# PERFORMANCE EVALUATION OF LINEAR ALGEBRA ROUTINES

**E. Caron**[1]
**F. Desprez**[1]
**M. Quinson**[2]
**F. Suter**[3]

## Abstract

In this paper we presents a tool for the dynamic forecast of performance of linear algebra routine as well as communication between clusters. FAST (Fast Agent's System Timer) is a software package allowing client applications to obtain an accurate forecast of routine needs in terms of completion time, memory space, and number of communication, as well as current system availability. FAST relies on existing low-level software packages, i.e. network and host monitoring tools, and some of our developments in sequential and parallel computational routines modeling. The FAST internals and user interface are presented, as well as experimental results and validation on different applications.

Key words: Performance evaluation, parallel computing

*The International Journal of High Performance Computing Applications*, Volume 18, No. 3, Fall 2004, pp. 373–390
DOI: 10.1177/1094342004046046


## 1 Introduction

Numerical computing on clusters or federations of clusters requires us to have an accurate performance evaluation of communications and computations. Today's machines are often shared between several users. A runtime performance evaluation is thus necessary to ensure a perfect load-balancing of applications. This knowledge can be expressed by static information about the memory space and the computational power needed by a routine and the theoretical performance of an interconnection network. When dealing with parallel routines, this static information also concern the decomposition in calls to sequential counterparts, the communication scheme of the routine, and the architecture of the target platform. Besides this static information, the scheduler has to take dynamic information into account. Indeed, in the case of a non-dedicated platform, it is mandatory to take into account parameters such as processor load and the actual availability of the network at runtime.

Several performance criteria can be used to select a solution among several. Indeed a "best" solution can be the one with the smallest completion time or one minimizing the communication amount or also one achieving a "satisfying" execution time but using fewer processors than the others. If the end-user has to select his/her performance criterion, the computation and communication time acquisition process has to be as transparent as possible. Furthermore, the response time of the performance forecasting tool has to be low.

In this paper, we present the FAST library (Desprez et al., 2001; Quinson, 2002), a dynamic performance forecasting tool, and its extension to handle parallel routines (Caron and Suter, 2002). The main goal of FAST is to constitute a simple and consistent Software Development Kit (SDK) to provide a client application (typically an agent scheduler) pertinent and accurate information about the different components of the execution platform, 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.

FAST is able to forecast time and space needs of routines using a model of the routines involved in the computation. Moreover, appropriate monitoring tools allow

[1] LABORATOIRE DE L'INFORMATIQUE DU PARALLÉLISME UMR CNRS-ÉNS LYON-INRIA-UCBL 5668, 69364 LYON CEDEX 07, FRANCE

[2] COMPUTER SCIENCE DEPARTMENT, UNIVERSITY OF CALIFORNIA, SANTA BARBARA, SANTA BARBARA, CA 93106 USA

[3] LABORATOIRE INFORMATIQUE ET DISTRIBUTION, UMR CNRS-INP-INRIA-UJF 5132 ZIRST 51, AVENUE JEAN KUNTZMANN 38330 MONTBONNOT SAINT MARTIN, FRANCE

us to measure the dynamically changing availabilities of different computation and communication resources. Finally, FAST is able to aggregate these two types of information in order to forecast the current computation time of a given task on a given machine. The last version of FAST also handles parallel routines. These routines are harder to time and thus to benchmark but easier to analyze. Indeed, they can often be reduced to a succession of computation and communication phases. Computation phases are composed of calls to one or several sequential routines while communication phases are made of point-to-point or global exchanges. Timing becomes even more tedious if we add the handling of data redistribution and the choice of the virtual processor grid where computations are performed. So it seems possible and interesting to combine code analysis and information given by FAST about sequential execution times and network availability.

The remaining of this paper is organized as follows. In Section 2, we give a motivating example. Then, in Section 3, we give some references on previous work both about system availability measurements and computational routines modeling. In Section 4, we present the FAST SDK and its different features. Then we present the modeling of several parallel linear algebra routines. Finally, and before a conclusion, we present an experimental validation of our developments using three different applications.

## 2 Motivating Example

Let us assume that a client wants to solve a problem involving two sets of data, $A$ and $B$. To perform this operation, three servers with different computational powers are available. Data $A$ are distributed on the first server ($S_1$) while data $B$ are distributed on the second server ($S_2$). Figure 1 shows this configuration. In this figure, a fourth server ($S_4$) is available which aggregate the resources of $S_1$ and $S_2$.

With such a configuration and assuming that data have to be aligned to perform the computation, several solutions can be considered:

1. redistribute $B$ on $S_1$, and compute;
2. redistribute $A$ on $S_2$, and compute;
3. redistribute $A$ and $B$ on $S_3$, and compute;
4. redistribute $A$ and $B$ on $S_4$, and compute.

The scheduler has to choose one solution among all these configurations. Each configuration has a cost which can be computed by carefully analyzing the load of the servers, the cost of the communications between the different elements, and the cost of each computation. This last cost is also a function of the number of processors involved. All these costs must be computed at run-time using static information (from the benchmarking and the analysis of the algorithms) and dynamic ones.

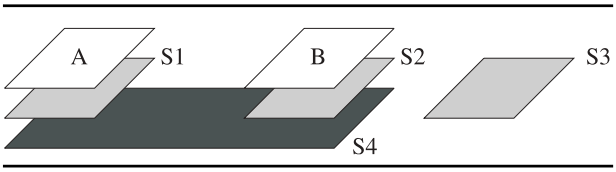


**Fig. 1 Sample of configuration for an application using computational servers.**

## 3 Related Work

### 3.1 SYSTEM AVAILABILITY

Several tools exist that are able to monitor a distributed system and extract information that can be used by a scheduler. In the following we describe some of these tools, PingER, Performance Co-Pilot, Bricks and NWS, and explain why NWS has been chosen to develop FAST.

PingER (Matthews and Cottrell, 2000) is a distributed monitoring architecture deployed on more than 600 hosts in 72 countries. Some tests are conducted periodically between hosts using the well-known `ping` tool, and the resulting data are made available from a web page. The goal of this project is to ease the search of partners in computer-intensive projects (e.g. in nuclear physic) or to help the design and understanding of the networks. Moreover, since PingER is based on the ping tool, it is limited to the latency between hosts and cannot estimate the bandwidth. Even if it would be interesting to provide such information to a scheduler, nothing seems to be done so far to allow an interactive use of this system.

Performance Co-Pilot (Silicon Graphics Inc., 2001) is a system developed by Silicon Graphics Inc. (SGI) to monitor a set of distributed machines. This project is based on daemons which collect data. It provides an API to allow a client application to access those data in a remote fashion. This project only provides an access to low-level data such as the communication outflow, the memory size of a process, etc. The priority is to acquire some knowledge about the use of each kind of resource instead of determining their availability for new processes. This type of information cannot be used directly by a scheduler to forecast the execution time of a routine. Moreover, deducing the availability of a resource from its actual use is not trivial. For instance, because of the priority mechanism used for Unix processes, a machine overloaded by low-priority processes is still available for processes of higher priority. This makes it very difficult to deduce the time-slice that a process will get from the raw load of the CPU (Wolski et al., 2000).

Bricks (Aida et al., 2000) is a simulation tool developed to compare different scheduling policies on Grid computing platforms under the same execution conditions. Indeed, measuring the real impact of a policy choice becomes very tedious because of non-reproducible disruptions caused by the resource sharing. Even if its goal differs from ours, this project is related to FAST because simulating the Grid implies to model it. Bricks proposes to model a complete Grid computing platform with queues. For instance, a server is a queue where tasks are stored when they arrive, and are released once performed. Each task is modeled by the amount of computation and communication needed for its completion. This approach has 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 allow to approach them. Furthermore this approach is relatively computation-intensive, which prevents its use in an interactive framework.

The Network Weather Service (NWS; Wolski et al., 1999) is a project led by Rich Wolski at the University of California, Santa Barbara. NWS includes some distributed sensors to monitor the actual state of the system and statistical forecasters to deduce the future evolution. It allows us 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 resource use, but also about their availability. One of the major advantages of NWS is that it allows us not only to obtain dynamic information about the platform availability, but also to predict their future evolution using statistical analysis.

## 3.2 COMPUTATIONAL ROUTINES MODELING

To obtain a good schedule for a parallel application, it is mandatory to initially determine the computation time of each of its tasks and communication costs introduced by parallelism. The most common method to determine these times is to describe the application and model the parallel computer that executes the routine.

