Dynamic Performance Forecasting for Network-Enabled Servers in a Metacomputing Environment*

Martin Quinson
ReMap / LIP
École Normale Supérieure de Lyon
46, allé d'Italie
69364 Lyon Cedex 07, France.
E-mail: `Martin.Quinson@ens-lyon.fr`

Abstract

This paper presents a tool for dynamic forecasting of Network-Enabled Servers performance. FAST *(*Fast Agent's System Timer*) is a software package allowing client applications to get an accurate forecast of routines needs in terms of completion time, memory space and number of communication, as well as of current system availability.* FAST *relies on existing low level software packages, i.e., network and host monitoring tools, and some of our developments in computation routines modeling. The* FAST *internals and user interface are presented, as well as experimental results.*
KEYWORDS: *Performance forecasting and modeling, Resource monitoring, Metacomputing.*

1 Introduction

Metacomputing consists in federating distributed computational resources in order to aggregate their power. The Grid [3] denotes the resulting virtual computer formed by a large-scale and heterogenous set of machines sharing their local resources with each other. The Network-Enabled Server (NES) paradigm is based on *servers* which can fulfill computational requests with the help of sequential or parallel libraries. This constitutes a simple yet powerful RPC-style programming model to achieve the metacomputing goals. In most systems based on this approach an *agent* is in charge of selecting the most appropriate server for a request issued by the *client* in terms of performance and software availability.

*This work has been supported by the ARC OURAGAN of INRIA.

Requests mapping onto servers is one of the most challenging problems to solve due to the dynamic, heterogenous and shared nature of the platform. This complicates the forecast of system characteristics such as the needs of the tasks to schedule and the system availabilities. Moreover the response time have to be minimized because of the interactive use of the system.

This paper presents the FAST library (*Fast Agent's System Timer*), which goal is to provide the information needed by a scheduler. FAST models the needs of the tasks both in terms of time and memory space. Appropriate tools like NWS [10] are used to monitor the dynamic availabilities of system resources. FAST is also able to aggregate these two kinds of information in order to forecast the current computation time of a given task on a given machine. The goal of FAST is however not to perform task placement, but to acquire the required knowledge to achieve it.

Scheduling the tasks on computers comes down to mapping task needs to system availability. We now describe these values more precisely.

Needs of routines group principally the time and the memory space necessary to their execution, as well as the amount of generated communication. These values depend naturally on the chosen implementation and on the routine input parameters, but also on the machine on which the execution takes place. Indeed, the implementation of numerical library often benefits of strong optimizations. Material parameters such as cache sizes influence in a significant manner on the achieved performance.

System availabilities represent the number and the speed of the machines, as well as their status (down, available or allocated through a batch system). One must also know the topology, the capacity and the protocols of the network between these machines. From the scheduling point of view, the actual availability of these resources is more important than their use or the theoretical peak performance.

FAST's goal is to constitute a simple and consistent Software Development Kit (SDK) providing client applications with accurate information about routine needs and about system availabilities, regardless of how theses values are acquired. The library is optimized to reduce its response time, and to allow its use in an interactive environment.

The rest of this article is organized as follows: section 2 presents a quick state of the art. Then, we explore FAST's internals in section 3, before presenting its interface in section 4. Before a conclusion, section 5 gives some experimental results.

2 State of the art

2.1 Distributed monitoring

The tools presented below monitor uses and availabilities of a distributed system. Most often, sensors are deployed to each machine and collected data are then dispatched to consumers.

PingER [7] is a distributed monitoring architecture deployed on more than 600 hosts in 72 countries. Periodically some tests are conducted between hosts using the well known `ping` tool, and the resulting data are made available from a web page. This project goal is to ease the search of partners in computer intensive project (like in nuclear physic) or help the design and understanding of the networks. Even if it would be interesting to provide these informations to a scheduler, nothing seems to be done so far to allow an interactive use of the system.

Performance Co-Pilot [8], developed by Silicon Graphics Inc (SGI), monitors a set of distributed machines using sensors, and an API allows access to the resulting data. Unfortunately, this project provides only raw data about the resource *use*, like the communication outflow. Since a scheduler wants to know the time a function needs to complete, these information cannot be used directly. Moreover, deducing the availability of a resource from its actual use is not trivial. For example, given the priority mechanism used for Unix processes, it is very difficult to deduce the time-slice that a process will get from the actual load of the CPU ([9]). A machine overloaded by low priority processes is still available for high priority ones.

