New Dynamic Heuristics in the Client-Agent-Server Model

Yves Caniou*
LORIA, INRIA-Lorraine
France
caniou@loria.fr

Emmanuel Jeannot
LORIA,
Université Henri Poincaré
France
ejeannot@loria.fr

Abstract

MCT is a widely used heuristic for scheduling tasks onto grid platforms. However, when dealing with many tasks, MCT tends to dramatically delay already mapped task completion time, while scheduling a new task. In this paper we propose heuristics based on two features : the historical trace manager *that simulates the environment and the* perturbation *that defines the impact a new allocated task has on already mapped tasks. Our simulations and experiments on a real environment show that the proposed heuristics outperform MCT.*

Keywords : time-shared resources, dynamic scheduling heuristics, historical trace manager, perturbation, MCT

1. Introduction

Recent developments in grid environment have focused on the need to efficiently schedule tasks onto distributed computational servers.

Thus, environments based on the client-agent-server model such as NetSolve [5], Ninf [8] or DIET [3] are able to distribute client requests on a set of distributed servers. Performances of such environments greatly depend on the scheduling heuristic implemented.

For instance, NetSolve (among many other Problem Server Environments) uses MCT (Minimum Completion Time) [10] to allocate tasks. MCT tries to map each task to the resource that finishes that task the soonest. Since in that kind of environment, each resource can execute more than one task at a time, MCT can map a task to a resource that already executes previously allocated tasks. In that case, MCT's main drawback is that it does not take into account the *perturbation* the new mapped task induces on already allocated and running tasks. Indeed, already running tasks will have less CPU power and therefore will finish later than expected. Thus MCT tends to delay too many task completion dates, which is not what users want. Moreover, MCT tends to minimize the makespan. The makespan is an application-centric metric, which in the case where many users request to have tasks scheduled by the same agent on the same resources, is not suited for grid computing. Indeed, in grid-computing environments task throughput as well as system metrics have to be optimized. In this paper, we tackle the problem of dynamically scheduling a set of tasks onto distributed resources in the client-agent-model. Our goal is to propose heuristics that are able to optimize different metrics (sum-flow, max-flow, max-stretch), without degrading the makespan. We propose and study heuristics based on an *historical trace manager* that is able to keep track and simulate the execution of the tasks on all resources. The historical trace manager predicts the *perturbation* a task can induce if mapped on a given server. Our simulations and experiments on real grid environment show that its use is relevant and combining it to heuristics based on minimizing the perturbation leads to better performances than MCT.

The remaining of this paper is organized as follows. In section 2, we describe models, the historical trace manager and define the perturbation. In section 3, we define the metrics we have observed and tried to optimize. In section 4, we present and discribe three heuristics based on the historical trace manager. Experiments and discussion are the object of section 5. Section 6 references related works. Finally, we conclude in section 7.

2. Models

The heuristics proposed in section 4 are conceived and studied for the Client-Agent-Server model. They focus on shared resources, aiming at better exploiting and less perturbing the system. These notions are explained

*This work is partially supported by the Région Lorraine, the French ministry of research ACI GRID

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 1. Two metatask executions

in this section.

2.1. Client-Agent-Server Model

The client-agent-server model is an extension of the client-server model where the agent plays a central role for dispatching client requests on already registered servers. Next, we will indifferently use the terms 'server', 'agent' and 'client' to refer to both the program and the machine or user.

Some *grid middlewares* rely on the Client-Agent-Server model, like PSE (Problem Solving Environment) built on top of grid RPC[1], allowing the use of distant, optimized numerical libraries.

In this model, the agent is the central part. 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 for computational resources. It asks the agent to find a set of the most suitable servers that are able to solve its problems.

Several tools instantiate this model like NetSolve, Ninf, DIET ([5], [8], [3]).

In this approach, the agent is launched first, then the servers can register to 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 performs the request to the server using an RPC–like call, by sending the input data. When the chosen server has completed the request it sends back the output data to the client.

[1]Remote Procedure Call

In this diagram, 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.

2.2. Information Model

In order to select the "*best*" possible server, the agent's scheduler needs accurate information on the system state (dynamic information) as well as on the problem and on the servers (static information).

Dynamic information concern each server (current CPU load average, current bandwidth and latency between the agent and the server).

Static information concern each server (CPU and network peak performances) and problem descriptions (size of input and output data as well as the task cost : number of operations requested to perform the problem).

