Granularity Control for Distributed Execution of Logic Programs

George Xirogiannis
Dep. of Computing & Elec. Engineering
Heriot-Watt University
Edinburgh, EH14 4AS, Scotland, UK
G.Xirogiannis@hw.ac.uk

Abstract

Distributed execution of logic programs requires a match of granularity between a program and the distributed multi-processor it runs on to exploit its potential for performance fully. This paper presents methods to control the granularity of tasks on distributed heterogeneous processors effectively. It considers the characteristics of such platforms and relates the amount of local computation with the significant communication overheads by introducing the notion of a collection of parallel tasks. The experimental results indicate that the proposed controls can model all kinds of predicates (recursive, mutually recursive etc.) satisfactorily and improve the performance of various forms of parallelism (AND, OR, combinations).

1 Introduction

Granularity analysis has been proposed as a method to avoid exploiting parallelism in a fine-grained way by coalescing tasks into larger grains to be executed on a single processor. In general processes that are too coarse-grained for a multi-processor system unnecessarily limit its ability to exploit parallelism, while fine-grained processes introduce excessive communication overheads. This paper presents a scheme for controlling the grain size of tasks (goals) of Prolog programs on a process-based parallel logic programming system running in a distributed manner on the nodes of a virtual multiprocessor. PAN [22] runs on a LAN of workstations with each Prolog engine running on a different workstation. Engines employ PVM to communicate with each other either synchronously or asynchronously using extra message passing primitives added to SICStus Prolog. PAN is able to exploit various forms of parallelism (AND, OR, combinations). Some of its particular merits are its robustness, ease of use and its ability to exploit highly available hardware. The communication cost of the distributed platform is significant dictating that any further task analysis should add little execution overhead, otherwise performance could degrade rapidly.

The following sections discuss the design of the controls used in PAN. Section 2 discusses recent granularity analysis models and indicates existing pitfalls. Section 3 introduces the controls used in PAN and section 4 presents several experimental results. Finally section 5 summarizes the proposed techniques.

2 Relevant Research

Recent proposals for granularity analysis in logic programming have focused on measuring the complexity of a process (goal) but have paid little attention to how to use such information. In order to make granularity decisions it is necessary to estimate the time spent initiating and conducting communication and the time spent performing useful computation. The characteristics of the platform used in this project dictate that the communication overhead must not be disregarded otherwise theoretical frameworks for controlling the granularity of goals may fail to improve performance. Early proposals like [26] and [15] investigated the automatic inference of the complexity of logic programs but only under several restrictive assumptions that rule out many interesting programs. For example these systems were unable to deal with recursive predicates. The complexity of functions to control parallelism in the parallel evaluation of functional programs has been investigated in [19] as well. Böckle in [4] presented detailed techniques to exploit fine-grained parallelism. But he focused on parallelism where the items processed are machine instructions offered by superscalar or VLIW architectures rather than coarse-grained distributed platforms like PAN.

Tick [24] describes a heuristic algorithm for estimating granularity using *weights* to quantify the grain of tasks. However this analysis is crude and does not model recursive predicates satisfactorily. Performance

measurements showed that this scheme does not perform significantly better than conventional distribution methods. This result is attributed to a combination of factors: sensitivity to system overheads, low cost of spawning a goal and the increase of synchronization caused by the method. Moreover this scheme can not model OR-parallelism efficiently. This scheme was designed for shared-memory multiprocessors and would require significant re-engineering in order to be used in PAN. The algorithm in [25], which is designed for committed-choice languages, revises the previous model by introducing iteration parameters to handle recursive predicates. Even if heuristics and the iteration parameters estimate complexity at compile-time, run-time execution may diverge from these estimates. Another problem associated with [25], is how to incorporate further information to derive more precise estimations while keeping analysis costs low.

