executed on a server as soon as the total amount of the input data has been received by the server. We consider that a task is finished when the data output is completely received by the client. We want to test how the heuristics react under different environments and we want to achieve the best possible overview when comparing each of them against MCT. Therefore, same environments and same task sets are generated for each heuristic: comparisons are conducted on the same sets of tuples (servers, tasks, arrival dates, etc.).

Experiments are conducted with a number of servers held to 25. They have one processor and their computing capacity is drawn from a uniform distribution in the range $[150000, 500000]$. Their network cards have the same performance (100Mbits/sec).

We use a uniform distribution to draw from the range [1Kb,300Mb] the input and output sizes of data to be transferred between the client and the server. The computation cost is generated from a uniform distribution, with the following rules:

- the computation phase costs more than 10 times than the transfer phase;
- the computation phase must not be greater than 600 seconds on the fastest server of the 25 available.

To draw task arrival dates, we use a Poisson distribution whose parameter μ varies from 0 to 70. Simulation results indicate a practical limit at $\mu < 10$: for these rates, MCT schedules more than five tasks on a server, and results are consequently distorted. On the other hand, when $\mu = 70$, each server is executing at most one task at a given moment for half of the heuristics tested. For each value of μ, we have varied the number of tasks from 10 to 250. Each plot in a graph is the mean of results of 1000 different tasks sets: each heuristic required 200000 simulations.

B. Results

Results are given in Figures 9, 10, 11 and 12. Each is composed of five 3D graphics, one for each metric. They present the gain (in percentage) of the heuristic over MCT. For example, the first graph shows the percentage of 'tasks that finish sooner'. It shows a gain if the value is greater than 50. In the others, a benefit is realized as soon as the percentage is positive. Note that due to the kind of the submission, the gain demonstrated on the sumflow is the same than the one obtained on the response time.

Considering that the makespan of an application composed of independent tasks is mainly due to the last completed task starting time [9], [3], we cannot expect *a priori* a great gain on the makespan.

To compute servers scores in our simulations, MCT uses information which is far more precise than in a real environment such as NetSolve. In consequence, our modelization of MCT behaves better than the algorithm does in reality. Therefore, if we build a heuristic that outperforms our simulation of MCT, this heuristic will certainly outperform MCT in reality.

1) HMCT: The goal of this heuristic is the same as MCT's: it expects to minimize the makespan of the application by minimizing the completion date of incoming tasks.

Figure 9 shows that there is a gain greater than 8% on the makespan for $\mu \leq 20$. For $\mu \geq 30$ the gain is still positive even if almost null. The sumflow is greater for HMCT than MCT for $\mu > 10$ (leading to degraded performance). Because HMCT tends to optimize the use of fast servers for new tasks, it delays the running ones (for $\mu \leq 10$, the rate is so high that the flow is increased for both MCT and HMCT). It also results in a percentage of 'tasks that finish sooner' lower than 50% (under 20% for $\mu = 30$). For $\mu \geq 40$, the maxstretch shows that a task is 30% longer than if scheduled with MCT: there is contention for fast servers, even at a low rate.

The main drawback of this heuristic is that it tends to overload the fastest servers, which has two effects: it unnecessarily delays task completion dates and in a highly heterogeneous environment, servers may collapse, mainly due to a lack of memory.

2) Minimum Perturbation: MP: MP aims to provide a better quality of service to each task by delaying as little possible already allocated tasks.

