New Generation Computing, 20(2002)373-399
Ohmsha, Ltd. and Springer-Verlag

PAN: A Portable, Parallel Prolog: Its Design, Realisation and Performance

George XIROGIANNIS and Hamish TAYLOR
Heriot-Watt University
Riccarton, Edinburgh, EH14 4AS, Scotland
ceegx@cee.hw.ac.uk, hamish@cee.hw.ac.uk

Received 12 August 1999
Revised manuscript received 16 May 2001

Abstract PAN is a general purpose, portable environment for executing logic programs in parallel. It combines a flexible, distributed architecture which is resilient to software and platform evolution with facilities for automatically extracting and exploiting AND and OR parallelism in ordinary Prolog programs. PAN incorporates a range of compile-time and run-time techniques to deliver the performance benefits of parallel execution while retaining sequential execution semantics. Several examples illustrate the efficiency of the controls that facilitate the execution of logic programs in a distributed manner and identify the class of applications that benefit from distributed platforms like PAN.

Keywords: Logic Programming, Parallelisation, PVM.

§1 Introduction

PAN is a distributed logic programming system designed to exploit divide-and-conquer and speculative parallelism while being resilient to the software and hardware evolution that rapidly makes most parallel programming technology obsolete. A major design goal has been to assemble it in a modular way from widely used, off-the-shelf components with long term software support. Such a system is less likely to become obsolete than if its main components were designed and built from scratch. The burden of evolving the system falls mainly on the component developers, and the smaller burden of evolving the modular interfaces is rather more manageable. This leads to a distributed design using sequential Prolog processes glued together by a distributed communications and process management infrastructure, PVM.[20] It allows PAN to run on widely available platforms such as networks of workstations as well as on parallel machines. A separable composite interface gives users access to all parts of the system.[50]

Effective exploitation of a distributed architecture for executing Prolog programs also requires significant analysis to determine how best to partition up the resolution work load among the processing elements while preserving standard execution semantics. This paper describes a practical scheme called ADEPT for doing this on PAN in an automatic way.

Section 2 discusses alternative approaches to logic programming in parallel, while Section 3 presents the PAN approach. Section 4 reviews techniques which detect good system configurations and introduces the tools incorporated in PAN to detect parallelism. Section 5 explains the granularity controls and provides experimental results. Section 6 presents the scheduling scheme and Section 7 presents a class of applications that can run effectively under PAN.

§2 Relevant Research

2.1 AND and OR Parallelism

Parallelism in Prolog programs can be exploited by evaluating alternative solutions of a goal in parallel (OR-parallelism) or by simultaneously executing the goals in the body of a clause (AND-parallelism). Where the goals in a clause body share variables, schemes for parallel execution (eg[8]) co-ordinate their evaluation to avoid the same variable being bound to different values.[24] In the generalised version of Independent AND-parallelism (IAP)[26] goals are deemed independent when no variable conflicts arise and/or the complexity of the search expected by the programmer can be preserved. DeGroot's RAP model[17] combines compile-time analysis with run-time checking to identify goals with shared variables and conditionalise their parallel execution. Such methods reduce the complexity of identifying parallelism solely at run-time. The RAP-WAM[25] is a WAM-based implementation of RAP for shared-memory systems. Parallel systems like &-Prolog,[23] ACE[41] and PDP[2] identify independent AND-tasks based on RAP.

Andorra-I[13] and NUA-Prolog[40] exploit deterministic AND-parallelism based on the Andorra Principle[56] as an alternative to IAP. A pre-processor identifies parallelisable tasks automatically. Implementations of the DDAS model[45] also exploit dependent AND-parallelism under conditions that yield the same results as sequential Prolog execution. These schemes have yielded promising results but heavily depend on specialised abstract machine technology that seems unlikely to attract long term software support. By contrast ACE adopts a copying-based approach to simplify the exploitation of parallel tasks.

Despite useful recent research in logic programming on program analysis, the automatic detection of parallelism has not yet been much used in parallel and/or distributed systems. Systems like ANDOR-II[49] Delta-Prolog,[12] CS-Prolog[18] and PMS-Prolog[59] cannot detect parallelism automatically and rely on explicit user declarations to exploit parallelism. These declarations require significant expertise to employ correctly.