The modeling of a parallel computer and more precisely of communication costs can be considered from different points of view. A straightforward model can be defined ignoring communication costs. A parallel routine can thus be modeled using the Amdahl law. This kind of model is not realistic in the case of distributed memory architectures. Indeed, on such a platform, communications represent an important part of the total execution time of an application and cannot be neglected. Furthermore, this model does not allow us to handle the impact

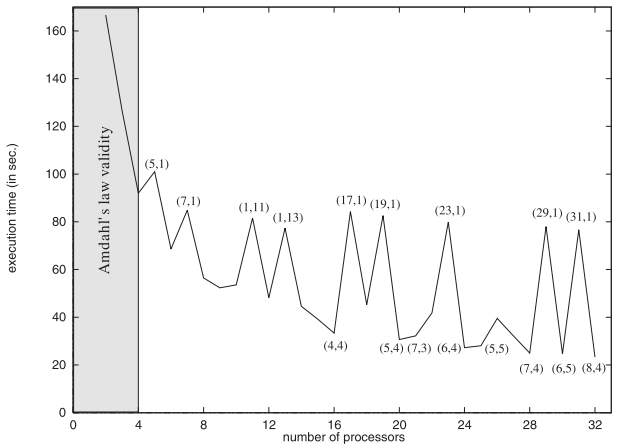


**Fig. 2 Execution times of a 4096×4096 matrix–matrix multiplication achieved on the best grid using a given number of processors.**

of processor grid shape on routine performance. In such conditions, it becomes important to study the communication scheme of the considered routine to determine the best size and shape that should be used. For instance, in the matrix–matrix multiplication routine of the SCALAPACK library, compact grids achieve better performance than elongated ones. However, Figure 2 shows the weakness of this law. Indeed, in this example the Amdahl law suggests that four processors should be used because the execution time increases for five processors because of the shape of the grid. However, better execution times can be obtained with more processors.

The *delay* (Rayward-Smith, 1987) and *LogP* (Culler et al., 1996) models have been designed to take communications into account. The former as a constant delay $d$ while the latter considers four theoretical parameters: transmission time from a processor to an other ($L$), computation overhead of a communication ($o$), network bandwidth ($g$), and number of processors ($P$). However, the delay model may be not accurate enough while *LogP* is too complicated to model a large heterogeneous platform (i.e. Grid environment) in a simple but realistic way.

About the modeling of parallel algorithms such as those used in libraries such as SCALAPACK, several approaches are possible. Dongarra and Roche (2002) aim at using parallel routines on clusters when it is possible to achieve better performance than a serial execution. Their model consists in identifying the sequential calls in the code of the parallel routine and to replace them by functions depending on data sizes and relative performance of the sequential counterpart of this routine on a node of the cluster. However, this model does not take into account

the load variation of the execution platform and the optimizations based on pipelining made in the routines. In Domas et al. (1996), a model of parallel routines is proposed based on estimation of sequential counterparts. The result of this estimation is a polynomial whose variables depend on the matrix sizes. Coefficients are specific to the couple {algorithm, machine}. These parameters are determined by interpolating, dimension per dimension, a set of curves obtained from the execution of the routine with small data. However, the model proposed in Domas et al. (1996) is highly focused on the *LU* factorization routine. Furthermore this method is not generic since a generated C code has to be recompiled depending on the problem studied. The *ChronosMix* environment (Bourgeois et al., 1999) uses micro-benchmarking to estimate execution times of parallel applications. This technique consists of performing extensive tests on a set of C/MPI instructions. Some source codes, written in this language and employing that communication library, are then parsed to determine their execution times. The use of this environment is therefore limited to such codes, but most of numerical libraries are still written in Fortran. Moreover, in the particular case of SCALAPACK, communications are handled through the BLACS library implemented on top of MPI but also on top of PVM.

Dimemas (Badia et al., 2003) is a performance prediction tool developed at the Technical University of Catalonia and distributed by PALLAS GmbH. The goal of this project is to use the traces from the MPI analysis tool Vampir to predict the performance of a parallel program on a given architecture. According to the authors, this tool is designed for "what-if analysis", i.e. the analysis of the performance variations induced by changes in the platform, and not for the forecasting of program performances given the current load of the system.

Led by a team at Rice University, the goal of the Grid Application Development Software (GrADS) project is to enable the routine development and performance tuning of Grid applications by simplifying distributed heterogeneous computing. In Petitet et al. (2001), a performance modeler is defined for the modeling of ScaLAPACK routines in a Grid environment. The performance modeler is used to find a collection of machines available for the problem. When a new machine is found, it is added to the collection until the time estimate for the application execution time increases. The experiments made on the PDGESV routine prove the validity of this approach on such a large-scale (and heterogeneous) platform.

Same project another way, Pablo group members and researchers from six other universities are working with the project's leadership at Rice to investigate the real-time monitoring and adaptive control (Vetter and Reed, 2000). The transient, rarely repeatable behavior of the Grid means that the standard model of post-mortem performance optimization must be replaced by a real-time model that optimizes application and run-time behavior during program execution. This model of closed-loop control will require design of new performance specification interfaces and assertion mechanisms, measurement tools, and decision procedures for adaptive configuration of application modules and run-time system components, as well as compiler interfaces for just-in-time compilation and performance prediction. Via run-time validation of these performance contracts, the real-time measurement and control system will activate dynamic recompilation and run-time resource configuration.

## 4 FAST, Fast Agent's System Timer

### 4.1 INTRODUCTION

As shown in Figure 3, FAST is composed of several modules. The first module (FAST benchmarker) is launched when FAST is deployed on a new server. This module is in charge of the modeling of the sequential routines and the analysis of parallel routines that can be executed on this server. The intermediary module includes the external tools used by FAST. Finally, we have the library itself (and its components) which is called for each estimation.

To obtain the needed information about system availabilities, FAST uses the NWS. FAST also relies on another software, LDAP (Lightweight Directory Access Protocol; Howes et al., 1999), which is a distributed and hierarchical database system widely used in the grid computing community. It allows us to split the database and map each part to an administrative organization. That way, the manager of each site handles the data relative to his set of machines. However, LDAP is explicitly optimized for read-and-search operations but not for write operations. This system is thus not designed to store highly dynamic data, but is perfectly adapted to the use made by FAST to store static data such as routine cost models. The next sections detail FAST's modules and their interactions.

### 4.2 SYSTEM AVAILABILITY ACQUISITION MODULE

The main goal of the system availability acquisition module is to discover the external load using NWS to take into account that the platform can be shared with other users. In its current version, FAST can monitor the CPU and memory load of hosts, as well as latency and bandwidth of any TCP link. Monitoring new resources such as non-TCP links should be easy, provided that these information are available from NWS.

The integration of NWS raised some of its shortcomings we tried to tackle. First, FAST significantly reduces

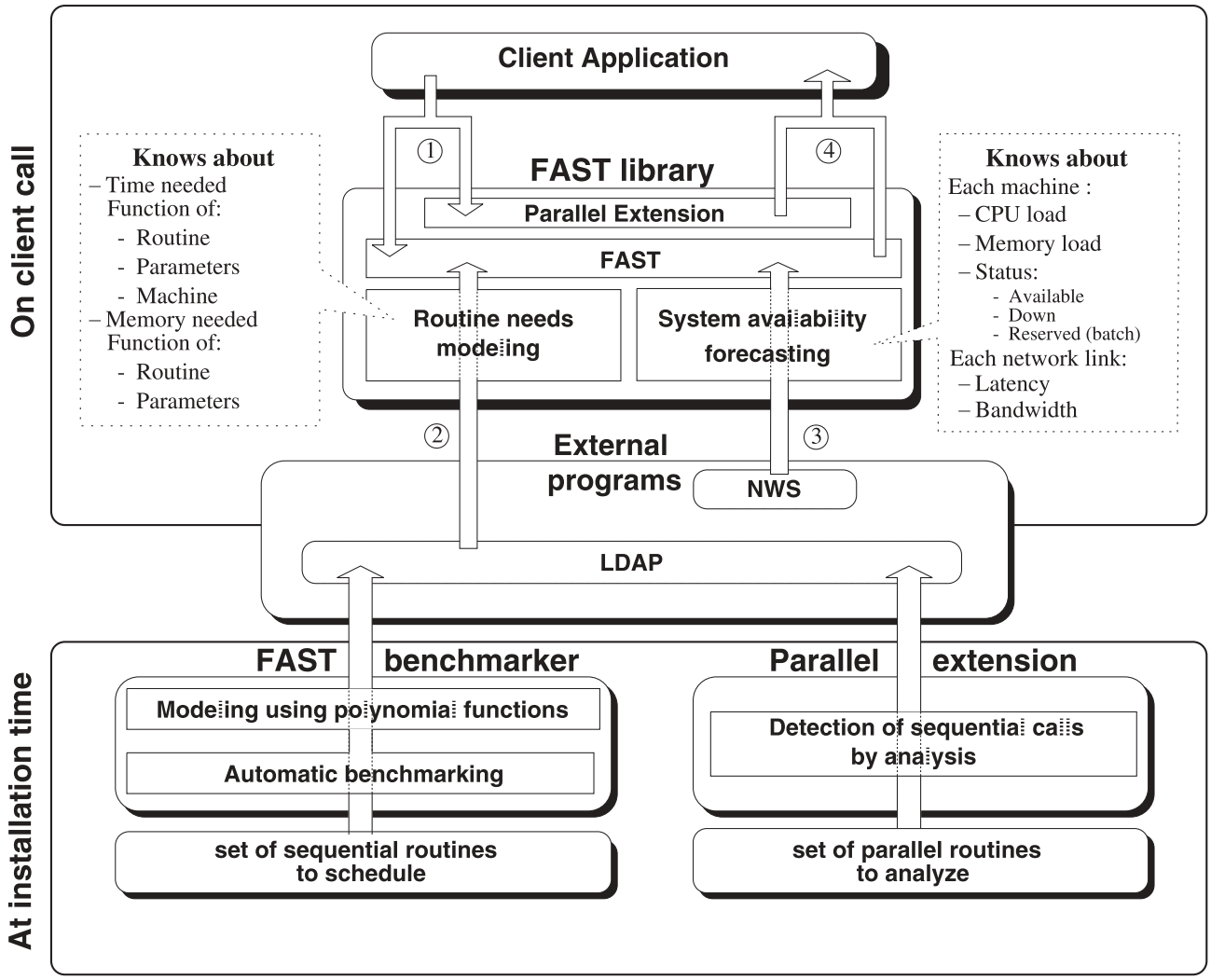


**Fig. 3 FAST Overview.**

NWS response time. Indeed, NWS is composed of four types of daemon, distributed across the network. This causes a non-negligible response time. Since the measurements are performed periodically, it is unnecessary to ask NWS before the next measure has been performed. FAST includes a cache system to call NWS only when necessary. To handle dynamic changes, a validity deadline is associated to each record stored in the cache. If this deadline is exceeded, NWS may have performed a new measurement and the value is refreshed.