Debray *et al* [10] explain how to deal with recursive predicates by deriving complexity functions for predicates at compile-time to provide a better approximation of the *weight*. Once the size of the data is known at run-time these functions can be evaluated. The size of the data is checked against a threshold to determine whether or not the goal should be evaluated in parallel. Experimental results show that this model can improve performance. However in some cases the model did not provide the expected results and decreased speed mainly because too much information is processed at run-time. This model does not take into consideration some important factors. The maintenance of grain size information and tests add a certain execution overhead which should be somehow included in time complexity estimation. This model disregards the fact that some predicates may fail or satisfy the granularity tests at compile-time and thus no run-time overhead is associated with them. The granularity decisions for OR-parallel tasks do not depend only on time complexity properties. Other characteristics like the number of clauses should be considered as well to make analysis more accurate. The basic idea of Debray's analysis [10] is close to PAN's needs because it considers the cost of creating parallel tasks. Garcia *et al* [12] refines most of the techniques used by Debray but still relies on the basic granularity control principle. The performance of this model will be compared with the performance of the controls used in PAN.

King and Soper [16] proposed a different technique for controlling the granularity of tasks at compile-time only. They point out that in some circumstances the overhead of thresholding can cause a slow-down. Their idea is to coalesce processes together if the complexity of their communication dominates the complexity of the computation on all sizes of possible data. This model does not add any run-time overhead. However it may fail to exploit parallelism in cases where the time required by a goal to communicate dominates its computation time because the model does not have the notion of a collection of parallel tasks and it does not quantify efficiently what useful computation can be performed during inter-engine communication. It analyzes goals one by one and does not consider parallel execution of a goal in relation to the execution of other goals. OR-parallelism is not supported as well.

The benefit of automatic grain size control has not yet been used in distributed implementations. Recent distributed Prolog systems like Delta-Prolog [8], CS-Prolog [11], PMS-Prolog [27] and Multi-Prolog [5] still do not accommodate a mechanism for determining when distributed execution improves performance. Granularity control mechanisms could also be used in parallel systems running on shared-memory multiprocessors like NUA-Prolog [18], Andorra-I [7], PEPSys [3], ANDOR-II [21], Aurora [17], Muse [1] to improve performance further. The absence of use of such an analyzer may result in unnecessary communication between engines that degrades performance and reduces the attractiveness of these platforms to programmers. The PDP system [2] controls the grain size of parallel tasks based on Debray's model. Its results will be compared with the results obtain by the techniques used in PAN.

3 Controlling Granularity in PAN

The proposed model concentrates mainly on how to use granularity information effectively to improve the performance of the distributed platform. Most of the analysis is performed at compile-time, using only few run-time tests to increase the accuracy of granularity control without imposing significant run-time overheads. The controls do not require any user declarations or expertise. We assume that an appropriate program analyzer [28] has determined which goals are candidates for parallel evaluation. We also assume that there is an automated mechanism [9] to estimate the time complexity of a goal at compile-time and estimate the time required to process that goal. In cases where the time estimation depends on input values, a time function is generated.

3.1 Basic Controls

Consider a clause C :- B_1, B_2,.....,B_n. Assume that program analysis has determined that a set of

B_i (written as G_i for convenience) are candidates for AND-parallel processing. Let these goals be $G \equiv G_1, G_2, ..., G_k$. This set of goals obeys sequentiality constraints with the rest of the body goals. Let the extra time required to process a goal G_i on a remote engine be $T_{lat}(G_i) = T_{com}(G_i) + T_{sched}(G_i)$. It represents the communication overhead T_{com} and any scheduling cost T_{sched}. Let T(G') be the time required to process $G' \equiv G_1, ..., G_{i-1}, G_{i+1}, ..., G_k$ sequentially.

Basic Granularity Control for AND-parallelism: *An AND-goal G_i should be executed in parallel with the rest* $G' \equiv G_1,...,G_{i-1},G_{i+1},...,G_k$ *if* $T(G') \geq T_{lat}(G_i)$.