A major problem with implementing OR-parallelism is how to represent different bindings for the same variable corresponding to different branches of the search tree as discussed in Reference.[22] This affects the cost of creating and accessing variable bindings either at task creation time, variable access time or task switching time. The SRI model[55] used by the Aurora[35,48] system keeps the multiple variable binding in an array in shared memory but incurs non-trivial task switching costs. MUSE[1] and Delphi[9] use a distributed-memory space for each parallel execution and do not need a representation for the multiple bindings of a variable. MUSE transfers an explicit copy of the data of an executing process while the Delphi model reconstructs a process environment by recomputing the initial goal, which increases the task creation time. Kacsuk's version of the recomputation model[28] makes the least changes to the sequential WAM. Multi-sequential models OPERA[6] and PloSys[36] implement TWAM, an extension to the WAM, to allow more efficient copy operations in the portion of stacks shared between engines. The Reduce-OR model[30] uses an alternative approach to exploit both full OR-parallelism and IAP parallelism. Data-join graphs represent computations for execution on non-shared memory multi-processors.

All these approaches re-engineer mainstream Prolog technology. They make users dependent on specialised components that are unlikely to be supported in the long term and risk becoming rapidly obsolete as mainstream technology evolves. They also tend to introduce new control and I/O mechanisms and fail to support well established I/O predicates and control mechanisms making it laborious and awkward to port existing Prolog code to these platforms.[21]

2.2 Granularity of Parallelism

Distributed execution of logic programs requires a match of granularity between a program and the multi-processor, it runs on to exploit its potential for performance fully. Processes that are too coarse-grained for a multi-processor system unnecessarily limit its ability to exploit parallelism. Processes that are too fine-grained introduce excessive communication overheads. Granularity analysis should enable parallelism to be exploited at the right grain.

Early proposals[31,57] investigated the automatic inference of the complexity of logic programs but only under several restrictive assumptions and did not model recursive predicates satisfactorily. Tick[52] used weights to quantify the grain of tasks. However, recursive predicates presented difficulties because the quantity of computation is data-dependent and difficult to determine. More recent proposals like[53] model satisfactorily all kinds of predicates but focus mainly on measuring the complexity of a process (goal) and have paid little attention on how to use this information.

Debray *et al.*[15,16] derive complexity functions for predicates at compile-time. Once the size of the data is known at run-time these functions can be evaluated. The size is checked against a threshold to determine whether or not the goal should be evaluated in parallel. The scheme models predicates better and provides a better approximation of the *weight* of recursive predicates. Experimental results show that this model can improve performance although it can also impair performance when too much information is processed at run-time. This model does not take into consideration some important factors. The maintenance of size information and the grain size tests add a significant execution overhead which is not included in the time complexity estimation. This model also disregards the fact that some predicates may fail or pass the granularity tests at compile-time and thus no run-time overhead is associated with them (i.e. for non-recursive or very complex recursive predicates). Garcia *et al.*[19] refine most of Debray's techniques but still rely on the basic granularity control principle.

King *et al.*,[32,33] have proposed a different technique for controlling the granularity of tasks at compile-time. This work is addressed towards concurrent languages. They coalesce tasks 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 but it does not consider parallel execution of a goal in relation to the execution of other goals (like Debray's model) and does not effectively relate the cost of processing a task in parallel with the amount of local computation of the remaining tasks.

Shen *et al.*[46,47] propose a granularity control metric based on the idea that if the gain obtained by executing a task in parallel is less than the overheads required to support parallel execution, then the task is better executed sequentially. They argue that to minimise the overheads associated with parallel control, the number of creation points of parallel tasks should be reduced, hence the "distance" between the points that create parallel work should increase.

Granularity control mechanisms could also be used in parallel systems running on shared-memory multiprocessors like NUA-Prolog, Andorra-I, Parallel NU-Prolog, ANDOR-II, Aurora and Muse to improve performance to reduce unnecessary communication. PDP controls the grain size of parallel tasks based on Debray's model. It also uses heuristic observations concerning memory usage to control the grain of OR-tasks better. Experimental results presented in Reference[2] indicate that performance improves in some cases when the mechanism is used to control the grain size of potential parallel tasks.

2.3 Scheduling Parallel Tasks

Scheduling may be *engine driven* where engines look for tasks or *task driven* where tasks look for engines. &-Prolog and Andorra-I[13] use the first method. The latter comprises a top scheduler (reconfigurer) and 2 sub-schedulers each responsible for AND-parallel and OR-parallel execution respectively. The schedulers partition the engines into flexible teams to distribute parallel tasks. Engine-driven approaches are also used by MUSE and Aurora and mainly address the problem of efficient scheduling of OR-parallel tasks controlling speculative OR-work.[4]

Such scheduling strategies tend to identify many small parallel tasks at run-time. They are designed for platforms with low communication costs and fail to allow for the run-time overheads of distributed heterogeneous platforms like networks of workstations. The task switches and the search for new work depend proportionally on the communication overheads among engines. Engine-driven scheduling can not always efficiently relate the actual task load with the composition of each team imposing run-time task and engine migration[39] overheads.