Moreover, NWS allows the administrator to specify which network tests to run in order to reduce the number of host pairs leading measurements to make the system less intrusive. However, NWS is then unable to combine the direct measurements automatically. For example, given three machines A, B, and C, if the administrator asked for measurements for the pairs (A,B) and (B,C), it is then impossible to obtain an estimation between A and C from NWS. In this case, FAST produces a combination of the direct tests. To do so, FAST search a path between

A and C in the graph of monitored links. The estimated bandwidth is the minimum of those of the path found while the estimated latency is the sum of the results of direct tests. Even if this value is less accurate than a real measurement, it is still interesting when no other information is available.

## 4.3 COMPUTATION ROUTINE MODELING

One of the most fundamental contributions of FAST is its capacity to forecast 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 considered.

**4.3.1 Sequential Routines.** Some routines are regular and easy to time. There are typically sequential numerical routines such as those of the BLAS library (Dongarra et al., 1988). To estimate the execution time of such routines, the classical approach consists of extracting a model from a manual or automatic source code analysis. However, this kind of approach is not always possible as it requires an access to the source code. For instance, on most computers, an highly optimized version of the BLAS library is provided by vendors or can be generated automatically (Dongarra et al., 2001). However, even when the source code is available, many parameters have to be taken into account to forecast the performance of a sequential code on a given machine. Indeed, the forecasting tool has to handle not only the implementation of the routine but also the optimizations performed by the compiler and/or the processor, some of the operating system characteristics or even memory and cache policies. Taking all these parameters into account implies an extensive study of each couple {code; machine}, which represents a huge amount of work on large heterogeneous platform.

To tackle this issue, another classical approach consists of using a generic benchmark of the machine to determine how many elementary operations can be computed per unit of time, and then to count the number of such operations for each routine to estimate. However, this approach does not take cache effects in account, which leads to important performance variation. Indeed the main optimization of sequential kernels is based on block algorithms, which allow a better use of hierarchical memories. This leads to drastic improvements (Dongarra et al., 1991), for example, for matrix–matrix routines (level 3 BLAS) in terms of Flop per second. Assuming that performance of a computer can be given as a constant amount of flop/s is therefore wrong.

The approach chosen in FAST consists of benchmarking the performance of each routine for a representative set of parameters when FAST is deployed on a new computational server. The results are then modeled by polynomial regression and stored in a LDAP tree. 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.

As said before other routines are harder to time, but can be easily and accurately studied, typically parallel regular routines. In the next section we detail how are handled these routines into FAST.

**4.3.2 Parallel Routines.** In its current version, FAST only handles some routines of the parallel dense linear algebra library SCALAPACK. For such routines, the description step consists in determining which sequential counterparts are called, their calling parameters (i.e. data sizes, multiplying factors, transposition, etc.), the communication schemes and the amount of data exchanged. Once this analysis completed, the computation part can easily be forecasted since FAST is able to estimate each sequential routines called from the parallel code studied.

Furthermore, processors executing a SCALAPACK code have to be homogeneous to achieve optimal performance, and the network between these processors has also to be homogeneous. It allows us two major simplifications. First, processors being homogeneous, the benchmarking phase of FAST can be executed on only one processor. Then concerning communications we only have to monitor a few representative links to obtain a good overview of the global behavior.

For the estimation of point-to-point communications we chose the $\lambda + L\tau$ model where $\lambda$ is the latency of the network, $L$ the message size and $\tau$ the time to transfer an element, i.e. the inverse of the network bandwidth. $L$ can be determined during the analysis while $\lambda$ and $\tau$ can be estimated by FAST using NWS. In a broadcast operation, the $\lambda$ and $\tau$ constants are replaced by functions depending on the processor grid topology (Caron et al., 2000). If we consider a grid with $p$ rows and $q$ columns $\lambda_p^q$ will be the latency for a column of $p$ processors to broadcast their data (assuming a uniform distribution) to processors that are on the same row and $1/\tau_p^q$ will be the bandwidth. In the same way $\lambda_p^q$ and $\tau_q^p$ denote the time for a line of $q$ processors to broadcast their data to processors that are on the same column. These functions directly depend on the implementation of the broadcast. For example, on a cluster of workstations connected through a switch, the broadcast is actually executed following a tree. In this case $\lambda_p^q$ will be equal to $\lceil log_2 q \rceil \times \lambda$ and $\tau_p^q$ equal to $(\lceil (log_2 q)/p \rceil) \times \tau$ where $\lambda$ is the latency for one node and $1/\tau$ as the average bandwidth.

Samples of parallel routine modeling will be given in Section 5.

### 4.4 FAST API

The API of FAST is composed of two levels. The low-level interface includes functions to acquire raw information about routine needs and system availabilities. Routines of the high-level interface combine these results into values which can be directly used by a client application.

First the system has to be initialized:

```
fast_init (LDAP_host, LDAP_root, NWS_ns,
          NWS_fore).
```

This function tells FAST how to find needed information. The LDAP tree is located on `LDAP_host` under `LDAP_root`. NWS name server and forecaster are respectively on the `NWS_ns` and `NWS_fore` hosts.

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

```
fast_comm_time (source, dest, msg_size,
          &value)
```

computes the needed time to move data of size `msg_size` from `source` to `dest`, taking in account the actual network load. The result (in seconds) is stored in `value`.

```
fast_comp_time (host, function, data_desc,
          &value)
```

gives the forecasted computation time for `function` executed on `host` for the parameters described by `data_desc` (which format is detailed at the end of this section). An error code is returned whether this host can not compute this function or if there is not enough free memory.

There are two other functions to forecast the execution time of a parallel routine. The first is:

```
fast_parallel_comp_time (host_list,
          function, data_desc, &value)
```

which allows the client to get in `value` the execution time of `function` on a processor grid for the parameters listed in `data_desc`. Hosts listed in `host_list` can be a subset of those composing the grid. FAST needs at least two hosts to acquire network information. The description of the processor depends on the type of the routine considered. In the case of dense linear algebra routines, it is a two-dimensional (2D) grid defined by its number of rows and columns. Furthermore, an extra parameter has been added in `data_desc` with regard to the sequential version to handle the data distribution. This parameter is the distribution block size, assuming data are distributed in a bi-dimensional block-cyclic way. The second function is:

```
fast_parallel_comp_time_best (host_list,
function, data_desc, nbp, &res).
```

In this call, the description of the processor grid is replaced by `nbp`, the maximal number of processors to allocate for the execution of `function`. FAST is then in charge to determine what are the optimal shape and size of grid for that execution. The description of this optimal grid and the execution time associated are returned to the client application in `res`.

The low-level interface is composed of two functions.

```
fast_need (resource, host, function,
data_desc, &value)
```

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

```
fast_avail (resource, host1, host2,
&value)
```

gives the available amount of `resource` on `host1`. (or between `host1` and `host2` in the case of network resources). 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. The `host2` is only significant for network resources and is ignored for other resource queries.

FAST groups the description of arguments passed to the routines studied in a table called `datadesc`. Each of these arguments can be a scalar, vector or matrix, with base type being integer, double or char. `datadesc` stores the information needed about the routine arguments to predict the execution time and memory space needed to perform the operation. For scalar arguments, the value is stored directly while we only store descriptive characteristics about matrices and vectors like their size and shape (triangular or band matrix, etc.).

## 5 Parallel Routine Model Samples

### 5.1 MATRIX–MATRIX MULTIPLICATION ROUTINE

The routine `pdgemm` of the SCALAPACK library computes the product

$$C = \alpha\, op(A) \times op(B) + \beta C$$

where *op(A)* (resp. *op(B)*) can be $A$ or $A^t$ (resp. $B$ or $B^t$). In this paper we have focused on the $C=AB$ case.[1] $A$ is a $M\times K$ matrix, $B$ a $K\times N$ matrix, and the result $C$ a $M\times N$ matrix. As these matrices are distributed on a $p\times q$ processor grid in a block-cyclic way with a block size of $R$, computation time is expressed as

$$\left\lceil \frac{K}{R} \right\rceil * \texttt{dgemm\_time}, \qquad (1)$$

where `dgemmtime` is given by the following FAST call

```
fast_comp_time (host, dgemm_desc,
&dgemm_time),
```

where `host` is one of the processors involved in the computation of `pdgemm` and `dgemm_desc` is a structure containing information such as the size of the matrices involved, if they are transposed or not, and so on. Used matrices are of size $\lceil M/p \rceil \times R$ for the first operand and $R\times\lceil N/q \rceil$ for the second one.

