HAL
open science

Parallel Extension of a Dynamic Performance Forecasting Tool

Eddy Caron, Frédéric Suter

To cite this version:

Eddy Caron, Frédéric Suter. Parallel Extension of a Dynamic Performance Forecasting Tool. [Research Report] RR-4470, INRIA. 2002. inria-00072118

HAL Id: inria-00072118

https://inria.hal.science/inria-00072118v1

Submitted on 23 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Parallel Extension of a Dynamic Performance Forecasting Tool

Eddy Caron, Frédéric Suter

No 4470

June 2002

THÈME 1

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--4470--FR+ENG

Parallel Extension of a Dynamic Performance Forecasting Tool

Eddy Caron, Frédéric Suter

Thème 1 — Réseaux et systèmes
Projet ReMaP

Rapport de recherche n° 4470 — May 2002 — 15 pages

Abstract: This article presents an extension of the FAST library to handle parallel routines. FAST is a dynamic performance forecasting tool in a metacomputing environment. Here we propose to combine estimations given by FAST about sequential computation routines and network availability to parallel routine models coming from analysis. This association will be defined and detailed on two examples and validated experimentally.

Key-words: Performance forecasting and modeling, parallel routines.

(Résumé : tsvp)

This text is also available as a research report of the Laboratoire de l'Informatique du Parallélisme `http://www.ens-lyon.fr/LIP`.

Unit´e de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN (France)
T´el´ephone : 04 76 61 52 00 - International : +33 4 76 61 52 00
T´el´ecopie : 04 76 61 52 52 - International : +33 4 76 61 52 52

Extension Parallèle d'un outil de prédiction dynamique de performances

Résumé : Cet article présente une extension de la bibliothèque FAST pour la gestion de routines parallèles. FAST est un outil de prédiction dynamique de performances dans un environnment de metacomputing. Nous proposons ici de combiner les estimations fournies par FASTconcernant les routines séquentielles de calcul et les disponibilités du réseau avec des modèles issus de l'analyse. Cette association sera définie, détaillée sur deux exemples et validée expérimentalement.

Mots-clé : Prédiction de performances et modélisation, routines parallèles.

1 Introduction

Thanks to metacomputing [1, 2, 9] it is now possible to perform computations on world wide spread machines. This is the promise of a huge computational power because most interactive machines are under-used. Moreover, computations could be performed on the most appropriate computer. But such solutions are difficult to achieve while keeping good efficiency. One of the reasons is that these machines can either be workstations, clusters or even shared memory parallel computers. This heterogeneity in terms of computing power and network connectivity implies that the execution time of a routine may vary and strongly depends on the execution platform. A trend of metacomputing, called *Application Service Providing*, allows multidisciplinary applications to access computational servers via agents. These agents are in charge of the choice of the most appropriate server and of task scheduling and placement. In such a context it becomes very important to have tools able to collect knowledge about servers at runtime to forecast the execution times of the tasks to schedule. Furthermore, in a metacomputing context, we might deal with hierarchical networks of heterogeneous computers with hierarchical memories. Even if software packages exist to acquire information needed about both computer and network using monitoring techniques [14], they often use a flat and homogeneous view of a metacomputing system. To be closer to reality we have to model computation routines with regard to the computer which will execute them.

In the next section we give an overview of FAST (Fast Agent's System Timer) [7, 11], the dynamic performance forecasting tool we have extended. In section 3 we propose a combination between dynamic data acquisition and code analysis which will be the core of the extension of FAST to handle parallel routines. Then we detail the extension on two examples of parallel routines in section 4. Finally in section 5 we give some experimental results validating our approach.

2 Overview of FAST

FAST (Fast Agent's System Timer) [7, 11] is a dynamic performance forecasting tool in a metacomputing environment designed to handle these important issues. Information acquired by FAST concerns sequential computation routines and their execution platforms. For example it can be useful to a scheduler to optimize task mapping.