The control qualifies even if the amount of computation of the remote task G_i is less than $T_{lat}(G_i)$ in contrast to the proposals of Debray and Garcia. Intuitively, the basic granularity control for AND-parallelism will improve performance because:

$$T_{parallel}(G) = \max\{T(G'), T_{lat}(G_i) + T(G_i)\}$$
$$\leq \max\{T(G'), T(G') + T(G_i)\}$$
$$\leq T(G') + T(G_i) \leq T_{sequential}(G)$$

Consider now a predicate P with clauses $C_1, C_2, ..., C_n$. Assume that a program analyzer has determined that P can explore its clauses in OR-parallel order. Let T(P) be a time estimation of P for a single solution. Let $T_{lat}(C_i) = T_{com}(C_i) + T_{sched}(C_i)$ be the extra cost for processing P over the head of clause C_i on a remote engine. Also let $C' \equiv C_1, ..., C_{i-1}, C_{i+1}, ..., C_n$. The use of a predicate level directive dictates that either all clauses are candidates for OR-parallel execution or none.

Basic Granularity Control for OR-parallelism: *The clauses of P should be explored in OR-parallel if* $\frac{(n-1)}{n} T(P) \geq max\{T_{lat}(C_i), i = 1...n\}$.

In order to process the clauses of P in parallel the extra time required to process any C_i on a remote engine should be less than the time required to process C' locally. Ideally this should translate to $T(C') \geq \max\{T_{lat}(C_i),\ i=1...n\}$, where T(C') is the time required to process C' sequentially. However, mainly due to practical restrictions imposed by the program analyzer which make the estimation of the time complexity of C' a non-trivial task, this condition can apply only to non-recursive predicates. To model satisfactorily recursive and mutual recursive predicates the current mechanism assumes that all clauses of a predicate equally contribute to its time complexity. It assumes that $T(C_i) \simeq T(P)/n \Leftrightarrow T(C') \simeq \frac{(n-1)}{n} T(P)$. and imposes the condition $\frac{(n-1)}{n} T(P) \geq \max\{T_{lat}(C_i), i = 1...n\}$. Garcia suggests using *follow sets* from the theory of context free grammars to address the problem of estimating the complexity of C'. Intuitively this basic control will improve performance because

$$T_{parallel}(P) = \max\{\frac{(n-1)}{n} T(P), \max\{T_{lat}(C_j)\} + T(C_i)\}$$
$$\leq \max\{\frac{(n-1)}{n} T(P), \frac{(n-1)}{n} T(P) + T(C_i)\}$$
$$\leq \frac{(n-1)}{n} T(P) + T(C_i) \leq \frac{(n-1)}{n} T(P) + \frac{1}{n} T(P)$$
$$\leq T_{sequential}(P)$$

The controls are particularly suitable for heterogeneous distributed platforms like PAN where communication costs are considerable and local engines are effective. The basic controls propose new criteria to control granularity and relate communication costs with the amount of useful computation that can be performed locally by a set (collection) of goals.

3.2 Further Controls

In order to improve granularity in PAN, not to generate considerable run-time overheads and to avoid problems discussed in section 2, the basic controls are extended with the following criteria which try to make compile-time granularity controls more accurate and reduce run-time overheads.

1) Granularity control becomes more accurate when $T_{com}(G_i)$ is estimated for each goal G_i.
2) The time complexity of a goal includes possible size-checking overheads $T_{sc}(G_i)$ and granularity control is adjusted by setting $T_{lat}(G_i) = T_{com}(G_i) + T_{sched}(G_i) + T_{sc}(G_i)$.
3) If time complexity depends on the size of the input data, then the analyzer generates "cheap" (in terms of time) tests to be checked at run-time. These tests check for the minimum size of input data for which parallel execution will improve performance.
4) Unfolding partially loop tests and testing the term sizes only to the point at which the granularity threshold is reached can reduce overheads.
5) If the minimum size of input data for a goal is "too large" to occur in practice then this goal is not considered worth parallelizing, hence no run-time size-check is added.
6) If analysis detects that goals satisfy the granularity tests at compile-time, then no run-time test is added. This is the case for non-recursive predicates or for very complex recursive predicates where even input data of very small sizes guarantees speed-up.