The **Network Weather Service** (NWS, [10]) includes some distributed sensors to monitor the actual state of the system and statistical forecasters to deduce the future evolution. It allows to monitor the latency and bandwidth of every TCP link, the CPU time-slice a new process would get, or the available memory and disk space. Contrary to the systems described above, NWS not only provides information about the resources use, but also about their availability.

2.2 Performance forecasting in a metacomputing environment

Bricks [1] is a Grid simulator. It allows to reproduce metacomputing experiments within the same conditions. This is very useful to compare scheduling algorithms on this kind of platform. This is normally not possible because of unreproducible disruptions caused by the resource sharing. The model used in Bricks is based on queues. For example, a server is a queue where tasks are stored when they arrive, and are released after "handling". Each task is characterized with the amount of computation and communication needed for its completion. This approach have two major drawbacks. First the speed of a machine is supposed to be constant whereas cache effects can lead to great variations. The peak performance (in Flops) can seldom be achieved, but optimizations may permit to approach them. Furthermore this approach is relatively computation-intensive, which prevents its use in an interactive framework.

2.3 Metacomputing systems

We now present several metacomputing Problem Solving Environment (PSE), stressing their solutions concerning system monitoring and performance forecasting.

Globus [4] is one of the best known and advanced metacomputing PSE. The associated monitoring tool, called GloPerf [5], is composed of monitoring daemons leading network measurements periodically. This system have several drawbacks: Each daemon maintains a list of all existing sensors, which prevents the scalability of the system. Then the system cannot forecast the future evolutions of the system, contrary to NWS. Moreover this system is tightly related to the whole Globus system and cannot be used in other projects easily. Finally it can only monitor the network, and not the CPU and memory of the machines.

NetSolve [2], used its own set of sensor to monitor the system until the version 1.3. In the current version, NWS is used for the monitoring issues. The routine model is merely expressed by an expression of the form $x\ s^y$, where s is the size of the parameters, and x, y express the complexity of the operation. The speed of the computer is supposed constant for all operations, and the memory need is not taken in account.

3 FAST's internals

3.1 Overview

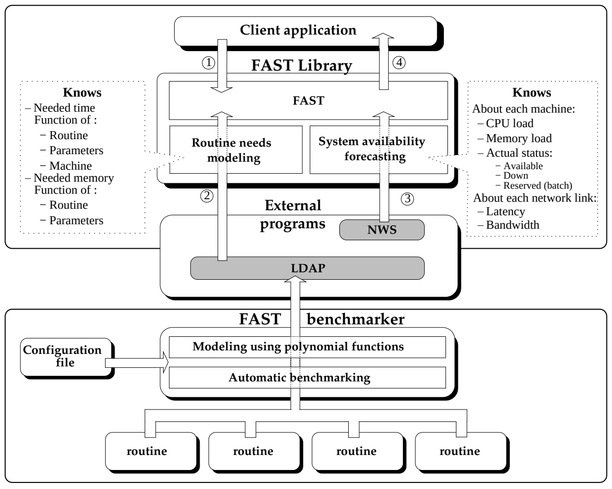


Figure 1. FAST's architecture.

We now present FAST, and show how this library can help to solve the monitoring and routine modeling problems. Figure 1 gives an overview of FAST's architecture, which is composed of two main parts: on the bottom of the figure, a benchmarking program can discover the routines needs on every machine of the system. Then, on the top, a shared library provides accurate forecasting to the client application. This library is divided in two submodules: the right one on the figure forecasts the system availabilities while the left one models the routines needs.

After a description of the used tools, we study the acquisition of system availability in section 3.3. Then section 3.4 will present how FAST models the routines needs at installation time and forecasts them at execution time.

3.2 External tools

Figure 1 shows that FAST uses principally two external tools (in grey). The first one is NWS, a monitoring tool described in section 2.1. The second one is LDAP (Lightweight Directory Access Protocol, [6]). This is a distributed and hierarchical database system. It allows to split the databases and map them to the administrative organization. That way the manager of each site handles the data relative to his set of machines. The openness of this system and its wide use seem to guarantee the durability of this protocol.

3.3 System availability acquisition

The system availability acquisition module allows FAST to take in account the external load due to resource sharing. We based our work on NWS to have solid and well tested foundations. But FAST would be able to get the needed data from any other library providing the same kind of information. In its actual version, FAST can monitor the CPU and memory load of hosts, as well as latency and bandwidth of any TCP link. Monitoring new resources like free disk space or non-TCP links would be pretty easy.