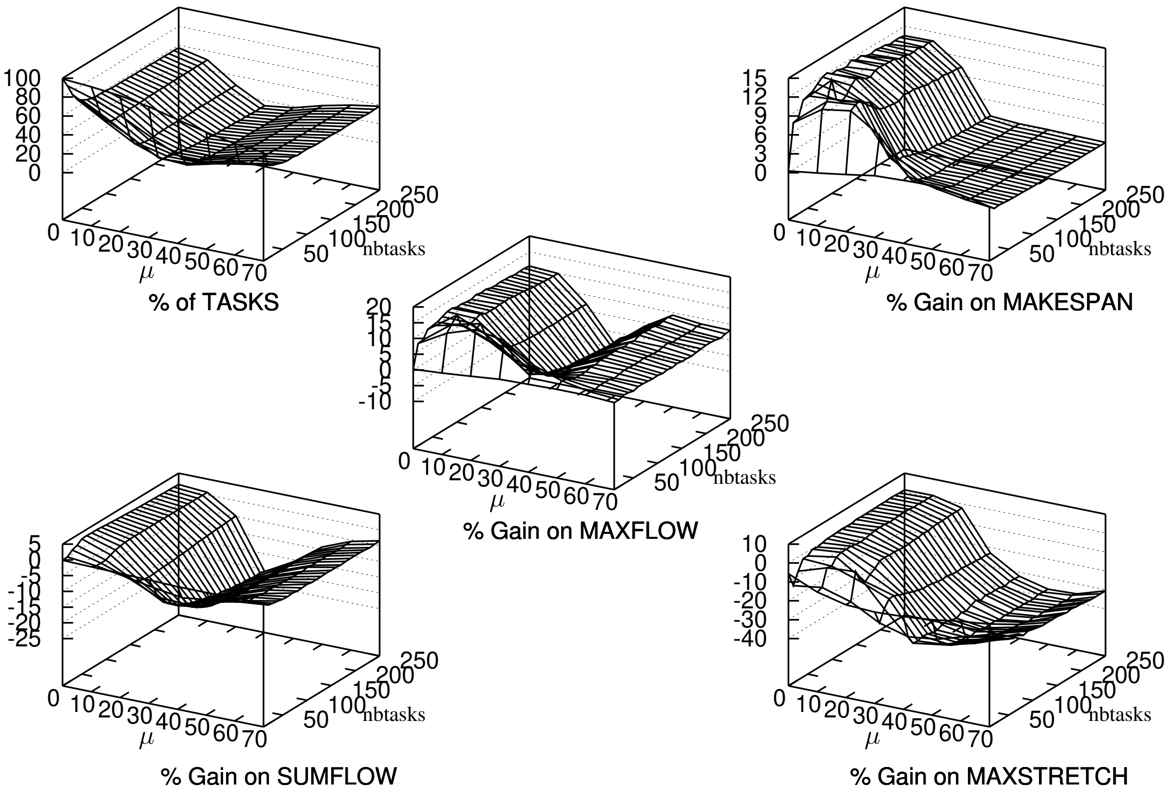

Fig. 9. Results for HMCT vs. MCT on 25 servers

Figure 10 shows that when $nbtask > 80$, the gain on the makespan is positive (greater than 5% for $nbtask = 250$ and $\mu < 30$). MP allows gain on the sumflow, with a peak at 15% for $\mu = 30$. The percentage of 'tasks that finish sooner' is always greater than 60%, with a peak at 70% for $\mu = 30$, (when in most cases, no more than one task is running on a given server). Due to the use of slower servers, there is always a gain on the maxstretch: in some cases, the gain reaches 90%. The maxflow is always better except for $\mu = 30$.

Despite these good results, MP presents a major drawback: when only one server is idle, it is chosen regardless of its speed, possibly jeopardizing its performance. This happens when dealing with highly heterogeneous resources and/or a high rate of costly requests: for example, when $\mu < 30$ and $nbtasks < 50$, MP is outperformed by MCT on the makespan.

3) Minimum Sum Flow: Minimum Sum Flow is an attempt to mix the advantages of HMCT and MP (to keep the makespan objective of HMCT, with less risk of collapsing fastest servers, and to give a better quality of service to each task) and to reduce the cost of resources.

MSF performs well on the makespan, around 9% better when $\mu \leq 20$, against MCT (Fig 11). The percentage of 'tasks than finish sooner' is slightly worse than MCT's for $25 \leq \mu \leq 50$, but is always above 40%. For $\mu \geq 35$, tasks are slowed down by a maximum factor of 20% (occuring when $50 \leq \mu \leq 60$). Sumflow performances depend on the parameter μ: MSF maximizes the use of fast servers when $30 \leq \mu \leq 40$ and thus delays tasks. It induces a greater flow cost. But when $\mu \leq 30$, MCT overloads faster servers and MSF, taking into account interferences, performs slightly better than MCT.

4) Minimum Length: Except for $\mu = 0$, the gain on the makespan is always positive (Fig 12). ML has also a positive gain on the sumflow, with a peak at 15% when $\mu = 10$. On the maxstretch, MCT performs slightly better for $20 \leq \mu \leq 40$ and yielding around 60% on the percentage of 'tasks that finish sooner', with a peak at 80% for $\mu = 50$.

