Cost prediction for load-balancing: application to Algebraic Computations

J.L. Roch, A. Vermeerbergen and G. Villard

Laboratoire LMC-IMAG, 46 Av. F. Viallet,
F38031 Grenoble Cédex
e.m. *jlroch@imag.fr, vermeer@mistral.imag.fr, gvillard@imag.fr*

Abstract

A major feature of Computer Algebra, and more generally of non-numerical computations, is the dynamical and non-predictable behaviour of the executions. We then understand that statical analysis should imperatively be completed by dynamical analysis in order to reach the best distribution of the tasks among the processors. In this paper, we present a new load-balancing system for parallel architectures with great numbers of processors. Being well suited for Computer Algebra and based on the notion of granularity, it is original in the sense that it takes into account the tasks complexity as a consistent information in order to achieve efficiency.

1 Introduction

Algebraic computations are intrinsically dynamic: in most cases, the effective cost of a computation cannot be predicted. The main reason is that the intermediate coefficients growth is difficult to estimate precisely since it relies on a lot of parameters. Therefore, the taking into account of a small number of these parameters, in order to characterize the inputs (as it is done from a practical point of view), leads to too large and often unrealistic upper bounds on the results. For instance, a matrix multiplication may be efficiently parallelized on 64 processors using a block repartition [9]. If the input data are *well conditioned*, i.e., at the level of the algorithm the basic operations have almost the same cost, this parallelization is efficient. However, in general, the basic operations are not equivalent. For instance, some coefficients may be polynomials with 1000 monomials, while others may be small integers or even zeros. Although it is useful, the static repartition is insufficient.

Consequently, even if the considered problem is highly parallel ($\mathcal{NC}$ class), the static parallelization of a general algebraic algorithm is a difficult issue. Indeed, given a number of processors, a good parallelization consists in equitably distributing the work load, so as to guarantee the best possible processors use. But this work load is related to the inputs and cannot be statically determined. A dynamic load-balancing mechanism is thus necessary to decide how the parallelism has to be exploited (i.e., at which level of granularity it is needed to work, which computations need to be distributed...) in order to decrease the total computation time [1].

In the framework of Computer Algebra, we show how the cost of an algorithm at its current level of granularity (assuming that all the operations have the same cost), is a consistent information which the load-balancer has to take into account in order to achieve efficiency. This fundamental property leads to the specification of a new load-balancer, based on tasks complexity, and dedicated to computer algebra. This approach is different from the ones which do not take into account the cost of tasks to be distributed [1, 6, 10].

Moreover, our target model suffers no limitation: we assume that a large number of processing units are available (more than one thousand). This number imposes two constraints to a load-balancing mechanism:

- it leads to thiner granularities. A new general methodology is thus needed to develop high-level parallel applications. At a given level of granularity, the design should take advantage of previous improvements obtained at lower granularity levels. We show how a load-balancer taking into account tasks granularity can satisfy this condition.
- In order to ensure coherence in the distribution, a global control is needed. Even if the former is centralized, its management must be distributed to avoid bottlenecks. The solution we chose consists in using a classical token-ring strategy [2].

Load-balancing characteristics and specifications given in this paper are dedicated to computer algebra algorithms, but the system is designed for any target architecture (as far as it includes many processors). The described specifications may be implemented on either distributed memory machines, shared memory machines or workstation networks. Two groups of parameters (corresponding to either machine-dependent characteristics or to algorithm-dependent properties) are used to achieve adequation between the load-balancer and the target architecture. An evaluation is in progress on a 128 transputers network [14]. Our target domain of computer algebra are arithmetic, linear algebra [13] and lattice basis reduction.

2 Algebraic Computations Requirements

This section intends to point out the specific characteristics and requirements of Algebraic Computations. These are mainly the dynamic behaviour of the executions and the fact that at a given level of granularity, a static parallelization is both irrelevant and incomplete.

