Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 6, Number 1, pp. 57–69. http://www.scpe.org

PARALLEL EXTENSION OF A DYNAMIC PERFORMANCE FORECASTING TOOL

EDDY CARON[†‡], FRÉDÉRIC DESPREZ[†] AND FRÉDÉRIC SUTER[†]

Abstract. This paper presents an extension of a performance evaluation library called Fast to handle parallel routines. Fast is a dynamic performance forecasting tool in a grid environment. We propose to combine estimations given by Fast about sequential computation routines and network availability to parallel routine models coming from code analysis.

Key words. Performance forecasting and modeling, parallel routines.

1. Introduction. Thanks to metacomputing [10] it is now possible to perform computations on the aggregation of machines all over the world. This is the promise of a huge computational power because many 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 using a system based on agents. These agents are in charge of the choice of the most appropriate server and task scheduling and mapping. 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.

The computational servers we target can be sequential or parallel. Moreover, they are running libraries such as the BLAS, LAPACK and/or their parallel counterparts (PBLAS, ScaLAPACK). The former libraries are using virtual grids of processors and block-sizes to distribute the matrices involved in the computations. The performance evalutation has thus to take into account the shape of the grid as well as the distributions of the input and output data. In this paper, we propose a careful evaluation of several linear algebra routines combined with a run-time evaluation of the parameters of the target machines. This allows us to optimize the mapping of data-parallel tasks in a client-server environment.

In the next section we give an overview of Fast (Fast Agent's System Timer) [5, 12], the dynamic performance forecasting tool we have extended. After a description of some related work, in section 4 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. We detail the extension on two examples of parallel routines. For each model we provide some experimental results validating our approach. Finally in section 5 we show how can our tool be used in a scheduling context.

2. Overview of Fast. Fast is a dynamic performance forecasting tool in a metacomputing environment designed to handle the important issues presented in the previous section. Informations acquired by Fast concern sequential computation routines and their execution platforms (memory, load) as well as the communication costs between the different elements of the software architecture. Fast should be useful to a scheduler to optimize task mapping on a heterogeneous and distributed environment.

As shown in Figure 2.1, Fast is composed of several modules. The first one, launched when Fast is installed on a computational server, is a tool that executes extensive benchmarks of sequential routines on that server and then extracts models from the results by polynomial regression. The models obtained are stored in a LDAP [11] tree. The second part is the library itself called by the client application. This library is based on two modules and a user API: the *routine needs modeling* module and the *system availabilities* module. The former extracts from the LDAP tree the time and space needs of a sequential computational 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, ... To acquire such dynamic data, Fast is essentially based on NWS (Network Weather

[†]ReMaP – LIP, École Normale Supérieure de Lyon. 46, Allée d'Italie. F-69364 Lyon Cedex 07.
[‡](Eddy.Caron@ens-lyon.fr) Questions, comments, or corrections to this document may be directed to that email address.

Service) [14] from the Universiy of Santa Barbara. NWS is a distributed system that periodically monitors and dynamically forecasts performance of various network and computational resources using sensors. 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.