ML achieves a makespan at least as good as MCT for $\mu \neq 0$, an adequate performance on the maxstretch and

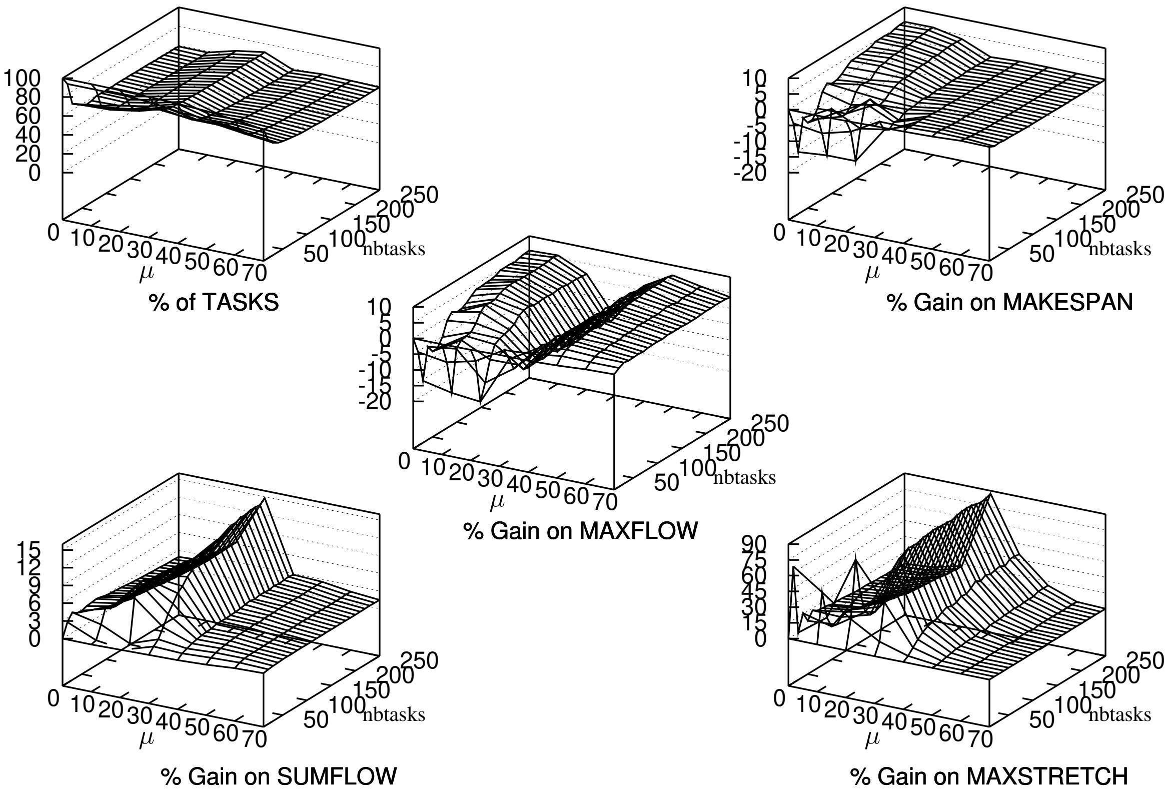

Fig. 10. Results for MP vs. MCT on 25 servers

gains on the sumflow, the maxflow and on the percentage of 'tasks that finish sooner'. Since the mapping decision is partly made using the cost of the new task, ML does not exhibit MP's drawback. Therefore, this heuristic tends to overcome the drawbacks of HMCT and MP, while keeping the advantages of both.

C. Discussion

To summarize our heuristics performances, we can firstly conclude that our heuristics outperform MCT on the makespan. However, the percentage of 'tasks that finish sooner' of HMCT is poor and the maxstretch is always negative. In a client-agent-server context, this needs to be improved. MP shows great performances on every metrics but can present the drawback that it unnecessarily utilizes slow servers. MSF behaves better than MCT. Indeed, it combines HMCT and MP performance. It outperforms ML on the makespan and on the maxflow, but loses on the other metrics. ML tends to overcome HMCT's and MP's drawbacks in addition to delivering good performances on every metrics, except when all tasks are submitted at the same time (e.g. $\mu = 0$, an almost impossible situation).

Since ML has positive results on all observed metrics and outperform all the other heuristics on the percentage of 'tasks that finish sooner than MCT', we consider that overall it is the best one among all the tested heuristics for this section.