2.1 Parallelization Driven by a High Level Cost

The execution cost of an algebraic algorithm strongly relates to its inputs, which are often very complex to describe. For example, let us consider a generic algorithm for matrix multiplication with coefficients in a commutative ring $\mathcal{R}$. Its cost may only be described by the number of operations performed in $\mathcal{R}$: this cost is thus related to the sole dimension of the input matrices, assuming that basic operations in the ring $\mathcal{R}$ are performed in constant time. However, the effective time spent for two given entry matrices, with coefficients in a given ring, actually corresponds to the number of binary operations performed. For instance, the effective cost of a matrix multiplication may only be due to the product of two coefficients, which cost can be distinguished only at a lower granularity level.

For instance, let $\mathcal{R} = \mathbb{Z}[X]$: the effective cost depends on the dimensions of two matrices, on the number of monomials in each polynomial coefficient, and on the size of the integer coefficients of each monomial. If the multiplication to is the following:

$$\begin{pmatrix} P & 0 & 0 & \dots & 0 \\ 0 & Q & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{pmatrix} \times \begin{pmatrix} P & 0 & 0 & \dots & 0 \\ 0 & Q & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{pmatrix} \quad with \quad \begin{cases} P = 1000! X^{10} + 1 \\ Q = \sum_{i=0}^{1000} X^i \end{cases}$$

then the effective cost will mainly results from the multiplication of large integers, i.e. $1000! \times 1000!$ and the product of large polynomials, i.e. $Q \times Q$.

This is a major remark in computer algebra: it means that, most of the time, the cost of a high level computation cannot be predicted. In the example, the main operations needed to parallelize are the multiplication of integers and the product of polynomials; in these operations, the parallelism involved concerns lower granularity operations. This parallelism, related to the inputs, cannot be found statically in the algorithm, but rather dynamically when this algorithm is to be applied on specific data.

Nevertheless, the cost of an algorithm, evaluated at a given level of granularity, constitutes a significant information. If this cost is high enough, then a parallelization of the algorithm at this level may be worth doing. If it is low, no parallelization should be useful at this level –because of communication overheads– though lower level operations may be efficiently performed in parallel (such as in the above, for example). Let us notice that this cost could be automatically evaluated [8].

2.2 Levels of Parallelization: An Example

Let us consider the following recursive algorithm for matrix product over a ring $\mathcal{R}$:

```
MatrixMultiplication (M ∈ R^{r×n}, N ∈ R^{n×n}) ==
  if (r == 1) return( VectorMatrixProd(M, N) )
  else
    M_H := the r/2 first rows of M ; M_L := the r/2 last rows of M
    R_H := MatrixMultiplication(M_H, N)      (1)
    R_L := MatrixMultiplication(M_L, N)      (2)
  return( MatrixMultiplication(R_H, R_L) )
```

Let $T(r, n)$ be the cost of this computation, assuming that computations in $\mathcal{R}$ are performed in constant time. The cost of the sequential execution for two $n \times n$ matrices is $T_{seq}(n, n) = 2.T_{seq}(\frac{n}{2}, n)$. Computations (1) and (2) may easily be performed in parallel: the first splitting of the recursion involves two processors, and thus the cost of the total parallel execution is: $T_{par}(n, n) = T_{par}(\frac{n}{2}, n) + T_{com}(M_H, M_L, N, R_H, R_L)$. A more precise definition of T_{com} relates to the parallel model:

- On a shared-memory architecture, T_{com} corresponds to the time spent in accessing the shared data (for instance, protected by semaphores). Thus, T_{com} may be assumed to be a machine-dependent constant, but independent on the size of the shared data.
- On a distributed memory architecture, T_{com} corresponds to the time needed for communicating data from one processor to another (point to point communication, between neighbour processors or performed by a router). Thus T_{com} may be assumed to be a linear function in the size of the shared data.

