Distributed OR Scheduling with Granularity Information

Patrícia Kayser Vargas [1] Jorge Luis V. Barbosa [1 2] Débora Nice Ferrari [1]
Cláudio F. R. Geyer [1] Jacques Chassin [3]

[1] Universidade Federal do Rio Grande do Sul
Instituto de Informática
Porto Alegre – RS, Brasil
[2] Universidade Católica de Pelotas
Escola de Informática
Pelotas – RS, Brasil
[3] Laboratoire Informatique et Distribution – IMAG
Grenoble, France

Abstract—
Logic Programming has some implicit sources of parallelism as OR parallelism that facilitates the automatic parallelization. There are several parallel logic systems (PLPS) that exploit only OR parallelism but most part consider shared memory in its design. Few distributed implementations exist. PLoSys is an example of this kind of distributed implementation of parallel Prolog.

This paper presents a proposal of granularity control for parallel logic system and the possible enhancement in PLoSys OR parallel system concerning scheduling using granularity information. Scheduling is one of the most important part in a distributed system since it decides the task attribution to processing elements and the execution order of each task. Several performance measures could be optimized in this allocation process.

Keywords— **parallel logic programming, static analysis, task scheduling, granularity analysis**

I. Introduction

Parallel processing is a very important alternative to obtain good performance from programs. It can be exploited implicitly or explicitly. Most well-known parallel systems are based on *imperative* programming model and parallelism is exploited *explicitly*, making the programming task hard. Contrasting with imperative programming, *declarative* programming presents a higher-level programming model where the programmer needs to worry on **what** to solve and not on **how** to solve a problem. Because of this declarative characteristic, logic programming offers some opportunities to exploit implicit parallelism [18].

In Logic Programming there are some implicit sources of parallelism that facilitate the automatic execution. The two main sources are *OR-parallelism* – parallel execution of clauses that composes a predicate – and *AND-parallelism* – parallel execution of goals that composes a clause. The main inherent difficulty of AND-parallelism is to obtain coherent bindings for variables shared by several goals executed in parallel. This has lead to design execution models where goals which may possibly bind shared variables to conflicting values are either serialized (*independent AND-parallelism*) or synchronized (*dependent AND-parallelism*).

Several systems exploit only AND-parallelism [21, 22], only OR-parallelism [2, 6, 28, 32] or a combination of AND- and OR-parallelism [13, 19, 7]. Systems that combine both AND- and OR-parallelism are still under development and research.

OR parallelism is present in many application areas as such as parsing, optimization problems and databases. OR parallelism is one of the simplest forms of implicit parallelism, since it can be executed without including any annotation in the code. At the same time, the execution model is close to the sequential one. This helps reusing efficient sequential techniques.

Systems implemented for shared memory architectures present several limitations: (1) the hardware is not scalable, (2) algorithms and data structures designed for shared-memory architectures sometimes are not scalable if the hardware is scalable. Distributed implementations of PLPS would be desirable, but are complex and there are few implementations. PLoSys [31] is one example of OR-parallel system for distributed memory systems. It was initially implemented for SP/2 machines using a centralized scheduler. PLoSys was used as a study case to test important concepts concerning PLPS over distributed environment.

Scheduling PLPS is difficult since the granularity of tasks is in general unknown. Most schedulers use heuristics as the original PLoSys scheduler did. This paper presents one improvement with respect to the original scheduling strategy of PLoSys [30, 31]. The main idea is to use a static analyzer to infer granularity information at compile time. This information is then used at runtime by the PLoSys scheduler.

The rest of this text has the following structure: section II presents some details of PLoSys implementation, section III presents the way granularity informations are obtained, IV

proposes the use of static information for improving scheduling, V presents some related works, and, finally, section VI presents the final considerations.

II. PLoSys System Outline

PLoSys implements an implicit OR-parallelism approach without including any annotation in the code. PLoSys considers that no shared memory is available and all communications must be done through message passing. As any other efficient implementation, it uses the *multi-sequential* model [25]. The multi-sequential model is characterized by a set of processes (*workers*) that have a full Prolog Engine, i.e. each worker can solve a complete branch of the search tree. The set of workers must cooperate sharing tasks to get the program final result. Each worker implements a Prolog abstract machine based on WAM (Warren Abstract Machine [1]). The standard Prolog compilation technique consists in generating WAM-like code whose architecture is stack based. This WAM code can be emulated or translated to machine specific code.

In OR-parallel systems, task exportation consists in giving a branch of the tree to be remotely solved (an OR alternative). A task is thus composed by one or more clauses (alternatives) of one predicate and its execution context. The next untried clause of a predicate is stored in a special data structure called *choice point* [1]. Choice points are stored in a stack called *Local*.

A. General characteristics

The fundamental problem to solve in an OR-parallel logic programming system is the maintenance of multiple bindings for the same variable location in different branches of the search tree [13, 25]. In theory, a new resolvent is generated at each unification of a goal against the head of a clause, by renaming all the variables of the current resolvent. In practice, resolvents are not copied and the same memory locations are used to compute alternative resolvents [25]. As a goal can match to several different clauses and produce different bindings for a variable, a system that allows several clauses to proceed in parallel need a mechanism to avoid conflicting bindings. So, optimization techniques used in sequential implementations, establish that the same variable location holds several single assignment logical variables (bindings) corresponding to alternative unifications of the same goal of a resolvent with different clause heads of a predicate. In sequential Prolog implementations, such a variable location is reset during the backtracking operation, before a new unification is attempted. The variable locations to be reset at backtracking are stored in a stack called *Trail Stack*.