As shown in Figure 1, FAST is composed of two main modules offering a user API: the *static data acquisition* module and the *dynamic data acquisition* module. The former models the time and space needs of a sequential computation routine on a given machine for a given set of parameters, while the latter forecasts the behavior of dynamically changing resources, *e.g.,* workload, bandwidth, memory use, ... FAST relies on low–level software packages. First, LDAP [10] is used to store static data. Then to acquire dynamic data FAST is essentially based on the NWS (Network Weather Service) [14], a project initiated at University of California San Diego. It is a distributed system that periodically monitors and dynamically forecasts performance of various network and computational resources. NWS

can monitor several resources including communications links, CPU, disk space and memory. Moreover, NWS is not only able to obtain measurements, it can also forecast the evolution of the monitored system.

FAST extends NWS as it allows to determine theoretical needs of computation routines in terms of execution time and memory. The current version of FAST only handles regular sequential routines like those of the dense linear algebra library BLAS [8]. However BLAS kernels represent the heart of many applications, especially in numerical simulation. The approach chosen by FAST to forecast execution times of such routines is an extensive benchmark followed by a polynomial regression. Indeed this kind of routines is often strongly optimized with regard to the platform, either by vendors or by automatic code generation [13]. A code analysis to find an accurate model thus becomes tedious. Furthermore those optimizations are often based on a better use of memory caches. Then an extensive benchmark of a computer will lack of accuracy because the number of floating operations that can be achieved on a computer is not constant as it depends on how a routine uses of cache.

In this paper we focus on the integration of parallel routine handling into FAST. These routines are harder to time and thus to benchmark but easier to analyze. Indeed they often can 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.

3 Extension of FAST to Handle Parallel Routines

To obtain a satisfying 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. First if we ignore communication costs a parallel routine can be modeled using Amdahl's law. This kind of model is not realistic in the case of distributed memory architectures. Indeed on such platforms communications represent an important part of the total execution time of an application. Furthermore this model does not allow to handle the impact of processor grid shape on routine performance. Other models like *delay* [12] and *LogP* [4] 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 these models introduce less or more parameters than what is necessary to obtain estimation sufficiently close to reality.

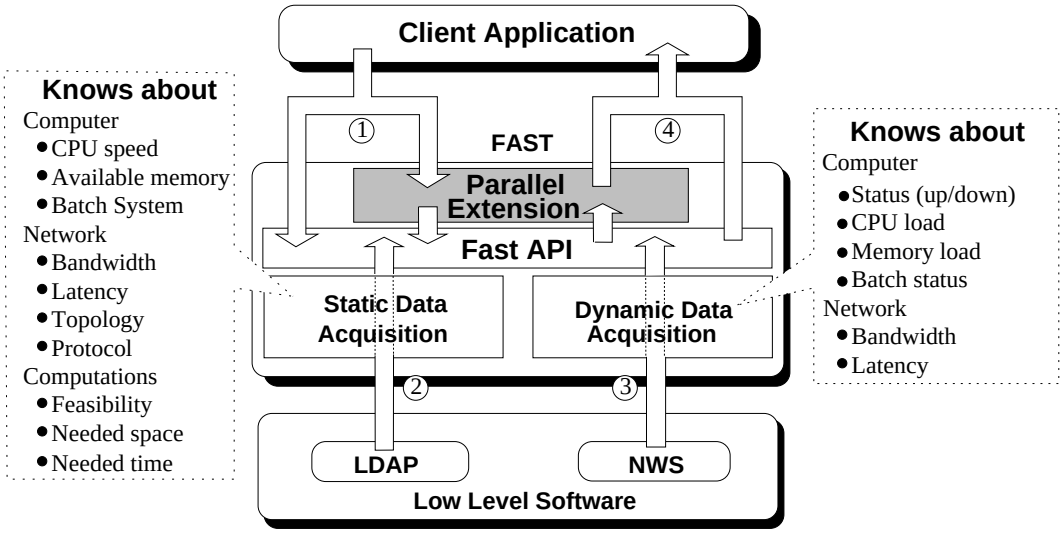

Figure 1: Overview of the FAST architecture with the extension to handle parallel routines.

The first version of the extension only handles some routines of the parallel dense linear algebra library ScaLAPACK. For such routines the description step only consists in determining which sequential counterparts are called, their calling parameters (*i.e.,* data sizes, multiplying factors, transposition, ...), the communication schemes and the amount of data exchanged. Once this analysis is completed the computation part can easily be forecasted. Indeed since FAST is able to estimate each sequential counterpart, FAST calls are enough to determine their execution times.