The communications costs of distributed platforms have been considered in the scheduling techniques used in OPERA[6] and PloSys[36] which exploit OR-parallelism. A hierarchy of schedulers run in parallel with the workers using an approximate representation of the system's state, while OPERA's multi-sequential computational model does not create more parallelism than is available to exploit. Similar multi-sequential models have also been adopted by more recent distributed systems like PDP. To improve their performance OPERA and PDP significantly re-engineer the WAM by pushing most of the parallel control to the engine level and adding primitives to optimise run-time execution. Thus they depart from using mainstream Prolog technology on distributed platforms and make themselves liable to being marginalised as mainstream technology evolves. PDP's current implementation only uses one scheduler to support the number of available engines. This results in a centralised scheduling scheme with reduced flexibility that is a potential bottleneck during distributed execution.

§3 Design Choices of PAN

PAN combines *standard SICStus Prolog processes*, a *model of multiprocessing*, a *virtual multiprocessor* PVM and a Tcl/Tk *user interface* to get a systems architecture with evolutionary resilience and high portability. PAN relies on its automatic paralleliser ADEPT to multiprocess ordinary Prolog programs. PAN's model of multiprocessing is *control driven*, exploits parallelism in a *coarse-grained way*, creates processes *statically* and communicates by synchronous and asynchronous *message passing* using extra primitives added to the Prolog language to control message passing and enable synchronisation.

PAN runs one Prolog on each host of a PVM session. It is presented to users under X Window as a multi-headed extension of single processor Prolog systems. An interface to each Prolog engine is displayed in a separate X widget together with a console window to the PVM, and window based methods are supported for simultaneously invoking all engines together.[50]

PAN has been implemented under several versions of UNIX and on several heterogeneous hardware platforms. PAN uses SICStus Prolog's foreign language interface to add a few extra primitives to Prolog to allow program threads to pass messages among each other. The main communication primitives are:

rx(Term, Id)	blocks until caller synchronises with transmission from Prolog Id and then Term is unified with message sent
tx(Term, Id)	blocks until caller synchronises with receiving Prolog Id and then Term is sent
rxnb(Term, Id)	Term is unified with message (to be) sent from Prolog Id
txnb(Term, Id)	sends Term asynchronously to Prolog Id

Arbitrarily large terms can be passed between Prolog engines. Unbound variables in them get freshly renamed on reception. Ordinary Prolog programs are run in parallel on PAN by being transformed into its extended version of Prolog so as to give the same results as with sequential execution. AND-parallelism is exploited in a *fork-join* manner. Tasks are distributed to remote engines and the results are returned to the parent engine. To facilitate OR-parallelism the algorithm in Reference[43] is implemented in Prolog. This implementation of the algorithm (Reference,[61] pp. 322) is run in an AND-parallel manner and in contrast to Reference[43] contains sequential calls to process (on demand) part of the OR-tree sequentially. AND-OR parallelism is exploited by combining both forms of parallelism. Parallel tasks run in AND-parallel, which reduces the complexity of interfacing between separate schedulers for OR and AND-parallelism (e.g. Andorra-I).

3.1 ADEPT

The **A**utomatic **D**istributed **E**xecution *of* **P**rolog *by* **T**ransformations system[61] provides a set of tools and facilities that can analyse the configuration state of PAN, transform the programs of Prolog applications and can effectively exploit parallel execution in a distributed manner on process-based heterogeneous platforms. The *ADEPT* system (Fig. 1) runs on top of the PAN system and consists of compile-time and run-time facilities that rely on the PAN extensions to control parallel computations. It offers automatic parallel program control requiring little user intervention, ensures correct results (in terms of sequential execution) and exploits good degrees of parallelism. PAN, augmented by ADEPT, provides a distributed platform which requires little (if any) user intervention to exploit parallelism efficiently (in contrast to platforms like PEPSys,[3] Delta-Prolog,[12] CS-Prolog,[18] PMS-Prolog,[59] etc.), exploits various forms of parallelism, and uses standard Prolog technology (unlike OPERA, PDP, etc.).