In parallel Prolog implementations, variable locations that may lead to conflicting bindings — these locations are also stored in the Trail Stack — are called conditional variables. Each processor needs to have its own copy of the conditional variables, as processing of each branch can be done separately and simultaneously with other processors. The problem is to mix efficiently sequential Prolog implementations — since most of the computation performed by the processors will be done sequentially — with solutions providing processors with their own copies of the conditional variables. Several solutions have been proposed to cope with this problem. One of the frequently used classifications consider three groups: copying, sharing and recomputation of stacks [25].

1. *copying of stacks*: In this scheme, each worker maintains a complete copy of the stacks in its workspace. An importer worker copies the stacks of the computation state down to the node providing work. The importer will need to restore the stack bindings to the previous state, i.e. when the imported choice-point was created. This way, they keep their own independent environments sets to produce multiple bindings to a variable. Some systems that use this approach are Muse [2] and OPERA [6]. The overhead of copying can be reduced through the *incremental copying of stacks* technique. Incremental copying is based in fact that one idle worker can share a part of the search tree with the exporter. So, both share parts of the stacks. In this technique, the importer backtracks until the last common choice-point before copying the segments of the stacks that are *younger* than the last common choice-point.
2. *sharing of stacks*: workers share the parts of the stacks that they have in common. Since several logical variables may have the same identification, which is a location in a shared stack, special data structures are used to store logical variables as in Aurora [28] OR-system.
3. *recomputation of stacks*: in this technique each worker computes a pre-determined path of the search tree described by an "oracle" allocated by a specialized process called "controller". Delphi [8] is the main example of this approach.

Sharing is still prohibitive for distributed systems due to the high cost of distributed shared memory maintenance. Recomputation is an interesting approach, however it also implies a high management cost. The technique used in PLoSys is *copying of stacks* since it seems to be the best alternative to distributed environments. However incremental copying of stacks is not already used.

Each technique has an associated non-constant-time cost that is incurred at task creation time, at variable access time, or at task switching time. All three operations cannot be performed in constant-time [20]. Therefore in order to reduce the overhead costs, systems need to reduce these three costs to as low as possible. Unless the system employs a static scheduling strategy to execute the programs, task switching is always non-constant-time, and in this case, it becomes one

of the critical parts of the system to be optimized [18]. This is the main reason of our study concerning scheduling in OR parallel systems.

On the other hand, if a parallel Prolog system wants to maintain the *Prolog semantics*, the execution of *side-effects* requires special attention: workers should guarantee that the execution of order-sensitive predicates across the branches of the OR-tree is realized in the correct order. This typically involves delaying the execution as long as the current branch is not leftmost in the OR-tree. PLoSys was designed to deal with this problem [30]. Side-effects had influence in the option for centralized scheduling since it would be less complex to detect the left most branch.

Other important design and implementation decisions in the first version of the PLoSys system can be characterized by the following aspects [32]:

- each processor has only one worker and each worker has a full Prolog engine (figure 1);
- there is one machine (processor) that controls initialization and termination of executions. In addition to a normal worker, this machine holds the scheduler process (centralized scheduling) and all workers must notify the scheduler of state transitions and get tasks;
- each worker has its private data area where WAM stacks are manipulated. When a task is exported, stacks that represent the context execution are copied to the importer environment (copying of stacks);
- since there is no granularity information available, each worker's load is estimated by its number of choice points (indicating the number of pending alternatives). However choice points do not represent the real load, since one worker with two complex choice points can have more pending work than another with three choice points;
- task exportation in PLoSys is done by copying of the WAM stacks and of all non tried alternatives of the oldest choice point from the worker with the highest number of choices points. This decision is based on the heuristic that the oldest choice points (highest in the search tree) are likely to have the greatest granularity. Besides, this decision simplifies memory management since the highest choice point is the closest of the top of the search tree and also of the top of the Local Stack.