VIII. Task Graphs Submission: Real Platform Results

In section VI, we have shown that the time share model may not be very accurate when submitting task graphs. Therefore, we have tested our heuristics on task graphs on a real platform.

The HTM and the heuristics HMCT, MP, MSF and ML have been implemented in the NetSolve agent in order to study the accuracy of the HTM predictions and to compare the heuristics with the MCT algorithm present as default scheduling heuristic in NetSolve [6].

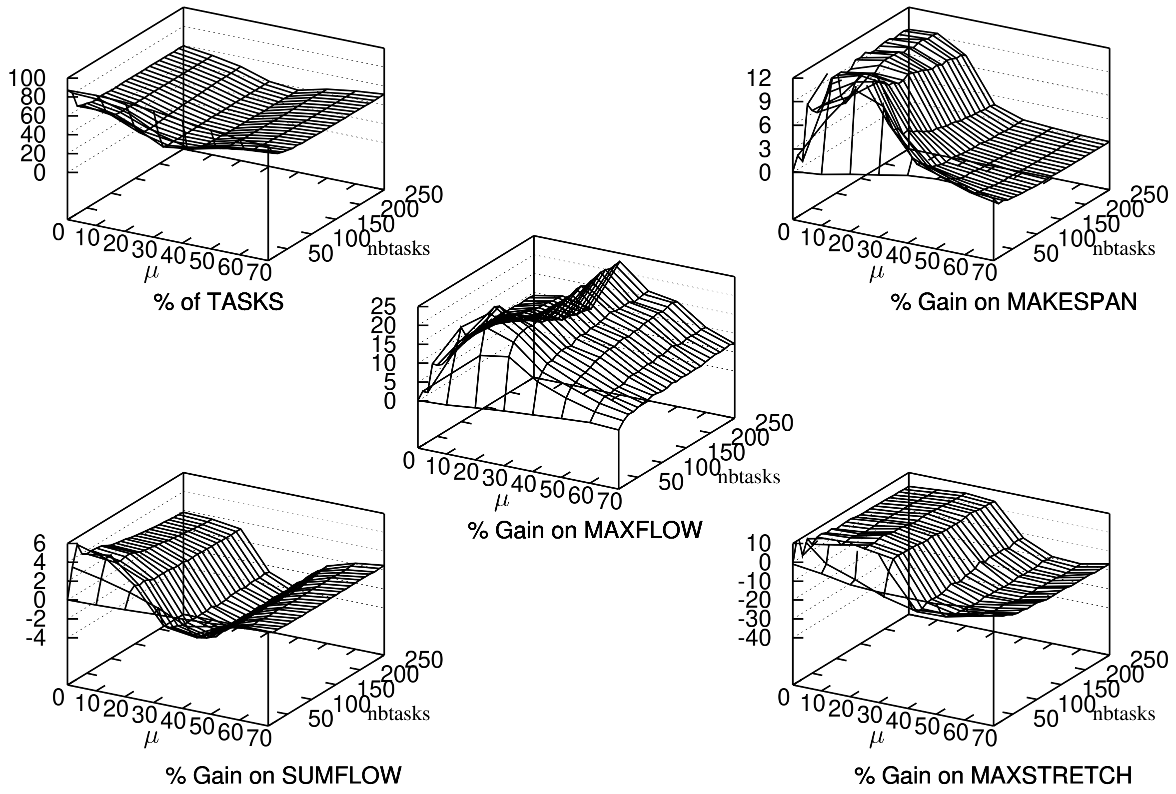

Fig. 11. Results for MSF vs. MCT on 25 servers

Type	Machine	Processor	Speed	Memory	Swap	System
Server	spinnaker	xeon	2 GHz	1 Go	2 Go	Linux
	artimon	pentium IV	1.7 GHz	512 Mo	1024 Mo	Linux
	soyotte	sparc Ultra-1		64 Mo	188 Mo	SunOS
	fonck	sparc Ultra-1		64 Mo	188 Mo	SunOS
Agent	xrousse	pentium II bipro	400 MHz	512 Mo	512 Mo	Linux
Client	zanzibar	pentium III	550 MHz	256 Mo	500 Mo	Linux

TABLE II
EXEPRIMENTAL PLATFORM

A. Experiments

Several experiments have been conducted onto an heterogeneous environment whose resources are given in Table II.