Dynamic information are computed by monitors. A NetSolve server runs its own monitors. The agent relies on the information sent by the server but may also use monitors beforehand installed such as NWS [12].

In NetSolve, the communication time is computed by dividing the size of the data by the bandwidth and adding the latency. The computation time is evaluated by dividing the fraction of the currently available CPU speed by the task cost.

2.3. Shared Resources 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, even if it depends on the scheduling policy, the agent will be very likely to often select the fastest servers).

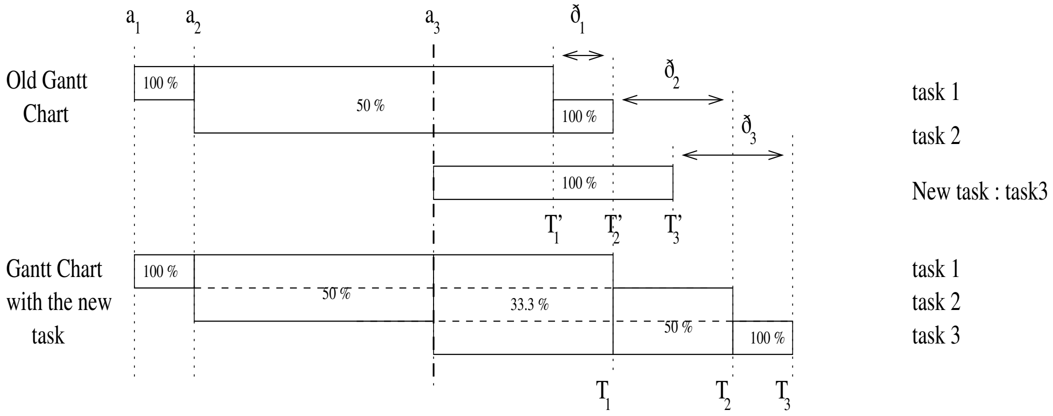

Figure 1. Notations for the historical trace use

This is true even if servers are dedicated to the grid middleware.

We consider a simple but realistic model when a server executes n tasks : each task is given $1/n$ of the total power of the resource. This model does not take into account the priorities of tasks (each task is supposed to have the same importance). We have experimented this model on LINUX systems when tasks are matrix multiplications and have the same priorities (two examples are given in Table 1. The percentage error is defined by 100 multiplicated by the absolute value of the difference divised by the real duration of the task). We have shown small variations between the simulated and real execution dates (a mean of less than 3% with regard to the duration).

We have designed a *historical trace manager* (HTM) that stores and keeps track of information about each task. It simulates the execution of tasks on resources and is able to predict the completion time of each task assigned to a server. It is used by our scheduling heuristics.

In order to do this, the HTM performs a discrete simulation of the execution of each task : all tasks mapped on a given server progress at the same speed until a new task arrives or a running task finishes.

The HTM can therefore build or update the Gantt Chart for each server when a new incoming task is mapped. This can lead to accurate prediction of the finishing time of the tasks assigned to a server. Indeed, when assigning a new task, the HTM does not consider that the load of the server is constant to the one at the arrival date of the new task.

The new or updated Gantt chart can be used by the agent to schedule the tasks more accurately.

The simulation of the distributed environment is performed for each three parts of the tasks, input data transfer, computing phase, output data transfer, for each server (fig 1).

Usefulness of the HTM Here follows an example that shows how the Historical Trace Manager can help in taking 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 T_1 and T_2, whose durations on each server is 100 and 1000 seconds respectively, with no input data. Let the agent schedules T_1 on the server 1 and T_2 on the server 2 for example. At time 80, let a client request the agent to schedule a task T_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 are not able to decide which is the best server to schedule T_3 (in practice, as there are dynamic information and as the evolution of the load average is not necessarily exactly the same on the two machines, the decision is blurred).

However, the historical trace manager simulates the execution of tasks on each server and the agent knows that 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.

2.4. Terms and Notations

Let a server be loaded with n tasks. The oldest one, not yet finished when a new one arrives is of local number 1. When a new task is scheduled on this server it is given the local number $n + 1$. Note that in the $n - 1$

tasks, some, all or none can be terminated. We note $J_{i,j}$ the task/job with the local number i on server j.

Let $a_{i,j}$ be the arrival date of the task $J_{i,j}$, $d_{i,j}$ its duration on the unloaded server and $C_{i,j}$ its real completion time.