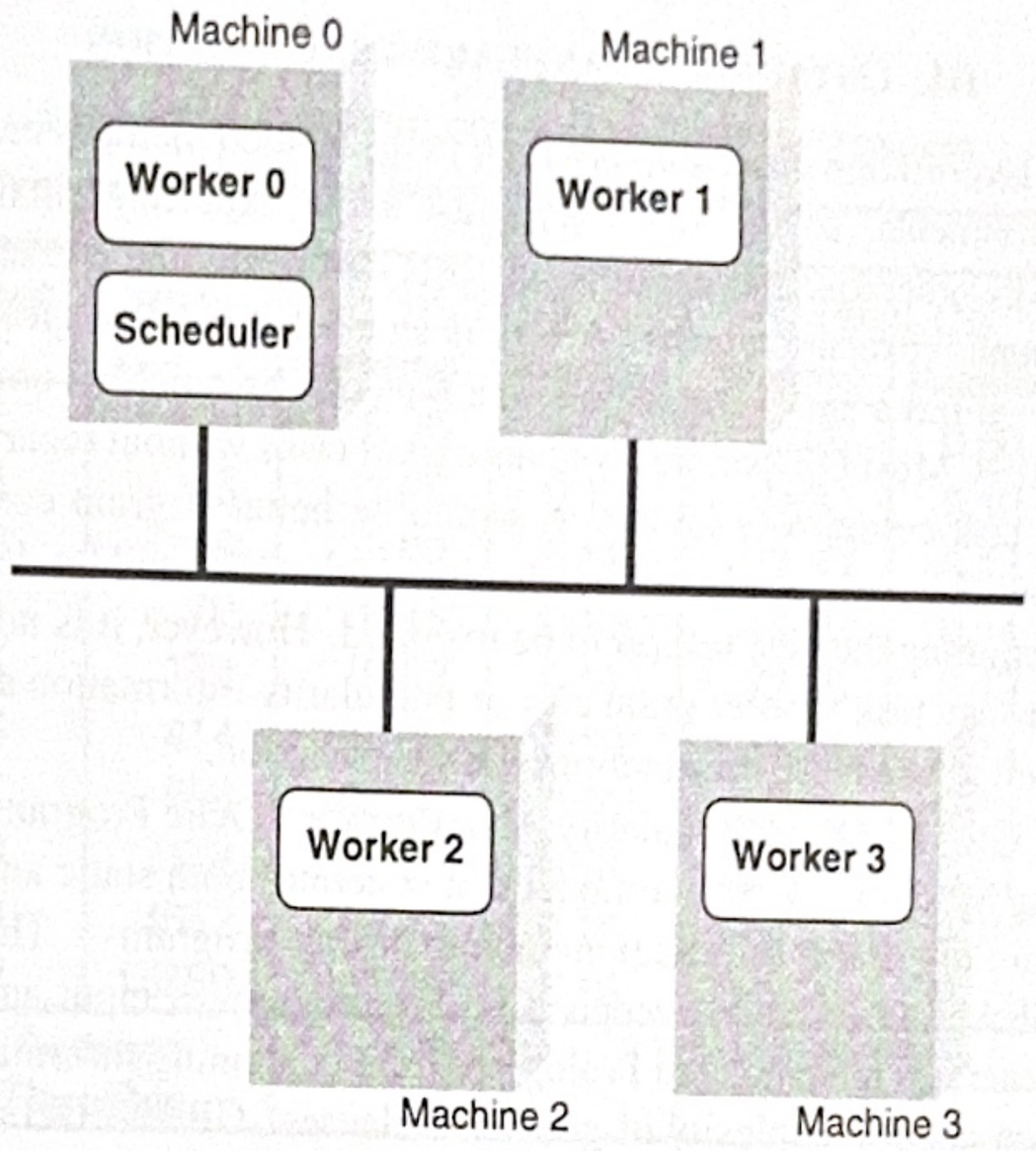

Fig. 1. Worker's organization in PLoSys with four nodes environment

B. Scheduling characteristics

One of the most important elements of a parallel system is the scheduling algorithm. Scheduling [15] aims at deciding the task attribution to processing elements and the execution order of each task. The task scheduler is the entity responsible for deciding what, when and to whom export tasks. All scheduling policies must distribute tasks over the available computational resources improving one or more performance measures. Scheduling is a fundamental aspect in any parallel system since it is closely related to system performance and thus is an important research field in parallel logic programming [18].

Since ideal scheduling is a NP-complete problem, heuristics are frequently used to obtain a good scheduling while minimizing the scheduling costs. In PLoSys two heuristics are used: (a) the oldest choice point is the most complex; and (b) the worker with the greatest number of choice points is the worker with more load.

In PLoSys, a worker can assume three states as illustrated in figure 2: idle, quiet and overloaded. This transaction is done using the number of choice point as load measure. This is a heuristic which is not much precise as will be discussed later.

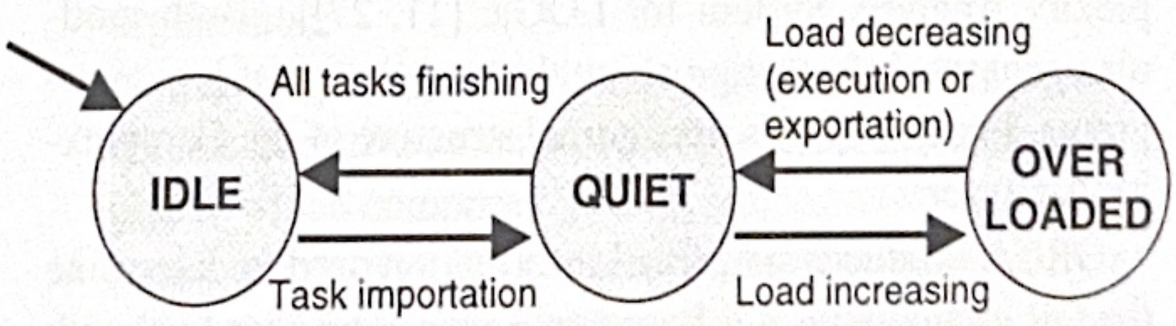

Fig. 2. Worker's state transition in PLoSys

When a worker is in a idle state, the centralized scheduler is notified about that. When the scheduler identifies a pair of *idle* and *overloaded* workers, it sends a message to the idle worker that will try to import a task from the overloaded worker. So, the exportation process will be realized directly between an importer (idle) and an exporter (overloaded).

III. Obtaining granularity information

Exportation costs are very high in distributed memory environments, so the scheduler must avoid exporting small tasks otherwise the exportation costs can overcome the sequential execution costs. A task is small if it has fine grain i.e. it has a small complexity and thus can be quickly computed. Most OR-parallel systems export tasks without regard to task complexity issues. It would be better if grain cost could be evaluated both for detecting worker load and for determining the best task(s) to be exported. However, it is not an easy task to infer grain cost or granularity information as well to use these informations during execution.

GRANLOG (GRanularity ANalyzer for LOGic Programming Systems) [3, 5] is a model that generates both static and partially static informations about Prolog programs. The GRANLOG system accepts a Prolog program as input and generates an annotated Prolog program containing information about complexity of goals and clauses. GRANLOG is composed of three parts: Global Analyzer (GLA), Grain Analyzer (GRA) and Complexity Analyzer (CA).