On both models, T_{com} is at most linear in the size of the shared data. As T_{seq} here is a super-linear function and T_{com} is linear, it exists n_0 so that: $n > n_0 \Longrightarrow T_{par}(n, n) \leq T_{seq}(n, n)$. Parallel execution of sub-multiplications (1) and (2) will be interesting if: $T_{par}(n, n) \leq T_{seq}(n, n)$. if $n < n_0$, a parallel execution of both sub-multiplications will be interesting only if the operations performed in $\mathcal{R}$ (considered as basic operations at the level of the algorithm) can themselves efficiently be performed in parallel. In this case, the benefit brought by parallelism is not due to parallel execution of (1) and (2), but by the inherent parallelism of (1) and (2).

A simple way to exploit parallelism at every granularity level is to perform the operations in $\mathcal{R}$ using similar recursive algorithms. This process is repeated and controlled by the effective cost of the operations at the current granularity level. Thus, it automatically decreases the granularity, exploiting parallelism at each level where gains can be guaranteed.

2.3 Theoretical Overhead

The overhead of this scheduling strategy may be theoretically evaluated if enough processors are available. Let us consider a problem involving n entries, which may be split into p instances of the same problem, each involving $\frac{n}{K}$ entries (the time of the lifting process is assumed to be a constant). Let us denote $T_{//}$ the time of the direct parallelization of this problem, involving $n^{\log_2 p}$ processors. On a shared memory model, the total overhead of the load-balancing will be $\log_k n$ for a parallel execution involving $n^{\log_2 p}$ processors, and the total parallel time will be $T_{//} + \log_k n$. On a distributed memory model, this total execution time will be $T_{//} \log_k n$.

As an example, if we consider Gaussian elimination, the overall cost of our strategy will be $n \log n$ using n^2 processors, which is within a $\log n$ factor of the best known parallelization. Morerover, on the one hand the dynamic strategy avoids useless computations (of zeros, for instance), and on the other hand it is well suited to a direct computation.

3 Load-Balancing : High-Level Specifications

The specific requirements of Algebraic Computations presented above and the related *top-down* approach for the granularity, led to the main choices we have made for the load-balancer. In the following, these high-level specifications are presented both from the user's point of view and from the machine designer's point of view. The user manages the parallelism at its own level of granularity, usually a coarse one. In other words, the user specifies a static mapping of his coarse granularity tasks which is relevant at this level. The inherent parallelism to the arithmetic operations called by the user, although appropriate to a static mapping because of its lower granularity, is then automatically exploited by the load-balancer.

To express the dynamic parallelization, several choices are possible [3, 5, 10]. As far as algebraic computation is concerned, mainly by independent AND parallelism (modular computations, distributed computations) and OR parallelism (probabilistic algorithms, several strategies with no deterministic choice) [11], we have chosen, in order to express parallelism, to use *Fork* (OR-parallelism) primitives and *Fork-Join* (AND-Parallelism) primitives, well suited for an effective computation. In any case, this choice is not restrictive: any primitive allowing *dynamic process creation* and *Rendez-Vous* is convenient. A peculiar choice does not modify the general specification of the presented load-balancer.

In order to stick to the framework of the *remote processes activations*, we have increased the usual *Fork* and *Join* functionalities. In this section, we will use two new functions, *RemThreadFork* and *RemJoin*, which stand for *Fork* and *Join* on possibly distinct processors. They will be described in detail in section 5.

3.1 User Application

The user will manage the parallelism of its application at two levels: he starts by writing a distributed application using the usual rules and parallel language facilities to communicate.

These communications will transit onto the *user network* (see section 4 for a description of the different networks of the load-balancer). At this level, we indeed offer the possibility to completely control the data distribution, the synchronization points and the data exchanges. This corresponds to the highest level of granularity, with a relevant user mapping. Between two data transfers, the user may then exploit a parallelism also inherent to the arithmetic operations he calls, but too difficult to map because of its lower granularity. The example in section 3.2 below shows how to operate in this case, i.e., how to call the load-balancer. Our approach is illustrated in figure 1: the user statically maps its application on its private network, his processes may then call the load-balancer.

