An Environment to Study Dynamic Load Balancing Functions and its Application to the Parallel Logic System *PLoSys*

S. E. Kannat, E. Morel and J. Briat

LGI-IMAG-INPG
46, avenue Félix Viallet
F-38031 - Grenoble - France
email: {kannat, morel, briat}@imag.fr
http://amon.imag.fr:80/Les.Groupes/PLoSys/

Abstract

Efficient parallel execution of Prolog programs, on scalable distributed memory architectures, need appropriate scheduling rules. This paper presents a platform to evaluate dynamic load balancing strategies for the Parallel Logic System PLoSys.

The PLoSys computational model exploits implicit OR-parallelism following a multisequential approach. In this kind of systems the parallelization is driven by the load balancing function. Many functions have been proposed in the literature according to the targeted parallel architecture. To identify an appropriate load balancing function we built an evaluation platform based on a modelling technique. Our approach consists in modelling the pure Prolog program execution by a directed acyclic task graph. The evaluation environment is currently developed on a Supernode computer whose processing elements are Transputers. We believe that our platform is a good testbed to improve and validate an OR-parallel strategy for the implementation of PLoSys on a specific parallel programming environment.

1 Introduction

A parallel logic system is defined by the characteristics of the supported language, its execution model, the control strategy of the parallel execution and the target parallel architecture [6][9]. *PLoSys* takes advantage of the implicit *OR-parallelism* of a *pure* Prolog program following a multisequential approach on *NORMA*[1][2] machine. In this class of systems, a program execution can be seen as an exhaustive parallel search of all possible solutions. The program execution starts on one processor with the evaluation of the initial goal. Meanwhile, the other processors ask for work, trying to get an unexplored part of the search tree. The search space of each processor is represented by its choice point stack. The depth of this stack represents the number of awaiting choice points. An execution control strategy fixes the search space exploration given by the evaluation of a Prolog program, in a way to optimize the efficiency. The deep-first strategy tries to minimize memory consumption, whereas the goal of a parallel strategy is to maximize the execution speed using a kind of width-first approach. Each processor executes a complete Prolog engine. Interprocessor communication is reduced to the exchanging of work (stack copying) and the load information between processors. In this type of systems, the parallelization is driven by the load balancing function which tries to equalize continuously the average load on each processor. The load balancing function selects a task from an overloaded processor and transfers it to an idle processor. A task represents a part of the search tree exploration, i.e. a part of the choice point stack awaiting for evaluation.

The efficiency of the load balancing function is determined by analyzing its behaviour which depends on the overall cost of the following elements that implement the strategy:
• the load estimation cost,
• the task selection cost,
• the transfer cost of the selected task,
• the number of migrations performed during the program execution.

An efficient load balancing function is the one that minimizes the overall cost. The scheduling overhead can be decreased by limiting the frequency with

[1] *NO Remote Memory Access.*

which the scheduler is invoked, and by optimizing the scheduling algorithm.

Many functions have been proposed in the literature. Therefore it is necessary to evaluate these functions to answer the following questions:
• is this function an efficient general load balancing strategy?
• is this function appropriate to a given program? a given class of programs?
• is this function dynamically adaptable to different intances of a large class of programs?

The real efficiency evaluation of the dynamic load balancing function requires a prototype. Given the heavy cost of realizing such a prototype, we chose to use a modelling technique. We focus our study on the load balancing function architecture and the dynamic load distribution strategy. We believe that such an evaluation is an important step before a real implementation on a specific parallel programming environment.

The remainder of this paper is organized as follows: Section 2 introduces the load balancing function of *PLoSys*. Section 3 presents the evaluation model. Section 4 describes the evaluation environment of the load balancing function. Finally, Section 5 outlines our futur work and concludes this paper.

2 The load balancing function

The load balancing function fixes the parallel search space traversal generated by the Prolog program evaluation in a way that optimises the execution time and the computation resources use.