GLA employs global analysis to determine modes, types and argument sizes for each predicate. Moreover, GLA analyzes accurately the dependencies between clause literals. Based on this analysis, the Grain Analyzer (GRA) module establishes the grains of a clause body, i.e., which are the tasks that can be executed in parallel. The Complexity Analyzer evaluates the cost involved in the execution of the grains (granularity) detected by the Grain Analyzer. In addition, it determines the complexity of the *pending resolvent* for each literal in a clause. The pending resolvent consists of all continuation literals that follow a clause literal [4]. It is used to get the complexity of a search tree branch during the execution of programs (*OR complexity* [4]). During the execution the runtime must use this static information to get the actual OR complexity at a give branch of the Prolog tree.

The Complexity Analyzer is composed by two modules: *ORCA* (OR Complexity Analyzer [4]) and *CASLOG* (Complexity Analysis System for LOGic [11, 27]). Both modules generate information about the complexity of logic programs. Figure 3 shows the internal structure of the Complexity Analyzer.

ORCA produces simplified information used to determine the OR complexity. It is based on Tick's algorithm [34] with some modifications to increase precision as presented in [4]. CASLOG was developed by Lin and Debray and generates highly accurate and complex recurrent expressions that can be used to determine the complexity of each predicate in a program (AND complexity). The information generated by ORCA is called simplified because it does not consider real recursion and gets all information at compile time. In contrast, CASLOG generates expressions which can be evaluated at run time give really precise complexity information.

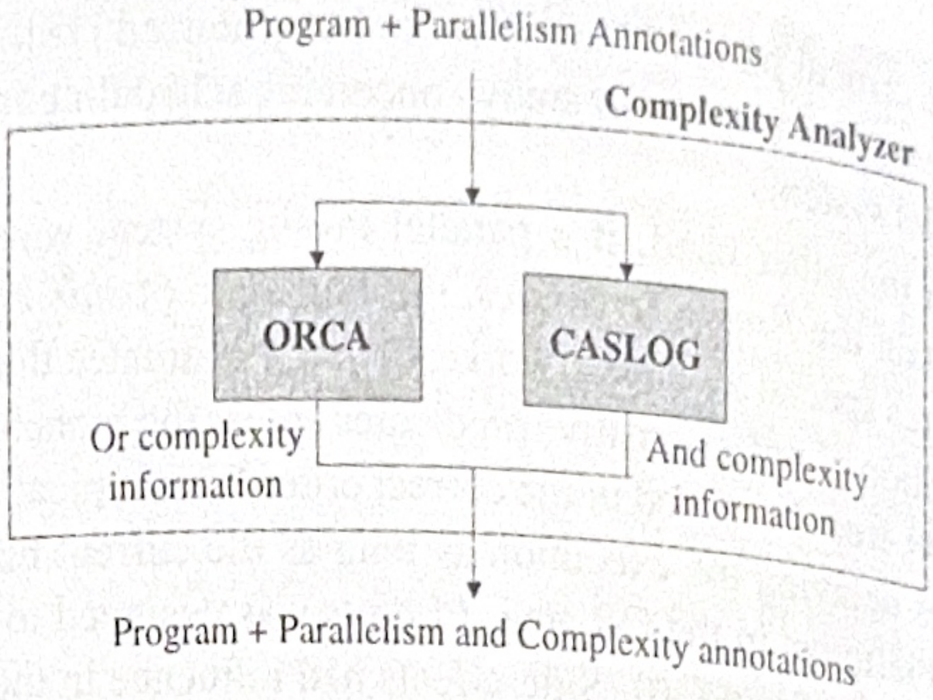

Fig. 3. Complexity Analyzer structure

The Complexity Analyzer includes complexity annotations. We present here a small piece of Prolog code (figure 4) and the corresponding GRANLOG notation generated after its analysis (figure 5). Figure 5 presents the annotations performed in the hanoi predicate. The `granularity` annotation contains the granularity expression. This expression can be used to calculate the computational cost of a grain considering the size of its inputs. The `out_size` annotation contains the relation between the sizes of the input and output grains. The notation used in these expressions is the same as those utilized in CASLOG [11, 27]. Besides that, the Complexity Analyzer annotates the simplified information generated by ORCA. This information includes a constant used to perform the OR complexity analysis (for example, 39 in the second `grain_clause` annotation). In the `grain_clause` annotation, this number represents the complexity of a clause. In the `grain_goal` and `grain_goals` annotations, the number represents the complexity of all next goals in the clause (pending resolvent). Besides that, `grain_proc` presents, after predicate identification, modes and argument measures, the following simplified predicate informations: predicate complexity, number of clauses, a list containing complexity of each clause, and recursive call complexity. At the end, it also presents the corresponding links to `granularity` and `out_size` annotations.

```
hanoi(1,A,B,C,[mv(A,C)]).
hanoi(N,A,B,C,M) :-
  N > 1, N1 is N - 1,
  hanoi(N1,A,C,B,M1), hanoi(N1,B,A,C,M2),
  append(M1,[mv(A,C)],T), append(T,M2,M).
```