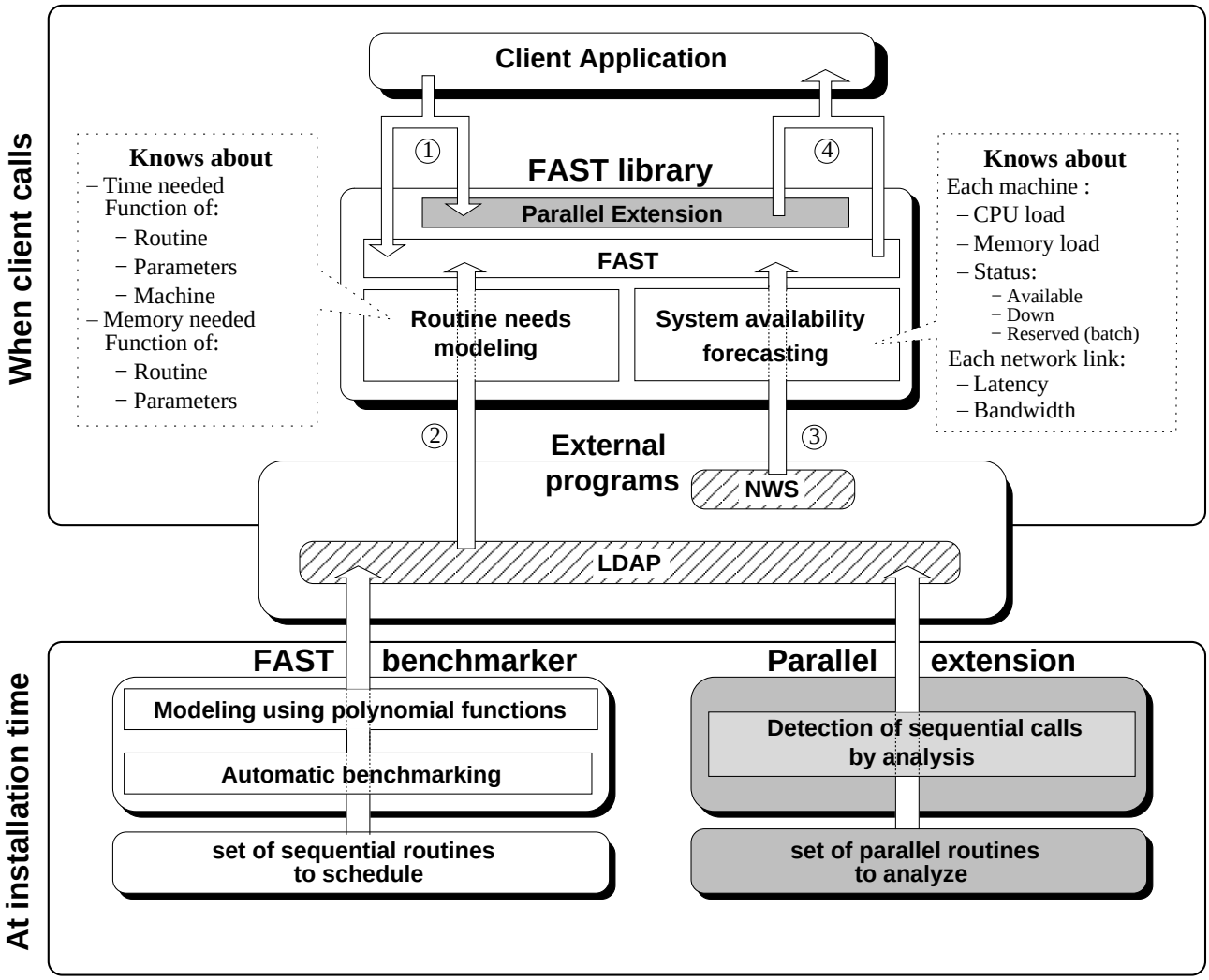

FIG. 2.1. *Overview of the* FAST *architecture with the extension to handle parallel routines.*

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 [7]. However, BLAS kernels represent the heart of many scientific 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 [8]. A code analysis to find an accurate model thus becomes tedious. Furthermore, those optimizations are often based on a better use of hierarchical memories. Then a generic benchmark of a computer will lack of accuracy because the performance that can be achieved on a computer is not constant as it depends on how a routine uses cache memories.

The current version of FAST does not take into account the availability of parallel versions of the routines. In this paper we focus on the integration of parallel routines handling into FAST. These routines are difficult 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 point–to–point or global communication patterns. Timing becomes even more tedious

if we add the handling of data redistribution between servers and the choice of the "optimal" 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 shaded parts in Figure 2.1 show where our work takes place in the FAST architecture. Our parallel extension of FAST is decomposed in two parts. The former is based on code analysis made at the installation of FAST on a machine while the latter is an extension of the API. Indeed FAST's estimations 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. Both parts will be detailed in Section 4.

3. Related Work. 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 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 and can not be neglected. Furthermore, this model does not allow to handle the impact 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 to use. For instance the matrix–matrix multiplication routine of the ScaLAPACK library, compact grids achieve better performance than elongated ones. We can see on Figure 3.1 that a too simple use of Amdahl's law, consisting in adding a processor to the execution platform while the obtained computation time is lower than the one previously estimated has a limited validity and may not detect the optimal execution platform.

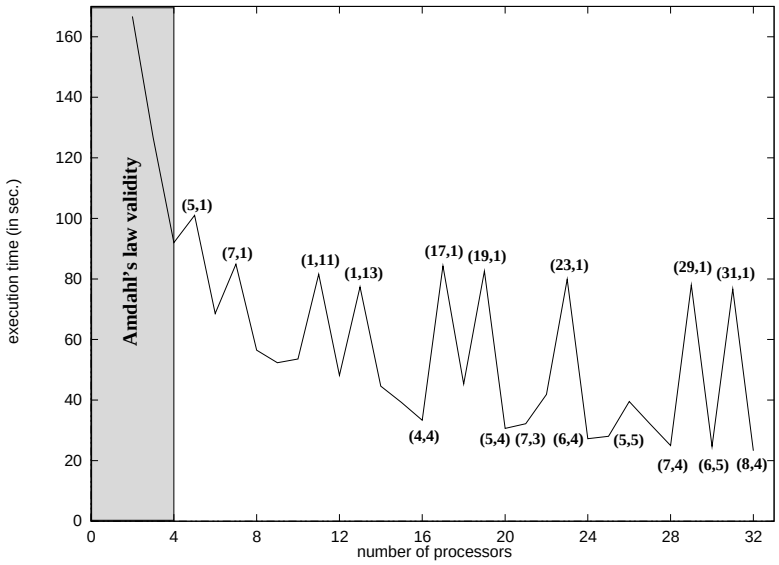

FIG. 3.1. *Execution times of a* 4096×4096 *matrix–matrix multiplication achieved on the best grid using a given number of processors.*

The *delay* [13] and *LogP* [3] 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 metacomputing platform in a simple but realistic way.