Our use of NWS revealed some of its shortcomings we tried to solve: NWS is composed of four kind of daemons, distributed across the network. This causes a non negligible response time. Since the measurements are done periodically, we introduced a cache system to avoid submitting the same request when the answer is already known. The cached value is refreshed only if it is older than the periodicity of the measurements.

Moreover, NWS can forecast future evolutions of the system, applying statistic functions to past measurements. This process smooth the predictions and reacts slowly to radical condition changes like a process startup. We added the possibility to indicate to NWS that old measurements should be discarded because of condition changes. This allows a better collaboration between the scheduler and NWS through FAST.

Finally it is possible to reduce the number of host pairs leading measurements to make the system less intrusive. But NWS is then unable to combine the direct measurements automatically. For example, given three machines A, B and C, if we asked for measurements for the pairs (A,B) and (B,C), it is then impossible to get an estimation between A and C from NWS. In this case, FAST produce a combination of the direct tests. This value can be less accurate than real tests, but is still interesting when no other information is available.

3.4 Routine needs forecasting

This part constitutes the most fundamental contribution of FAST. It is about forecasting the theoretical needs of a routine in terms of completion time and memory space, depending on its parameters and the machine on which the execution takes place. This problem is complex, and several approaches are possible, depending on the routine.

Some functions, like the sequential numerical routines, are regular and easy to time. The classical approach consists in modeling the execution time manually, based on the source analysis. But this is tedious because of the amount of parameters to take in account. One should obviously study the implementation, but also the optimizations done by the compiler, the operating system and hardware characteristics concerning the memory hierarchy and its use. This profusion of parameters leads to a detailed study of each pair {software ; hardware}, which represents a colossal amount of work on a metacomputing platform.

To solve this problem, another classical approach consists in using a generic benchmark on the machine to know how much elementary operations can be done per unit of time, and then to count the necessary operation of each routine. This approach seems too simple because it does not take cache effects in account, which lead to important performance variation.

Our approach consists in benchmarking the function on each machine for a representative set of parameters during the installation. The results are then modeled by polynomial regression and stored in a LDAP database. This benchmarking phase can be time-consuming, but it has to be done only once, and can be done in parallel over an homogeneous set of machines.

Other routines are harder to time, but can be easily and accurately studied. The parallel numerical computation functions can be decomposed in sequential sub-routines and communications between the nodes. Benchmarking the function is harder in this case, because of the amount of parameters to take in account and because it is almost impossible to cut off from the influence of resource sharing.

In that case, FAST will permits to express the structural decomposition of the function, and the need of the routines will be deduced from the needs of each sub-functions and communications. Naturally, the needs of each sequential subroutines may be obtained by FAST using the first method explained above.

4 FAST's interface

This section presents FAST's interface. In fact, there are two interfaces. The first one makes it possible for the client application to get the needed informations while the second one allows FAST to model new routines without modification to FAST itself.

4.1 FAST's API

FAST's API is designed to be very simple to mask the underlying complexity to the calling application. It is composed of two levels: a scheduler gets ready to use values from the high level interface while a visualization tool can get more details from the lower one.

After initialization, a scheduler requieres only two functions to get the needed information:

`fast_comm_time(source, dest, msg_size)` computes the needed time to transmit data of size `msg_size` from machine `source` to machine `dest`, taking in account the actual network load.

`fast_comp_time(host, function, data_desc)` gives the forecasted computation time for this `function` on this `host` for the parameters described by `data_desc`. An error is returned if this host cannot do this function or if there is not enough free memory.

The lower level interface is composed of two functions:

`fast_need(resource, host, function, data_desc)` computes how many of the `resource` (either time or space) the `function` needs to complete on this `host`, for the parameters described by `data_desc` while

`fast_avail(resource, host1, host2)` gives the available amount of `resource` on `host1` (or between `host1` and `host2` in the case of network resource). The resource can be either the CPU load, the CPU time-slice a new process would get, the available memory or the free disk of a host, or the latency and bandwidth of a network link.

4.2 Extending FAST

To be able to forecast their needs, FAST must have some knowledge about the functions which can be scheduled in the system. In particular, one should describe how to do the benchmarks, and the function's arguments. All the needed information about a function are grouped in a XML configuration file.