Fig. 4. Hanoi predicate – fragment of Prolog code

Clause and goal complexity are expressed in terms of number of resolutions [4]. Resolution express the cost of calling a predicate in a Prolog program. Data in table I present some results of GRANLOG evaluation [3]. Using the expressions

```
:- granularity(e1,$1*exp(2,$1)+exp(2,$1-1)-2).
:- granularity(e2,$1*exp(2,$1)+3*exp(2,$1-1)-2).
:- granularity(e3,$1+1).

:- out_size(r1,[0,0,0,0,1]).
:- out_size(r2,[$1,0,0,0,exp(2,$1)-1]).
:- out_size(r3,[$1,0,0,0,exp(2,$1)]).
:- out_size(r4,[$1,0,$1+1]).
:- out_size(r5,[$1,$2,$1+$2]).
:- out_size(r6,[0,$2,$2]).

:- grain_proc(hanoi/5,[i,i,i,i,o],
   [int,void,void,void,length],
   40,2,[1,39],14,e1,r2)
:- grain_clause(hanoi/5,g1,[i,i,i,i,o],
   [int,void,void,void,length],1,1,r1).
:- grain_clause(hanoi/5,g2,[i,i,i,i,o],
   [int,void,void,void,length],39,e1,r2).
:- grain_goals(hanoi/5,g2_3,[N1,A,B,C,T],
   [i,i,i,i,o],[int,void,void,void,length],
   [int,atom(3),atom(3),atom(3),
   list(?,[struct(2,2, [atom(3)])],19,e2,r3).
:- grain_goal(hanoi/5,g2_3_1,[i,i,i,i,o],
   [int,void,void,void,length],[int,atom(3),
   atom(3),atom(3),list(?,[struct(2,2,
   [atom(3)])])],24,e1,r2).
:- grain_goal(hanoi/5,g2_3_2,[i,i,o],
   [length,void,length]), [list(?,
   [struct(2,2,[atom(3)])]),
   list(1,[struct(2,2,[atom(3)])]),
   list(?,[struct(2,2,[atom(3)])])],19,e3,r4).
:- grain_goal(hanoi/5,g2_4,[i,i,i,i,o],
   [int,void,void,void,length], [int,atom(3),
   atom(3),atom(3), list(?,[struct(2,2,
   [atom(3)])])],5,e1,r2).
:- grain_goal(hanoi/5,g2_5,[i,i,o],[length],
   [list(?,[struct(2,2,[atom(3)])])],0,e3,r5).
hanoi(1,A,B,C,[mv(A,C)]).
hanoi(N,A,B,C,M) :-
   N > 1, N1 is N - 1,
   (hanoi(N1,A,C,B,M1),append(M1,[mv(A,C)],T)) &
   hanoi(N1,B,A,C,M2), append(T,M2,M).
```

Fig. 5. Hanoi predicate – fragment of GRANLOG code

generated by the CASLOG system, the number of resolution is calculated for several input sizes. As presented in figure 5, the complexity expression for hanoi is:

Chanoi = $1 * exp(2,$1) + exp(2,$1-1) - 2

Note that $1 represents the size of the first argument. Then, for each input size, execution time is divided by the expected number of resolutions. With this results it is possible to verify the accuracy of CASLOG and resolution time for each Prolog system. This experiment was done using Sicstus 2.1 and CProlog 1.5 under SunOS system.

ORCA module generates static information considering that a fact has 1 resolution while the complexity of rules is obtained by the sum of the complexities of each goal in the clause plus one call (considers the head call). The complexity of a procedure is given by the sum of the complexities

TABLE I
SEQUENTIAL EXECUTION FOR HANOI AND CORRESPONDING RESOLUTION TIME

HANOI					
SIZE	RES.	SICSTUS		C-PROLOG	
		Time(ms)	Time/Res.	Time(ms)	Time/Res.
1	1	0.00	0.00	0.00	0.00
2	8	1.00	0.12	0.83	0.10
3	26	3.00	0.11	3.33	0.13
4	70	8.45	0.12	10.00	0.14
5	174	18.50	0.11	22.50	0.13
6	414	43.00	0.10	54.17	0.13
7	958	99.50	0.10	125.00	0.13
8	2174	220.00	0.10	279.17	0.13
9	4862	486.00	0.10	618.33	0.13
10	10750	1054.45	0.10	1395.00	0.13
Average Time/Res.		0.10		0.13	

of the clauses belonging to that procedure. The complexity of a recursive call is computed as the complexity of the corresponding procedure by considering each recursive call as having weight 1. Since the actual number of recursive calls is unknown, this number is only an approximation. For example, ORCA will calculate for Hanoi the recursive call complexity 14 and the complete predicate complexity 40.

IV. Applying Granularity Information to PLoSys System

Among static information generated by GRANLOG, three informations about predicates are used in this integrated model:

- number of clauses (Nc): this information indicates the number of clauses which composes the analyzed predicate. Each predicate is composed of a set of one or more clauses with the same name and arity (number of arguments);
- clause complexity (Cc): is an integer number which represents a relative amount of work to execute one specific clause which composes a predicate;
- goal complexity (Cm): each clause is composed by a set of zero or more goals (zero if it is a fact). This value represents the complexity of executing one specific goal inside a clause.

All these informations are obtained from `grain_proc` annotations. The PLoSys-GRANLOG version uses the static annotation for OR parallelism. The static OR simplified annotation gives information about predicates and clauses as fixed values. Obviously, it is an imprecise value since the real complexity can only be evaluated at runtime with the actual data values.

GRANLOG's static informations are helpful but are not sufficient to be used in OR scheduling. At a given step of the execution of a program, there is a set of goals that remain to be executed called the current *resolvent*. It is necessary to calculate dynamically the work available considering the obtained granularity of tasks plus the granularity of its resolvent. Therefore, three other informations are calculated during program execution using these static informations:

- resolvent complexity (ACRPL): an accumulator of local resolvent complexity, i.e., the amount of untried alternatives in the execution tree;
- choice point complexity (CPE): each choice point has this value which represents the complexity of alternative clauses plus accumulated resolvent at that point in the execution tree;
- worker load (RCT): sum of all choice point complexities. This sum means the complexity of all tasks waiting for execution.