Each time we need to simulate the execution of a new task on a server, we use the HTM. The HTM provides the following numbers after simulating the execution of tasks (fig 1) : $T'_{i,j}$ refers to the simulated finishing date of the job $J_{i,j}$ before the arrival of the new task. $T_{i,j}$ is its finishing date given after the simulation of the execution of the new task on server j (hence, $T'_{i,j} = T_{i,j}$ if the task is allocated to the server $j_0 \neq j$ for each j).

Finally, we define the *perturbation* of the new task on the local task i as $\delta_{i,j} = T_{i,j} - T'_{i,j}$.

3. Metrics

Usually, the makespan metric is used, in order to minimize the execution date of the entire application submitted to the agent. The heuristics employed in our tests are not specifically designed to only minimize the makespan, because we do not believe it to be the main or the only scheduling metric to use. Hence, our observations have been conducted on metrics among which some come up from system environments and continuous job stream studies. We have observed our experiments on the following metrics:

- the makespan : it is the completion time of the last finished task, $\max_{i,j} C_{i,j}$. The makespan is much more an application metric, for it is its finishing date. So, even if it is the most used (basically with the MCT heuristic in Legion [7], NetSolve [5]), we do not think it is the appropriate metric to use when considering the grid. The agent can be requested by 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 sum-flow [1] : this is the amount of time that the completion of all tasks has taken on all the resources, $\sum_{i,j}(C_{i,j} - a_{i,j})$. 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 leads to estimate the profit realized by using a given heuristic when the cost of each resource is the same.

- the max-stretch [9] : we know by this value by what maximum factor, $\max_{i,j}((C_{i,j} - a_{i,j})/d_{i,j})$, 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 his 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 max-flow [9] : this is the maximum time a task has spent in the system, $\max_{i,j}(C_{i,j} - a_{i,j})$. In a loaded system, a task will generally cost more than expected. This is even truer if it is allocated on a fast server (which is generally more sollicited). This value can inform about high contention or about the use of the slowest servers in a high heterogeneous system.

- 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 $\mid \{t_i | C_{t_i H_1} < C_{t_i MCT}\} \mid$

The user point of view is not that the last allocated task finishes the soonest (optimizing the makespan) but 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 MCT's without delaying too much other task completion dates (that can be verified with the sum-flow for example), we can claim that this heuristic, to the user point of view, outperforms MCT.

4. Proposed Heuristics

We have conducted several simulated experiments with the Simgrid API [4]. Our investigations on several heuristics are reported in [2] : among them, some gave good results on most of the metrics observed here. Indeed, in the simulated experiments, these did generally not only optimized the makespan but also one or more other metrics. We only give in this section the ones that we have implemented and tested in the real NetSolve environment.

Their interactions with the HTM and their objectives are also presented.

4.1. Historical Minimum Completion Time

HMCT is the Minimum Completion Time algorithm relying on the HTM. When a new task arrives, the HTM

For each new task t
  For each server j that can resolve the new submitted problem
    The task t is of local number l_j
    Ask the HTM to compute $C_{l_j,j}$
  Map task t to server j_0 such as $C_{l_{j_0},j_0} = \min_j C_{l_j,j}$
  Tell the HTM that task t is allocated to server j_0

Figure 2. *HMCT algorithm*

For each new task t
  For each server j that can resolve the new submitted problem
    Ask the HTM to compute $P_j = \sum_i \delta_{i,j}$
  If all P_j are equal
    map task to server j_0 that minimizes $C_{n+1,j}$
  Else Map task 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

Figure 3. *MP algorithm*

simulates the mapping of the task on each server until its completion. Therefore, we have an estimation of the finishing date of this task on each server. The agent then matches the task to the server that minimizes its finishing date (fig 2). 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.

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

4.2. Minimum Perturbation

In MP, the new task is mapped to server j that minimizes the sum of perturbations given by $\sum_i \delta_{i,j}$. In the case of equality, for instance at the beginning, the server that minimizes the completion date of the last incoming task is chosen (fig 3). MP aims to provide a better quality of service to each task by delaying as less as possible already allocated tasks.

Its main drawback is that the utilization of resources is sub-optimal : a task can be allocated to a slow server unnecessarily.