Figure 2 presents a simplified example of such file for the matrix multiplication operation. The problem description is split in two parts: the `<description>` contains the high level description of the problem. For example, in this case, we indicate that the operation take two double matrices A and B in input, and its output is a matrix, resulting in the multiplication of A and B.

Then, the part `<implementation>` gives some informations about how to do the benchmarks and about how to call the underlying function. The `<implInput>` sub-part indicate that the real function takes six arguments. All implementation input arguments are converted from the high level ones using a `<implConvert>` statement. For example, `<implConvert func="numLines" data="A"/>` indicate that M is the number of line of the matrix A, while `<implConvert data="A"/>` indicate that the

```
<problem name="mult">
  <description>
    <descInput>
      <descMat name="A" baseType="double"/>
      <descMat name="B" baseType="double"/>
    </descInput>
    <descOutput>
      <descMat name="C" baseType="double"/>
    </descOutput>
    <descCode>
      <mult><arg name="A"/><arg name="B"/></mult>
    </descCode>
  </description>
  <implementation language="C">
    <implInput>
      <implInt    name="M">
        <implConvert func="numLines" data="A"/>
        <benchNum min=128 max=1152 incr=128/>
      </implInt>
      <implInt    name="N">
        <implConvert func="numRows"  data="A"/>
        <implConvert func="numLines" data="B"/>
        <benchNum min=128 max=1152 incr=128/>
      </implInt>
      <implInt    name="K">...
      <implMat    name="A">
        <implConvert data="A"/>
      </implMat>
      <implMat    name="B">...
      <implMat    name="B">...
    </implInput>
    <implCode>
      <benchBody>
        ... A=malloc(...);memset(A,2,... );...
      </benchBody>
      <implBody>
        mymult(M,N,K, A,B,C);
      </implBody>
    </implCode>
  </implementation>
</problem>
```

Figure 2. Matrix multiplication description.

matrix A of the implementation level is the matrix A of the high level without any modification.

5 Experimentations

After the study of Fast internals and interfaces, we now present some experiments. We first report the result of our contributions to NWS in section 5.1. Section 5.2 presents some experiments showing the quality of Fast forecasts. Finally, we report the result of Fast's integration to others projects in section 5.3.

5.1 Contributions to NWS

As indicated in section 3.3, we tried to solve some of NWS's shortcomings. The first point concerns the response time we tried to reduce using a cache system. Figure 3 presents the measured times. (3a) is the measured time for a direct interrogation to the NWS (when all the parts of the system are on the same host), (3b) is the time for an interrogation through Fast when the asked value is not in the cache, resulting in an interrogation to NWS. (3c) is the time to get the value from the cache (avoiding completely NWS).

This experiment shows that Fast's overhead is quite low in case of cache default (about 1%), while getting the value from the cache is more than 4000 times quicker than an interrogation to NWS.

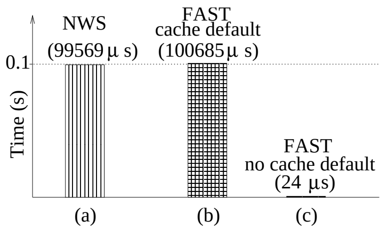


Figure 3. NWS and FAST response time.

NWS mechanism to forecast the evolution of the system seemed to lack of reactivity to the condition changes like the start or the stop of a process. So, we modified NWS to change this. It is now possible to indicate such changes to NWS so that the system reacts quicker.

We now present our experiment showing the benefit of this change. Two NWS systems are running on a single machine. The first one is a non-modified version, and the second one can be notified of condition changes. At the beginning of the experiment, the machine is free of any load. Then, we launch a CPU-consuming task taking exactly one minute to complete. The second NWS system is informed of these changes.

Figure 4 shows the values each NWS system reported during this experiment. The theoretical value is 1 during the idle time (*i.e.*, 100% because the processor is free), and 0.5 when the task runs (because a new process would get only 50% of the processor). We notice that notifying NWS of the changes allows it to react quicker. When not notified, the indicated load increase very slowly, and after one minute, it only reach 7% where the theoretical value is 50%. On the contrary, when notified of the changes, NWS's error is within 5% after only one measurement.