Furthermore processors executing a ScaLAPACK code have to be homogeneous to achieve good performance. Similarly the network between these processors also has 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 λ is the latency of the network, L the size of the message and τ the time to transfer an element, *i.e.,* the inverse of the network bandwidth. L can be determined during the analysis while λ and τ can be estimated with FAST calls. In a broadcast operation the λ and τ constants are replaced by functions depending on the processor grid topology [3]. If we consider a grid with p rows and q columns λ_p^q will the 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 λ_q^p and τ_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 depend directly on the implementation of the broadcast. For example, on a cluster of workstations connected through a switch, the tree broadcast is actually executed following a tree. In this case λ_p^q will be equal to $\lceil log_2 q \rceil \times \lambda$ and τ_p^q equal to $(\lceil log_2 q \rceil / p) \times \tau$ where λ should be interpreted as the latency for one node and $1/\tau$ as the average bandwidth.

Our work can then be considered either as client application of FAST or as an extension to handle parallel routines. The shaded part in Figure 1 shows where our work takes place in the FAST architecture. Indeed estimations of FAST are injected into a model coming from code analysis. Once this combination performed the execution time of the modeled parallel routine can be estimated by FAST and is thus available through the FAST standard API. In the next section we give a view of the content of the extension by detailing both examples of dense matrix–matrix multiplication and triangular solve.

4 Examples of Parallel Extension Models

4.1 Matrix–Matrix Multiplication Model

The routine `pdgemm` of the ScaLAPACK library performs 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 then an $M \times K$ 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 can be expressed as

$$\left\lceil \frac{K}{R} \right\rceil * dgemm_time, \tag{1}$$

where $dgemm_time$ is given by the following FAST call

$$fast_comp_time_best\,(host,\ dgemm_desc, dgemm_time),$$

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, their transposition, ... Matrices passed in arguments to that FAST call are of size $\left\lceil \frac{M}{p} \right\rceil \times R$ for the first operand and $R \times \left\lceil \frac{N}{q} \right\rceil$ 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 broadcasted 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 of these broadcasts is performed block by block. The communication cost of the `pdgemm` routine is thus

[1] The other cases are similar.

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

Assuming that broadcast operations are performed following a tree, τ_p^q, τ_q^p, λ_p^q and λ_q^p can be replaced by their values depending on τ and λ. These two variables are estimated by the following FAST calls

$$fast_bandwidth\,(source, dest, TAU), \text{ and}$$
$$fast_latency\,(source, dest, LAMBDA),$$

where the link between *source* and *dest* is one of those monitored by FAST. It leads us to the following estimation for `pdgemm` communication cost

$$\frac{\left(\frac{\lceil log_2\, q\rceil \times M \times K}{p} + \frac{\lceil log_2\, p\rceil \times K \times N}{q}\right)}{TAU} + (\lceil log_2\, q\rceil + \lceil log_2\, p\rceil) \times \left\lceil \frac{K}{R} \right\rceil \times LAMBDA. \tag{3}$$

4.2 Triangular Solve Model

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 being two $M \times N$ matrices.

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		

Figure 2: Correspondence between values of calling parameters and actually performed computations for the `pdtrsm` routine.

Figure 2 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 2. These cases can also be grouped into two family which achieve different kinds of performance depending on the shape of the processor grid. The former includes cases 1 and 3 and will execute faster on row–dominant grids, while the latter contains cases 2 and 4 and will achieve better performance on column–dominant grids.

To analyze why some shapes are clearly better than some other, we focus on the case $XA = B$ where A is a non-unitary upper triangular matrix.

This routine is actually composed of calls to the subroutine `pbdtrsm` which computes the same operation as `pdtrsm` but the number of rows of B is less or equal to the size of a distribution block. So in a call to `pdtrsm`, we have $\lceil M/R \rceil$ calls to `pbdtrsm`, where R is the distribution block size.

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

Figure 3: Computations performed in a `pbdtrsm` call, where A is an $n \times n$ block matrix.