Both the *System Analysis* component and the *Program Analysis* component have been custom-made for PAN. The *System Analysis* component uses the commands and tools of PAN (e.g. PVM console, extra primitives added to SICStus by PAN, etc.) as well as heuristics and rules of thumb to identify a suitable platform configuration to run the applications. The *Program Analysis* component is called ADEPT. Apart from determinacy analysis (which follows the ideas presented in Reference[42]), all other controls and tools (namely freeness analysis, call synchronisation graphs, task granularity control, task distribution analysis and engine allocation strategy) are either novel products of ADEPT or augment existing research in the area of distributed execution of Prolog. The *System Analysis* component is described briefly in Section 4.1 while *Program Analysis* is presented in more detail in Sections 4.2, 5 and 6.

§4 Analysis

4.1 System Analysis

Analysis mechanisms identify those workstations that combine high processing capabilities with low workload. System analysis is performed at compile-time in the following steps:

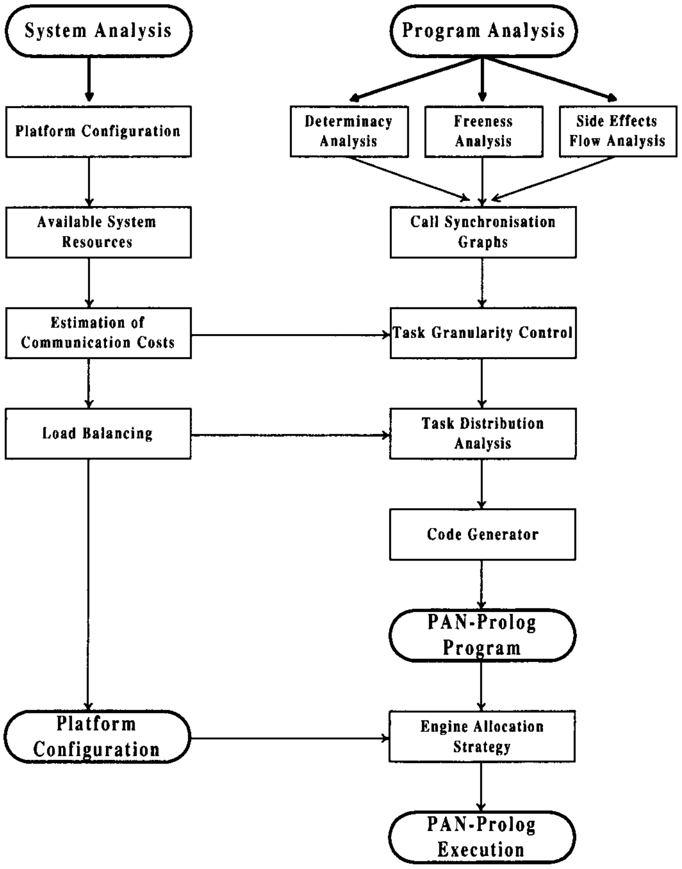

Fig. 1 Overview of ADEPT

- Identification of Platform Components
- Estimation of the Available System Resources
- Estimation of Communication Costs
- Load Balancing

Participating hosts are identified at compile-time, using the facilities of the PVM console. Each processor's current workload is measured as the average length of its current run-time queues. The processing capabilities of each engine are measured in LIPS while the communication speed of the LAN is measured in Mbits/sec. PAN sessions utilise available Prolog engines taking these costs into account. The number of LIPS each engine can perform has been measured by running quicksort. The maximum rating in LIPS for that program does not change for a given engine if the hardware and software configuration of that engine remain the same. The length of the run-time queues is identified using the UNIX command *uptime*. The current communication speed of the LAN is estimated by running tests that communicate messages over it.

PAN's performance is sensitive to the LAN's communication traffic and to changes in the workload of the engines participating in a given session. The current implementation of the system analyser detects network overheads and engine workload mainly at compile-time, hence it may fail to anticipate a sudden change during program execution and adjust the configuration of the platform accordingly.