Three types of tasks have been used for these experiments. The duration of each task on each server is given in Table III. Tasks are computing intensive and require less than one second of data transfer.

Table IV gives a summary of the series of tests that we have performed. 8 scenarii (from (a) to (h)) have been submitted to the platform. Two types of graphs have been used: 1D meshes and stencil graphs. Each stencil graph is submitted by at most one client, while numerous 1D meshes are submitted by different clients (5 or 10). Stencil graphs have a width of 5 or 10 and a depth ranging from 25 or 50.

In scenarii (c), (f), (g) and (h), independent tasks are submitted by other clients during the execution of the graph. The duration of each task has a uniform probability [7] to be of one of the three types given in Table III. In this case, inter-arrival rate is drawn from a Poisson distribution of parameter μ.

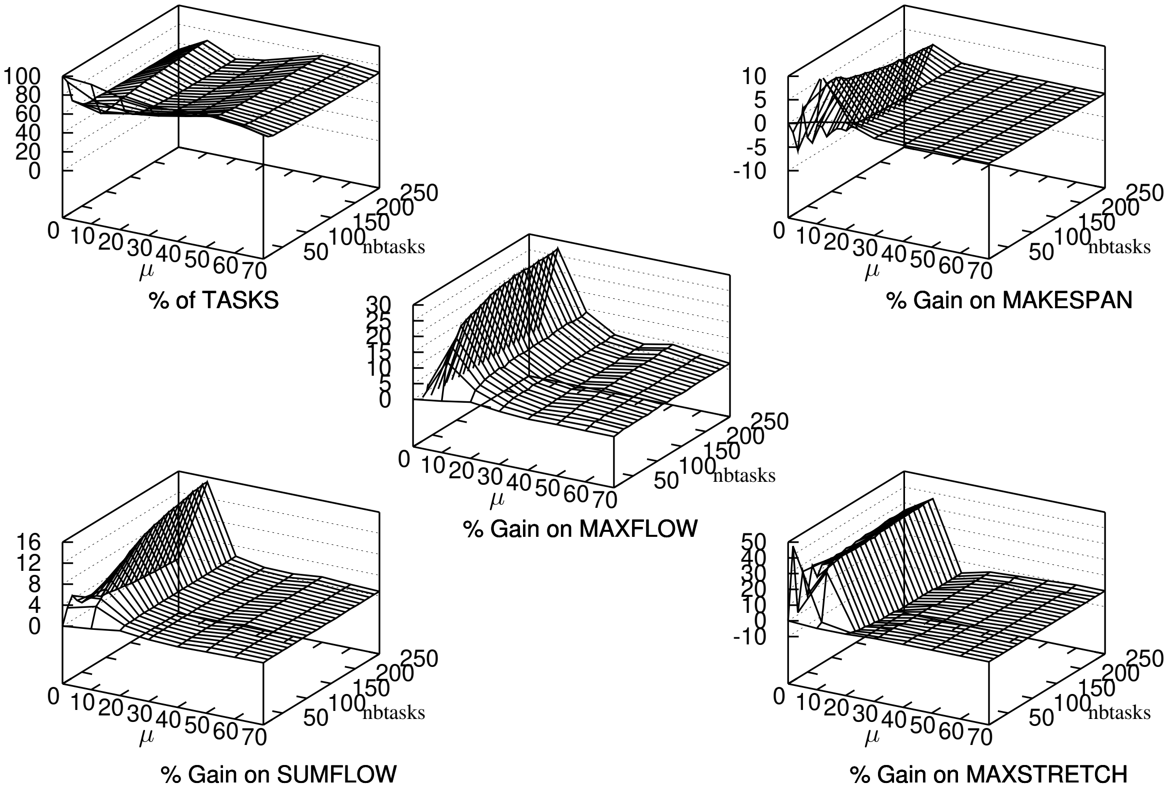

Fig. 12. Results for ML vs. MCT on 25 servers

Server	Task 1	Task 2	Task 3
spinnaker	15	30	43
artimon	17	33.5	49.5
soyotte	128	256	382.5
fonck	127.5	254	380.9

TABLE III
DURATION OF THE TASKS ON THE UNLOADED SERVERS

Each scenario is randomized by a different seed and runs 6 times per heuristic. 6 runs were sufficient for the average values given in Section VIII-B to be stabilized. This leads to 720 experiments and more than 55 days of computation.

B. Comparison Between Heuristics

For each scenario we compare each heuristic on three criteria. MCT is used as the comparison baseline.