4.3. Minimum Sum Flow

Minimum Sum Flow is a willing attempt to mix the advantages of HMCT and MP (to keep the makespan objective of HMCT and give a better quality of service to each task) and to reduce the cost of resources. The heuristic uses the HTM to compute the sum of the whole flow when assigning the last task to each server. Hence, the heuristic returns the identity of server j_0 that minimizes the system sum flow, e.g. $\min_j(\sum_{k\neq j,i=1}^{i=n}(T'_{i,k} - a_{i,k}) + \sum_{i=1}^{i=n+1}(T_{i,j} - a_{i,j}))$. 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 $\sum_{i=1}^{n} \delta_{i,j} + T_{n+1,j} - a_{n+1,j}$ 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 the same as *MTI* (minimize total interference) proposed by Weissman in [11].

5. Experiments

Each heuristic as well as the HTM have been implemented in the NetSolve code [5]. We have compared them to the MCT algorithm that is implemented in NetSolve. We made two sets of experiments that differ on

1 **For each** new task t
2 **For each** server j that can resolve the new submitted problem
3 The task t is of local number l_j
4 Ask the HTM to compute $P_j = \sum_i \delta_{i,j} + T_{l_j,j} - a_{l_j,j}$
5 Map task t to server j_0 such as $P_{j_0} = \min_j P_j$
6 Tell the HTM that task t is allocated to server j_0

Figure 4. *MSF algorithm*

type	machine	processor	speed	memory	swap	system
server	chamagne	pentium II	330 MHz	512 Mo	134 Mo	linux
	cabestan	pentium III	500 MHz	192 Mo	400Mo	linux
	artimon	pentium IV	1.7 GHz	512 Mo	1024 Mo	linux
	pulney	xeon	1.4 GHz	256 Mo	533 Mo	linux
	valette	pentium II	400 MHz	128 Mo	126 Mo	linux
	spinnaker	xeon	2 GHz	1 Go	2 Go	linux
agent	xrousse	pentium II bipro	400 MHz	512 Mo	512 Mo	linux
client	zanzibar	pentium III	550 MHz	256 Mo	500 Mo	linux

Table 2. Resources of the testbed

the environment and on the type of tasks that are submitted as we will see further.

In each set, the number of machines in the experimental environment is held to six whose characteristics are given in Table 2. We should note that, as the machines are scattered in the laboratory, links are not exclusive to our experiments whereas servers are dedicated to the system.

We call an experiment the submission of a metatask composed of 500 independent tasks to the agent. It is given in next sections how many times an experiment has been carried out (e.g. the number of submissions of the same metatask in the same re-initialised environment). Two types of experiments are conducted : the same set of tasks, e.g. the same metatask, is considered with different arrival dates. The difference between two arrivals is drawn from a Poisson distribution with a mean of $\mu = 20$ seconds or $\mu = 15$ seconds.

In fact, these tasks are of the same type, whose input data differ (leading to different durations). A task has a uniform probability to be of each duration. As we consider in this paper three possible inputs, one can generate in each set 500^3 metatasks for each arrival rate.

We note that in each experiment of each set, a scheduling decision cost is negligible compared to the duration of the shortest task (less than 0.01 second in most of cases) for all the proposed heuristics.

In the following, we will describe more precisely the two sets of experiments, doing some remarks particuliar to each set, then we will comment all the results.

5.1. First Set of Experiments

In the first set of experiments, the tasks are multiplications of square matrix of size 1200, 1500 and 1800. Each multiplication has been run on each unloaded server hence determining its time cost (transfer and computing), which have been placed in the NetSolve code (Table 3). Hence, the NetSolve MCT uses also these information. For each heuristic, the environment is made of :

- client : zanzibar ;
- agent : xrousse ;
- servers : chamagne ; pulney ; cabestan ; artimon.

Results for $\mu = 20$ seconds are given in Table 5. Results for MCT and HMCT presented in Table 6 ($\mu = 15$) are those maximizing the number of completed tasks.