7) It is more efficient to let the scheduler [29] perform the run-time tests rather than include them as part of the program code as suggested by [12]. That is because if there are no available engines (information only know to the scheduler) then there is no reason to perform any granularity test.
8) Stricter granularity controls are imposed on slower engines. Let W_n be a "weight" of the processing capabilities of some engine n, then $T_{lat}(G_i) = T_{com}(G_i) + W_n * T_{sched}(G_i) + W_n * T_{sc}(G_i)$.

The following heuristics (not considered by [10], [24], [26], [15] and [19]) may improve performance further. They are based on the observations that the time required to process a time consuming goal might get significantly larger by remote computation.

- Given two goals for parallel execution that both satisfy the granularity control criteria, the one with the smaller sum of computation and communication time should be executed on the remote processor.
- Given more than two parallel goals, performance may be improved by distributing them in order of non-increasing computation and communication cost.

Granularity analysis is independent of the number of available engines. If there are more tasks than currently available engines a sophisticated scheduler may have to coalesce tasks further to fit the number of engines while if there are more engines than tasks, some engines may stay idle in order to guarantee speed-up even if that may limit exploitable parallelism. In the case of OR-parallelism the combination of the following tests might control granularity better and improve performance further. These tests also were not considered by some models described in section 2. They can be performed at compile-time and the information can be retrieved at run-time.

- The clauses of a predicate should be considered for exploitation in OR-parallel if there are more than k of them (the integer k is determined by the implementation).
- If a clause contains a minimum number (determined by the implementation) of user-defined sub-goals then it should be considered for exploitation in OR-parallel with other clauses.

3.3 Comparisons

The proposed controls are able to overcome many of the pitfalls and problems of other methods as described in section 2. The controls used in PAN are able to relate granularity information effectively with the communication cost which is significant in distributed heterogeneous platforms like PAN. In comparison to other models he proposed controls

- Relate more efficiently the amount of local computation with the communication cost using the notion of a collection of tasks.
- Are sensitive to communication overheads that affect the performance of distributed systems, hence they are more suitable for platforms like PAN.
- Can model satisfactorily different forms of parallelism (AND, OR, combinations) and introduce new explicit controls for OR-parallel execution.
- Control the maintenance and testing of size information better.
- Are able to make certain granularity decisions at compile-time only, to reduce run-time overheads.
- Assist run-time schedulers to limit certain run-time overheads and distribute tasks more effectively.

However the controls maintain a conservatism like most of the techniques presented in section 2, mainly because the analysis is performed at compile-time with few (if any) run-time tests only. The fact that it does not add unnecessary run-time overheads proves beneficial for distributed platforms which are sensitive to run-time communication overheads. More run-time tests would make the controls more accurate and more effective but could degrade performance significantly.

4 Experimental Results

A heterogeneous distributed architecture like PAN provides a suitable platform to determine if the proposed controls adapt well to the changing needs of a LAN-based multiprocessor. Large input sizes have been used on purpose in most benchmarks to provide long running non-trivial problems to push the controls and the platform to their limits. The significant communication overhead in PAN generates large granularity thresholds; therefore, only large input sizes can be used in practice to illustrate the performance of the proposed controls. In contrast the models described in section 2 were tested mainly on shared-memory multiprocessors with low communication overheads, small input sizes and small granularity thresholds.

The numbers in the following tables represent the performance improvement (PI) due to the use of the controls. Given a program, an input goal and a platform configuration, let T_{WC} be the time required to process that goal using the proposed granularity controls and T_{NC} the time required to process the same goal without the proposed granularity controls.