First, we compare the application completion time, e.g. the *makespan*, this comparison is done for all the scenarii.

Second, when independent tasks are involved (scenarii (c), (f), (g) and (h)), we compare the average *response time*, e.g. in our context (see section II), the mean flow of the task. The response time is only measured for independent tasks (the response time of a graph is its makespan).

Third, when independent tasks are involved we give the percentage of them that finish sooner if scheduled with our heuristic than with MCT. This, in relation to the response time, gives information about the quality of service proposed by the heuristic. Indeed, if a heuristic is able to give a good response time and if a majority of tasks scheduled with this heuristic finish sooner than if scheduled with MCT then, the client sees a better quality of service with this heuristic.

Figure 13 shows the average gain on the makespan performed by each heuristic over MCT *for each client* and for each scenario. We see that HMCT, MSF and ML always outperform MCT up to 25%. MP outperforms HMCT, MSF and ML when applications are 1D-mesh graphs (scenarii (a), (b) and (c)). However, MP shows

Scenario	Application(s)		Independent Tasks		Experiment	
	nbclients	widht x depth	nbtasks	μ (sec)	nbseeds x nbrun	total nbtasks
(a) 1D-mesh	10	1x50	-	-	4 x 6	500
(b) 1D-mesh	10	1x variable	-	-	4 x 6	500
(c) 1D-mesh + 250 independent tasks	5	1x50	250	20	4 x 6	500
(d) stencil (task 1)	1	10x50	-	-	1 x 6	500
(e) stencil (task 3)	1	10x50	-	-	1 x 6	500
(f) stencil + 175 independent tasks	1	10x25	175	28	2 x 6	425
(g) stencil + 87 independent tasks	1	10x25	87	40	4 x 6	337
(h) stencil + 87 independent tasks	1	5x25	87	25	4 x 6	212

TABLE IV
SCENARII, MODALITIES AND NUMBER OF EXPERIMENTS

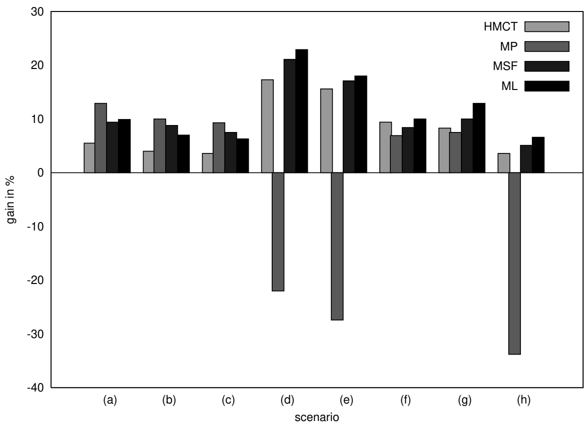

Fig. 13. Gain on the Makespan for each Client of each Scenario

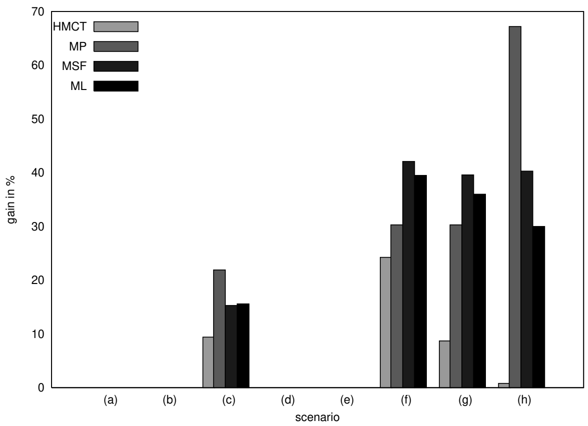

Fig. 14. Gain on the Response Time for each Client of each Scenario

negative performance for some stencil graphs scenarii. This is explained as follow. Due to its design, MP can unnecessarily map a task on a slow server, for example when it is the only idle one. This delays the completion of the graph because some critical tasks are mapped on slow servers. The client must wait for the completion of these tasks before being able to resume the execution of the application.

Figure 14 shows, for each heuristic, the average gain on the response time that *each task* perceives when scheduled with the considered heuristic. We see that HMCT, MP, MSF and ML outperform MCT for all the scenarii. On scenarii (f) and (g), MSF is the best heuristic with a gain up to 40%. On scenarii (c) and (h), MP is the best heuristic, with more than 65% of gain in scenario (h). For this scenario HMCT shows nearly no gain. ML achieves poorer performance than MSF, except on scenario (c) where it gives slightly the same performance.