To accurately estimate the communication time, it is important to consider the `pdgemm` communication scheme. At each step, one pivot block column and one pivot block row are broadcast to all processors, and independent products take place. Amounts of data communicated are then $M\times K$ for the broadcast of rows and $K\times N$ for the broadcast of the columns. Each broadcast is performed block by block. The communication cost of the `pdgemm` routine is thus

$$(M\times K)\tau_p^q + (K\times N)\tau_q^p + (\lambda_p^q + \lambda_q^p)\left\lceil \frac{K}{R} \right\rceil. \qquad (2)$$

This leads us the following estimation for the routine `pdgemm`:

$$\left\lceil \frac{K}{R} \right\rceil \times \texttt{dgemm\_time}$$

$$+ (M\times K)\tau_p^q + (K\times N)\tau_q^p + (\lambda_p^q + \lambda_q^p)\left\lceil \frac{K}{R} \right\rceil. \qquad (3)$$

If we assume that broadcast operations are performed following a tree, $\tau_p^q$, $\tau_q^p$, $\lambda_p^q$ and $\lambda_q^p$ can be replaced by their values depending on $\tau$ and $\lambda$. These two variables are estimated by the following FAST calls

```
fast_avail (bandwidth, source, dest, &tau)
```

and

```
fast_avail (latency, source, dest, &lambda),
```

| Upper / Lower | Left / Right | Transposition | Operation | Case |
|---|---|---|---|---|
| U | L | N | B = A \ α B | 1 |
| U | R | T | | |
| U | L | T | B = α B / A | 2 |
| U | R | N | | |
| L | L | N | B = A \ α B | 3 |
| L | R | T | | |
| L | L | T | B = α B / A | 4 |
| L | R | N | | |

**Fig. 4 Correspondence between values of calling parameters and actually performed computations for the `pdtrsm` routine.**

where the link between `source` and `dest` is one of those monitored by FAST. Equation 2 thus becomes

$$\frac{\left(\frac{\lceil log_2 q \rceil \times M \times K}{p} + \frac{\lceil log_2 p \rceil \times K \times N}{q}\right)}{\texttt{tau}}$$

$$+ \left\lceil \frac{K}{R} \right\rceil (\lceil log_2 q \rceil + \lceil log_2 p \rceil) \times \texttt{lambda} \qquad (4)$$

## 5.2 TRIANGULAR SOLVE ROUTINE

The routine `pdtrsm` of the SCALAPACK library can be used to solve the following systems: $op(A)*X = \alpha B$ and $X*op(A) = \alpha B$ where $op(A)$ is equal to $A$ or $A^t$. $A$ is a $N\times N$ upper or lower triangular matrix which can be unitary or not. $X$ and $B$ are two $M\times N$ matrices.

Figure 4 presents graphical equivalents of `pdtrsm` calls depending on the values of the three parameters `UPLO`, `SIDE` and `TRANSA` defining respectively whether $A$ is upper or lower triangular, $X$ is on the left or right side of $A$ and if $A$ has to be transposed or not. The eight possible calls can be grouped into four cases as shown in Figure 4. These cases can also be grouped into two families, which achieve different performance depending on the shape of the processor grid. The first includes cases 1 and 3 and will execute faster on row-dominant grids, while the second contains cases 2 and 4 and will achieve better performance on column-dominant grids. Finally, the routine `pdtrsm` is actually composed of calls to the subroutine `pbdtrsm` which computes the same operation but when the number of rows of $B$ is less or equal to the size of a distribution block, $R$. To obtain the computation cost of the `pdtrsm` routine of SCALAPACK, the cost of a call to `pbdtrsm` has then to be multiplied by $\lceil M/R \rceil$ as each call to that routine solves a block of rows of $X$.

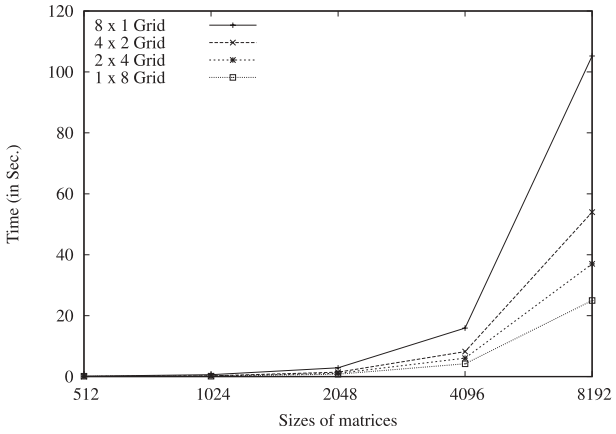


**Fig. 5 Performance of a pdtrsm for different grid shapes.**

To analyze why some shapes are clearly better than some others, we focus on the case $X*A = B$ where $A$ is a non-unitary upper triangular matrix. Figure 5 shows the results we obtained on eight processors for several grid shapes using SCALAPACK v1.6 (based on PBLAS v1.0). In this particular case (number 2 in Figure 4), we can see that an eight-processor row grid (i.e. 1×8) achieves performance 3.7 times better than an eight-processor column grid (i.e. 8×1). We first focused our analysis on this case where the execution platform is a row, and then extended it to the more general case of a rectangular grid.

**5.2.1 On a Row of Processors.** To solve the equation system presented in Figure 6 (where $A$ is an $n \times n$ block matrix), the principle is the following. The processor that owns the current diagonal block $A_{ii}$ performs a sequential triangular solve to compute $b_{1i}$. The resulting block is broadcast to the other processors. The receivers can update the unsolved blocks they own, i.e. compute $b_{1j} - b_{1i}a_{ij}$, for $i < j \leq n$. This sequence is thus repeated for each diagonal block, as shown in Figure 7.

`trsm_time` denotes the time to compute a sequential triangular solve. This time is estimated by

```
fast_comp_time (host, trsm_desc, &trsm_time),
```

where `host` is one of the processors involved in the computation of `pdtrsm` and `trsmdesc` is a structure containing information on matrices and calling parameters.

Blocks solved are broadcast along the ring but the critical path of `pdtrsm` follows also this ring. So we have

$$
\begin{aligned}
b_{11} &= b_{11}/a_{11} \\
b_{12} &= (b_{12} - b_{11}a_{12})/a_{22} \\
&\vdots \quad \vdots \\
b_{1j} &= (b_{1j} - b_{11}a_{1j} - \cdots - b_{1(j-1)}a_{(j-1)j})/a_{jj} \\
&\vdots \quad \vdots \\
b_{1j} &= (b_{1n} - b_{11}a_{1n} - \cdots - \cdots - b_{1(n-1)}a_{(n-1)n})/a_{nn}
\end{aligned}
$$

**Fig. 6 Computations performed in a `pbdtrsm` call, where $A$ is an $n \times n$ block matrix.**

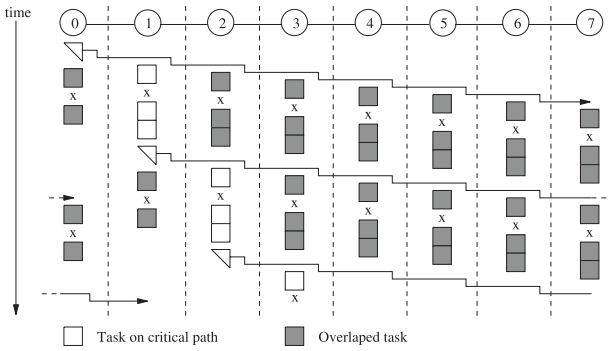


**Fig. 7 Execution on a ring of eight processors of the `pbdtrsm` routine.**

only to consider the communication between the processor that computes the sequential solve and its right neighbor. As the amount of data broadcast is $R^2$, this operation is then estimated by

$$T_{broad} = R^2\texttt{tau} + \texttt{lambda}, \qquad (5)$$

where `lambda` and `tau` are estimated by the FAST calls presented in the matrix multiplication model.

At each step, the update phase is performed calling the `dgemm` routine. The first operand for each of these calls is a copy of the block solved at this step and therefore is always an $R \times R$ matrix. The number of columns of the second operand depends on how many blocks have already been solved and can be expressed as $R\lceil (N - iR)/(qR) \rceil$ where $i$ the number of solved blocks. The corresponding FAST call is then

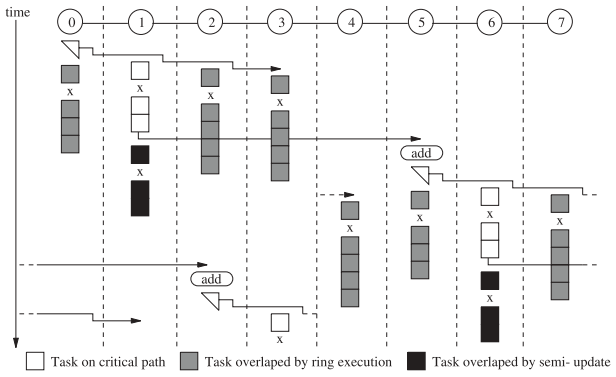


**Fig. 8 Execution on a 2×4 processor grid of the pbdtrsm routine.**

```
fast_comp_time (host, dgemm_desc_row,
&dgemm_time_row).
```

Idle times may appear if one of the receivers of a broadcast is still updating blocks from the previous step. As our model forecasts the execution time following the critical path of the routine and in order to handle these idle times, we need to apply to apply a correction $C_b$ as both sending and receiving processors have to wait until the maximum of their previously estimated times before performing this communication. Thus, equation (6) gives the computation cost model for the `pbdtrsm` routine on a row of processors:

$$\sum_{i=1}^{\lceil N/R \rceil} (\texttt{trsm\_time} + T_{broad} + C_b + \texttt{dgemm\_time\_row}). \quad (6)$$

**5.2.2 On a Rectangular Grid.** The main difference between the previous case and the general one appears in the update phase. Indeed, another pipeline is introduced in the general case. It consists of splitting the update phase into two steps. The first step updates the first ($p$–1) blocks (where $p$ is the number of processor rows of the grid) while the second deals with the remaining blocks. Once the first part has been updated, it is sent to the processor which is on the same column and on the next row. The receiving processor then performs an accumulation to complete the update of its blocks, as shown in Figure 8.