About the modeling of parallel algorithms such as those used in libraries like ScaLAPACK, several approaches are possible. In [9], authors aim at using parallel routines on clusters when it is possible to achieve better performance than with 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 neither of the optimizations based on pipelining made in the routines. In [6] 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. The *ChronosMix* environment [1] uses micro-benchmarking to estimate execution times of parallel applications. This technique consists in 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.

4. Extension of Fast to Handle Parallel Routines.

4.1. Modeling Choices. The first version of the extension only handles some routines of the parallel dense linear algebra library ScaLAPACK. For such routines the description step consists only 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 completed, the computation part can easily be forecasted. Indeed since FAST is able to estimate each sequential counterpart, FAST calls are sufficient enough to determine their execution times.

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 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 [2]. If we consider a grid with p rows and q columns λ_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 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 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 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 is the latency for one node and $1/\tau$ as the average bandwidth.

In the next section, we give a view of the content of the extension by detailing both examples of parallel dense matrix-matrix multiplication and triangular solve routines.

4.2. Matrix–Matrix Multiplication Model. The routine `pdgemm` of ScaLAPACK 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 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

$$\left\lceil \frac{K}{R} \right\rceil * \texttt{dgemm\_time}, \tag{4.1}$$

where `dgemm_time` 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

[1] The other cases are similar.

containing informations such as the size of the matrices involved, if they are transposed or not, ... Matrices given in arguments to this call to FAST 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 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 broadcast is performed block by block. The communication cost of the `pdgemm` routine is thus

$$(M \times K)\tau_p^q + (K \times N)\tau_q^p + \left(\lambda_p^q + \lambda_q^p\right) \left\lceil \frac{K}{R} \right\rceil . \tag{4.2}$$

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

$$\left\lceil \frac{K}{R} \right\rceil \times \texttt{dgemm\_time} + (M \times K)\tau_p^q + (K \times N)\tau_q^p + \left(\lambda_p^q + \lambda_q^p\right) \left\lceil \frac{K}{R} \right\rceil . \tag{4.3}$$