5.2 Forecasts quality

We made several experiments to demonstrate the validity of our approach and the quality of Fast's returned values. The first experiment only deals with

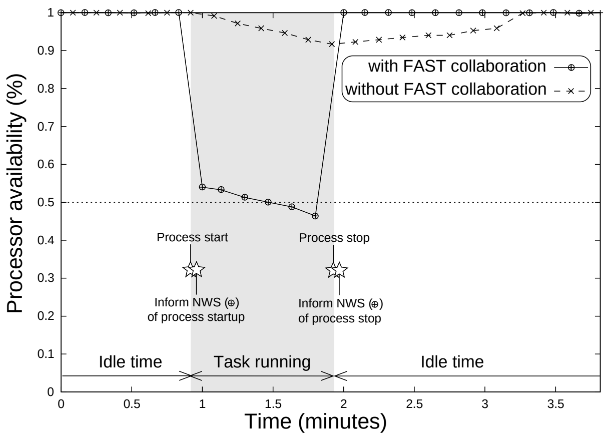


Figure 4. Collaboration of NWS with the platform.

the modeling of functions' needs. The second one is about FAST's ability to forecast the completion time of a single operation, while the third one is about complex sequence of operations.

The test platform is composed of two clusters: icluster and paraski. The first one is the PC cluster of the IMAG. It is composed of Pentium III, with 256 Mb of RAM. The second one is the PC cluster of the IRISA. Each node have two Pentium II and 256 Mb of RAM. All these machines run under Linux version 2.2 and both machine are connected to a nation-wide 2.5 Gb link.

5.2.1 Needs modeling

Before showing FAST's forecast quality, we thought it was important to measure the quality of our model concerning the function's need (in terms of time and space) without taking the load in account. For this, we compared the modeled and the real value for the completion time and needed space concerning three functions of the BLAS numerical library: the matrix addition `dgeadd` ; the matrix multiplication `dgemm`, and the triangular resolution `dtrsm`.

	dgeadd		dgemm		dtrsm	
	iclust.	para.	iclust.	para.	iclust.	para.
Max. error	0.02s 6%	0.02s 35%	0.21s 0.3%	5.8s 4%	0.13s 10%	0.31s 16%
Avg. error	0.006s 4%	0.007s 6.5%	0.025s 0.1%	0.03s 0.1%	0.02s 5%	0.08s 7%

Table 1. Temporal modeling quality for dgeadd, dgemm and dtrsm for machines icluster and paraski.

Table 1 shows the quality of the temporal model for these functions on the cluster machines, for size matrices variating between 128 and 1152. On the first line appears the maximal error in absolute value measured during the experiment. The corresponding percentage is indicated in parenthesis. On the second line is listed the average error in absolute value and in percentage.

We notice that the maximal error is generally lower than 0.2s, while the average error is about 0.1s. This is very acceptable in our context because we saw in section 5.1 that NWS's response time is about 0.3s. The relative error is lower than 7% in average and a bit more than 15% in the worst case (leaving alone the case of dgeadd on paraski, which will be discussed later). For the matrix multiplication on paraski, we observe an error going up to 5.8s, but this is pretty small compared to the computation time since it only represents 4% of the measured time.

In the case of the matrix addition `dgeadd`, the relative errors are higher. This may be explained by the fact that we choose to time the operation using the `rusage` system call in FAST. This permits to cut off from the external load, but the accuracy is about 0.02s. Since the matrix addition takes less than half a second, the timing error becomes significant.

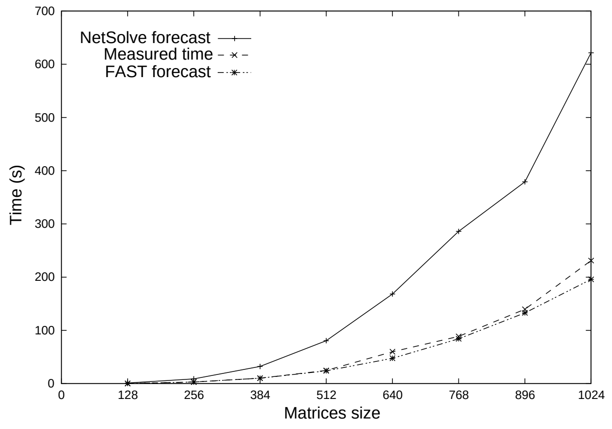
FAST manages to forecast the needed space perfectly (*i.e.*, with a maximal error of 0 byte) for these operations. This perfection is possible because the size of a program doing a matrix operation is the size of the code (which is constant) plus the size of the data (which is obviously a polynomial function).