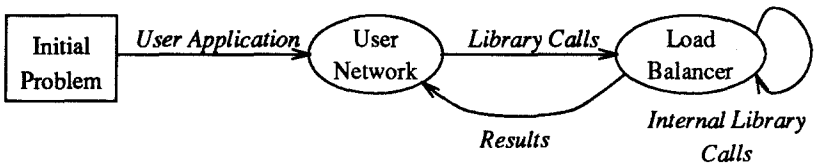

Figure 1 : General behaviour of a user application

3.2 Describing Load-Balancing : the Granularity Threshold

We have seen how the parallel processes are activated. We now show how the cost predictions are taken into account in order to dynamically allow the load-balancer to decrease the granularity and reach the more convenient one. Let us take a typical example. We can write a function which will realize the product of two quantities a and b.

```
mul( *res,a,b )
  case ugdecision( mul,a,b )
  seq :  /* Sequentially on this processor */
    seqmul( &res,a,b )
  par :
    aH = high weights of a ; al = low weights of a
    taskH = DefRemThread( fork,mul,ArithmCost(mul,aH,b),
      CommCost(aH,b), *resH, aH, b ,
      comm.  for resH, comm.  for aH, comm.  for b )
    myfork = DefFork(1)
    /* Computations on high and low weights are done
     in parallel, probably on distinct processors */
    RemThreadFork(myfork,taskH)
    mul( &res,al,b )
    RemJoin( myfork )
    add( &res,resH,resl )
  endcase
```

From the weights of a and b, a prediction will be done during the execution for the cost of the operation. The weight of an operand is a relevant information on it, much cheaper to manipulate than the operand itself, e.g. the binary length for an integer, the dimension for a matrix, the depth for an expression tree... This prediction will allow the evaluation of a *user granularity decision* (`ugdecision` in the example) to `seq` if it is not worthwhile running the parallel algorithm instead of the sequential one, and to `par` in the other case. In other words, the prediction may be formulated by using the notion of threshold : on this side of

a given *user granularity threshold* (the details are given in section 4), the product –at this level of granularity– will be made sequentially; beyond, part of the work will be taken on by the load-balancer. Let us yet notice that if the function parameters are known, the *user granularity threshold* may be statically computed (unlike the other thresholds involved in the load-balancer (section 4)).

3.3 Parameterizing the Load-Balancer

A general purpose load-balancer will behave differently according to the current application and the current machine: adjusting some basic parameters from some relevant criteria. As it is, the ones that have to be considered, are not known for Computer Algebra applications: our work consists in pointing them out and proving their relevance. In order to do so, the load-balancer must allow the consideration of the largest possible number of situation. A good solution is to parameter the load-balancer as much as possible. This will be done at three main stages, from the user level to the run-time system level:

- the load-balancer must let the user adjust the behaviour of the automatism, being given its own problem and its own knowledges.
- The hardware and software costs of the machine may considerably influence the decision whether to export a given task or not.
- During execution, the arithmetic load of the processors and the communication load of the network have also to influence the exportation decisions.

The decisions related to the first point will obviously belong to the user: he will guide the load-balancer behaviour via its own *user granularity thresholds* previously introduced. Relating to the other two points, the *machine threshold* and the *load threshold* will be detailed in section 4 in order to take into account both hardware and software characteristics of the machine and the loads of the system. These three thresholds are intended to dynamically lead the system to better tasks distributions.

3.4 Control and Communications

We terminate this section on the high-level specifications of the load-balancer with some basic principles. They have been applied in order to distribute a *global control* and to simplify the aspects of the different communications which take place.

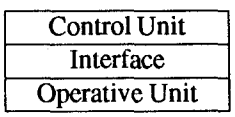

Figure 2 : Abstract view of the load-balancer.