This optimization implies that we have two values for the number of columns of the second operand of the `dgemm` calls. The former can be expressed as the minimum between $R\lceil (N-iR)/(qR) \rceil$ and $R(p-1)$ while the latter will be $R(\lceil (N-iR)/(qR) \rceil - (p-1)))$ if positive. The first operand is still an $R \times R$ matrix. We then have two FAST calls to estimate these two different matrix products of the update phase.

```
fastcomptime (host, dgemmdesc1,
          &dgemmtime1),
```

and

```
fastcomptime (host, dgemmdesc2,
          &dgemmtime2).
```

Two operations are still needed to estimate the general case model, the *send* and the *accumulation* operations. For both of these, we have the same restriction on the number of columns as for the first `dgemm` of the update phase. This leads us to the following expressions:

$$T_{send} = \left( R \times min\left( (p-1)R, R\left\lceil \frac{(N-iR)}{qR} \right\rceil \right) \right) \texttt{tau} + \texttt{lambda}, \quad (7)$$

and

```
fastcomptime (host, adddesc, &addtime).
```

Here again we handle idle times by applying the same kind of correction, $C_u$, to both sender and receiver of the send operation. Moreover, it has to be noticed that when the number of columns is equal to one, the broadcast operation is replaced by a memory copy. Equation 8 gives the computation cost models for the `pbdtrsm` routine on a rectangular grid of processors:

$$\sum_{i=1}^{\lceil N/R \rceil} \left( \begin{array}{c} \texttt{trsm\_time} + T_{broad} + C_b + \texttt{dgemm\_time\_1} \\ + T_{send} + C_u + \texttt{add\_time} \end{array} \right) \quad (8)$$

## 6 Experimental Validation

After studying the internals and the API of FAST, we now present some experimental results validating our approach. First, we show how FAST and its cache system allow to improve the response time of NWS. We then study the quality of our forecasting, for sequential and parallel routines. Finally we demonstrate how FAST can be used in a scheduling context to decide of the best execution scenario.

### 6.1 RESPONSE TIME IMPROVEMENT

As indicated in Section 4.2, we tried to solve some of the shortcomings of NWS. The first optimization concerns the response time we reduced using a cache system. To prove the efficiency of the changes made, we measured the average response time over 30 runs of a direct interrogation to the NWS, when all the parts of the system are on the same host. Our test computer is a Linux Pentium

**Table 1**
**Temporal modeling quality for sequential routines.**

| | Addition | | Multiplication | | Solve | |
|---|---|---|---|---|---|---|
| | *PIII* | *Bi-PII* | *PIII* | *Bi-PII* | *PIII* | *Bi-PII* |
| Maximal Error | 0.02 s | 0.02 s | 0.21 s | 5.8 s | 0.13 s | 0.31 s |
| | (6%) | (35%) | (0.3%) | (4%) | (10%) | (16%) |
| Average Error | 0.006 s | 0.007 s | 0.025 s | 0.03 s | 0.02 s | 0.08 s |
| | (4%) | (6.5%) | (0.1%) | (0.1%) | (5%) | (7%) |

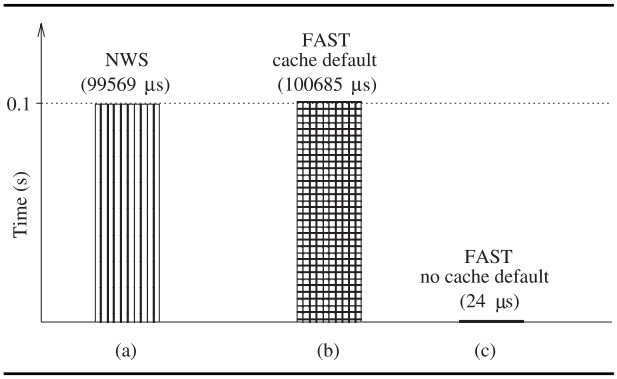


**Fig. 9 NWS and FAST response times.**

II workstation. Figure 9 presents the measured times. Figure 9(a) is the measured time for a direct interrogation to NWS, Figure 9(b) is the time for an interrogation through FAST when the asked value is not in the cache, resulting in an interrogation to NWS, and finally Figure 9(c) is the time to extract the value from the cache (avoiding completely NWS).

This experiment shows that the overhead introduced by FAST is quite low in case of cache default (about 1%), while obtaining the value from the cache is more than 4000 times quicker than an interrogation to NWS. This gain can be very interesting as a scheduler will have to ask for NWS values for each machine of the platform and for each request submission. Indeed, FAST may have to handle the same request several times per second in a regular use while NWS refreshes network measurements every 2 min and host measurements every 30 s by default.

## 6.2 FORECASTS QUALITY STUDY

We ran several experiments to study the quality of the forecasts produced by FAST. The first experiment only deals with the modeling of routine needs, both in terms of time and space, without taking into account the load of the CPU. The second is the same experiment but when the CPU is loaded. The third experiment measures the quality of the forecast for a sequence of operations executed on an heterogeneous platform. The fourth experiment concerns the estimation of the execution time of a parallel routine depending on the size and the shape of the processor grid. Finally, the last experiment shows the same kind of estimation for a given processor grid.

**6.2.1 Routine Needs Modeling.** Before showing the quality of forecasts provided by FAST, we thought it was important to measure the quality of our model concerning the function's needs (in terms of time and space) without taking into account the CPU load. We compared the modeled and the actual value for the completion time and space needed concerning three functions of the BLAS library, the matrix addition, the matrix multiplication, and the triangular solve. This comparison has been carried out on two different computers: a Pentium III, with 256 MB of memory, and a Bi-Pentium II 450 MHz also with 256 Mo of memory.

Table 1 presents the quality of the temporal model for the functions studied for matrix sizes varying between 128 and 1152. On the first line appears the maximal error in absolute value measured during the experiment. The corresponding relative error is indicated between parentheses. On the second line is listed the average error in absolute and relative values.

In the case of the matrix addition, the relative errors are higher. This may be explained by the fact that we choose to time the operation using the `rusage` system call. This allows us to cut off from the external load, but the accuracy is about 0.01 s. Since the matrix addition takes less than half a second, the timing error becomes significant.

In other cases, the maximal error is generally lower than 0.2 s, while the average error is about 0.1 s. This is very acceptable in our context as we saw in Section 6.1 that the response time of NWS's is about 0.3 s. The rela-

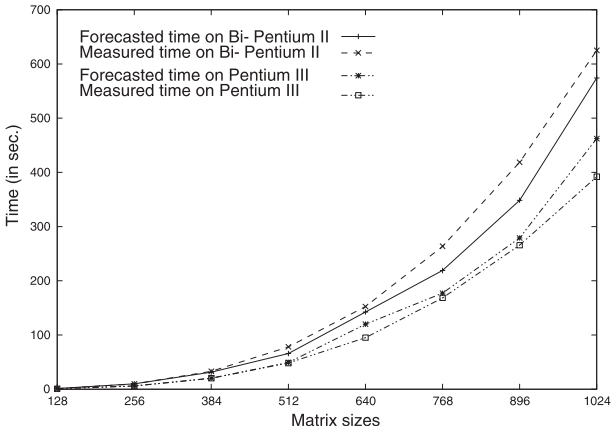


**Fig. 10 Comparison between measured and forecasted times for one execution of the `dgemm` routine.**

tive error is lower than 7% on average and slightly more than 15% in the worst case. For the matrix multiplication on the Bi-PII machine, we observe an error up to 5.8 s, but this is small compared to the computation time since it only represents 4% of the measured time.

In this case, 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 performing a matrix operation is the size of the code (which is constant) plus the size of the data (which is obviously a polynomial function).

**6.2.2 Forecast of one Sequential Routine Execution Time.** This experiment aims at validating FAST forecast when the CPU is loaded. This only deals with the completion time and not with the memory needed since FAST refuses to forecast the completion time if it detects that there is not enough memory to perform the operation. We compared our forecast with the measured time for the matrix–matrix multiplication operation. To simulate an external load, a computation task was running during the experiment. We used the same testbed as in the previous experiment.

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

**6.2.3 Forecast of the Execution Time of a Sequence of Sequential Routines.** To judge the quality of our predictions for a sequence of sequential routines, we chose to perform a complex matrix multiplication. This operation gives a task graph that leads to calls to high-level routines and some data redistribution between processor sets. Input data are two complex matrices $A$ and $B$ decomposed in real and imaginary parts. The problem to solve can be expressed as

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

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 heterogeneous platform composed of only two machines. The client is launched on a Pentium II workstation with 128MB of memory, while two servers are deployed on a SMP computer with four Pentium III and 256 MB of memory. The client asks FAST to obtain a estimation of the execution time of the complete sequence, then initiates the computation and measures the actual execution time. Figure 11 shows the (static) task allocation between computers.