Indeed, for $\mu = 15$, MCT and HMCT are not able to handle all tasks. HMCT and MCT overload the fastest servers that cannot accept any more jobs because it runs out of memory. However, the NetSolve MCT has fault tolerance mechanisms that permit to schedule almost all tasks. For $\mu = 20$, this phenomenon does not occur

size of the square matrix	memory need (Mo) input	memory need (Mo) output	phase	servers chamagne	cabestan	artimon	pulney
1200	21.97	10.98	input data cost	4	4	3	3
			computing cost	149	70	18	14
			output data cost	1	1	1	1
1500	34.33	17.16	input data cost	6	5	5	5
			computing cost	292	136	33	25
			output data cost	2	2	1	1
1800	49.43	24.72	input data cost	8	8	8	7
			computing cost	504	231	53	40
			output data cost	3	3	2	2

Table 3. Multiplication tasks' needs

parameter	phase	servers valette	spinnaker	cabestan	artimon
200	input data cost	0.08	0.09	0.1	0.12
	computing cost	91.81	16	74.86	17.1
	output data cost	0.03	0.05	0.03	0.03
400	input data cost	0.08	0.14	0.09	0.13
	computing cost	182.52	30.6	148.48	33.2
	output data cost	0.03	0.06	0.03	0.03
600	input data cost	0.13	0.09	0.08	0.14
	computing cost	273.28	45.6	222.26	49.4
	output data cost	0.03	0.05	0.03	0.03

Table 4. Waste-cpu tasks' needs

because the arrival rate lets faster servers complete more tasks before a new request.

One should note that for $\mu = 15$ seconds, the NetSolve MCT gives very high values, and the highest for all the observed metrics. As MP gives even better results on the max-flow, we can conclude that there is a huge time and space contention on the fastest servers. This is confirmed by the load average sent to the agent (more than 12 on pulney). Consequently, some servers collapsed during the experiment.

5.2. Second Set of Experiments

To prevent the memory problems that we do not yet handle, we designed a task, 'waste-cpu', that does not require any memory to be computed. The goal is obviously to replace the multiplication tasks, so its computation costs, dependent on the parameters, are similar to the multiplication tasks. Then, a task can have the parameter 200, 400 or 600 that reflects the time it costs (Table 4). The experimental testbed for this set of experiments is made up of:

- client : zanzibar ;
- agent : xrousse ;
- servers : valette ; spinnaker ; cabestan ; artimon.

We generated three different metatasks, submitted at two different arrival rates. Results that are given in Table 7 ($\mu = 20$ seconds) are obtained from 2 executions of the same metatask for each of the four tested heuristics. At this rate, the metatask finishes after about $10,000$ seconds. In Table 8 ($\mu = 15$), values of a metatask are the mean of 4 executions for the NetSolve MCT and 3 executions for the three others. At this rate, the metatask needs about $7,700$ seconds to complete. For a metatask scheduled according to a given heuristic, the number of tasks that finish sooner is the mean of the values obtained from the comparison between each run for this heuristic and each run for NetSolve.

We should remark, with relief, that two runs of the same experiment give slightly the same results.

All the tasks of all the metatasks of this set of experiments have been submitted, accepted and computed. The results for $\mu = 20$ and $\mu = 15$ are presented in Table 7 and Table 8 respectively.

5.3. Results

In this paper, we consider the scheduling of a metatask. Therefore, the makespan value is strongly de-

	NetSolve's MCT	HMCT	MP	MSF
number of completed tasks	500	500	500	500
makespan	9906	9908	10162	9905
sumflow	25922	19934	26383	19702
maxflow	230	103	517	97
maxstretch	12.8	5.8	3.7	5.3
number of tasks that finish sooner than with NetSolve's MCT	-	325	330	325

Table 5. results for $\mu = 20$ in seconds for multiplication tasks

	NetSolve's MCT	HMCT	MP	MSF
number of completed tasks	495	358	500	500
makespan	7880	5600	7648	7626
sumflow	89254	25092	34677	31375
maxflow	1780	500	720	250
maxstretch	99	27.8	6.3	11.3
number of tasks that finish sooner than with NetSolve's MCT	-	306	418	435

Table 6. Results for $\mu = 15$ in seconds for multiplication tasks

pendent on the latest task arrival. We cannot expect at the very outset a big difference between two heuristics on that metric especially at low rate. That is verified by our tests. Nonetheless, the sum-flow is different for each heuristic.

To see how useful the HTM is, we compare the sum-flow on the same heuristic : MCT and HMCT (in fact, these are not exactly the same. NetSolve has two load correction mechanisms. The first tries to take note of the allocation of a task to a server. This is useful if a request is received as the server has not sent a load report that shows the load increase. The second is a message sent by the server when a task finishes). HMCT needs one missing hour of computations for a metatask duration of less than three hours ($\mu = 20$) for the same makespan. Moreover, a better quality of service is offered (a better max-stretch and a mean of 327 tasks for HMCT versus 173 for MCT). Its performances are greater as the rate increases : a gain of 4.8 hours of computation for a duration of 2.1 hours, and there too, a better quality of service. The use of the HTM definitely leads to better results.