In an abstract manner, a load-balancer is composed of three main parts. The same division will be used in order to distinguish three main types of communication (in addition to the user communications). The three main parts of an abstract load-balancer are,

- a *Control Unit* taking the mapping decisions,
- an *Operative Unit* receiving orders from the *Control Unit*, appling them and executing the user application.
- a *Control/Operative Interface* linking the *Control Unit* and the *Operative Unit*. Via this interface, the *Control Unit* sends orders and the *Operative Unit* sends the loads measurements.

This division is directly implemented by our load-balancer (section 4). Indeed, we have divided the processes in *controllers* and *workers*, and the communications will transit onto the following inter-processes "parallel networks":

- the *user network* on which transit the messages explicitly communicated by the user,
- the network dedicated to the global control, i.e., to the management of the resources : the *token network*,
- the *mapping network* which plays the role of the interface between the *controllers* and the *workers*,
- the network used by the workers to exchange the data (parameters and results) of the exported tasks : the *inter-workers network*.

The reader will refer to figure 3 for a global view of these different networks. The position of our scheduler in the taxonomy[4] is dynamic, adaptive, heuristic and it has an intermediate position between centralized and distributed control. Note that we do not stick to this terminology, as our scheduler is not *load-balancing* in the sense of[4].

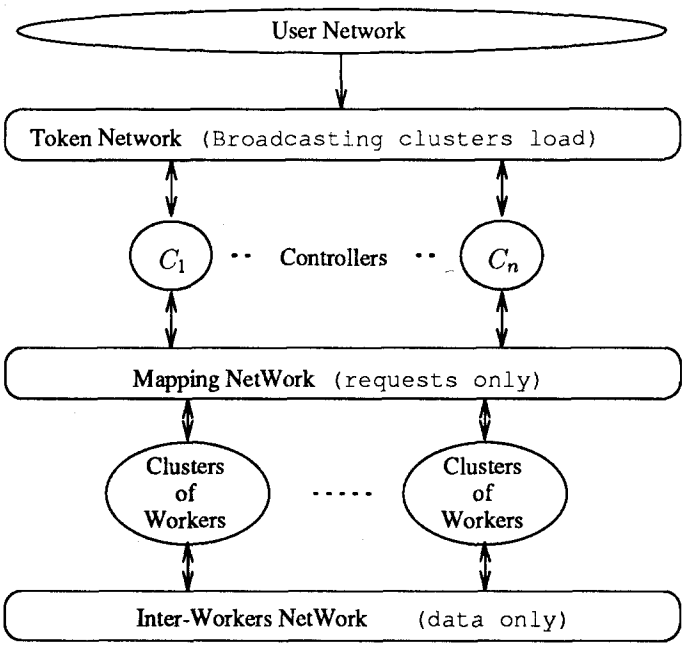

Figure 3 : Global view of the logical architecture

4 Logical Architecture of the Load-Balancer

We may now detail the logical architecture of the load-balancer. The algorithms of the principal tasks which are involved will be given in the next section which will be concerned by the balancer implementation on a distributed architecture. We have *processors* and *processes* which execute on the former.

4.1 Global View

We introduce here a classification of the different processors and processes of the balancer. The processors differ in the processes they execute and the networks they belong to.

Processors. Processors are of two types: the *controllers* and the *workers*. To each controller a cluster of workers is associated, being divided into two groups:

- the *global workers*, also called global resource, are ready to receive the tasks from any other worker.
- the *local workers*, also called *local resources*, which will only receive tasks from the processors of the same cluster or from themselves.

These two latter notions will allow to establish and test the influence of proximity relations between the processors.

Processes. In the same way, we point out three types of process:

- the *resource managers*, managing the arithmetic requests and transmitting the informations concerning the *mapping of the exported tasks*,
- the *arithmetic processes*.

The following sections give details and describe the overall behaviour of the different processes during the execution.

4.2 Distributing a Global Control

As previously noticed, the state of the global resources is kept up-to-date by the circulation of a token between the different controllers. This token encodes the number of global workers ready to be allocated for some extra work: it indicates to the controller to which cluster the workers belong. The following points are always satisfied:

- at a given time, only one processor has the token and can modify it,
- a processor allocates a global resource only when it gets the token; in this case, the only field of the token which is modified is the one corresponding to the cluster of the allocated worker.