To solve the system of equations presented in Figure 3 (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 processors on the same row. 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. Concerning the update phase, two strategies are applied depending on the shape of the processor grid. Indeed, the case where the grid is a row, which is the best case, does not employ pipelining while the others do. This pipeline consists in splitting the update phase in two steps. The first updates the $(P-1)$ first blocks (where P is the number of 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.

We have to distinguish these two strategies in our models, but some calls to FAST will be the same in both versions. For instance *trsm_time*, the time to compute a sequential triangular solve, will be estimated by

$$fast_comp_time_best\,(host,\ trsm_desc, trsm_time),$$

where *host* is one of the processors involved in the computation of `pdtrsm` and $trsm_desc$ is a structure containing information on matrices and calling parameters. Matrices passed in arguments to that FAST call are of size $R \times R$. Broadcasts of solved blocks are performed considering a $Q-$processor row as a ring. But the critical path of `pdtrsm` follows this ring. So we have only to consider the communication between the processor that computes the sequential solve and its right neighbor. It has to be noticed that idle times may appear in this pipeline as explained in [6] about the LU factorization. These idle times will be taken into account in the next version of our extension. As the amount of data communicated per broadcast is R^2, a broadcast operation can be estimated by

$$T_{broadcast} = R^2 TAU + LAMBDA, \tag{4}$$

where $LAMBDA$ and TAU are estimated by the FAST calls presented in the matrix multiplication model.

The difference between the row case and the other takes place in the number of columns of the second operand of `dgemm` computation. The first operand is always a $R \times R$ matrix. This value depends on how many blocks have already been solved and can be expressed as $R\left\lceil \frac{N-iR}{QR} \right\rceil$ where i the number of solved blocks. In the general case this value can not exceed $(P-1)R$ as only the first $(P-1)$ rows are updated. This leads us to the following FAST calls

$$fast_comp_time_best\,(host,\, dgemm_desc_row, dgemm_time_row),$$

and

$$fast_comp_time_best\,(host,\, dgemm_desc, dgemm_time).$$

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

$$T_{send} = \left(R \times min\left((P-1)R, R\left\lceil \frac{N-iR}{QR} \right\rceil \right) \right) TAU + LAMBDA, \tag{5}$$

and

$$fast_comp_time_best\,(host,\, add_desc, add_time).$$

Equations 6 and 7 gives the computation cost models for the `pbdtrsm` routine respectively on a row and in the general case.

$$\sum_{i=1}^{\lceil N/R \rceil} \left(trsm_time + T_{broadcast} + dgemm_time_row \right). \tag{6}$$

$$\sum_{i=1}^{\lceil N/R \rceil} \left(trsm_time + T_{broadcast} + dgemm_time + T_{send} + add_time \right). \tag{7}$$

To obtain the computation cost of the `pdtrsm` routine of ScaLAPACK, these costs have to be multiplied by $\lceil \frac{M}{R} \rceil$ as each call to `pbdtrsm` solves a block of rows of X.

5 Experimental Validation

To validate our handling of parallel routines into FAST, we ran several tests based on the ScaLAPACK matrix–matrix multiplication routine. These tests were executed on *icluster* which is a cluster of 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 tree broadcast is actually executed following a tree.

5.1 Study of Forecast Accuracy

Figure 4 presents a comparison between the estimated time given by our model (top) and the actual execution time (bottom) for the `pdgemm` routine on all possible grids from 1 up to 32 processors of *icluster*. 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 only 14.69% while the average error is 3.79%. Furthermore these figures also 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.

In the second experiment we tried to validate the accuracy of the extension for a given processor grid. Figure 5 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. Again our extension provides very accurate forecasts the average error rate is only $2,77\%$.

5.2 Utility in a Scheduling Context

The objective of our extension of FAST is to provide accurate information allowing a client application, *e.g.,* a scheduler, to determine which is the best solution among several scenari. Assume that we have two matrices A and B we aim to multiply. These matrices are of same size but distributed in a block cyclic way on two disjointed 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 virtual grid with all available processors. Figure 6(top) summarizes the framework of this experiment. These grids are actually sets of nodes from a single parallel computer (or cluster). Processors are then homogeneous. Furthermore inter– and intra–grids communication costs can be considered as similar.