Then the performance improvement (PI) due to the use of the controls is calculated using the expression $\mathrm{PI} = \frac{T_{NC} \Leftrightarrow T_{WC}}{T_{NC}} * 100\%$. The programs were run under PAN using SICStus Prolog 3.5 on a variety of SUN, DEC and IRIX workstations. Each PI number corresponds to the average value of the best three runs.

It is not crucial to be precise in determining the best grain size information for a problem as illustrated in figure 1. There is a reasonable amount of leeway in how accurate this information has to be mainly for the following reasons. The estimation of time complexity has a certain inaccuracy itself. Communication costs may vary at run-time, because of an transient network load. There is a trade off between accuracy and small run-time overheads. Complex grain information requires more time to be processed and in several cases the benefit of accurate analysis cannot outweigh the time required to perform such tests. However the amount of leeway detected suggests that granularity inference can usefully be performed by a compiler despite a certain inaccuracy.

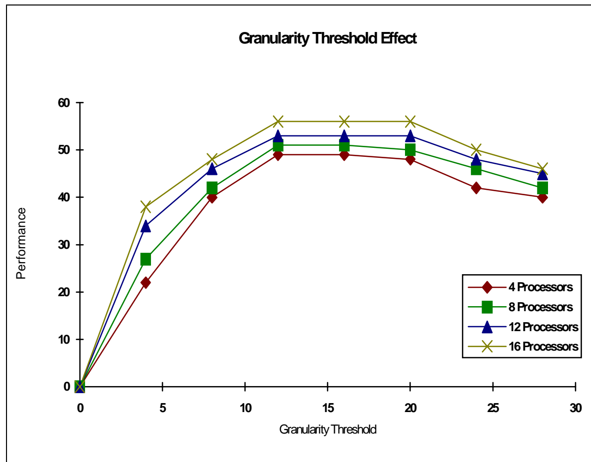

Figure 1: Performance Improvement vs. Task Granularity for QuickSort(3000)

4.1 AND-parallel Execution

Consider the QuickSort program (as presented in [20] page 56) and table 1.

Eng.	List Input Size			
	750	*1000*	*2000*	*3000*
4	5.341%	40.733%	45.709%	49.272%
8	34.043%	43.033%	46.510%	50.613%
12	46.765%	47.517%	53.038%	52.317%
16	38.000%	48.669%	53.623%	55.996%

Table 1: Granularity Control for Quick Sort

The effect of the proposed controls increases as the size of the input list and the number of engines participating in a PAN session increases indicating that the controls adapt well to the characteristics of the platform and the nature of the benchmark ranging up to 56%. The proposed controls cope adequately with large test sizes and impose small run-time overheads as well. Consider that when the size of the input list increases (2000 or 3000 elements), run-time checks become complicated. However the proposed controls reduce these tests by allowing the run-time task schedulers to impose granularity constraints only when there are available engines to process tasks in parallel. Debray's model provides a speed up of 3% under the ROLOG [14] system and a speedup of 16.2% under the &-Prolog [13] system for this benchmark. Under the PDP system performance does not improve at all when three and fifteen processors are used. Performance improves by 17.8% only when 8 processors are employed to process quicksort(700). Garcia's [12] model improves the performance of quicksort(1000) by 21% running on a hierarchical[1] implementation of &-Prolog with 4 processors.

The Perfect Numbers (as presented in [22]) benchmark is shown in table 2. This benchmark also illustrates that the proposed controls impose small run-time overheads. In contrast to Quick Sort only in the last few recursions do the input sizes become small enough to fail the granularity tests. However, as the table shows the proposed controls cope with such cases effectively, improving performance by 24.5%.