Given those assumptions, the management of the global resources is particularly simple: a controller receiving a request of global resource from its cluster waits for the token. Once the operation is performed, it may allocate an available global worker, and consequently modifies the token before sending it to the next controller.

4.3 Controllers and Workers

The controllers : they are dedicated to the management of the resources. In order to do so, they concurrently execute a *global resource manager* and a *local resource manager*. The first manager allocates resources only at each reception of the token, whereas the second one may always allocate some (provided a local worker is available): it itself manages the corresponding informations.

The workers : they execute the exported arithmetic operations. Notice that a worker is not only passive (reception of orders): since the operation it has been asked to execute may involve *remote thread activations*, the worker may become producer –active– and ask for the exportation of its tasks. For the parameters and results exchanges, the global workers use the *inter-workers network* while the local ones communicate within themselves using the intra-cluster *mapping network*.

4.4 Processes and Interconnections

In the present section, we are concerned by more general descriptions about their behaviours.

The global resource managers : on the one hand, the *global resource manager* has to receive the requests from the *local resource manager*, to wait for the token , to modify it for the resource allocation, and finally to answer to the request by communicating with the *mapping process* which is associated to it. The *global resource manager* is thus connected on the *token network* and has two other links: a first one able to receive the requests and a second one able to answer to them. On the other hand, when receiving the token, the manager compares it to its old value, to know if one of the workers of its cluster has been allocated. If yes, it gets in touch with the concerned worker via the mapping processes so that the connection between workers can be established.

The local resource managers : in an analogous way, the *local resource manager* has to receive the request from the local *mapping process*. And, whether there is an available local processor or not, it directly answers to the *mapping process* or relays the request to the *global resource manager*.

The mapping processes : placed on all the processors of the network, their role is to collect the resource requests from the user application and from the exported processes, to relay those requests to the *resource managers*, to get the corresponding answers, and finally to inform the source processor and the one chosen for the exportation of their respective identification (so that they can subsequently exchange their data).

The user or exported processes : these are all the processes which may ask for a resource to export some arithmetic tasks. We have yet seen that the requests are sent to the *mapping processes*. The answers may then be either the identification of a found worker or a negative answer. Once they are virtually connected by their respective *mapping process*, these application processes only communicate via their *inter-workers network* (in order to exchange parameters and results).

4.5 Global Description of Remote Process Activation

We are now ready to give an overall description of a *remote process activation* from the resource request to the reception of the operation result (assuming a free resource is available in the network). After a call to *RemThreadFork* (section 5), the father process (i.e., which has called the activation) will go on executing on the same processor. Concurrently, the requests are sent to the resources-management processes to find other processors for the execution of the sons arithmetic tasks. If those tasks have arithmetic costs which are high enough (depending on the overall load) and if other processors have been found to be unemployed, the corresponding works are exported. Otherwise, the sons will also execute on the original processor. After that, the *RemJoin* synchronization (section 5) simply consists in a transfer of the operation results between the processes concerned. Seven main steps are distinguished.

1. The source process asks for a request to its mapping process, and waits for an answer.
2. The mapping process relays the request to the mapping process of the controller and then to the local resource manager.
3. If a local resource is available the local resource manager sends its answer via the mapping processes, otherwise it relays the request to the global manager.
4. If it is in touch, the global manager waits for the token, takes a decision and sends its answer via the mapping processes.

5. The mapping processes establish the connection between the source process and the chosen destination resource.
6. The source and the destination resource exchange the parameters of the exported operation.
7. The source and the destination resource exchange the result of the exported operation.

In the situation where no free resource is found in the network the global manager simply sends a negative answer to the source process which then itself executes the work which could have been exported.