If we assume 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_avail (bandwidth, source, dest, &tau)
```

and

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

where the link between `source` and `dest` is one of those monitored by FAST. Equation 4.2 thus becomes

$$\frac{\left(\frac{\lceil \log_2\ q \rceil \times M \times K}{p} + \frac{\lceil \log_2\ p \rceil \times K \times N}{q}\right)}{\texttt{tau}} + \left\lceil \frac{K}{R} \right\rceil (\lceil \log_2\ q \rceil + \lceil \log_2\ p \rceil) \times \texttt{lambda}. \tag{4.4}$$

4.3. Accuracy of the Matrix–Matrix Multiplication Model. To validate our modeling technique we ran several tests on *i-cluster* 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 broadcast is actually executed following a tree.

In a first experiment we tried to validate the accuracy of the extension for a given processor grid. Figure 4.1 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%.

Figure 4.2 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 *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 a grid shape for a parallel routine call. Combined with a run-time evaluation of parameters like communication, machine load or memory availability, we can then build an efficient scheduler for ASP environments.

4.4. 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 are two $M \times N$ matrices.

Figure 4.3 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

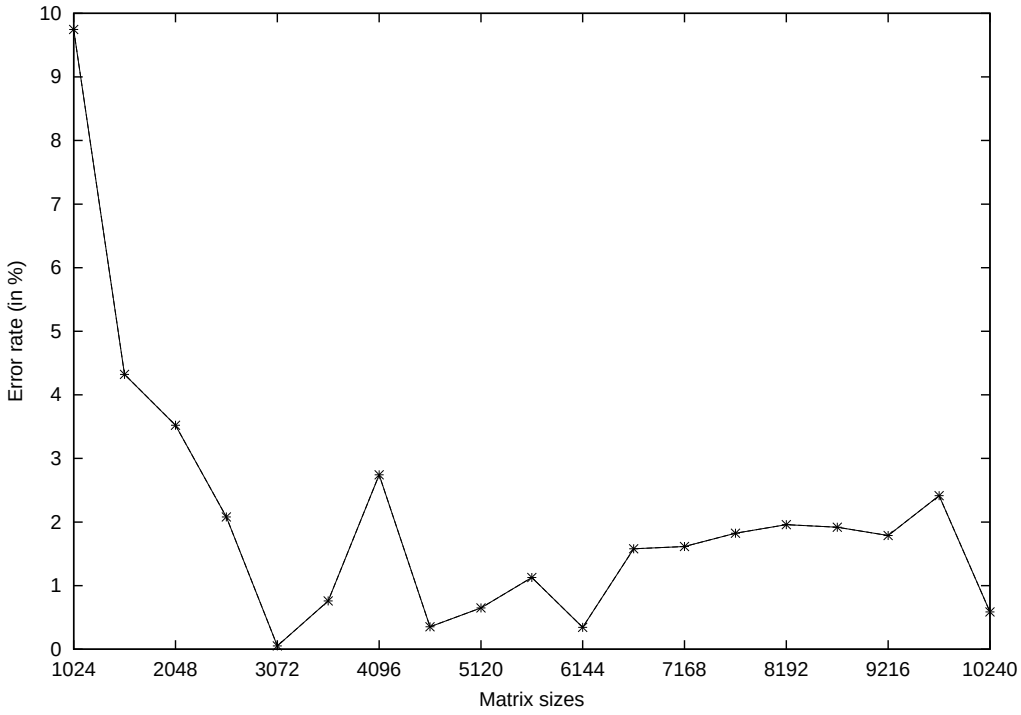

FIG. 4.1. *Error rate between forecasted and actual execution time for a matrix–matrix multiplication on a* 8×4 *processor grid.*

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 4.3. These cases can also be grouped into two family which achieve different 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. 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.

To analyze why some shapes are clearly better than some other, we focus on the case $X * A = B$ where A is a non-unitary upper triangular matrix. Figure 4.4 shows the results we obtained on 8 processors for several grid shapes. The library used is ScaLAPACK v1.6 which is based on PBLAS v1.0. In this particular case (number 2 in Figure 4.3), we can see that a 8 processors row grid (*i.e.,* 1×8) achieves performance 3.7 times better than a 8 processors 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 too the more general case of a rectangular grid.

4.4.1. On a Row. To solve the system of equations presented in Figure 4.5 (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 broadcasted 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 4.6.