Figure 12 compares the forecast 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 manages to forecast the completion time with an error smaller than 25% in the worst case and smaller than 12% on average.

**6.2.4 Execution Time of a Parallel Routine Depending on the Shape of the Grid.** To validate our parallel routine handling in FAST, we ran several tests on the *i-cluster* which is a cluster of 225 HP e–vectra nodes (Pentium III 733 MHz with 256 MB of memory per node) connected through a Fast Ethernet network via HP Procurve 4000 switches. So the broadcast is actually executed following a tree.

Figure 13 presents the estimated time given by our model for the `pdgemm` routine on all possible grids from 1 up to 32 processors and Figure 14 presents the actual execution time of the same routine on the *i-cluster*. Matrices are of size 2048 and the block size is fixed to 64. The $x$-axis represents the number of rows of the processor grid, the $y$-axis the number of columns, and the $z$-axis the execution time in seconds. We can see that the estimation given by our extension is very close to the experimental execution times. The maximal error is less than 15% while the average error is less than 4%. Furthermore, these figures confirm the impact of topology on performance. Indeed, compact grids achieve better performance than elongated ones because of the symmetric communication pattern of the routine. The different stages for row and column topologies can be explained by the log term introduced by the broadcast tree. These results shows that our evaluation can be efficiently used to choose

- Client:
  - Send $A_r$, $A_i$ and $B_r$ to server 1 ;
  - Send $A_r$, $A_i$ and $B_i$ to server 2.

- Server 1 :
  - $C_{r_1} = A_r \times B_r$ ;
  - $C_{i_2} = A_i \times B_r$ ;
  - Send $C_{i_2}$ to server 2 ;
  - $C_r = C_{r_1} - C_{r_2}$ ;
  - Send $C_r$ to client.

- Server 2:
  - $C_{r_2} = A_i \times B_i$ ;
  - $C_{i_1} = A_r \times B_i$ ;
  - Send $C_{r_2}$ to server 1 ;
  - $C_i = C_{i_1} + C_{i_2}$ ;
  - Send $C_i$ to client.

**Fig. 11 Task sharing for the complex matrix multiplication.**

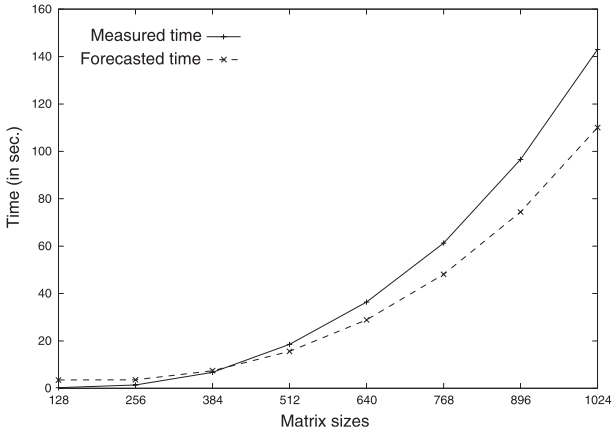


**Fig. 12 Comparison between measured and forecasted times for a sequence of operations.**

a grid shape for a parallel routine call. Combined with a run-time evaluation of parameters such as communication, machine load or memory availability, we can then build an efficient scheduler for Network Enabled Servers.

We ran the same kind of experiment for the triangular solve routine. Figure 15 shows estimations produced by our model. Comparing these results with those of Figure 5, we can see that our model allows us to forecast performance evolution with regard to changes in the processor grid shape. The average error rate is less than 12%. It has to be noticed that if this rate is greater than the one achieved for the multiplication, it comes mostly from the difficulty to model the optimizations done using pipelining. Our model is thus very inaccurate for the 2×4 case, as the execution time is underestimated. However, if we only consider the execution on a processor row, which is the best case in term of performance, the error rate is then less than 5%.

**6.2.5 Forecast of a Parallel Routine Execution Time.** We also tried to validate the accuracy of the extension for a given processor grid. Figure 16 shows the error rate of the forecast with regard to the actual execution time for matrix multiplications executed on a 8×4 processor grid. Matrices are of sizes 1024 up to 10240. We can see that our extension provides very accurate forecasts as the average error rate is less than 3%.

## 6.3 UTILITY IN A SCHEDULING CONTEXT

**6.3.1 Comparison of FAST and NETSOLVE Forecasts.** In order to test FAST in real conditions, 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. Again, the test platform is quite simple because we only want to study the forecasts, and not their impact. There was only one server and one server, located on machines from two different labs separated by about 700 km and connected through a WAN.

Figures 17 and 18 present the results of this experiment. Figure 17 compares the forecasts of NETSOLVE, FAST, and the actual time in terms of completion time while Figure 18 does the same for the communication time.

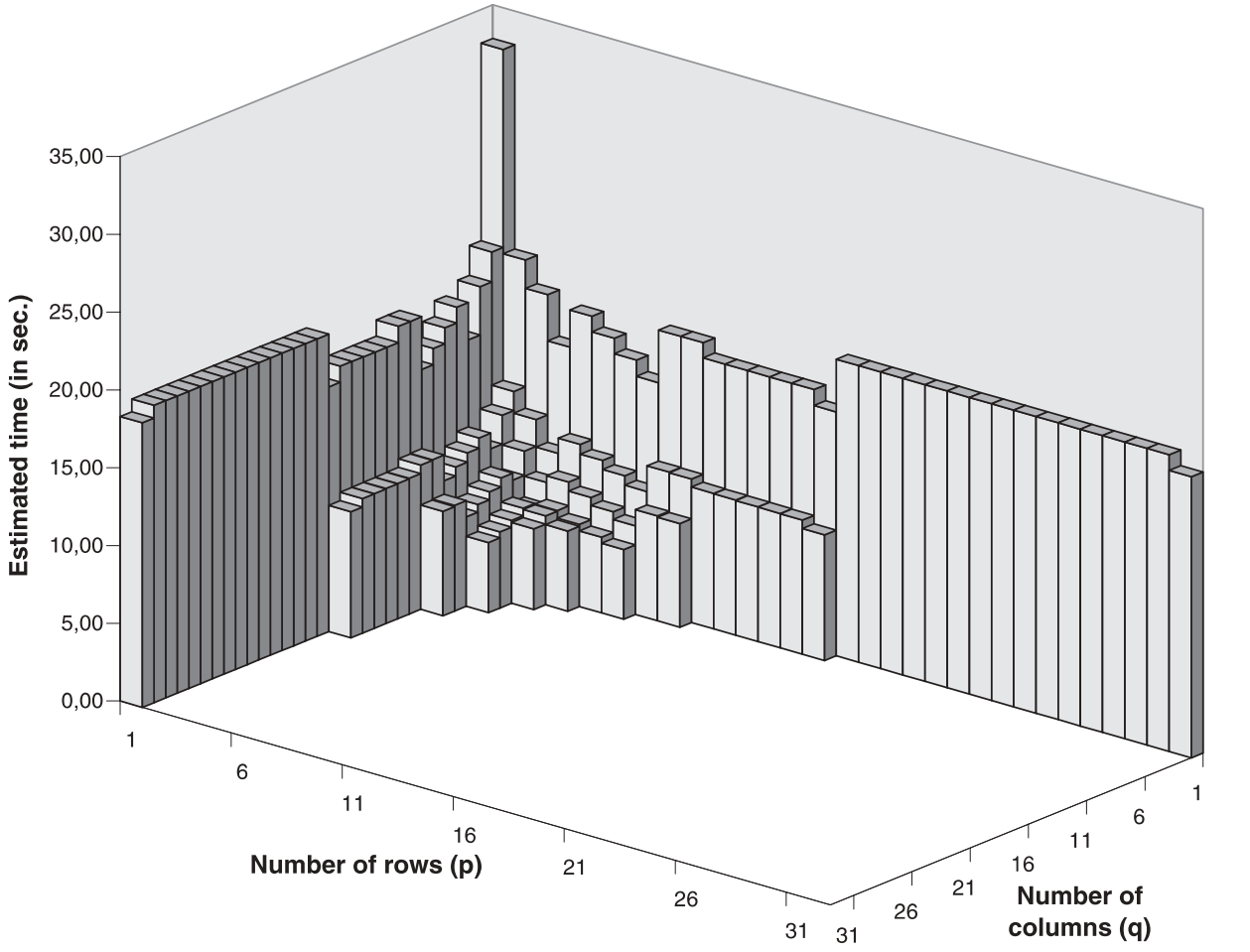


**Fig. 13 Estimated time for the `pdgemm` routine on all possible grids from 1 up to 32 processors of *i-cluster*.**

These results are very encouraging. Figure 17 clearly shows the advantages of a performance estimation depending also on the selected host while Figure 18 proves the advantage of the use of a specialized monitoring tool.