4.6 Thresholds

We have seen at section 3.2 and 3.3 that to use the notion of *threshold* is a good way to formulate the predictions and the exportation decisions. These decisions are taken at three different levels, to each level corresponds a particular *threshold.*

The granularity threshold : as presented in section 3.2, the user may use a threshold to decide whether to export an operation or not. For instance, after a study of the theoretical cost of his parallelization he may force the system to reach a desired level of granularity, to find the best compromise between arithmetic and communication costs before beginning the exportations. In order to do so, the user will write a *user granularity decision* function, according to the arithmetic operations he uses and the argument types he manipulates. Using constants delivered by the system (as elementary computation, communication or memory management cost) and given an operation and a type, this function will return `par` or `seq` indicating whether the work may be parallelized or not.

The machine thresholds : a lot of characteristics of the machine, both hardware and software, must statically influence the behaviour of the load-balancer. We may point out for example:

- the ratio of the atomic communication cost over the arithmetic one. If it increases, such a ratio will clearly decrease the volume of remote processes.
- The distances in the network. The *global manager* and the *local manager* must have different thresholds if the *local resources* are reachable at a low cost compared to the *global resources* one.
- The ratio of the process creation cost over the arithmetic one. The remarks are the same than above.
- The different arithmetic units capabilities. The availability of some dedicated or powerful arithmetic units must lead to thresholds depending on the types of the free resources.

Those characteristics will be taken into account by using system-defined constants and, as previously, a *machine-granularity-decision* function.

The load thresholds : The decisions and the parameters we have discussed above rely on statically defined data. This will be completed by dynamically assigned parameters to take into account mainly the load of the system. During execution, the load of the networks (the volume of routed data) and the arithmetic load of the processors should clearly influence the balancer behaviour and allow to take the final exportation decisions.

Calibrating the load threshold : Among the three types of threshold mentioned above, both the granularity and the machine ones can be determined in a deterministic way in the general case. The former is a function issued from concerns about algorithms

complexity and the later is a set of physical constants that can be determined by a series of preliminary tests. This is not the case of the load threshold in our general context: network load cannot be predicted during the execution. This problem can be tackled by using the temporal locality hypothesis on the network load, that is to say, the network load varies slowly and continuously with time. This method has already been successfully assumed in close contexts. For example, this hypothesis in used in [15] on communication costs and task executions time in a bi-processors to feed exact equations giving the frequency for load-balancer invocations. In our first implementation, we locally estimate an expected communication cost by using routers statistics. This local estimation is expected to be satisfying enough and meets the stability requirement of [7]. However, our model has an intrinsic global state stored in the token.

5 Implementation on a Distributed Architecture

The model we propose for load-balancing is designed for any parallel architecture. For instance, it could be easily implemented on shared memory computers: they are well suited for a global management of resources or to private data exchanges between pairs of processors. In the same way, the distributed memory architectures naturally provide the notion of network and the facilities for the management of many communicating processes. Furthermore, this second model does not lead to any serious limitation concerning the number of processors: this is the reason why (PAC [11, 12] has privileged this alternative) all the implementation part of the paper will rely on a distributed memory model.

5.1 Target Model and Mapping

Our abstract-machine model consists in processes communicating by using a message-passing protocol using `send` and `receive` functions with the appropriate processes identifier (receiver or sender). The processes presented at section 4.4 are gathered together and simultaneously executed on processors. This mapping of the processes on the processors satisfies the two following rules: 1) One *mapping process* is placed on each processor, 2) If a *global (resp. local) resource manager* is placed on a processor, a *local (resp. global) resource manager* is placed on the same processor. These two rules don't lead to any serious limitation, the processes may be combined in many ways and mapped on many topologies. For instance, one may choose to map any number of controllers and workers on a given processor, and may fix any ratio between the numbers of local and global resources, depending on the different costs. The reader will refer to the section 6 for a detailed example of an implementation of the balancer on a 2-dimensional torus topology.