4.2 Program Analysis

One of the jobs of ADEPT's compile-time analyser is to process Prolog programs so that they can be transformed into some equivalent form to run in parallel on PAN. The query-independent analysis offers an automatic control driven approach to exploiting parallelism. The analyser imposes a low degree of extra run-time controls. Under PAN, pure goals are AND-parallel candidates if:

1. Goals do not share any variables.
2. Goals share free variables
3. Goals share variables but these goals are determinate.[*1]

Abstract program analyses (based on *determinacy analysis* and *freeness analysis*) determine when these conditions exist. *Determinacy Analysis*[42] ascertains the number of times that a goal can be (re)-satisfied. It considers the worst case where backtracking is exhaustive among independent goals with independent variables as arguments. The analysis calculates the upper bound for the determinacy of a conjunction/disjunction from the goals within the conjunction/disjunction. Goals with at most one solution are determinate. Such goals do not restrict the search space and they do not generate variable conflicts in and-parallel execution. Determinacy analysis is conservative mainly because it is performed entirely at compile-time, but it imposes no run-time condition-checking overhead.

Freeness Analysis detects at compile-time goal dependencies based on the instantiation state of shared variables. A mechanism[60,61] analyses the instantiation state of variables and classifies shared variables as *free, ground, non-free* and *aliases*. Free variables remain uninstantiated after the execution of the program. Grounding of variables occurs when one of the terms involved in a unification is ground. This causes the variables of the non-ground term to become ground too. Aliasing can occur when one free variable is unified with another free variable. Variables remain free but any consequent instantiation applied to one of them will also be applied to the other. Variables instantiated to non-ground terms are said to be non-free. The analyser determines that *shared free variables do not restrict the search space of their goals and goals with shared free variables do not generate variable conflicts.*

Analysis is performed entirely at compile-time adding no run-time condition-checking overhead. The mechanism for inferring freeness and sharing

[*1] This paper adopts the terminology proposed by Sahlin.[42] Determinate goals either fail or succeed once.

information in ADEPT is in a sense similar to the depth-k mechanism discussed in Reference.[34] However there is a fundamental difference. The depth (and the fixpoint) of the analysis in ADEPT is not pre-defined, in contrast to Reference.[34] It is determined dynamically during compile-time for different types of programs. ADEPT iterates as many times as is necessary to estimate the instantiation state of shared variables. The depth of the iteration restricts the depth of the terms and then ADEPT approximates the behaviour of the variables. On the other hand, the mechanism in Reference[34] restricts the depth of the terms to a fixed number first and this restricts the number of iterations. Then it approximates the behaviour of the variables. The dynamic nature of ADEPT allows greater flexibility in the approximation of the behaviour of terms but this on the other hand may affect the efficiency of the mechanism. The complexity of Reference[34] can be controlled by restricting the depth of terms. This is not the case for ADEPT. Its complexity can grow as the number of iterations increases. But it is encouraging to know that for the range of applications tested in ADEPT so far, iterations stopped at reasonably small depths, similar to the depth restriction imposed by Reference.[34]

Freeness analysis performed by ADEPT always terminates. The analyser estimates if the instantiation state[*2] of shared variables changes in a way that may affect the search space of goals. This estimation can be calculated in a fixed number of iterations. The algorithm iterates (and lets recursive goals recurse) as many times as it is necessary to prove that further iterations will not provide any different information as far as the instantiation state of variables is concerned. The *fixpoint* is reached when further iterations do not change the instantiation state of shared variables. The analysis is safe in the sense that goals classified as independent at compile-time, will never depend on each other at run-time (eg. restrict the search space of each other). However, the analysis maintains a certain conservatism because it cannot consider the actual run-time instantiations. In this sense, goals classified as dependent at compile-time, may not depend on each other at run-time. This is the case when a suitable run-time query grounds all shared variables, which effectively makes AND-goals with shared variables, independent. The accuracy and the effectiveness of the analysis performed by ADEPT can be compared with the Share,[27] Free,[38] Linear[37] and Gif[7] mechanisms. ADEPT performs as accurately and as effectively as these mechanisms. But there are examples in Reference[61] which indicate that ADEPT can perform even better.

Side Effects Flow Analysis is also used at compile-time to detect goal dependencies and OR-parallel execution restrictions due to the presence of certain non-logical Prolog primitives. The main disadvantage of Side-Effects Flow Analysis is that it does not allow any predicate to perform speculative OR-work in the presence of side-effects in its body. The results of the program analysis performed so far are recorded in abstract *Call Synchronisation Graphs* (CSG) which represent sequentiality constraints which are necessary to maintain standard Prolog semantics. The algorithm[60] that generates CSG extends the research

[*2] shared variables are classified as *free, ground, non-free* and *aliases*

in Reference[58] to encapsulate side-effects flow analysis, determinacy analysis as well as freeness analysis. CSG has similar expressive power to UDG.[38] CSG can be extended to include run-time conditions similar to CDG.[38]

PAN-Prolog programs are able to exploit AND and OR-parallelism. If further analysis (section 5) detects that some form of parallel execution may not improve the performance, the run-time engine component (section 6) disregards the corresponding transformations to reduce the extra code interpretation and processes Prolog programs sequentially.