**6.3.2 Use of FAST on the Motivating Example.** The objective of FAST is to provide accurate information allowing a client application, e.g. a scheduler, to determine which is the best solution among several scenarios. Let us assume we have two matrices $A$ and $B$ we want to multiply. These matrices have same size but distributed in a block-cyclic way on two disjoint processor grids (respectively $G_a$ and $G_b$). In such a case it is mandatory to align matrices before performing the product. Several choices are then possible: redistribute $B$ on $G_a$, redistribute $A$ on $G_b$ or define a new virtual grid with all available processors. Figure 19 summarizes the framework of this experiment. These grids are actually node sets from a single parallel computer (or cluster). Processors are then homogeneous. Furthermore, inter- and intra-grids communication costs can be considered as similar.

Unfortunately FAST is not able to estimate the cost of a redistribution between two arbitrary processor sets. This problem is indeed very hard in the general case (Desprez et al., 1998). Thus, for this experiment we have determined amounts of data transferred between each pair of processors and the communication scheme generated by the SCALAPACK redistribution routine. Then we use FAST to

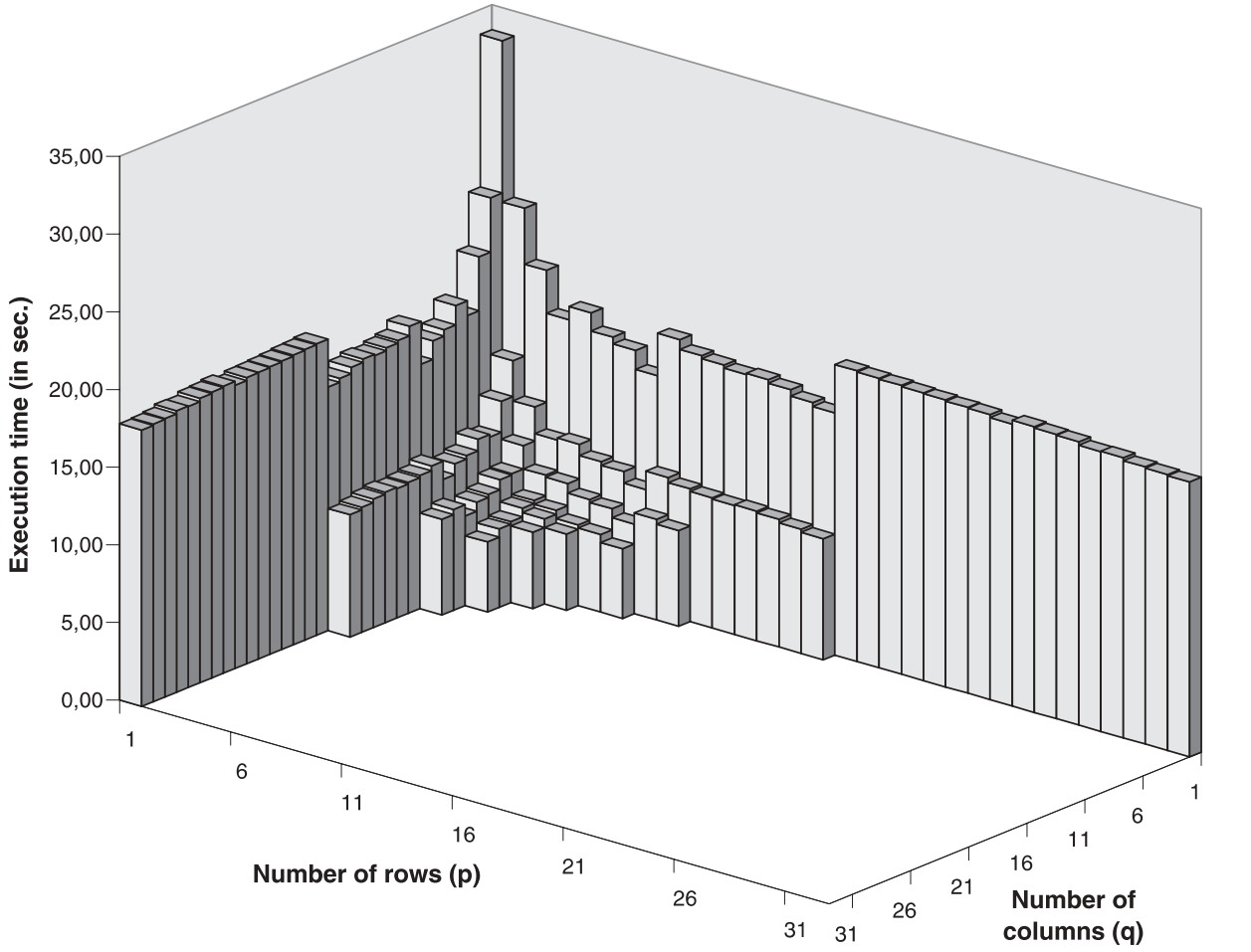


**Fig. 14 Actual execution time for the `pdgemm` routine on all possible grids from 1 up to 32 processors of *i-cluster*.**

forecast the costs of each point to point communication. Figure 20 gives a comparison between forecasted and measured times for each of the grids presented in Figure 19.

We can see that the parallel extension of FAST allows us to accurately forecast what is the best solution, namely a 4×3 processor grid. If this solution is the most interesting with regards to the computation point of view, it is also the least efficient from the redistribution point of view. The use of FAST can then allow us to perform a first selection depending on the processor speed/network bandwidth ratio. Furthermore, it is interesting to see that even if the choice to compute on $G_a$ is slightly more expensive, it induces less communications and releases four processors for other potential pending tasks. Finally a tool like the extended version of FAST can detect when a computation will need more memory than the available amount of a certain configuration and thus induce swap. Typically the 2×2 processor grid will no longer be considered as soon as we reach a problem size exceeding the total capacity of involved processors. For larger problem sizes, the 4×2 grid may also be discarded. This experiment shows that the extension of FAST to handle parallel routines will be very useful to a scheduler as it provides enough information to choose according to several criteria: minimum completion time, communication minimization, number of processors involved, etc.

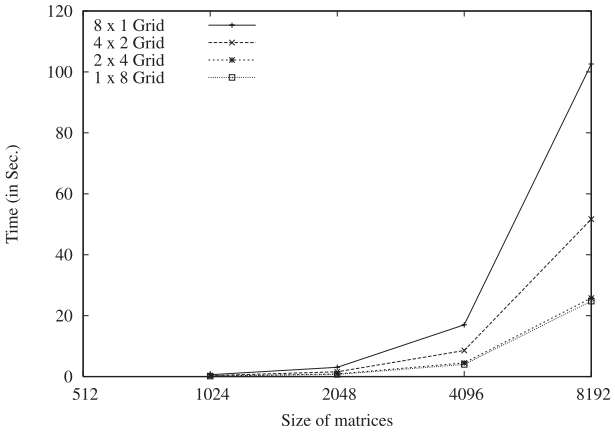


**Fig. 15 Estimations of the execution time of a `pbdtrsm` on different grid shapes.**

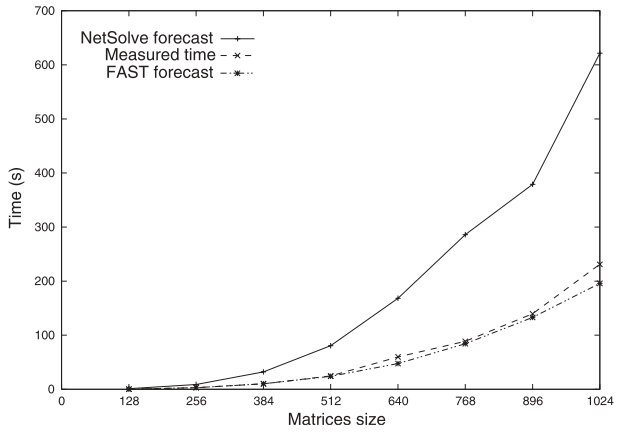


**Fig. 17 FAST and NetSolve forecasts for the computation time compared to measured time (`dgemm`)**

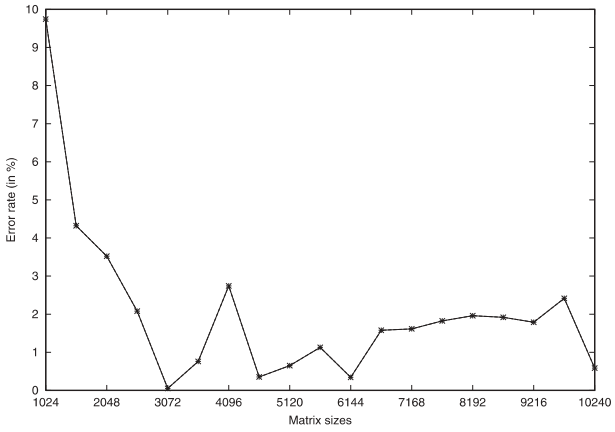


**Fig. 16 Error rate between forecasted and actual execution time for a matrix–matrix multiplication on a 8×4 processor grid.**

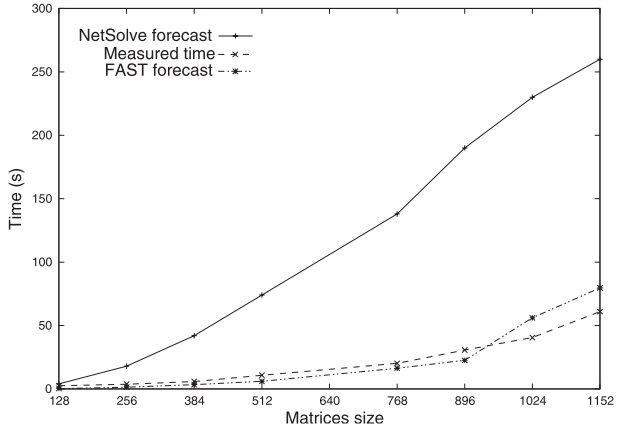


**Fig. 18 FAST and NetSolve forecasts for the communication time compared to measured time (`dgemm`).**

## 7 Conclusion

In this paper, we have presented FAST, a modeling and performance evaluation tool based on NWS's. We have presented the different modules incorporated to the SDK and their features. Experiments show that our predictions are very accurate and can be efficiently applied in a scheduling context.