We can present the problem of the load balancing in our parallel logic system as follow: a fixed number N of processors execute tasks with a finite duration. The execution of one task can generate an undefined (but finite) number of other independent tasks. If we consider at a given time t, that the risidual execution time $D_r(t)$ on one processor, is the execution duration sum of the waiting tasks on this processor and the risidual execution time of the current task. The execution time D_T of a task T is its own duration and the execution time sum of all the tasks it generates. Let E_T be the overhead caused by exporting the task T and I_T the time required to import and to start running the task T. The transfer time (E_T and I_T) depends on the multiprocessor architecture, the network communication caracteristics and the size of transferred data (cf. Figure 1). The condition under which it is worthwhile to parallelize the computation of the task T is given by the followimg inequality:

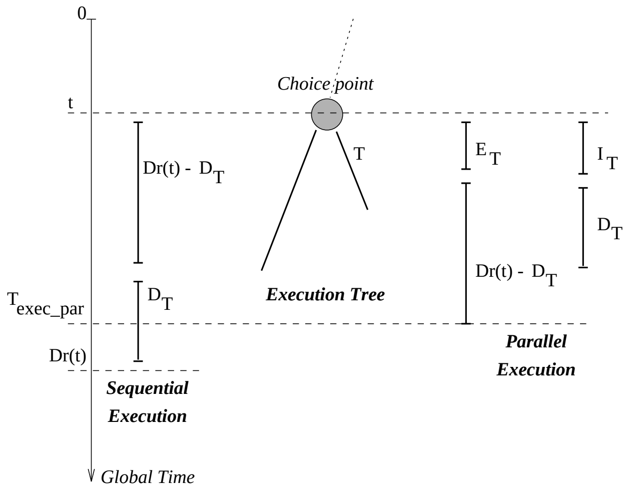

Figure 1: *parallelism constraints*

$$D_r(t) \geq Max[(D_r(t) - D_T) + E_T, (D_T + I_T)]. \quad (1)$$

This condition can be rewritten with the following constraints:

$$D_r(t) - D_T \geq I_T. \quad (2)$$

$$D_T \geq E_T. \quad (3)$$

The first constraint (2) represent the minimum exploitable grain of parallelism. The second (3) one shows the utility of the parallelisation regarding to the exporting processor state.

An efficient load balancing function is the one that satisfy the both constraints (2, 3) at each invocation. If the transfer cost (E_T and I_T) can be estimeted at runtime; it is not the case for the task duartion D_T and $D_r(t)$. An heuristic is requiered for predicting these durations.

The current structure of the *PLoSys* load balancing function is inspired from the OPERA scheduler [3]. Specialized central controller schedules slave workers. The scheduler is characterized by its system load management and by the work scheduling strategy.

2.1 System load evaluation

There are no simple and exact criteria to evaluate the load of a processor. Compile time analysis of the granularity of Prolog programs is currently an active research topic [8][11]. In *PLoSys*, the evaluation of workload is performed dynamically. The simplest heuristic is to measure the workload by the number of choice points held by the processor. Each processor can be either a producer or a consumer of choice points. As the total time of a program evaluation is a function of the number of created choice points [17], the processor load, at a given time, is estimated with the number of cumulated choice points. Several thresholds based on the number of choice points split the processors in three classes: *idle*, *quiet* or *overloaded*.

An *idle* processor does not have any Prolog task to compute. A *quiet* processor is active but does not have enough work to share with an *idle* processor without taking the risk of becoming rapidly *idle*. An *overloaded* processor is active and has enough work to share with an *idle* processor. The threshold between *quiet* and *overloaded* processors is simply a number of choice points.

The *idle* state is the initial state of every processor except one, on which we place the problem to solve. Each processor evaluates its load by a periodic sampling of its number of choice points. Whenever a processor has a load variation which overpasses a given amount, it sends its current load value to the load balancing function. This discrete sampling shows an approximate state of the global system, avoiding the small fluctuation of the choice points number on each processor. The scheduler is designed to use this approximate representation of the system, to avoid costly synchronization.

2.2 Work Scheduling strategy

When a processor is *idle*, it requests work from overloaded ones. The selected overloaded processor then sends a subset of its cumulated choice points. The size of the transferred data is known at execution time depending on the locality of the exported choice points. There are mainly two alternatives to select a choice point [15][16]:

- Task selection from the oldest choice point (*top-most scheduler*) [3][4][5].
- Task selection from the youngest choice point (*bottom-most scheduler*) [1][13].