We denote the time to compute a sequential triangular solve as `trsm_time`. 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 `trsm_desc` is a structure containing information on matrices and calling parameters. Matrices passed in arguments to that FAST call are of size $R \times R$.

Solved blocks are broadcasted along the ring but the critical path of `pdtrsm` follows also this ring. So we have only to consider the communication between the processor that computes the sequential solve and its right neighbor. As the amount of data broadcasted is R^2, this operation is then estimated by

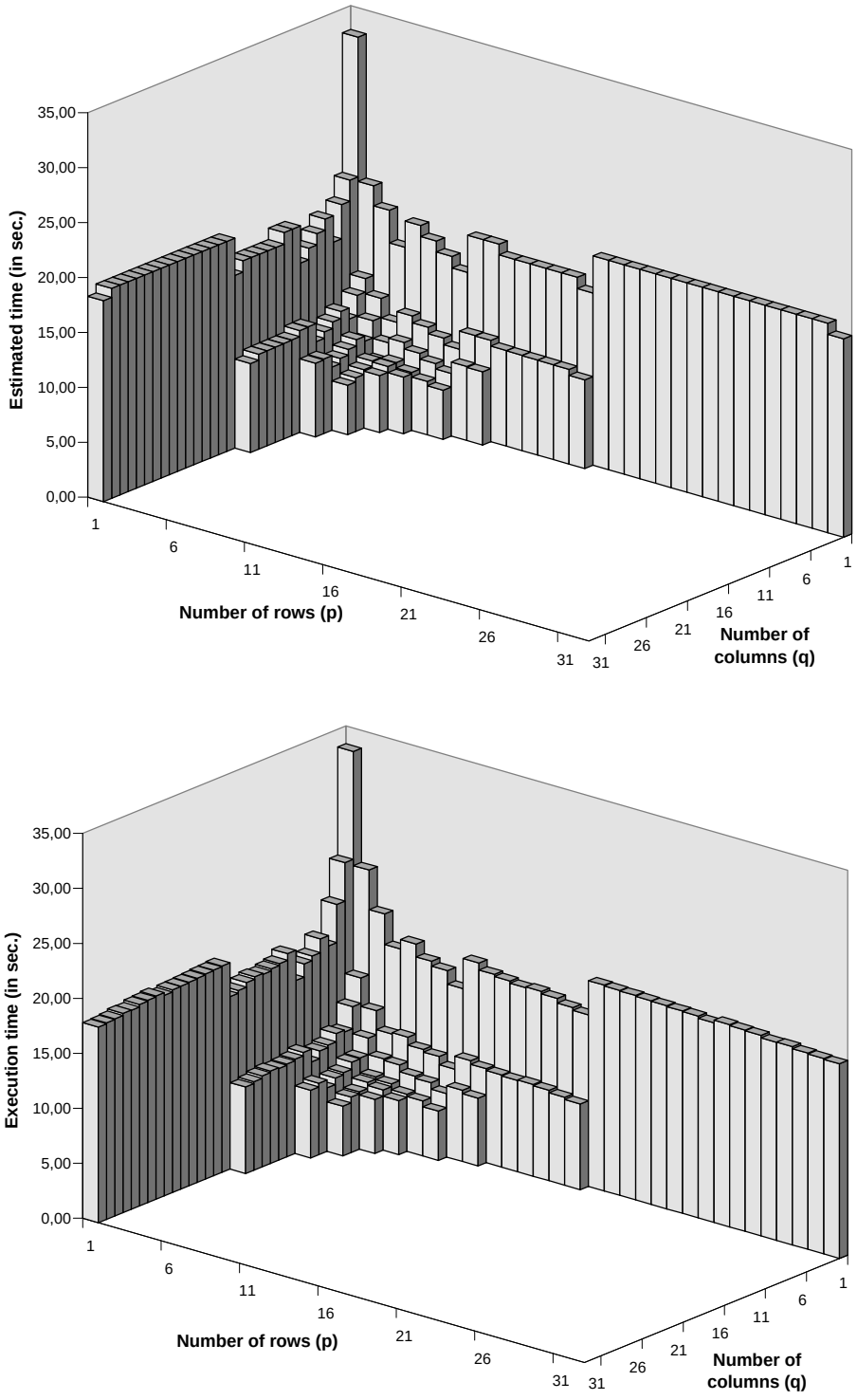

FIG. 4.2. *Comparison between estimated time (top) and actual execution time (bottom) for the* `pdgemm` *routine on all possible grids from 1 up to 32 processors of* i-cluster.

$$T_{broadcast} = R^2\texttt{tau} + \texttt{lambda}, \tag{4.5}$$

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

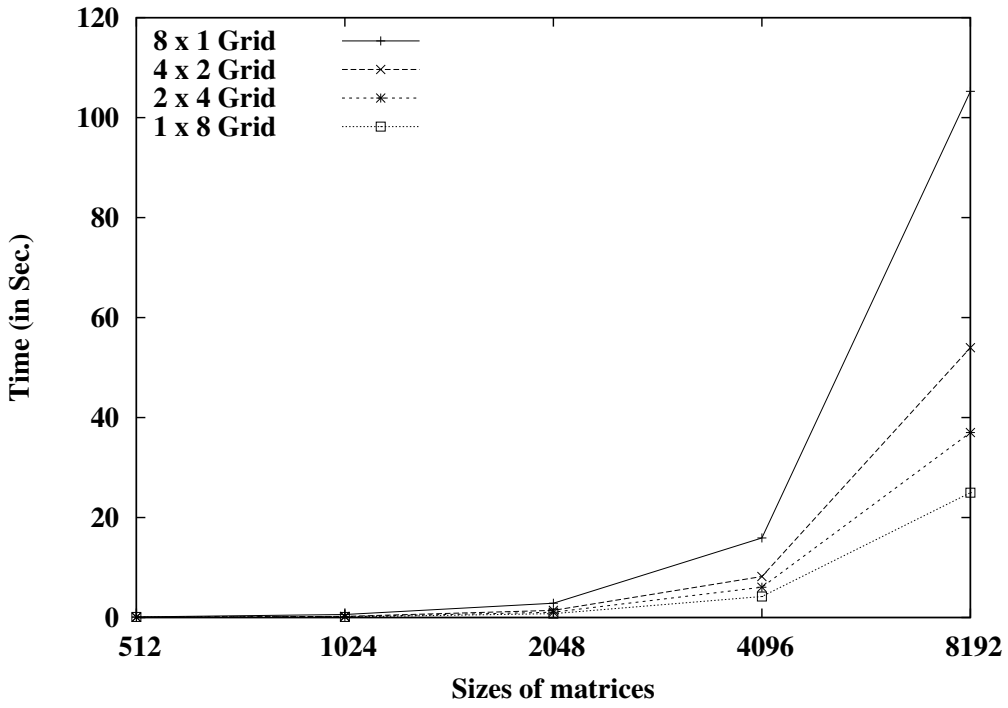

FIG. 4.4. *Performance of a* `pdtrsm` *for different grid shapes.*

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 a $R \times R$ 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

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

Idle times may appear if one of the receivers of a broadcast is still uptading blocks corresponding to the previous step. As our model forecasts the execution time following the critical path of the routine, to handle these idle times we apply a correction C_b as both sending and receiving processors have to wait until the

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

FIG. 4.5. *Computations performed in a* `pbdtrsm` *call, where A is an $n \times n$ block matrix.*

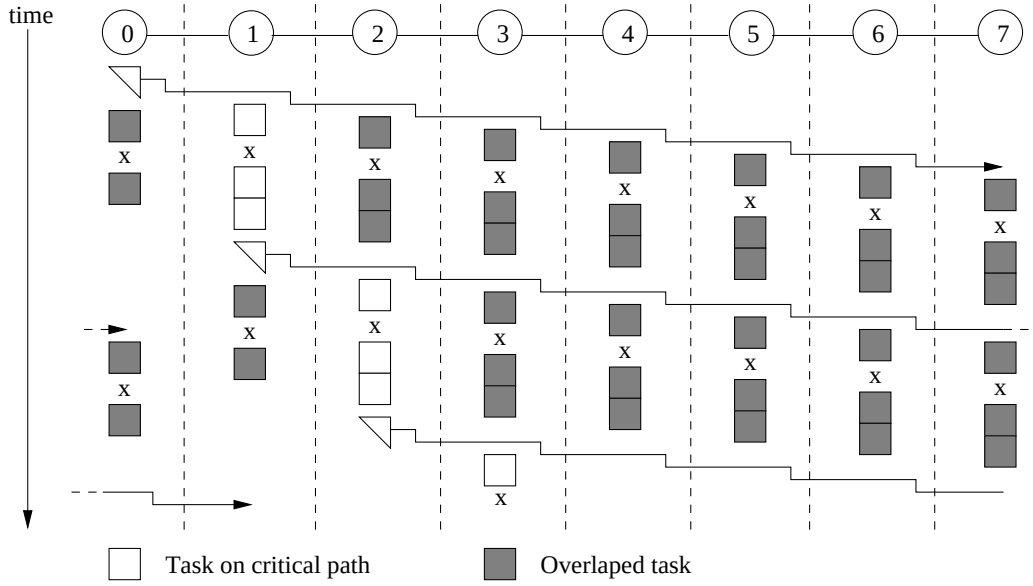

FIG. 4.6. *Execution on a ring of 8 processors of the* `pbdtrsm` *routine.*

maximum of their previously estimated times before performing this communication. Thus equation 4.6 gives the computation cost model for the `pbdtrsm` routine on a row of processors.

$$
\sum_{i=1}^{\lceil N/R \rceil} (\texttt{trsm\_time} + T_{broadcast} + C_b + \texttt{dgemm\_time\_row}) \,. \tag{4.6}
$$

4.4.2. 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 in splitting the update phase in two steps. The first 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 4.7.

This optimization implies 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 a $R \times R$ matrix. We then have two FAST calls to estimate these two different matrix products of the update phase.