We are working on many different developments. First we would like to be able to deploy the FAST and NWS's infrastructure automatically at run-time. This implies being able to connect the NWS's sensors in a P2P fashion and to modify the cliques to obtain the best performance. We would also like to use standard SDK (such as PAPI) to acquire information about the processors involved in a computation. Finally, we need to extend our sequential and parallel routines modeling to other routines from the BLAS, LAPACK, ScaLAPACK and other libraries.

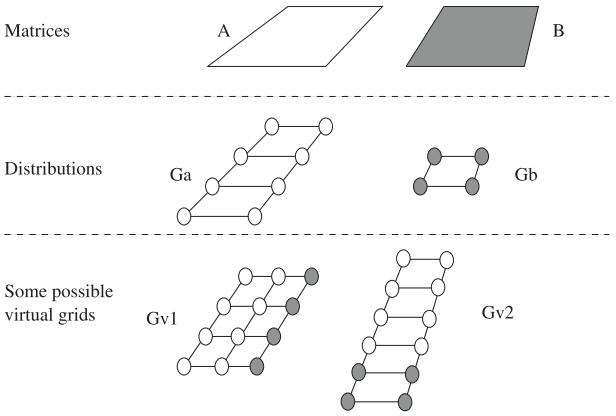


**Fig. 19 Initial distribution and processors grids used in this experiment.**

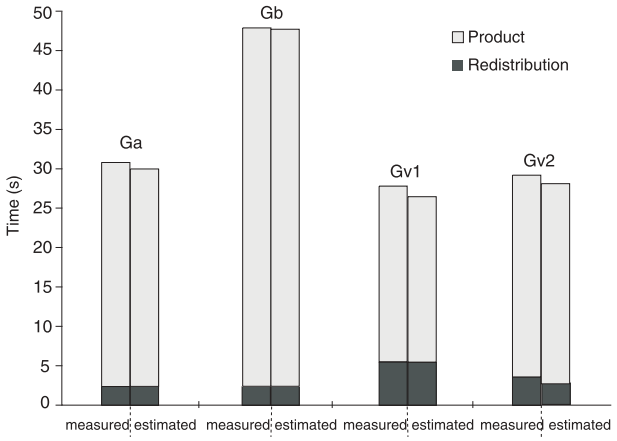


**Fig. 20 Validation of the extension in the case of a matrix alignment followed by a multiplication. Forecast times are compared to measured ones distinguishing redistribution and computation (matrix size 2048×2048).**

## ACKNOWLEDGMENTS

This work has been partially supported by the international scholarship program of INRIA. This work was supported in part by the projects ACI GRID–GRID ASP and RNTL GASP of the French Ministry of Research. We also thank the ID laboratory for granting access to its Cluster Computing Center, this work using the ID/HP cluster (http://icluster.imag.fr/).

## BIOGRAPHIES

*Eddy Caron* holds a position of assistant professor at École Normale supérieure de Lyon and a position at LIP laboratory (ENS Lyon, France). He is a member of GRAAL project and technical manager for the DIET middleware. He received a PhD in C.S. from University de Picardie Jules Verne in 2000. His research interests include parallel libraries for scientific computing on parallel distributed memory machines, problem solving environments, and grid computing. See http://graal.ens-lyon.fr/~ecaron for further information.

*Frédéric Desprez* is a director of research at INRIA and holds a position at LIP laboratory (ENS Lyon, France) where he leads the GRAAL project. He received a PhD in C.S. from the Institut National Polytechnique de Grenoble in 1994 and his MS in C.S. from the ENS Lyon in 1990. His research interests include parallel libraries for scientific computing on parallel distributed memory machines, problem solving environments, and grid computing. See http://graal.ens-lyon.fr/~desprez for further information.

*Martin Quinson* is a post-doc at University of California, Santa Barbara. He received a PhD in C.S. from École normale supérieure de Lyon in 2003 and his MS in C.S. from Université de St-Étienne in 1999. His research interests include grid computing, performance monitoring and forecasting, and large scale distributed application development. See http://graal.ens-lyon.fr/~mquinson for further information.

*Frédéric Suter* received a PhD in C.S. from the École normale supérieure de Lyon, France in 2002 and his MS in C.S. from the Université de Picardie in 1999. He then did a post-doctoral scholarship at the University of California, San Diego in the GRAIL Laboratory. His main research interests include mixed task and data parallelism, parallel routine modeling and grid computing. He is currently Assistant Professor at the Université Joseph Fourier, Grenoble, France. See http://graal.ens-lyon.fr/~fsuter for further information.

## NOTE

1 The other cases are similar.

## REFERENCES


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

Badia, R., Escale, F., Gabriel, E., Gimenez, J., Keller, R., Labarta, J., and Müller, S., 2003. Performance prediction in a grid environment. In *Proceedings of the 1st European Across Grids Conference*, Santiago de Compostela.

Bourgeois, J., Spies, F., and Tréhel, M., 1999. Performance prediction of distributed applications running on network of workstations. In H. R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, June, Las Vegas, NV, CSREA Press, Vol. II, pp. 672–678.

Caron, E. and Suter, F., 2002. Parallel extension of a dynamic performance forecasting tool. In *Proceedings of the International Symposium on Parallel and Distributed Computing*, July, Iasi, Romania, pp. 80–93.

Caron, E., Lazure, D., and Utard, G., 2000. Performance prediction and analysis of parallel out-of-core matrix factorization. In *Proceedings of the 7th International Conference on High Performance Computing (HiPC'00)*, Vol. 1593 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, pp. 161–172.

Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K., Santos, E., Subramonian, R., and von Eicken, T., 1996. LogP: a practical model of parallel computation. *Communications of the ACM*, 39(11):78–95.

Desprez, F., Dongarra, J., Petitet, A., Randriamaro, C., and Robert, Y., 1998. Scheduling block-cyclic array redistribution. In E.H. D'Hollander, G.R. Joubert, F.J. Peters, and U. Trottenberg, editors, *Parallel Computing: Fundamentals, Applications and New Directions*, North-Holland, Amsterdam, pp. 227–234.

Desprez, F., Quinson, M., and Suter, F. 2001. Dynamic performance forecasting for network-enabled servers in a heterogeneous environment. In H.R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, June, Las Vegas, NV, CSREA Press, Vol. III, pp. 1421–1427.

Domas, S., Desprez, F., and Tourancheau, B. 1996. Optimization of the ScaLAPACK LU factorization routine using communication–computation overlap. In *Proceedings of Europar'96 Parallel Processing Conference*, Vol. 1124 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, pp. 3–10.

Dongarra, J. and Roche, K. 2004. Deploying parallel numerical library routines to cluster computing in a self-adapting fashion. *Parallel Computing*, submitted.

Dongarra, J., Du Croz, J., Hammarling, S., and Hanson, R. 1988. An extended set of fortran basic linear algebra subroutines. *ACM Transactions on Mathematical Software*, 14(1):1–17.

Dongarra, J., Mayes, P., and Radicati di Brozolo, G. 1991. The IBM RISC System 6000 and Linear Algebra Operations. *Supercomputer*, 8:15–30.

Dongarra, J., Petitet, A., and Whaley, R.C. 2001. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35.

Howes, T., Smith, M., and Good, G. 1999. *Understanding and Deploying LDAP Directory Services*. Macmillan Technical Publishing, London.

Silicon Graphics Inc. 2001. *Performance Co-Pilot: Monitoring and Managing System-Level Performance*. http://www.sgi.com/software/co-pilot/.

Matthews, W. and Cottrell, L. 2000. The PingER project: active internet performance monitoring for the HENP community. *IEEE Communications Magazine*, 38(5):130–136.

Petitet, A., Blackford, S., Dongarra, J., Ellis, B., Fagg, G., Roche, K., and Vadhiyar, S. 2001. Numerical libraries and the grid: the grADS experiments with scaLAPACK. In *SC'2001 Conference*, November, Denver, CO, ACM SIGARCH/IEEE.

Quinson, M. 2002. Dynamic performance forecasting for network-enabled servers in a metacomputing environment. In *Proceedings of the International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems (PMEO-PDS'02)*, April, Fort Lauderdale, FL.

Rayward-Smith, V. 1987. UET scheduling with unit interprocessor communication delays. *Discrete Applied Mathematics*, 18:55–71.

Vetter, J. and Reed, D. 2000. Real-time performance monitoring, adaptive control, and interactive steering of computational grids. *International Journal of High Performance Computing Applications*, 14(4):357–366.

Wolski, R., Spring, N., and Hayes, J. 1999. The Network Weather Service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computing Systems, Metacomputing Issue*, 15(5–6):757–768.

Wolski, R., Spring, N., and Hayes, J. 2000. Predicting the CPU availability of time-shared Unix systems. *Cluster Computing*, 3(4):293–301.