§5 Granularity Control

The execution of a logic program and query divide naturally into subtasks for distributed execution. If this division is carried too far, the benefits of parallel execution are outweighed by the overheads of communication and run-time scheduling. One way to deal with tasks that are too fine-grained is to coalesce them into larger grained tasks. The main motivation for controlling grain size is to reduce the overheads so that overall execution time will decrease. ADEPT's controls[62] focus on efficiently using the data gathered by complexity analysis programs like CasLog[15] which estimate the time complexity of goals. They relate the amount of useful local computation with PAN's communication costs.

5.1 AND-Parallel Execution

Consider a clause C :- B_1, B_2,.....,B_n. Assume that program analysis has determined that a set (collection) of B_i (written as G_i for convenience) are candidates for parallel processing. Let these goals be G_1,G_2,...,G_k. Assume that PAN employs k engines.[*3] Let $T(G_m)$ be an estimate (by CasLog) of the time required to process goal G_m locally. Let W_n represent the processing capabilities (as discussed in section 4.1) of engine n. Let the extra time required to process goal G_m on a remote engine be

$$T_{lat}(G_m) = T_{com}(G_m) + T_{sched}(G_m) + T_{rt}(G_m) + \frac{W_{local} - W_{remote}}{W_{remote}} * T(G_m)$$

T_{lat} represents the communication overhead (T_{com}), any scheduling cost (T_{sched}), the cost of any run-time granularity test (T_{rt}) and the extra time required to process a task in a slower engine[*4] $\left(\frac{W_{local} - W_{remote}}{W_{remote}} * T(G_m)\right)$.

Let goal G_{local} be such that $T(G_{local}) = \max\{T(G_j) : G_j \in \{G_1, ..., G_k\}\}$. Goal G_{local} will be processed locally.

Basic Granularity Control for AND-parallelism: *AND-goal G_i should be executed remotely if* $T(G_{local}) \geq T_{lat}(G_i)$.

This granularity control dictates that in order to execute goal G_i in AND-parallel with goal G_{local}, the extra time required to process goal G_i remotely should be greater or equal to the time required to process goal G_{local} locally. This applies even if $T(G_i) < T_{lat}(G_i)$, in contrast to Debray and Garcia's proposal.

[*3] Alternatively assume that PAN is able to process all G_1,...,G_k in parallel.

[*4] In section 6 we will discuss why all remote engines are slower than the local engine

The basic granularity control for AND-parallelism will improve performance because $\forall G_i \in \{G_1, ..., G_{local-1}, G_{local+1}, ..., G_k\}$ that satisfy the granularity control

$$\begin{aligned} T_{parallel}(G) = \max\{T(G_{local}), \max_i\{T_{lat}(G_i) + T(G_i)\}\} \le \\ \max\{T(G_{local}), \max_i\{T(G_{local}) + T(G_i)\}\} \le \\ \max\{T(G_{local}), T(G_{local}) + \max_i\{T(G_i)\}\} \le \\ T(G_{local}) + \max_i\{T(G_i)\} \le T(G_1) + ... + T(G_k) \\ = T_{sequential}(G) \end{aligned}$$

The control still holds even if PAN employs less than k engines[*5] because

$$T(G_{local}) + T(\text{AND-tasks processed locally}) \ge T(G_{local}) \ge T_{lat}(G_i).$$

Performance may improve further by distributing every G_i using the rule $T(G_i) = \min\{T_{lat}(G_j) : G_j \in \{G_1, ..., G_{local-1}, G_{local+1}, ..., G_k\}\}$. This rule may reduce overheads by distributing goals with less communication cost first, hoping that goals with considerable communication costs will be processed locally.

5.2 OR-Parallel Execution

Consider now a predicate P with clauses $C_1, C_2, ..., C_n$. Assume that a program analyser 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

$$T_{lat}(C_i) = T_{com}(C_i) + T_{sched}(C_i) + T_{rt}(C_i) + \frac{W_{local} - W_{remote}}{n * W_{remote}} * T(P)$$

be the extra cost for processing P over the head of clause C_i on a remote engine defined similarly to the AND-parallel controls. Let C_{local} be such that $T(C_{local}) = \max\{T_{lat}(C_j) : C_j \in \{C_1, ..., C_n\}\}$.