The average percentage of tasks that finish sooner than MCT is given for each heuristic in Figure 15. On the opposite of the two former figures where a gain is observed as soon as the value is positive, the percentage here has to be superior to 50% to express a gain. We see that, as for the response time, all of our heuristics always outperform MCT. ML is the best heuristic for scenarii (f) and (g). MP is the best heuristic for scenarii (c), and (h). On scenario (h), about 90% of all the tasks finish sooner when scheduled with MP than if scheduled with MCT. For this scenario HMCT has the same performance as MCT.

Figure 14 and Figure 15, show together that MP, MSF and ML are able to offer a good quality of service to each independent task. This means a task scheduled by one of these three heuristics, on the average, finishes sooner and the average gain is high.

Experiment (h) shows that MP is outperformed by MSF and ML on the makespan but outperforms both of them on the quality of service. This is due to the fact that MP is better at scheduling independent tasks

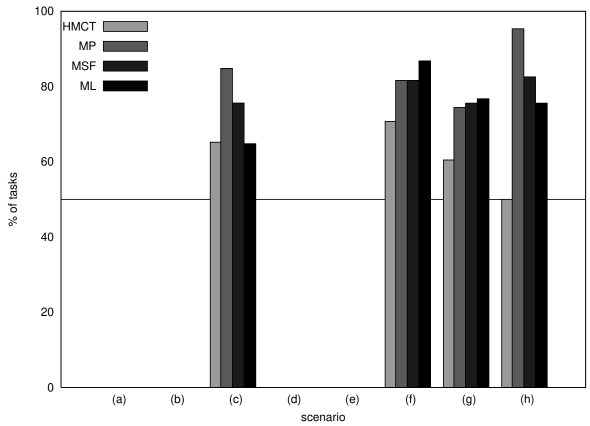

Fig. 15. Percentage of Tasks that Finish Sooner for each Scenario

and the two others are better at scheduling application graphs. Indeed, due to the use of slower servers by MP, the submission of the graph can suffer dramatically when waiting for a critical task to complete thus benefiting independent tasks which profit of the whole platform. Because MSF and ML take into account for their decisions the flow that the new task will require. They use slow servers only under heavily loaded conditions.

As a conclusion, in an heterogeneous GridRPC environment, MSF and ML gives the best results regardless the makespan of applications or response time of independent tasks. They achieve a higher number of tasks finishing sooner than when they are scheduled with MCT. This is more beneficial to *each client*, whose concern is that his own tasks, a subset of all task in the environment, finish as fast as possible. Depending on the main metric that the environment must optimize, ML can be employed to prefer application completion over independent tasks and MSF for the opposite, both of them giving good results in the other metrics.

Furthermore, we see that the "heuristic hierarchy" is conserved between simulation and real experiments. Both ML and MCT are shown to combine MP and HMCT advantage without the drawbacks.

IX. Conclusion

In the paper, we have addressed the problem of scheduling tasks in GridRPC systems. Our work relies on a non-intrusive predictive module called the Historical Trace Manager (HTM), which simulates the execution of tasks on the environment with the time-share model.

We have presented four heuristics: Historical MCT, which is MCT using the far more precise HTM estimations, Minimim Perturbation, which tries to minimize the delays that tasks induce on each other, Minimum Sum Flow, which tries to minimize the sum of all task durations and Minimum Length, which only considers the remaining duration of all the tasks at the submission date of the new request (including it).

We have shown that the HTM is precise for independent tasks. When dealing with task graphs, the HTM is accurate when the server load is not too high. In some cases, the HTM accuracy degrades during the execution. This is due to the error accumulation. It can regain accuracy without synchronization, depending on the heuristic and the rate/kind of submission.

Our approach is an experimental one. Almost two months of continuous computation (for simulation and real world tests) have been performed in order to evaluate our heuristics.

We have tested two kinds of submissions: independent tasks and task graphs. For independent tasks, we have performed simulation experiments. For task graphs, the time-share model is not accurate enough: we have tested our heuristics on a real platform.

Our heuristics were compared to Minimum Completion Time. MCT is a widely used heuristic designed for mono-client environment where servers executes tasks sequentially which is not the case in most GridRPC environments.