This experiment deals with FAST's forecasts, taking the load into account. This only deals with the completion time and not with the needed memory since the resource share only affects the former. We compared FAST's forecasts with the measured time for the operation `dgemm`. To simulate an external load, a computation task was running during the experiment. We used the same test platform: a node of each paraski and icluster were taken.

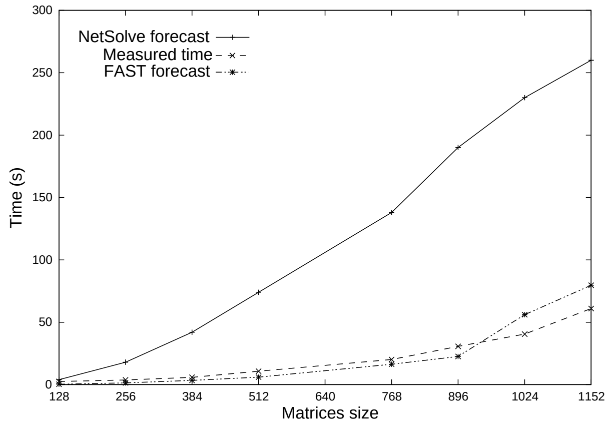
Figure 5 presents the results of this experiment. Despite the external load, FAST managed to forecast the completion time with a maximal error of 22%, and with an average error smaller than 10%.

The third experiment is about sequence of operations. Complex matrix multiplication was chosen: Let A and B be two complex matrices, decomposed in a real and imaginary parts, ones should compute the matrix C resulting of the multiplication of A and B. The formula to solve is therefore:

$$C = \begin{cases} C_r = A_r \times B_r - A_i \times B_i \\ C_i = A_r \times B_i + A_i \times B_r \end{cases}$$



(a) Computation time



(b) Communication time

Figure 7. FAST and NetSolve forecasts compared to measured time (dgemm).

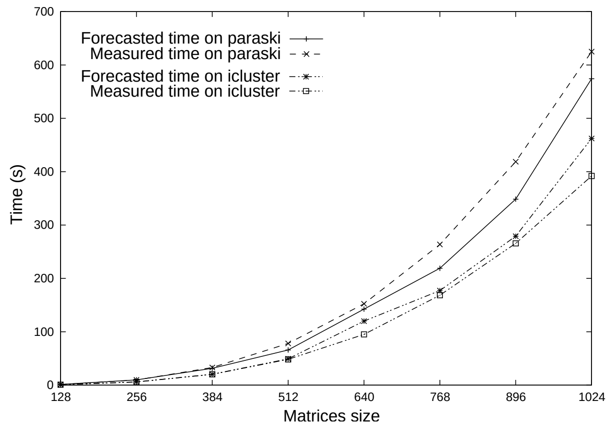


Figure 5. Real and forecasted time comparison for one operation (dgemm).

Since the goal of the experiment was to show the accuracy of the forecasts and not its impact on the scheduling quality, we used a simple yet heterogenous platform composed of two machines. *Pixies* is a Pentium II workstation with 128Mb of RAM, and *Kwad* is a computational server with four Pentium III and 256 Mb of RAM. Both machines run under Linux. During the experiment, *Pixies* had a client process, while two computation processes were placed on different processors of *Kwad*. Here is the course of the experiment:

- *Pixies* creates the four matrices $A_i, A_r, B_i\ and\ B_r$;
- *Pixies* sends $A_{\{i,r\}}$ and B_i to the first computational server on *Kwad* and then $A_{\{i,r\}}$ and B_i to the second one;
- Each server computes its matrix multiplications: $C_{r_1} = A_r \times B_r$ and $C_{i_2} = A_i \times B_r$ for the first one, and $C_{r_2} = A_i \times B_i$ and $C_{i_1} = A_r \times B_i$ for the second one;
- The servers exchange the matrices C_{i_2} and C_{r_2};
- They complete the computation, doing $C_i = C_{i_1} + C_{i_2}$ and $C_r = C_{r_1} - C_{r_2}$;
- The results are returned to the client on *Pixies*.