Eng.	Integer Input Size			
	75	*100*	*300*	*500*
4	17.018%	21.502%	23.650%	24.528%
8	6.526%	9.553%	12.234%	15.186%
12	8.213%	13.208%	15.530%	17.151%
16	14.074%	20.067%	20.383%	21.641%

Table 2: Granularity Control for Perfect Numbers

The Fibonacci Numbers benchmark (as presented in [6] is shown in table 3. similar to Perfect Numbers. However the controls improve performance more than the previous example mainly because the input sizes for Fibonacci Numbers are closer to the grain size threshold. Debray's model provides a performance improvement of 27.3% under the ROLOG system and 29.2% under the &-Prolog system. This is the only

[1] Essentially this is an &-Prolog implementation with arbitrary overheads added to task creation

benchmark with consistent results under both parallel systems. Garcia's model improves the performance of fib(19) by 24% running on a hierarchical implementation of &-Prolog with 4 processors.

Eng.	Integer Input Size		
	10	*15*	*20*
4	1.28%	7.23%	15.169%
8	7.011%	12.814%	21.81%
12	11.382%	17.972%	27.391%
16	17.021%	22.732%	31.551%

Table 3: Granularity Control for Fibonacci Numbers

Consider the MergeSort benchmark (as presented in [6] page 578) and table 4.

Eng.	List Input Size			
	750	*1000*	*2000*	*3000*
4	-3.922%	-1.492%	-4.059%	-10.791%
8	33.333%	35.514%	37.828%	43.878%
12	18.182%	18.816%	22.819%	29.825%
16	37.234%	43.167%	46.667%	50.881%

Table 4: Granularity Control for Merge Sort

This program is similar to QuickSort. But in this program the input size becomes small enough to fail the granularity tests only in the last recursions. The table indicates that the granularity mechanism copes adequately and improve the performance when many engines are being used. Debray's models improves performance by 14.1% under the ROLOG system and 1.44% under the PDP system. But the table indicates that the proposed controls are able to provide a best case performance improvement of 50.881%.

The Integer Matrix Multiplication benchmark (as presented in [13]) is shown in table 5.

Eng.	NxN Matrix Input Size			
	15	*30*	*45*	*60*
4	16.901%	19.672%	23.354%	26.979%
8	16.129%	19.170%	20.698%	27.983%
12	21.909%	22.377%	26.679%	33.287%
16	23.750%	24.493%	28.708%	36.935%

Table 5: Granularity Control for Matrix Multiplication

This benchmark generates four parallel tasks in a single recursion, which is twice as many as previous programs. Extra parallel tasks require extra granularity tests which impose extra run-time overhead. The table indicates that the controls are able to cope with such cases effectively and improve performance. Debray tests a small 8x8 matrix. The large granularity threshold of this benchmark in PAN makes the 8x8 matrix multiplication without any practical interest therefore it can not be compared with Debray's model. Garcia's model improves the performance of the multiplication of a 4x2 and a 2x100 matrix by 16.27% under the &-Prolog system using 4 processors. When a matrix 75x1 and a vector are multiplied under the PDP system performance does not improve at all regardless of the number of processors used.

4.2 OR-parallel execution

The set of OR-parallel controls contains several complex tools which consider properties like potential OR-branches, and the number of user defined primitives in the body of the clauses of predicates.

The Permutations benchmark (as presented in [23] page 91) is shown in table 6.

Eng.	List Input Size		
	6	*7*	*8*
4	0.0%	-5.932%	-3.833%
8	0.571%	34.118%	39.0%
12	6.061%	34.12%	40.825%
16	22.727%	32.632%	42.551%

Table 6: Granularity Control for Permutations

Permutations is a typical example of fine-grained OR-parallelism. The clauses generate a search tree with several OR-branches. The table shows that the proposed model controls granularity effectively, generating coarse-grained parallel tasks that improve performance on distributed platforms like PAN. Performance increases as the input size and the number of engines increases. This benchmark however does not perform that well when four processors are used. Since the percentage of exploitable parallelism is very small, the effect of granularity controls can not balance the time required to test the granularity constraints. Performance improves significantly when more processors are used, because the percentage of exploitable parallelism gets bigger.