MP has a better load balance property than MCT and HMCT. Hence, when μ is large (low arrival rate), it loads slower servers because they are idle. Whereas, when μ is low, no servers are idle, then MP tends also to load the fastest ones. MP is always the best on the max-stretch, i.e. the tasks are the least delayed when scheduled with this algorithm. It is normal for there is less contention : when all servers are loaded, the fastest are chosen. But even with the contention produced by that load, it does not create a really high stretch. A higher stretch would be obtained if slower servers had more contention. But, this means a higher rate, faster servers more loaded and even collapsed because of the heterogeneity of the environment. MP is also the heuristic that presents the highest max-flow. It seems logical considering that :

- for $\mu = 20$, as tasks on faster servers are not necessarily finished, lower servers are used. The MP max-flow is then the maximum cost of a task on the slowest server. But, as contention on faster servers is small when the metatask is scheduled by another heuristic, MP maximizes the max-flow.
- for $\mu = 15$, there is contention even on the slowest servers. A task that had already a higher duration than if allocated to a faster server requires even more time.

Therefore for a low rate, MP is sub-optimal, but is rather good at higher rates : less sum-flow (even compared to

HMCT) and a high number of tasks that finish sooner (410 versus 90 for MCT).

MSF tries to optimize the sum-flow, hence finds a good balance between minimizing the perturbation and minimizing the new task duration. Therefore, it gives good performances on the makespan, the sum-flow, even on the max-flow and the number of tasks that finish sooner than with MCT is always very high (same as MP's for $\mu = 15$!). While MSF is not explicitly designed to optimize the makespan as is MCT, it appears that it always outperforms MCT, as well as HMCT and MP. Indeed, an agent cannot guess the rate of the requests it will have to process and MSF gives the same performances than HMCT at low rates and better performances than others at higher rates.

6. Related Work

Figueira has developed a contention model in [6] for application performance on two-machine heterogenous platforms. Weissman in [11] has explored the interference paradigm that we have called in this paper the perturbation a task induces on already allocated and running tasks when being assigned to the same server. We use the same model for the contention of tasks. Moreover, Weissman designed some heuristics : MNI (that minimizes the number of tasks that experience interference) and MTI (equivalent to MSF). He compared them and MCT to each other on the mean delay recorded by a task due to later allocation on the same server, e.g. the average stretch. His goal was to schedule a set of tasks where 90% were sequential and 10% parallel jobs. He concluded that MTI is the heuristic that delaies the least the tasks. We have considered only sequential tasks but went further in our simulation work, considering other metrics and other heuristics as well [2]. Contrary to Weissman, we assume that all tasks can create communication bandwith interference for any other task. Moreover, we realized the implementation of three heuristics that seemed promising in this study in a real environment (NetSolve [5]).

7. Conclusions and Future Work

In this paper, we have studied the problem of scheduling a set of tasks in the client-agent-server model. We have tackled the problem in order to optimize needs of each users (a submitted task has to finish as soon as possible) without degrading the makespan.

We have proposed three heuristics based on a historical trace manager we have designed. Simulation experiments [2] were carried out and we modified the NetSolve code in order to test these heuristics on a real distributed architecture.

We made some tests to validate the model, hence the HTM approach. Then we have submitted to the agent some metatasks, firstly composed of multiplication tasks. This first set of experiments has been difficult to conduct mainly because the allocation model does not take the memory requirements into consideration (for MCT and HMCT). Hence, we designed a task similar in cost but that requires no memory to be processed and ran the second set of experiments with this kind of task.

Our results show that using the HTM to schedule a set of tasks is a real improvement. Better choices are made all along the execution of the metatask leading to better performances on the observed metrics : makespan, sum-flow, max-flow, max-stretch. The number of tasks that finish sooner than if scheduled with MCT is always very high (at least a factor of 1.7 to a factor of 5). Moreover, heuristics based on minimizing the perturbation outperform the standard MCT as implemented in NetSolve (for a low rate, this is not true only when MP schedules a waste-cpu metatask. For a metatask made up of matrix multiplications, it beats MCT). MSF outperforms NetSolve's MCT in all the cases.

Moreover, experiments performed in the first set show that MSF and MP balance the load in a better way than MCT and HMCT, leading to less memory consumption on servers.