We have performed an extensive simulation study on five metrics and various parameters. Our heuristics show better performance against the idealized version of MCT, which has a far better understanding of the environment than in reality. In this case, ML is the best overall heuristic.

For our real world tests, we have used NetSolve, a GridRPC environment, in which we have implemented the HTM and all of the four heuristics. We have performed an extensive study of our heuristics on a real platform, facing the same set of experiments. Numerous scenarii involving different kinds of submissions, with different application graphs running concurrently with an independent task workload, have been submitted to the modified NetSolve agent.

Results show that our heuristics outperform MCT on at least two of the three observed metrics: makespan, response time and percentage of tasks finishing sooner. Depending of the main metric the environment must optimize (thus depending of the kind of average submissions it will encounter), MSF or ML appear to be the best overall heuristics as they outperform MCT on all the criteria and do not present any drawback, like MP does.

In conclusion, ML is shown to be the best of the proposed heuristics if the application makespan is preferred to the quality of service that the environment can possibly give to each request and MSF in the opposite case. Both of them still give good performances on the other metrics.

In our future work, we will implement a mechanism to synchronize the HTM to the reality. This mechanism will be based on the task completion date. The goal is to limit the effect of error accumulation seen in some experiments. Early developments show an improvement in the stability of HTM prediction accuracy.

REFERENCES

[1] K. R. Baker. *Introduction to Sequencing and Scheduling*. Wiley, 1974.

[2] M. A. Bender, S. Chakrabarti, and S. Muthukrishnan. Flow and stretch metrics for scheduling continuous job streams. In *SODA: ACM-SIAM Symposium on Discrete Algorithms*, 1998.

[3] Y. Caniou and E. Jeannot. Dynamic mapping of a metatask on the grid: Historical trace, minimum perturbation and minimum length heuristics. Technical Report 4620, LORIA, nancy, oct 2002.

[4] A. Caron, F. Desprez, E. Fleury, F. Lombard, J.-M. Nicod, M. Quinson, and F. Suter. *Calcul réparti à grande échelle*, chapter Une approche hi´rarchique des serveurs de calculs. Hermès Science Paris, 2002. ISBN 2-7462-0472-X.

[5] H. Casanova. Simgrid: A toolkit for the simulation of application scheduling. In *Proceedings of the IEEE Symposium on Cluster Computing and the Grid (CCGrid'01)*. IEEE Computer Society, may 2001. Available at `http://www-cse.ucsd.edu/~casanova/`.

[6] H. Casanova and J. Dongarra. Netsolve : A network server for solving computational science problems. In *Proceedings of Super-Computing -Pittsburg*, 1996.

[7] M. Galassi and J. Theiler. The gnu standard library, 1996. Availble at `http://www.gnu.org/software/gsl/gsl.html`.

[8] A. S. Grimshaw and W. A. Wulf. The legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1):39–45, 1997.

[9] M. Maheswaran, S. Ali, H. J. Siegel, D. Hengsen, and R. F. Freund. Dynamic mapping of a class of independent tasks onto heterogeneous computing system. In *Journal of Parallel and Distributed Computing, Special Issue on Software Support for Distributed Computing*, June 1999.

[10] H. Nakada, S. Matsuoka, K. Seymour, J. Dongarra, C. Lee, and H. Casanova. A gridRPC model and API for end-user applications. December 2003. Available at `https://forge.gridforum.org/projects/gridrpc-wg/document/GridRPC_EndUse%r_16dec03/en/1`.

[11] H. Nakada, M. Sato, and S. Sekiguchi. Design and implementations of ninf: towards a global computing infrastructure. *Future Generation Computing Systems, Metacomputing Issue*, 15:649–658, 1999.

[12] M. Quinson. Dynamic performance forecasting for network-enabled servers in a metacomputing environment. In *International Workshop on Performance Modeling, Evaluation and Optimization of Parallel and Distributed Systems (PMEO-PDS'O2)*, april 15-19 2002.

[13] J. B. Weissman. The interference paradigm for network job scheduling. In *Proceedings of the 10th International Parallel and Distributed Processing Symposium, HCW*, 1996.

[14] R. Wolski, N. T. Spring, and J. Hayes. the network service : a distributed resource performance forecasting service for metacomputing. *Journal of Future Generation Computing Systems*, 15(5-6):757–768, october 1999.