```
fast_comp_time (host, dgemm_desc_1, &dgemm_time_1),
```

and

```
fast_comp_time (host, dgemm_desc_2, &dgemm_time_2).
```

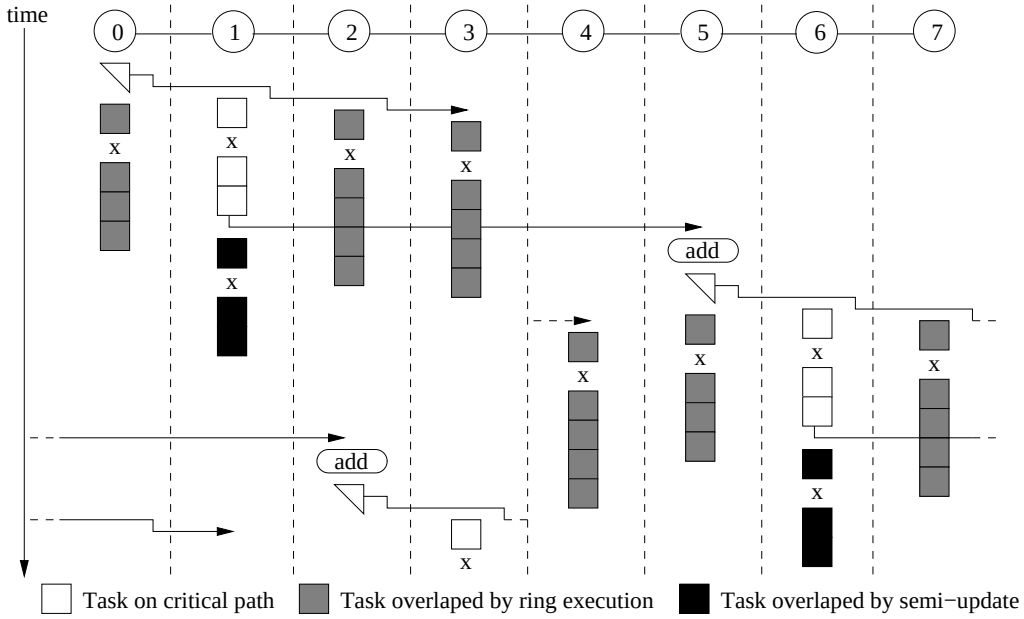

FIG. 4.7. *Execution on a* 2×4 *processor grid of the* `pbdtrsm` *routine.*

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 first `dgemm` of the update phase. This leads us to the following expressions

$$T_{send} = \left(R \times min\left((p-1)R, R \left\lceil \frac{N - iR}{qR} \right\rceil \right) \right) \texttt{tau} + \texttt{lambda}, \tag{4.7}$$

and