The previous example is the core program for other benchmarks like naive N-Queens (as presented in [23]

page 119). OR-parallelism in N-Queens is generated in a similar way to Permutations. As a result performance figures are very similar. The controls used in the PDP are able to provide a best case performance improvement of approximately 5% for this program.

The Tree Lookup benchmark (as presented in [20]) is shown in table 7. In contrast to previous examples, Tree Lookup may generate more than two OR-branches in each recursion (depending on the shape of the tree). Granularity controls are able to improve performance dramatically because they can exploit the fine-grained nature of the program and perform profitable size tests that coalesce many fine-grained tasks. The controls prove useful when the tree is unbalanced.

Eng.	Integer Input Size		
	5	*7*	*9*
4	52.5%	65.41%	77.69%
8	57.86%	73.10%	85.20%
12	64.17%	77.23%	90.37%
16	68.57%	81.04%	93.20%

Table 7: Granularity Control for Tree Lookup

Debray's model improves performance only by 3% under the ROLOG system.

5 Summary

Most models described in section 2 base their decisions on the relation between candidates for parallel execution tasks and communication costs. In contrast the proposed model makes better use of that information and proposes new criteria to relate the communication cost with the amount of useful computation that can be performed locally during communication. This method is particularly suitable for distributed systems for which communication costs are considerable and local engines are effective. The method does not require any user declarations or expertise. The model tries to reduce run-time tests that influence the performance of [10] but also includes estimations of such costs at any compile-time decision to make the proposed model more accurate. It bases the analysis on the notion of a collection of parallel tasks. The absence of that notion may prevent [16] from making correct classifications. Some further heuristics applied to the basic granularity controls may improve decisions. Moreover the proposed model makes OR-parallel granularity decisions as well.

This model exploits a good degree of parallelism because it adjusts to the communication costs which may change from time to time or from goal to goal. The restrictions implied by the controls are proportional to the system limitations. If the system characteristics improve, parallelism will become more fine-grained. Generally speaking granularity control is conservative, but has good reasons and appropriate justifications for restricting exploitable parallelism and tries to keep a balance between performance and distributed execution. The tables illustrate that performance may improve significantly, making distributed platforms like PAN more attractive to programmers.

References

[1] K.A.M. Ali and R. Karlsson. The MUSE approach to OR-parallel Prolog. *International Journal of Logic Programming*, 19(2):129–162, April 1990.

[2] L. Araujo and J.J. Ruz. A parallel Prolog system for distributed memory. *International Journal of Logic Programming*, 33(1):49–79, October 1997.

[3] U. Baron, J.C. de Kergommeaux, M. Hailperin, M. Ratcliffe, P. Robert, J.C. Syre, and H. Westphal:. The parallel ECRC Prolog system PEPSys: An overview and evaluation results. In ICOT, editor, *International Conference on Fifth Generation Computer Systems*, pages 841–850, Tokyo, November 1988.

[4] G. Böckle. Exploitation of fine-grained parallelism. *Lecture Notes in Computer Science*, 942, 1995.

[5] K. De Bosschere and J-M. Jacquet. Multi-Prolog: Definition, operational semantics and implementation. In D.S Warren, editor, *Proceedings of the 10th International Conference on Logic Programming*, pages 299–313, Budapest, June 1993.

[6] I. Bratko. *Prolog: Programming for artificial intelligence,* second edition. Addison Wesley, 1991.

[7] V.S. Costa, D.H.D. Warren, and R. Yang. The Andorra-I preprocessor: Supporting full Prolog on the basic Andorra model. In K. Furukawa, editor, *8th International Conference on Logic Programming*, pages 599–613, Paris, June 1991.

[8] J.C. Cunha, P.D. Medeiros, M.B. Carvalhosa, and L.M. Pereira. Delta-Prolog: A distributed logic programming language and its implementation on distributed memory multiprocessors. In

P. Kacsuk and M.J. Wise, editors, *Implementations of Distributed Prolog*, pages 335–356, John Wiley,Chichester, 1992.

[9] S.K. Debray and N. Lin. Automatic complexity for logic programs. In K. Furukawa, editor, *8th International Conference on Logic Programming*, pages 599–613, Paris, June 1991.