The information produced by GRANLOG can be very useful to guide scheduling decisions. The scheduler can select better pieces of work to assign to processors based on the number and grain size of alternatives in a choice-point. The centralized scheduler used by the PLoSys system was modified in order to support GRANLOG information [16].

A. Implementation Details

We tried to minimize the changes into `wamcc` abstract machine [9] used in the PLoSys implementation. Only the instructions related to choice point manipulation was changed. All other functionalities are introduced in compiled WAM code. This means that a parser/compilation strategy must also be added.

A special file is generated for each Prolog program with GRANLOG's static information. This file is a table of granularity informations related to each predicate in the Prolog program extracted from GRANLOG output.

The dynamic informations (ACRPL, CPE and RCT, see above) are calculated at runtime and must be updated continuously during the execution. Each choice point complexity (CPE) is associated to one specific choice point stack structure. We use a specific structure in our integration module to manipulate these informations, avoiding changes in the WAM data structures. This design decision aims at simplifying granularity information management into other PLoSys versions under development in our research group.

The compiled WAM code i.e., the application file is also modified to allow a correct evaluation of the workers' load. These changes are some variable settings to associate correctly the predicate and clause being currently executed with their complexity information stored in their granularity information table.

Static and dynamic informations are included into the PLoSys system in the following way:

- clause complexity (Cc), goal complexity (Cm) and number of clauses (Nc): all static informations come from the GRANLOG output file and are stored in a table to be used during execution;
- resolvent complexity (ACRPL): it is a variable updated at each predicate call to store the complexity of the resolvent;
- choice point complexity (CPE): this value must be associated to each created choice point. Two implementations are possible: include a field in the choice point structure to store CPE or have another structure that maps the complexity of each choice point. The last alternative was used to minimize WAM changes;
- worker load (RCT): it is a variable updated by the choice point operations (creation, updating and deletion). This avoids summing all CPEs to know the worker load during a scheduling decision.

To summarize, there are two main situations where the PLoSys Prolog abstract machine must be changed: (1) before a predicate call (ACRPL updating), and (2) in choice point operations (CPE and RCT updating). The PLoSys modules responsible for parallelism control were also modified. The main changes concern the treatment of the workers' load. The complexity stored in RCT is considered the load indication in the integration model. This load measure was also included in the scheduler module. The only change in the scheduling policy was the use of RCT as worker load value and new threshold value to classify a worker as overloaded. Some computation must also be done to keep the CPE of each choice point and the global RCT updated when an importation or an exportation occurs.

B. Experimental Results

Before presenting our initial experimental results, it is important to discuss a little about dynamic granularity informations used for the implementation. They are calculated using only simple mathematic manipulations as adding and subtracting of single variables. Even simple, there is a time spent to calculate and manipulate these values. However, this is the cost paid to get a less imprecise granularity information.

The methodology used was simple and it could be implemented with minor changes in parallel Prolog abstract machine. The first experiments indicate that the use of granularity minimizes the communications with the scheduler and the "false overloaded state detection". By false overloaded state detection we mean that the centralized scheduler can be notified that some worker *Wx* is overloaded due to current number of choice points while it actually has few tasks to be executed. So, the scheduler can inform an idle worker *Wy* that it can import work from this worker *Wx*. In this case two

things can happen: (1) the *Wy* worker imports a task and *Wx* will soon become idle, or (2) the *Wy* worker tries to import but the importation is canceled by *Wx* because *Wx* is no more in the overloaded state. If granularity information is used both cases can be minimized. In our experiments, no exportation was canceled using granularity information. Figure 6 illustrates these characteristics. The use of granularity informations decreased execution time even in a single processor execution. This behavior is due to scheduling implementation. All workers, including the one located in scheduler processor, communicates with scheduler thread using message passing. Each worker has a *spy* thread which send load information to scheduler, even if there is a single worker.

queens 8	*with granularity*			*without granularity*		
	1	2	3	1	2	3
average	34,69	29,68	23,70	68,09	41,33	51,00
std dev	4,43	4,35	3,45	13,15	20,19	12,22