The criteria used by the scheduler in *PLoSys* is to maximize the benefit provided by the selected work. To achieve this goal the load balancing function selects the highest possible work in the tree, following the heuristic that "higher is the work, larger is the granularity". This approach also minimizes the size of data (the task context) to be transferred by the initialization of a new task. To achieve that, each choice point is stamped with its creation date, which corresponds to its depth in the search tree. This allows to identify classes of overloaded processors depending on the depth of their oldest choice point.

According to the system load evaluation approach and the work scheduling strategy, three main parameters rule our dynamic load balancing function:

- the sampling rate of processor load evaluation.
- the number of exported choice points during a transfer operation.
- the threshold of the load level.

The different values of these parameters establish the efficiency of the strategy. These values depend on the Prolog program execution behaviour:

- The structure behaviour of the search tree.
- The rate of production/consumption of choice points.
- The overall number of choice points created by the program execution.

In order to evaluate the influence of each parameter value on the load balancing function and to validate our approach, we built an evaluation platform based on a modelling technique.

3 Evaluation model

Our approach consists in modelling the pure Prolog program execution by an acyclic task graph (*PLoSys_DAG*). Each task has two main parameters: a *memory cost* and a *processing cost*. The task graph is obtained either by trace execution analysis or from synthetic Prolog programs. The trace execution analysis allows modelling a particular real program execution on a given set of data. A *synthetic* Prolog program is obtained from the source code, by excluding all the variables from the program. This preserves the resolution mechanism avoiding the processing unification steps. A *synthetic* Prolog program allows us to control the program behaviour and to generate different search tree structures [12].

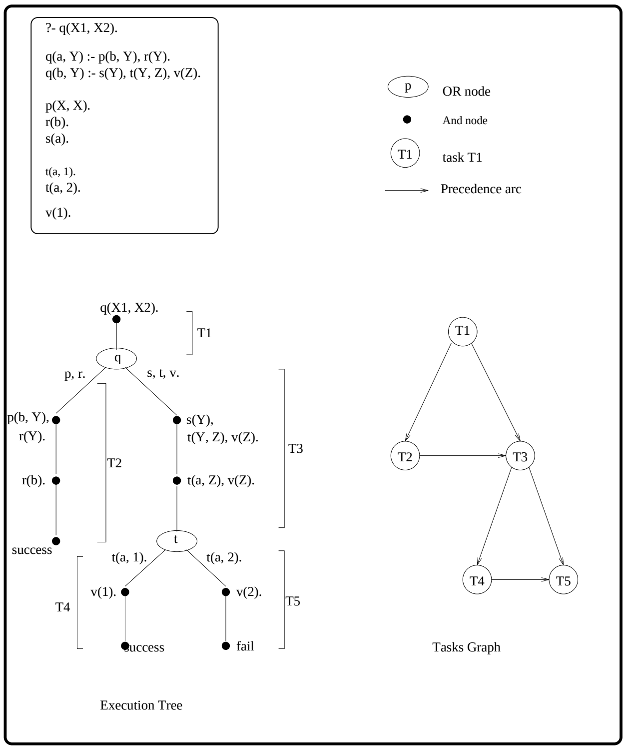

Figure 2: *Task graph extraction*

Task graph extraction: The execution of a Prolog program can be seen as an exhaustive traversal of an *AND/OR* tree (cf. Figure 2). The root of the tree is an *AND* node called the initial goal. Each *AND* node is a conjunction of sub-goals. Each *OR* node is the root of an alternatives set. Our approach consists in modelling the pure Prolog program execution P by an acyclic task graph (*PLoSys_DAG*) G defined by a set N of nodes ($N = \{1, ..., Card(N)\}$) and a set A of arcs. Each node in the *PLoSys_DAG* models a *synthetic* task which represents a sequence of *AND* nodes computation between two *OR* nodes, or between an *OR* node and a leaf of the Prolog execution tree. The *root* task represents the processing between the initial goal and the first created choice point (*OR* node).

Each node in the resulting graph is described by four parameters:

- a *synthetic processing cost* which describes the task computation sequence amount,
- a *memory cost*, modelling the data size to transfer if the task execution is not performed locally. It represents the *task context*,
- a number of *input arcs* and a number of *output arcs*. The arcs define a temporal precedence relation between nodes. This relation corresponds to a data dependency. An arc (i, j) of the set A means that the execution of the task j can start only after the end of the task i execution.