Unfortunately the current version of FAST is not able to estimate the cost of a redistribution between two processor sets. This problem is indeed very hard in the general case [5]. So for this experiment we have determined amounts of data transfered between each pair of processors and the communication scheme generated by the ScaLAPACK redistribution

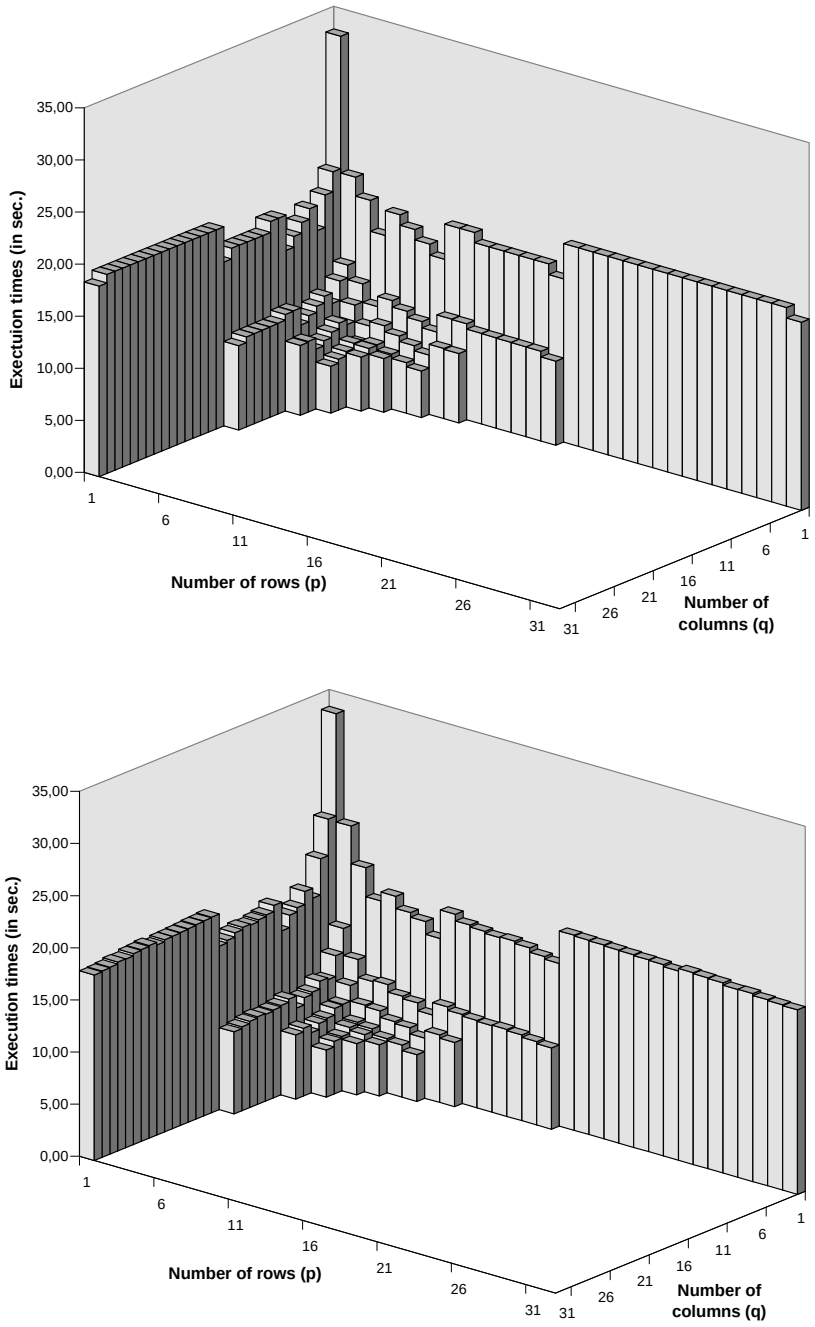

Figure 4: Comparison between estimated time (top) and actual execution time (bottom) for the `pdgemm` routine on all possible grids from 1 up to 32 processors of *icluster*.