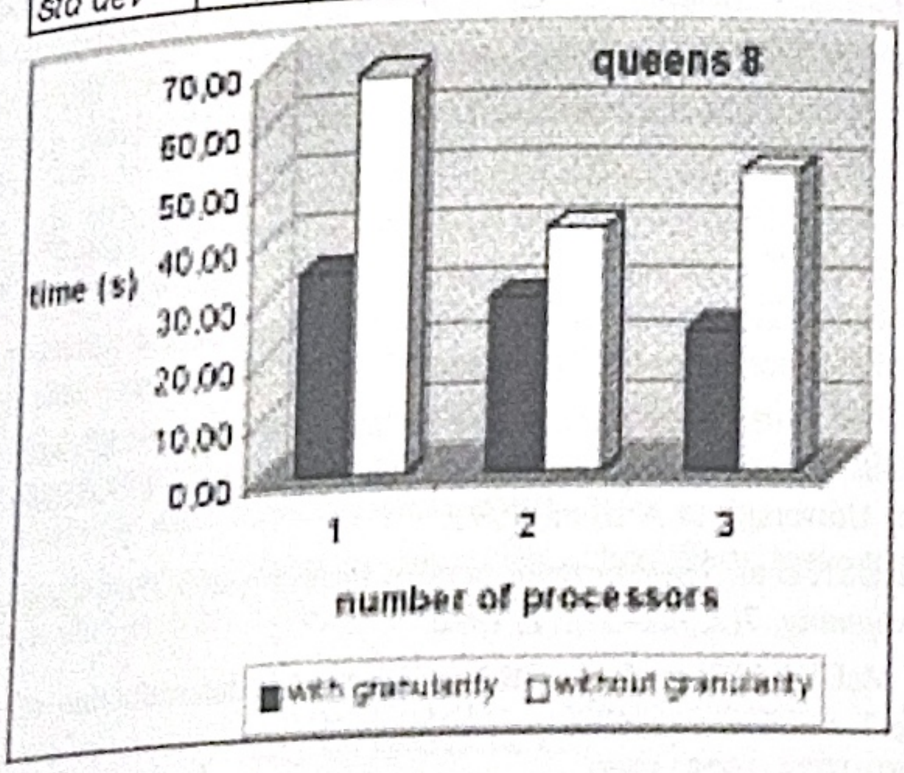

Fig. 6. PLoSys execution time for queens 8 benchmark

Figure 6 presents our initial evaluation of scheduling using granularity. Queens problem was chosen as our benchmark. Queens is a classical problem which aims to place *n* queens in chessboard *n* x *n* in such a way that none queen attack the other. For our experiments, it was executed at least 10 times for placement of eight queens using one, two and three workstations. The execution time (*average*) was presented in seconds in the graph and in the data table which also have the standard deviation (*std dev*).

V. Related Work

Granularity analysis can be done using different methodologies. Normally, it depends on programming paradigm being used. The first studies on complexity and granularity analysis were done for imperative languages [26, 29]. Later, granularity analysis studies about functional [23] and logic [11, 24, 27, 33, 34, 35] programming were published. Granularity information can be computed at compile time (low precision, low execution cost) [34] or at runtime (high precision, high cost). It is also possible to use an hybrid approach: to get some informations at compile time to be evaluated at runtime as proposed in the CASLOG system [27]. The latter approach would be the best combination since it infers the most informations at compile time without losing much precision due to runtime informations used. However, GRANLOG also presents fully static informations to be used when execution cost must be minimized. So, GRANLOG system [3, 5] offers two approachs: it can collect either static or hybrid informations. The underlying philosophy of GRANLOG is to provide as much information as possible so that the user can decide what and how to use it.

The use of granularity informations to help parallelization is not a new concept. However, there are few implementations in real systems. For example, DSLP [10] is a distributed AND/OR scheduling that uses granularity information to make decisions. DSLP currently has only a simulator prototype of AND scheduling. Other classical approaches were investigated by Debray, Hermenegildo et al [12, 17]. GRANLOG's granularity informations were already incorporated to help scheduling of an AND/OR parallel system for shared memory machines. Static informations were used in the Andorra-I system and good results were obtained [14].

VI. Conclusion

Granularity informations obtained at compile time can be really helpfull to guide scheduling decisions. GRANLOG is an example of granularity analyser. Using its informations, PLoSys OR scheduler was improved to use a more precise heuristic for worker load. The approach presented in this paper can be applied to any kind of scheduling policy, since no assumption is made about how scheduler(s) interact with workers to evaluate worker load. So, besides describing a specific implementation experience, it also presents a methodology to dynamic load evaluation using static granularity information.

Initial evaluation indicates that our method is worthwhile. Several extensions are possible to our current implementation. As a future work, several benchmarks will be submitted to PLoSys and the real impact of granularity information will be evaluated. Using these results the algorithm could be improved.

Acknowledgments

We would like to acknowledge Inês de Castro Dutra, Cristiano Costa, Ana Paula Centeno, Luís Fernando Pias de Castro and Silvana Campos de Azevedo for their contribution. This work has been partially supported by the CNPq/ProTem-CC project Appelo (1995-1998) and by funds granted to Instituto de Informática/UFRGS .

REFERENCES

[1] H. AÏT-KACI. *Warren's Abstract Machine — A Tutorial Reconstruction*. MIT Press, 1991.

[2] K. A. M. ALI and R. KARLSSON. The Muse Or-parallel Prolog Model and its Performance. In *Proceedings of the North American Conference on Logic Programming*, pages 757–776. MIT Press, October 1990.

[3] J. L. V. BARBOSA. Granlog: Um modelo para análise automática de granulosidade na programação em lógica. Dissertação de mestrado, Universidade Federal do Rio Grande do Sul, 1996.

[4] J. L. V. BARBOSA and C. F. R. GEYER. Análise de complexidade na programação em lógica: Taxonomia, modelo granlog e análise ou. *XXIII Conferência Latino Americana de Informática/V Encontro Chileno de Computação da Sociedade Chilena de Computação (CLEI - PANEL'97)*, 1997. ValParaíso, Chile.

[5] J. L. V. BARBOSA, O. WERNER, and C. F. R. GEYER. Automatic granularity analysis in logic programming. In *TENTH LOGIC PROGRAMMING WORKSHOP - WLP94*, ZURICH: INSTITUT FOR INFORMATIK DER UNIVERSITE ZURICH, Sept. 1994.

[6] J. BRIAT, M. FAVRE, C. F. R. GEYER, and J. CHASSIN. Scheduling of or-parallel prolog on a scalable, reconfigurable, distributed-memory multiprocessor. In *PARLE91*, 1991.

[7] L. F. P. CASTRO, V. SANTOS COSTA, C. F. R. GEYER, F. SILVA, P. K. VARGAS, and M. E. CORREIA. DAOS – Scalable And-Or Parallelism. In *EUROPAR '99*, pages 899–98, Aug 31 – Sept 3 1999. Toulouse, France.

[8] W. F. CLOCKSIN. Principles of the DelPhi parallel inference machine. *Computer Journal*, 30(5):386–392, 1987.

[9] P. CODOGNET and D. DIAZ. Wamcc: Compiling prolog to c. In L. Sterling, editor, *Proceedings of the Twelfth International Conference on Logic Programming*, pages 317–331, Tokyo, Japan, June 1995. MIT Press.