The advantage of using a synthetic task is the flexibility when varying the task parameters values. This allows the evaluation of the stability and the communications influence on the dynamic load balancing function.

4 Evaluation Environment

The evaluation environment is an extension based on the *ALPES* environment [14]. This extension allows the support of quantitative evaluation of dynamic load balancing functions.

4.1 Machine architecture

Our platform is currently implemented on a Meganode machine whose processing elements are general purpose parallel processor T800 floating point Transputers.

The *Meganode* is a distributed memory multiprocessor. This machine is an extended version of the T-Node machines, a serie that resulted from the European Esprit Project 1085, called Supernode [10]. T-Node computers have a reconfigurable topology, thanks to switching devices.

4.2 Platform processes structure

The platform is implemented by three kind of servers (cf. Figure 3): *the worker*, *the communication manager* and the *scheduler*. All the servers are virtually fully connected: each server s_i is connected by a logical communication channel to all other server s_j (even for $i = j$). The message routing between processors is achieved by VCR *(Virtual Channel Router)* [7].

The worker This server runs in every processor of the *Meganode*. It has three roles:
• receiving the task graph discription *(PLoSys_DAG)* from the controller processor. This information is kept in the *(PLoSys_DAG_Table)*,
• executing the synthetic load,
• counting the idle time of the processor.

This server is implemented by two processes:

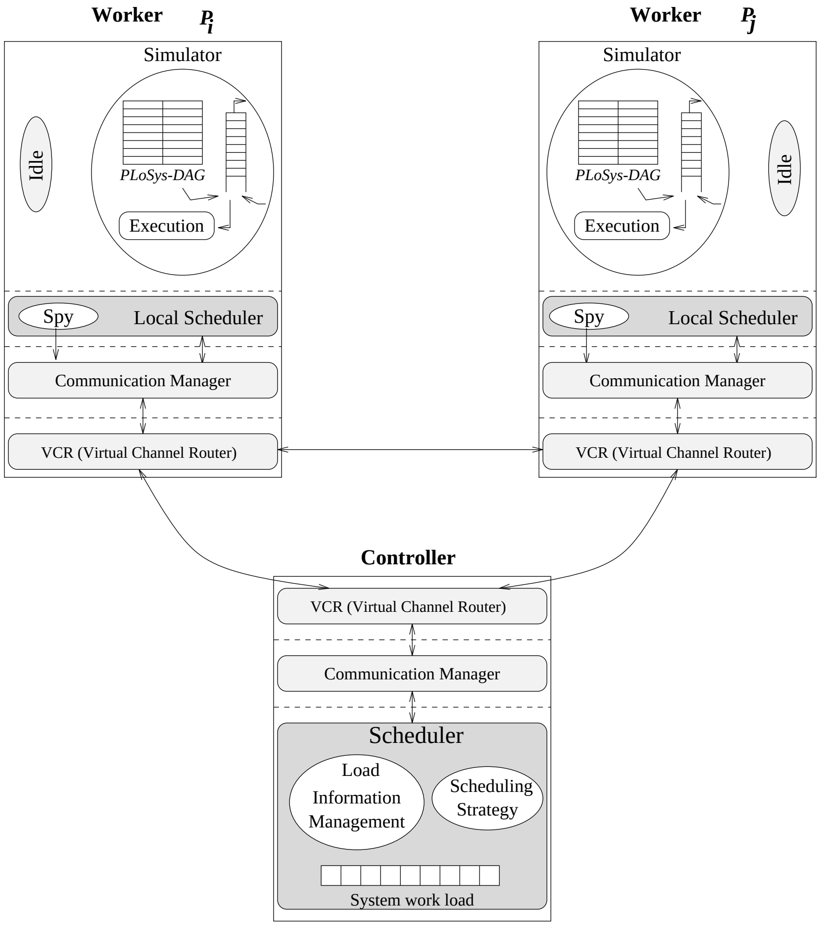

Figure 3: *PLoSys platform evaluation structure.*

- The **simulator:** it modelises the Prolog inference engine. This process manages a queue of ready tasks to be executed. It selects one and executes the corresponding workload. This execution consists in an empty loop execution whose number of iterations is associated to the computation cost of the node. At the end of this loop, the execution process puts in the task queue buffer the data concerning the successors (the "sons") of the task that has just finished to be executed. The task queue buffer is implemented as a *deque* (double ended queue) to store the task. Each processor uses such a task queue to maintain the different task nodes. Tasks which are generated by the own processor are inserted at one end of the queue. According to the load balancing function we evaluate, when this processor needs a task for its own, a potentially small one is taken from this end. Hence, the processor uses the *deque* in stack-like way leading to a depth first evaluation strategy. When a processor has to transfer a task, a potentially big one is taken from the other end;