```
fast_comp_time (host, add_desc, &add_time).
```

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 4.8 gives the computation cost models for the `pbdtrsm` routine on a rectangular grid of processors.

$$\sum_{i=1}^{\lceil N/R \rceil} \left(\texttt{trsm\_time} + T_{broadcast} + C_b + \texttt{dgemm\_time\_1} + T_{send} + C_u + \texttt{add\_time} \right). \tag{4.8}$$

4.5. Accuracy of the Triangular Solve Model. We ran the same kind of experiment as for matrix multiplication to test the accuracy of our triangular solve model. Figure 4.8 shows estimations produced by our model. Comparing these results with those of Figure 4.4, 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, the error rate is then less than 5%.

5. 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. Let us assume we have two matrices A and B we aim to multiply. These matrices have same

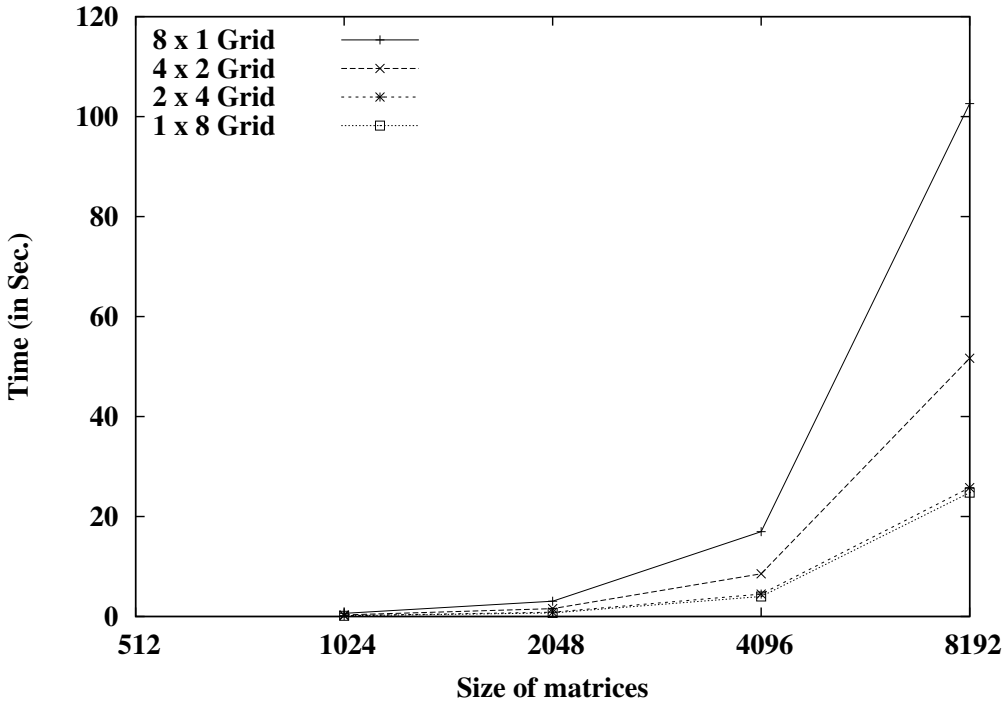

FIG. 4.8. *Estimations of the execution time of a* `pdtrsm` *on different grid shapes.*

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 new virtual grid with all available processors. Figure 5.1 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.

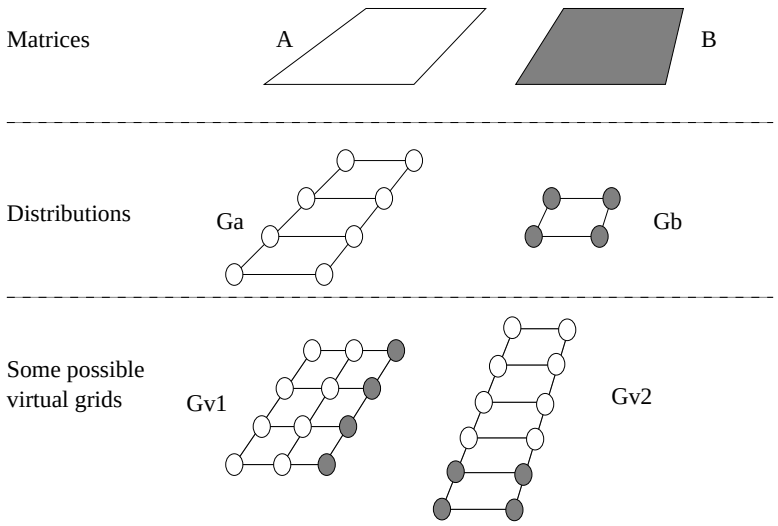

FIG. 5.1. *Initial distribution and processors grids used in this experiment.*

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 [4]. 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 routine. Then we use FAST to forecast the costs of each point to point communication. Figure 5.2 gives a comparison between forecasted and measured times for each of the grids presented in Figure 5.1.

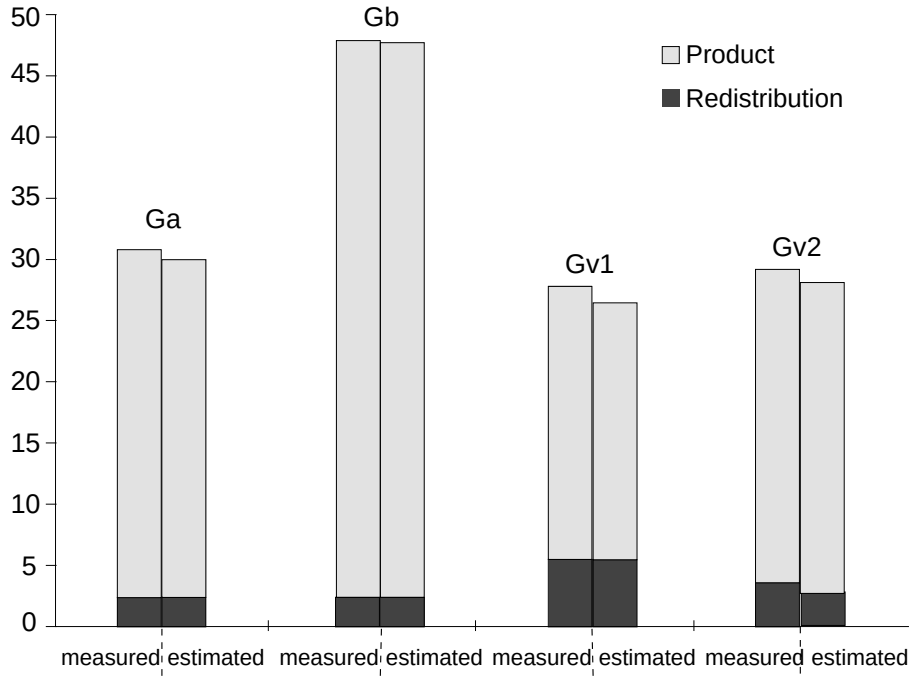

FIG. 5.2. *Validation of the extension in the case of a matrix alignment followed by a multiplication. Forecasted times are compared to measured ones distinguishing redistribution and computation (matrix size* 2000×2000*).*

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 with regards to 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 releases 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 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 informations to be able to choose according to several criteria: Minimum Completion Time, communication minimization, number of processors involved, ...