We conclude that among all the tested heuristics, the use of the Minimum Sum Flow algorithm would increase the quality of service given to each user and decrease at the same time the amount of computation required. Our experiments on a real testbed confirm the simultation results found by Weissman [11].

Our immediate future works will follow two directions. First, we need to incorporate memory requirements into the model. Second, we will improve the synchronization between the HTM and the execution of the tasks on the platform.

8. Acknowledgment

The authors would like to thank the reviewers for their comments and the Cassis research group for the use of chamagne and pulney for the first set of experiments.

	NetSolve's MCT				HMCT				MP				MSF			
	M_1	M_2	M_3	**Avg**	M_1	M_2	M_3	**Avg**	M_1	M_2	M_3	**Avg**	M_1	M_2	M_3	**Avg**
makespan	9913	10044	10210	**10056**	9901	10041	10210	**10051**	10009	10047	10265	**10107**	9903	10041	10209	**10051**
sumflow	25768	22036	20727	**22844**	20151	18151	17364	**18555**	26729	24384	24239	**28451**	20306	18138	17317	**18587**
maxflow	193	168	124	**162**	123	109	82	**105**	286	275	273	**278**	116	135	85	**112**
maxstretch	4.1	4.1	2.8	**3.7**	3.1	2.2	2.1	**2.4**	1.8	1.8	2	**1.9**	2.8	2.8	2.2	**2.6**
number of tasks that finish sooner than with NetSolve's MCT	-	-	-	–	344	314	324	**327**	339	317	321	**326**	331	309	320	**320**

Table 7. Results for $\mu = 20$ in seconds for waste-cpu tasks

	NetSolve's MCT				HMCT				MP				MSF			
	M_1	M_2	M_3	**Avg**	M_1	M_2	M_3	**Avg**	M_1	M_2	M_3	**Avg**	M_1	M_2	M_3	**Avg**
makespan	7690	7615	7644	**7650**	7655	7565	7626	**7615**	7672	7545	7765	**7661**	7672	7544	7626	**7614**
sumflow	63364	49529	50014	**54302**	39973	36675	34821	**37156**	33913	30860	30158	**31644**	34347	30279	29744	**31457**
maxflow	344	283	290	**306**	234	231	228	**231**	340	314	314	**323**	234	182	162	**193**
maxstretch	7.5	6.5	6.7	**6.9**	4.9	5	4.6	**4.8**	3.8	3.2	2.9	**3.3**	5	3.4	3.2	**3.9**
number of tasks that finish sooner than with NetSolve's MCT	-	-	-	–	393	368	388	**383**	416	403	410	**410**	423	402	412	**412**

Table 8. Results for $\mu = 15$ in seconds for waste-cpu tasks

References

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

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

[3] E. Caron, F. Desprez, E. Fleury, D. Lombard, J. Nicod, M. Quinson, and F. Suter. Une approche hirarchique des serveurs de calcul. *to appear in Calculateurs Parallles, numro spcial metacomputing*, 2001. http://www.ens-lyon.fr/ desprez/DIET/index.htm.

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

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

[6] S. M. Figueira and F. Berman. Modeling the effects of contention on the performance of heterogeneous applications. In *Proceedings of the high Performance Distributed Computing Conference*, august 6-9 1996.

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

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

[9] M. l 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.

[10] M. Maheswaran, S. Ali, H. J. Siegel, D. Hengsen, and R. F. Freund. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing system. In *Proceedings of the 8th Heterogeneous Computing Workshop (HCW '99)*, april 1999.

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

[12] R. Wolski, N. 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.

Yves Caniou is currently a PhD student in one of the five INRIA centre, the Laboratoire lOrrain de Recherche en Informatique et ses Applications, Nancy, France. He has received his M.S. degrees in computer science from the University Henry Poincaré in 2001. His research focuses on scheduling for metacomputing platforms. He is also interested in simulation of distributed environments.

Emmanuel Jeannot received the B.S from the University of Lyon in mathematics and the M.S from the École Normale Superieure of Lyon in computer science and modelization. He received the Ph.D. degree in computer science in 1999 from the École Normale Superieure of Lyon.

He is currently associate professor of computer science at the Université Henri Poincaré, Nancy. His research interests include algorithms for parallel and distributed computing, scheduling, online compression, data redistribution, software tools and environments for grid computing.