C_{local} will be processed locally. The use of a predicate level directive dictates that either all or no clauses are candidates for OR-parallel execution.

Basic Granularity Control for OR-parallelism: *Clauses of P should be explored in OR-parallel if*

$$\frac{1}{n}T(P) > \max\{T_{lat}(C_i), i = 1, ..., (local - 1), (local + 1), ..., n\}$$

The basic control for OR-parallelism ideally should dictate that in order to process the clauses of P in parallel $T(C_{local}) > T_{lat}(C_i)$. However, mainly due to practical restrictions which make estimating the time complexity of C_{local} non-trivial, this condition could apply only to non-recursive predicates. To model recursive and mutually recursive predicates, ADEPT currently assumes that all clauses of such predicates equally contribute to time complexity:

$$T(C_i) \simeq T(C_{local}) \simeq \frac{1}{n}T(P)$$

[*5] Alternatively assume that PAN is not able to process all $G_1, ..., G_k$ in parallel.

This basic control will improve performance because $\forall C_i$ that satisfy the granularity control where $\forall C_i \in \{C_1, ..., C_{local-1}, C_{local+1}, ..., C_n\}$

$$\begin{aligned} T_{parallel}(P) = \max\{T(C_{local}), \max_i\{T_{lat}(C_i) + T(C_i)\}\} \leq \\ \max\{\frac{1}{n}T(P), \max_i\{\frac{1}{n}T(P) + T(C_i)\}\} \leq \\ \frac{1}{n}T(P) + \max\{T(C_i)\} \leq \frac{1}{n}T(P) + \frac{1}{n}T(P) \leq T_{sequential}(P) \end{aligned}$$

Performance may improve further (similarly to AND-parallelism) if we distribute every C_i using the rule

$$T(C_i) = \min\{T_{lat}(C_j) : C_j \in \{C_1, ..., C_n\}\}$$

To make the analysis more accurate, Garcia suggests using *follow sets* from the theory of context free grammars to estimate the complexity of C'. Garcia also suggests using clusters of clauses such that within each cluster clauses are executed sequentially and the different clusters are executed in OR-parallel instead of using predicate level parallelism. The program is re-written to generate the clusters. This technique may tend to balance the work-load better in contrast to the "all clauses in parallel" directive adopted by ADEPT's analyser.

ADEPT's analysis uses a new control metric to make better use of the notion of the collection of parallel tasks. The general idea is that parallel candidates should be executed in a distributed manner if the time required to communicate all but one of the candidates remotely, is less than the time required to process the remaining task locally. This metric relates local computation to the overheads of executing a set of tasks over the distributed platform which makes it different to other proposed control metrics. In contrast, Shen *et al.* use a metric that compares parallel execution time with the overheads of supporting parallelism. They claim that the execution time of all parallel tasks should be considered, while we claim that the execution time of the local task suffices. The approach by King and Soper coalesces processes together if the complexity of their communication dominates the complexity of their computation. Debray *et al.* and Garcia *et al.* use a metric that compares the execution time of a parallel candidate with the extra cost of handling this candidate in parallel.

The proposed controls (in many cases) may explore more parallelism than other proposed controls. For example, Debray and Garcia's proposal will restrict AND-parallelism of an AND-candidate G, if $T(G) < T_{lat}(G)$. ADEPT, on the other hand, will make a more informative decision based on the execution time and overheads of the collection of related AND-candidates, by parallelising T(G) only if there exists an AND-candidate K (to be processed locally) such that $T(K) > T_{lat}(G)$. Similarly, the analysis by King and Soper might fail to AND-parallelise AND-candidate goals G, K if $T(G) < T_{lat}(G)$ even if $T_{lat}(G) < T(K)$.

The new controls hold regardless of the number of engines employed by the distributed platform or the tool or methodology that estimates the execution time. Most of the analysis is performed at compile-time adding little (when necessary) run-time tests. If $F(n)$ is a function that estimates the time complexity of local tasks depending on the value of the input size n, the compile-time analysis calculates an integer k, such that $\forall n > k$, $\mathsf{F}(\mathsf{n})$ satisfies the granularity control criteria. $\mathsf{F}(\mathsf{n})$ is generated by CasLog.