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 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 rows 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 Ministery 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] J. Bourgeois, F. Spies, and M. Trhel, *Performance Prediction of Distributed Applications Running on Network of Workstations*, in Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99), H. R. Arabnia, ed., vol. II, Las Vegas, June 1999, CSREA Press, pp. 672–678.

[2] E. Caron, D. Lazure, and G. 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), vol. 1593 of Lecture Notes in Computer Science, Springer-Verlag, Dec. 2000, pp. 161–172.

[3] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, *LogP: A Practical Model of Parallel Computation*, Communications of the ACM, 39 (1996), pp. 78–95.

[4] F. Desprez, J. Dongarra, A. Petitet, C. Randriamaro, and Y. Robert, *Scheduling Block-Cyclic Array Redistribution*, in Parallel Computing: Fundamentals, Applications and New Directions, E. D'Hollander, G. Joubert, F. Peters, and U. Trottenberg, eds., North Holland, 1998, pp. 227–234.

[5] F. Desprez, M. Quinson, and F. 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), H. Arabnia, ed., vol. III, Las Vegas, June 2001, CSREA Press, pp. 1421–1427. ISBN: 1-892512-69-6.

[6] S. Domas, F. Desprez, and B. Tourancheau, *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, Aug. 1996, pp. 3–10.

[7] J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson, *An Extended Set of Fortran Basic Linear Algebra Subroutines*, ACM Transactions on Mathematical Software, 14 (1988), pp. 1–17.

[8] J. Dongarra, A. Petitet, and R. C. Whaley, *Automated Empirical Optimizations of Software and the ATLAS Project*, Parallel Computing, 27 (2001), pp. 3–35.

[9] J. Dongarra and K. Roche, *Deploying Parallel Numerical Library Routines to Cluster Computing in a Self Adapting Fashion*, Submitted to Parallel Computing, (2002).

[10] I. Foster and C. Kesselman, eds., *The Grid: Blueprint for a New Computing Infrastructure*, Morgan-Kaufmann, 1998. ISBN 1-55860-475-8.

[11] T. Howes, M. Smith, and G. Good, *Understanding and Deploying LDAP Directory Services*, Macmillian Technical Publishing, 1999. ISBN: 1-57870-070-1.

[12] M. 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), Fort Lauderdale, Apr. 2002.

[13] V. Rayward-Smith, *UET Scheduling with Unit Interprocessor Communication Delays*, Discrete Applied Mathematics, 18 (1987), pp. 55–71.

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

Edited by: Dan Grigoras, John P. Morrison, Marcin Paprzycki
Received: October 01, 2002
Accepted: December 21, 2002