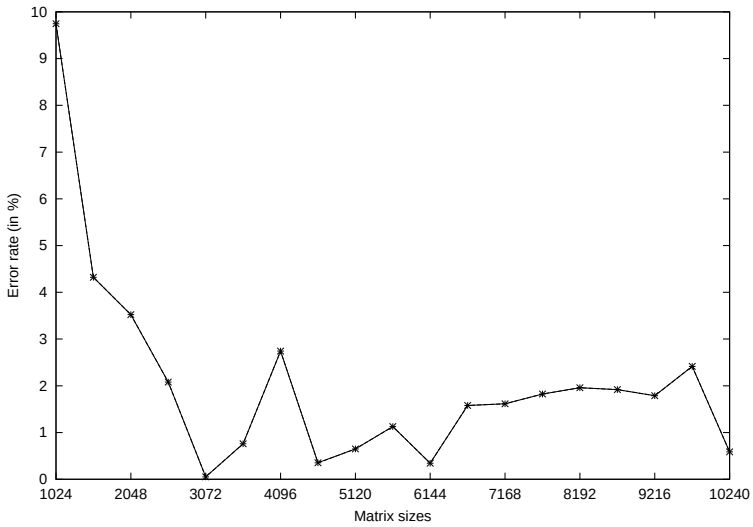

Figure 5: Error rate between forecasted and actual execution time for a matrix–matrix multiplication on a 8×4 processor grid.

routine. Then we use FAST to forecast the costs of each point to point communication. Figure 6(bottom) gives a comparison between forecasted and measured times for each of the grids presented in Figure 6(top).

What we can see that the parallel extension of FAST allows to accurately forecast what is the best solution, namely a 4×3 processor grid. If this solution is the most interesting from the computation point of view, it is also the less efficient from the redistribution point of view. The use of FAST can then allow 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 a little more expensive it induces less communications and discharges 4 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 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 be able to choose according to several criteria: Minimum completion time, communication minimization, number of processors involved, ...

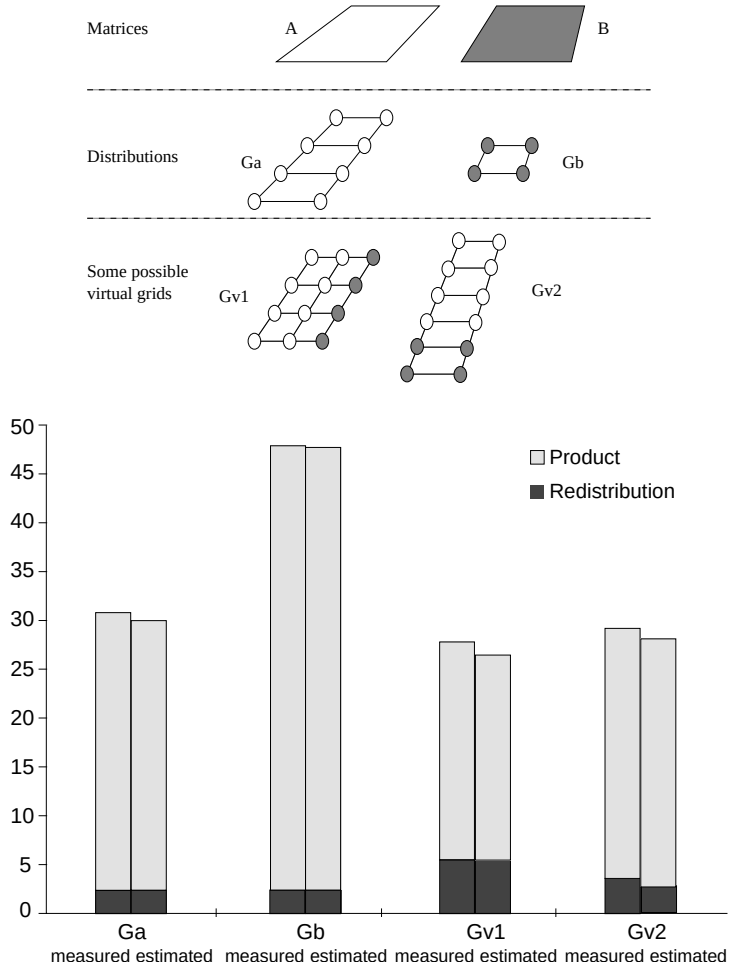

Figure 6: Validation of the extension in the case of a matrix alignment followed by a multiplication for 4 virtual processor grids (top). Forecasted times are compared to measured ones (bottom) distinguishing redistribution and computation.