- The **idle:** this process executes only when no other process executes. This task only increments a counter: the final value of this counter is used after the execution in order to estimate the idle time of the processor.

The communication manager This server, implemented on top of VCR layer, manages the communication messages between processors. It is started on all processors at the system initialisation. A message is caracterised by its size, its type and the data to be transmited. There are two type of messages:
• the control messages (initialisation, acknowledgement and ending),
• the scheduling messages (workload, tasks transfer).

The scheduler This server implements the load balancing function. It is organised by three processes: *the central scheduler* which runs on a dedicated processor (contoller processor), *the local scheduler* and *the spy* runing on each processor of the *Meganode*. Between the begin and the end of the execution, those processes cooperate to control the parallelism and the task migration between processors.

- the **central scheduler**: It reads a file containing the description of the workload (the *PLoSys_DAG*) and sends to each *worker* w_i the data concerning the whole *PLoSys_DAG*. The *central scheduler* keeps the identifier of the root computation nodes of the *PLoSys_DAG*. After this phase of data loading, the *central scheduler* sends a start signal to all processors. The start signal can be set in order to achieve either a simple sequential execution on one processor or a parallel execution on a group of processors. Starting a parallel execution, the *central scheduler* chooses one processor and sends a start signal to this processor in order to begin the execution of the synthetic load. The remaining processors receive a signal of work request. The time clock is initialized. The *central scheduler* manages the system load information coming from the worker processors and take scheduling decisions. These decisions are taken according to the implemented scheduling strategy. The algorithm try to match an overloaded processor with an idle one. The scheduling directives are sent to each selected processors' *local scheduler*. At the end of this execution, all the leaf computation nodes send an end signal to the *central scheduler*. The time clock is stopped to measure the execution time. Finally, the *central scheduler* waits for trace data coming from the different *worker* processors.
- the **local scheduler**: This process manages the scheduling directives coming from the *central scheduler* (controller processor). It implements a strategy to select tasks to be transfered to the idle processor. The transfer operation consists on:
  - removing the selected task from the overloaded worker task queue,
  - sending a message size from the selected overloaded processor to the idle processor according to the selected task memory cost,
  - pushing the transferred task in the idle processor task queue.

  When the transfer is over, both processors acknowledge the transfer to the *central scheduler* and the previous idle processor resumes work from the granted task.
- the **spy**: this process estimates the workload of the processor. It consults the number of tasks held by the processor in the task queue and sends periodically the size and the depth of the oldest task to the *central scheduler*. With this information the *central scheduler* maintains an approximated global state of the system and decides for the load balancing function activation;

4.3 Performance evaluation

Two types of performance metrics are obtained with our evaluation platform. The first one is the execution time of the synthetic load. The second one is the *load* information. The global load of a processor is composed of three parts:

1. *useful time*: it is the fraction of the total execution time corresponding to the execution of the empty loops. This fraction can be estimated before execution and after the use of the implementation strategies because we know the power of the emulated processor and the number of total empty loops to be executed on each processor;
2. *idle time*: it corresponds to the fraction of the total execution time when a processor does not work;
3. *overhead time*: it corresponds to the time spent doing housekeeping (operating system) and managing the synthetic load execution (the executive kernel).

The performance of the dynamic load balancing function is estimated by comparing it to:

- the sequential execution of the task graph (*PLoSys_DAG*) to establish the effective speedup,
- the fastest parallel execution of the task graph achieved on *ALPES* environment [14] by a static mapping strategies.