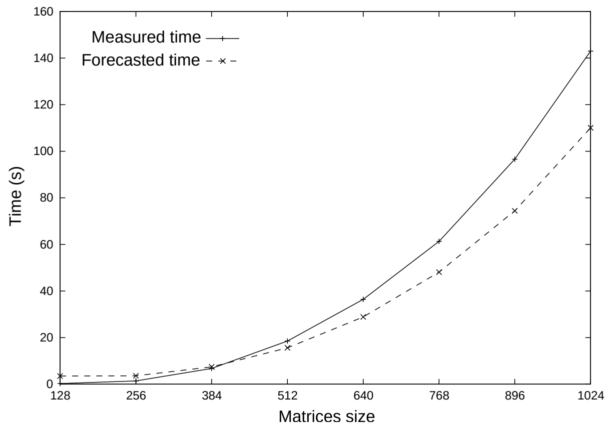


Figure 6. Real and forecasted time comparison for a sequence of operations.

Figure 6 compares the forecasted and the measured time for the whole sequence of operations. We notice that both values vary the same way. Despite the fact that this sequence is composed of six matrix operations (of two different kinds), and six matrix exchanges over the network, FAST managed to forecast the completion time with an error smaller than 25% in the worst case and smaller than 13% in average.

5.3 FAST's integration to other projects

5.3.1 NetSolve

The goal of this integration was to test FAST in real condition. For that, we changed the scheduler of NetSolve to use FAST. This way, we could compare the result of our library to the ones of NetSolve 1.3. One more time, the test platform is quite simple because we only want to study the forecasts, and not their impact. There was only one server located on a node of paraski while the client was on a node of icluster.

Figure 7 presents the results of this experiment. The (7a) part compares the forecasts of NetSolve and FAST to the actual time in terms of completion time while the part (7b) does the same for the communication time. These results are very encouraging. (7a) clearly shows the advantages of a performance estimation depending also on the selected host while (7b) proves the advantage of the use of a specialized monitoring tool.

5.3.2 A visualization tool

We also developed a FAST-based visualization tool using the OpenGL library for a 3D rendering. xOur work is based on the Cichlid[1] library, developed at the National Laboratory for Applied Network Research, UCSD. It is possible to navigate in a platform representation in order to zoom in parts of the network for a more detailed study, or in contrary, to zoom out to obtain a global view of the system.

6 Conclusion and future work

In this article, we presented FAST, a performance modeling and forecasting tool usable in a metacomputing environment. After a brief state of the art, we studied the internal mechanisms of FAST, as well as the proposed interface. We also presented some experimental results. First about some contributions we made to NWS, and then about the quality of FAST's results for single operation or full sequences of several operations.

The presented implementation is fully functional: it was successfully integrated to NetSolve, and DIET, our multi-agent metacomputing platform. We also developed a real-time visualization tool based on FAST. The library is freely available under BSD-license from our web site[2]. It is known to work on various platform like Linux, Solaris or Tru64.

But this version still presents some shortcomings we plan to solve. FAST should also be able to handle the parallel routines by structural decomposition. It is also planed to port FAST to the operating systems Irix and AIX. We would finnally like to enhance NWS to monitor non-TCP links, like the Myrinet- or SCI-based ones.

[1]`http://moat.nlanr.net/Software/Cichlid/`

[2]`http://www.ens-lyon.fr/~mquinson/fast.html`

References

[1] K. Aida, A. Takefusa, H. Nakada, S. Matsuoka, S. Sekiguchi, and U. Nagashima. Performance evaluation model for scheduling in a global computing system. *International Journal of High-Performance Computing Applications*, 14(3):268–279, 2000.

[2] H. Casanova and J. Dongarra. NetSolve: A network-enabled server for solving computational science problems. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(3):212–223, Fall 1997.

[3] I. Foster and C. K. (Eds.). *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[4] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputing Applications*, 11(2):115–128, 1997.

[5] I. Foster and C. Kesselman. The Globus project: A status report. In *Proceedings of the Heterogeneous Computing Workshop*, pages 4–18, 1998.

[6] I. A. Howes, M. C. Smith, and G. S. Good. *Understanding and deploying LDAP directory services*. Macmillian Technical Publishing, 1999. ISBN: 1-57870-070-1.

[7] W. Matthews and L. Cottrell. The pinger project: Active internet performance monitoring for the henp community. *IEEE Communications Magazine*, 38(5), 2000.

[8] SGI. Performance co-pilot: Monitoring and managing system-level performance. `http://www.sgi.com/software/pcp/`.

[9] R. Wolski, N. Spring, and J. Hayes. Predicting the cpu availability of time-shared unix systems. Technical Report CS98-602, UCSD, 1998.

[10] R. Wolski, N. T. Spring, and J. Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Future Generation Computing Systems, Metacomputing Issue*, 15(5–6):757–768, Oct. 1999.