6 Conclusion and Future Work

In this paper we proposed an extension to the dynamic performance forecasting tool FAST to handle parallel routines. This extension uses the information provided by the actual version of FAST about sequential routines and network availability. This information is injected into a model coming from code analysis. The result can be considered as a new function it is possible to estimate by call to the FAST API. For instance it will be possible to ask FAST to forecast the execution time of parallel matrix–matrix multiplication or parallel triangular solve.

Some experiments validated the accuracy of the parallel extension either for different grid shapes with a fixed matrix size or for different sizes of matrices on a fixed processor grid. In both cases, the average error rate between estimated and measured execution times is under 4%.

We also showed how a scheduler could benefit of our extension to FAST. Indeed it allows a scheduler to make mapping choices based on a realistic view of the execution platform and accurate estimations for execution times of tasks and data movement costs.

Our first work will be to properly handle pipelining and idle times in the `pdtrsm` routine. Then we plan to extend the work presented in this paper to the entire ScaLAPACK library in order to provide performance forecasting tool for a complete dense linear algebra kernel. Another point to develop is the cost estimation of redistribution. If the general case is a difficult problem, we think we can base our estimations upon a set of redistribution classes. These classes are built depending on modifications made to the source grid to obtain the destination grid. For instance the redistribution from G_a and G_b to G_{v2} made in Section 5 might be elements of the same class, where redistributions only use a proportional increase of the number of lines of the processor grid.

Acknowledgements

This work was supported in part by the projects ACI GRID–GRID ASP and RNTL GASP funded by the French ministry of research.

We would like to thank the ID laboratory for granting access to its Cluster Computing Center, this work having been done on the ID/HP cluster (`http://icluster.imag.fr/`).

References

[1] Rajkumar Buyya, editor. *High Performance Cluster Computing*, volume 1: Architectures and Systems. Prentice Hall, 1999. ISBN 0-13-013784-7.

[2] Rajkumar Buyya, editor. *High Performance Cluster Computing*, volume 2: Programming and Applications. Prentice Hall, 1999. ISBN 0-13-013784-7.

[3] Eddy Caron, Dominique Lazure, and Gil Utard. Performance Prediction and Analysis of Parallel Out-of-Core Matrix Factorization. In *Proceedings of the 7th International Conference on High Performance Computing (HiPC'00)*, pages 161–172, Dec 2000.

[4] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Eric Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: A Practical Model of Parallel Computation. *Communications of the ACM*, 39(11):78–95, Nov 1996.

[5] Frédéric Desprez, Jack Dongarra, Antoine Petitet, Cyril Randriamaro, and Yves Robert. 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*, pages 227–234. North Holland, 1998.

[6] Frédéric Desprez, Jack Dongarra, and Bernard Tourancheau. Performance Study of LU Factorization with Low Communication Overhead on Multiprocessors. *Parallel Processing Letters*, 5(2):157–169, 1995.

[7] Frédéric Desprez, Martin Quinson, and Frédéric Suter. Dynamic Performance Forecasting for Network Enabled Servers in a Heterogeneous Environment. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, June 2001.

[8] Jack Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard Hanson. An Extended Set of Fortran Basic Linear Algebra Subroutines. *ACM Trans. on Mathematical Soft.*, 14(1):1–17, 1988.

[9] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan-Kaufmann, 1998.

[10] Tim Howes, Marc Smith, and Gordon Good. *Understanding and deploying LDAP directory services*. Macmillian Technical Publishing, 1999. ISBN: 1-57870-070-1.

[11] Martin Quinson. 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)*, Apr 2002.

[12] Victor Rayward-Smith. UET Scheduling with Unit Interprocessor Communication Delays. *Discrete Applied Mathematics*, 18:55–71, 1987.

[13] R. Clinton Whaley and Jack Dongarra. Automatically Tuned Linear Algebra Software (ATLAS). In *Proceedings of IEEE SC'98*, 1998.

[14] Rich Wolski, Neil Spring, and Jim 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.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399