5 Conclusion

We have presented in this paper an original method to evaluate a load balancing functions for our parallel logic system *PLoSys*. Actually, some benchmarks used on the OPERA prototype [3] show that the speedup is nearly linear when the size of the problem is large but levels off rapidly for smaller ones. Modelling several problem size with our platform, will exhibit such phenomenon leading us to improve an *OR-parallel* strategy, and to evaluate the load balancing function regarding to the scheduler architecture and the network interconection characteristics for the real implementation of *PLoSys* on the IBM SP1 machine. We also believe that with the flexibility of our model, we will be able to adapt the platform to study dynamic load balancing functions for every parallel system which can be modelled by a direct acyclic graph.

References

[1] Athony Beaumont, S. Muthu Raman and Péter Szredi, "Flexible scheduling of Or-parallelism in Aurora: The Bristol Scheduler" *Proc. of Parallel Architectures and languages Europe, ESPRIT,* June 1991.

[2] R.C. Bedichek, "The Meerkat Multicomputer: Tradeoffs in Multicomputer design", *PhD Thesis, University of Washington,* juin 1994.

[3] J. Briat et al, " OPERA: OR-parallel Prolog system on supernode," *In P. Kacsuk and M. Wise, editors,* Distributed Prolog, Chapter 3. John Wiley and Sons, 1992.

[4] Ralph Butler, Terry Disz, Ewing Lusk, Robert Olson, Ross Overbeek and Rick Stevens, "Scheduling Or-parallelism: An Argonne Perspective", *Proc. of the int'l. conf. on Logic Programming, MIT Press,* pp. 1590-1605, August 1988.

[5] Alain Calderwood and Péter Szeredi, "Scheduling Or-parallelism in Aurora: The Manchester Scheduler", *Proc. of the sixth int'l. Conf. on logic Programming, MIT Press,* June 1989.

[6] M. Cannataro, G. Spezzano and D. Talia, "A Parallel Logic System on a Multicomputer Architecture", *Future Generation Computer System - Elsevier Science Publishers B.V, North-Holland,* vol. 6 , pp. 317-331, 1991.

[7] M. Debbage, M. Hill, D. Nicole, "Virtual Channel Router: Version 2.0 User Guide" *Department of Electronics and Computer Science, University of Southampton,* October 1991.

[8] Saumya K. Debray and Nai-Wei Lin, "Automatic Complexity Cost Analysis of Logic Programs", *Proc. of the Eight Int'l. Conf. on Logic Programming, The MIT Press, Paris,* pp. 599-613, 1991.

[9] G. Gupta and B. Jayaraman, "Analysis of Or-Parallel Execution Models", *ACM Transactions on Programming Languages and Systems,* vol. 15, num. 4, pp. 659-680, September 1993.

[10] Harp et al, "The Development and Application of Low Cost High Performance Multiprocessors Machine", *ESPRIT'86: Results and Achievements, North-Holland,* 1986.

[11] S. K. Debray, N-W. Lin and M. Hermenegildo, "Task Granularity Analysis in Logic Programs" *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation, White Plains, New York,* June 1990.

[12] S. E. Kannat et al, "A Platform to Study Dynamic Load Balancing Functions for Parallel Logic Systems", *IEEE Proceedinds of the First International Workshop on Parallel Processing,* pp. 580-585, December 26-31, 1994 Bangalore, India.

[13] Khayri Ali and Roland Karlsson, "Full Prolog and scheduling Or-parallelism in Muse", *IJPP,* vol. 19, num. 6, pp. 445-475, 1990.

[14] J.P. Kitajima, C. Tron and B. Plateau, "ALPES; a tool for the performance evaluation of parallel programs", *In Jack J. Dongarra and Bernard Tourancheau, editors,* Environments and tools for parallel Scientific computing, pages 213-228, Amsterdam, North-Holland, 1993.

[15] V. Kumar and V. Nageshwara Rao, "Parallel Depth First Search, Part 1: Implementation", *Department of Computer Sciences, The university of Texas at Austin, TR-88-28, Austin, Texas,* July 1988.

[16] V. Kumar and V. Nageshwara Rao, "Parallel Depth First Search, Part 2: Analysis", *Department of Computer Sciences, The university of Texas at Austin, TR-88-29, Austin, Texas,* July 1988.

[17] Evan Tick, *Parallel Logic Programming,* In The MIT Press, Cambridge, Massachusetts, England, 1991.