[10] S.K. Debray, N.W. Lin, and M. Hermenegildo. Task granularity analysis in logic programs. In *Proceedings of the 1990 ACM Conference on Programming Language Design and Implementation*, pages 174–188, New York, June 1990.

[11] I. Futo. The real time extension of CS-Prolog professional. In J. Barklund, B. Jayaraman, and J. Tanaka, editors, *ICLP'94 - Workshop on Parallel and Data Parallel Execution of Logic Programs*, Santa Margherita Ligure, June 1994.

[12] P.L. Garcia, M.V. Hermenegildo, and S.K. Debray. A methodology for granularity based control of parallelism in logic programs. *Journal of Symbolic Computation, Special Issue on Parallel Symbolic Computation*, 22:715–734, 1996.

[13] M.V. Hermenegildo and K.J. Greene. The &-Prolog system: Exploiting independent AND-parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.

[14] L.V. Kale. The REDUCE/OR process model for parallel execution of logic programming. *International Journal of Logic Programming*, 11(1), July 1991.

[15] S. Kaplan. Algorithmic complexity of logic programs. In R.A. Kowalski and K.A. Bowen, editors, *Fifth International Conference and Symposium on Logic Programming*, pages 780–793, Seattle, August 1988.

[16] A. King and P. Soper. Heuristics, thresholding and a new technique for controlling the granularity of concurrent logic programs. Technical Report CSTR 92-08, Dep. of Electronics and Computer Science - Southampton University, Southampton S09 5NH, 1992.

[17] E. Lusk, D.H.D. Warren, and S. Haridi. The Aurora OR-parallel system. *New Generation Computing*, 7(2,3):243–271, 1990.

[18] D. Palmer and L. Naish. NUA-Prolog, an extension to the WAM for parallel Andorra. In K. Furukawa, editor, *8th International Conference on Logic Programming*, pages 599–613, Paris, June 1991.

[19] F.A. Rabhi and G.A. Manson. Using complexity functions to control parallelism in functional programs. Technical report, Dep. of Computer Science, University of Sheffield, England, January 1990.

[20] E. Shapiro and L. Sterling. *The art of Prolog*. MIT Press, 1988.

[21] A. Takeuchi. *Parallel Logic Programming*. PhD thesis, University of Tokyo, Japan, July 1990.

[22] H. Taylor. Assembling a resolution multiprocessor from interface, programming and distributed processing components. *Computer Languages*, 22(2,3):181–192, 1996.

[23] E. Tick. *Parallel Logic Programming*. MIT Press, 1991.

[24] E. Tick. Compile-time granularity analysis of parallel logic programming languages. *New Generation Computing*, 7(2), January 1990.

[25] E. Tick and X. Zhong. A compile-time granularity analysis algorithm and its performance evaluation. *New Generation Computing*, 1(3,4), 1993.

[26] B. Wegbreit. Mechanical program analysis. *Communications of the ACM*, 18(9):528–539, September. 1975.

[27] M.J. Wise, D.G. Jones, and T. Hintz. PMS-Prolog: A distributed, coarse-grain-parallel Prolog with processes, modules and streams. In P. Kacsuk and M.J. Wise, editors, *Implementations of Distributed Prolog*, pages 379–404, John Wiley,Chichester, 1992.

[28] G. Xirogiannis. Compile-time analysis of freeness and side-effects for distributed execution of Prolog programs. In T. Sellis and G. Pagkalos, editors, *6th Hellenic Conference on Informatics*, pages 701–722, Athens, December 1997.

[29] G. Xirogiannis and H. Taylor. A Dynamic Task Distribution and Engine Allocation Strategy for Distributed Execution of Logic Programs. In *1998 International Conference on High-Performance Computing & Networking* to appear, Amsterdam, April 1998.