[10] C. A. COSTA and C. F. R. GEYER. Uma Proposta de Escalonamento Distribuiido para Exploração de Paralelismo na Programação em Lógica. In *X Simpüsio Brasileiro de Arquitetura de Computadores e Processamento de Alto Desempenho*, 1998. Búzios, RJ.

[11] S. K. DEBRAY and N. LIN. Cost analysis of logic programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, Nov. 1993.

[12] S. K. DEBRAY, N.-W. LIN, and M. HERMENEGILDO. Task granularity analysis in logic programs. In *Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation*, pages 174–188. ACM Press, 1990.

[13] I. C. DUTRA. *Distributing And- and Or-Work in the Andorra-I Parallel Logic Programming System*. PhD thesis, University of Bristol, Department of Computer Science, February 1995. PhD thesis.

[14] I. C. DUTRA, V. SANTOS COSTA, J. L. V. BARBOSA, and C. F. R. GEYER. Using Compile-Time Granularity Information to Support Dynamic Work Distribution in Parallel Logic Programming Systems. In *X Simpósio Brasileiro de Arquitetura de Computadores, SBAC-PAD*, October 1999.

[15] J. EL-REWINI, T. G. Lewis, and H. H. ALI. *Task Scheduling in Parallel and Distributed Systems*. Prentice-Hall, 1994.

[16] D. N. FERRARI, P. K. VARGAS, J. L. V. BARBOSA, and C. F. R. GEYER. Modelo de Integração Plosys-Granlog: Aplicação da Análise de Granulosidade na Exploração do Paralelismo OU. In *XXV Conferência Latino Americana de Informática/V Encontro Chileno de Computação da Sociedade Chilena de Computação (CLEI - PANEL'99)*, September 1999. Assuncion, Paraguay.

[17] P. L. GARCIA, M. HERMENEGILDO, and S. K. DEBRAY. Towards granularity based control of parallelism im logic programs. In *International Symposium on Parallel Computation*, Linz, Austria, Sept. 1994.

[18] C. GEYER, P. K. VARGAS, and I. C. DUTRA. Parallelism in logic programming. *International School on Advanced Algorithmic Techniques for Parallel Computation with Applications - CIMPA'99*, page 35, September 1999.

[19] G. GUPTA, M. V. HERMENEGILDO, E. PONTELLI, and V. SANTOS COSTA. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *Proceedings of the Eleventh International Conference on Logic Programming*, Italy, June 1994.

[20] G. GUPTA and B. JAYARAMAN. On Criteria for Or-parallel Execution of Logic Programs. In *Proceedings of the 1990 North American Conference on Logic Programming*, pages 737–756. MIT Press, October 1990.

[21] G. GUPTA, E. PONTELLI, and M. V. HERMENEGILDO. &ACE: A High Performance Parallel Prolog System. In *Proceedings of the First International Symposium on Parallel Symbolic Computation, PASCO'94*, 1994.

[22] M. V. HERMENEGILDO and K. J. GREENE. The &-prolog system: Exploiting independent and-parallelism. *New Generation Computing*, 9(3,4):233–256, 1991.

[23] L. HUELSBERGEN, J. R. LARUS, and A. AIKEN. Using run-time list sizes to guide parallel thread creation. In *Proc. ACM Conf. on Lisp and Functional Programming*, June 1994.

[24] S. KAPLAN. Algorithm complexity of logic programs. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 780–793. MIT Press, 1990.

[25] J. C. KERGOMMEAUX and P. CODOGNET. Parallel logic programming systems. *Computing Surveys*, 26(3):295–336, September 1994.

[26] B. KRUATRACHUE and T. LEWIS. Grain size determination for parallel processing. *IEEE Software*, 5(1):23–32, January 1988.

[27] N. LIN. Automatic Complexity Analysis of Logic Programs. PhD Thesis, Department of Computer Science, University of Arizona, Tucson: University of Arizona, 1993.

[28] E. LUSK et al. The Aurora or-parallel Prolog system. *New Generation Computing*, 7(2,3):243–271, 1990.

[29] C. McCREARY and H. GILL. Automatic determination of grain size for efficient parallel processing. *Communications of the ACM*, 32(9):1073–1078, 1989.

[30] E. MOREL, J. BRIAT, and J. CHASSIN DE KERGOMMEAUX. Cuts and side-effects in distributed-memory or-parallel prolog. *Parallel Computing*, 22:1883–1896, 1997.

[31] E. MOREL, J. BRIAT, J. C. D. KERGOMMEAUX, and C. GEYER. Side-Effects in PloSys OR-parallel Prolog on Distributed Memory Machines. In *JICSLP'96 Post-Conference Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages, Bonn, Germany*, September 1996.

[32] E. MOREL, J. BRIAT, J. C. d. KERGOMMEAUX, and C. F. R. GEYER. *Parallelism and Implementation of Logic and Constraint Logic Programming*, chapter Side-effects in PloSys OR-parallel Prolog on Distributed Memory Machines. Nova Science, Inc., 1999.

[33] K. SHEN, V. SANTOS COSTA, and A. KING. A New Metric for Controlling Granularity for Parallel Execution. In *ILPS97 Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages*, Port Jefferson, USA, Outubro 1997.

[34] E. TICK. Compile Time Granularity Analysis for Parallel Logic Programming Systems. *New Generation Computing*, 7(2,3):325–337, 1990.

[35] E. TICK and X. ZHONG. A compile-time granularity analysis algorithm and its performance evaluation. *New Generation Computing*, 11(3-4):271–295, 1993.