6 Evaluation

The evaluation of the model presented in this paper is currently under progress on a 128-processors computer [14]. Tests will be soon available, concerned with linear algebra problems [13] and lattice basis reduction. For the sake of simplicity the target topology is a 2 dimensioned torus of $P \times Q$ processors which has to be viewed as P horizontal rings and Q vertical rings. One of the P horizontal rings is both the *user network* and the *token network*. Each node of this ring is thus a *controller* and the corresponding vertical ring

gives the cluster of his *workers*. Consequently the *mapping processes* may be connected essentially using those vertical rings, and the messages for the *inter-workers network* will be routed using all the torus links except those of the controller ring. Besides its simplicity and its regularity, such a topology allows to easily implement deadlock-free routing algorithms for the three system networks, and the two dimensions offers a lot of different situations for each value of P and Q [13].

7 Conclusion

We have presented a new load-balancing system, well suited to Computer Algebra. An application written with our model of *Remote process activation*, and based at each granularity level on previous developments at lower granularity levels, will automatically execute at its best granularity on any given architecture. This has been made possible simply by considering the tasks complexity as a consistent information, and always by taking into account the machine constants. Besides, this model is general enough in order to to find other applications in other domains.

References

1. G. Bernard, D. Steve, and M. Simatic. Placement et migration de processus dans les systèmes reépartis faiblement couplés. *TSI*, 10 (5):375–392, 1991.
2. D.P. Bertsekas and J.N. Tsitsiklis. *Parallel and distributed computation*. Prentice-Hall, 1989.
3. G. Booch. *Software engineering with Ada*. Benjamin-Cummings Publishing Company, 1983.
4. T. Casavant and J. G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, 14(2):141–154, February 1988.
5. K. Clark and S. Gregory. Parlog: Parallel programming in logic. In J.S. Kowalik, editor, *Parallel Computation and Computers for Artificial Intelligence*, pages 109–130. Kluwer Academic Publishers, 1988.
6. A. Beaumont et al. Flexible Scheduling of OR-Parallelism in Aurora: The Bristol Scheduler. In *PARLE'91*, pages 403–420, Eindhoven, The Netherlands, 1991. Springer-Verlag, LNCS 506.
7. D. Ferrari and S. Zhou. An empirical investigation of load indices for load-balancing applications. In P.J. Courtois and G. Latouche, editors, *PERFORMANCE '87*. Elsevier Science Publishers B.V. (North-Holland), 1988.
8. Ph. Flajolet and J.S. Vitter. Average-case analysis of algorithms and data structures. In J. van Leuwen, editor, *Handbook of Theoretical Computer Science*, pages 431–524. Elsevier, 1990.
9. G. Fox and al. *Solving problems on concurrent processors*. Prentice-Hall, 1988.
10. R.H. Halstead. Parallel computing using multilisp. In J.S. Kowalik, editor, *Parallel Computation and Computers for Artificial Intelligence*, pages 21–49. Kluwer Academic Publishers, 1988.
11. J.L. Roch. The PAC System and its Implementation on Distributed Architectures. In *Computer with Parallel Architectures : T. Node, ed. D. Gassilloud, J.C. Grossetie, Kluwer Ac. Pub.*, 1991.
12. J.L. Roch, F. Siebert, P. Sénéchaud, and G. Villard. Computer Algebra on a MIMD machine. *ISSAC'88, LNCS 358 and in SIGSAM Bulletin, ACM*, 23/11, p.16-32, 1989.
13. F. Siebert and G. Villard. PAC : First experiments on a 128 transputers Meganode. In *International Symposium on Symbolic and Algebraic Computation, Bonn Germany*, 1991.
14. Telmat. TNode Overview. Technical Report Doc-1.02-3.2, Telmat Informatique, 1990.
15. M.C. Wikstrom, J.L. Gustafson, and G.M. Prabhu. A meta-balancer for dynamic load balancers. Technical Report TR91-04, Iowa State University/Ames, Iowa 50011, January 1991.

This article was processed using the LaTeX macro package with LLNCS style