The analysis partially unfolds loop tests and tests the term sizes but only to the point at which the granularity threshold is reached. This reduces overheads. If analysis detects that goals satisfy the granularity tests at compile-time, then no run-time test is added.[*6] Run-time testing is also part of the compiled-code, but the scheduler[63)] performs any run-time checks only when there are available engines to improve performance further.

Using a practical rule of thumb, the clauses of a predicate are considered (at compile-time) for exploitation in OR-parallel if they are more than k in number (k is set by the implementation, based on tests). The use of the $\frac{\mathsf{W}_{\mathsf{local}} - \mathsf{W}_{\mathsf{remote}}}{\mathsf{W}_{\mathsf{remote}}} * \mathsf{T}(\mathsf{G}_{\mathsf{m}})$ factor makes the mechanism adaptive to heterogeneous distributed platforms like PAN because $\mathsf{T}_{\mathsf{lat}}$ is re-adjusted for each engine n.

The controls generate run-time tests for all possible "local" tasks. This may increase the code size, but on the other hand it provides all possible run-time distribution tests. However, it is fair to say that this problem may appear in other proposals as well. A suitable implementation using appropriate heuristics can reduce the number of possible run-time tests. Consider the **QuickSort** program (presented in Reference,[44)] pp. 56). In terms of PAN-Prolog representation the program will be automatically annotated with the granularity controls and transformed for AND-parallel execution as follows.

```
qsort([],[]).
qsort([A|B],C) :-
   split(B,A,D,E),
   and_task([[grain(qsort(D,F),size(E,SIZE)), qsort(E,G)],
             [grain(qsort(E,G),size(D,SIZE)), qsort(D,F)]]),
   append(F,[A|G],C).
```

The expression `and_task([Set`$_1$`, ..., Set`$_n$`])` indicates that one (and only one) Set$_i$ can be executed in a distributed manner as long as it satisfies the granularity control criteria. The expression `[grain(qsort(D,F),size (E,SIZE)), qsort(E,G)]` dictates that if `size(E) > SIZE` then `qsort(D,F)` will be executed remotely and `qsort(E,G)` locally. Similarly, if `size(D) > SIZE` then `qsort(E,G)` will be executed remotely and `qsort (D,F)` locally.

§6 Scheduling Parallel Tasks in PAN

Detecting available system resources at run-time and migrating tasks among distributed processors incurs overhead. A traditional task scheduler relies heavily on shared resources i.e. shared memory or the interconnection network to perform its functions. As the scale of the distributed platform increases and the speed of local computation improves, task scheduling becomes increasingly frequent but necessary to synchronise the engines. PAN incorporates a mechanism, which consists of 2 components:

[*6] This is the case for non-recursive predicates or for very complex recursive predicates.

1. The *compile-time component* generates *Task Distribution Functions* that estimate the *relative difficulty* of potential parallel tasks.
2. The *run-time component*[63] provides a task distribution and engine re-allocation strategy suitable for heterogeneous distributed platforms.

The compile-time component reduces the overhead of identifying the distribution order of potential parallel tasks. Detailed discussion of this component would clutter this paper, therefore the reader is referred to Reference[61] pp. 194–228. The general idea behind this component is that we can use a suitable abstract interpreter (e.g. a suitable modification of CasLog) to generate a *Task Distribution Function*, which estimates the number of parallel tasks generated by a program and a given goal. It is then "easy" (and in fact of greater value to PAN) to estimate the ratio of the number of parallel tasks generated by pairs of goals, especially when these goals can be executed in parallel with each other. This ratio represents the *relative difficulty* of parallel tasks, which provides an estimate of the number of engines a parallel task may require in comparison to other parallel tasks. The relative difficulty dictates a *best-first* task distribution order, which allocates more system resources to difficult tasks first. Task Distribution Functions can be particularly useful when estimating the relative difficulty of different parallel tasks. Consider the Integer Matrix Multiplication benchmark (as presented in Reference,[61] pp. 333) which generates two sets of parallel tasks, each set generating two different parallel goals (in total four different parallel goals). On the other hand QuickSort (presented in Reference,[44] pp. 56) generates a set of two "similar" parallel tasks (in the sense that both parallel tasks call the same clauses).

6.1 Task Distribution and Engine Re-allocation

Run-time task distribution and engine re-allocation use a *fully distributed farmer-worker* scheme (chapter 8 [61]) which generates dynamic task-driven relations among engines. A *hierarchy* of farmers and workers is generated which corresponds to a hierarchy of goals and sub-goals. Each node in it corresponds to a *distributed component* which consists of a distributed scheduler, its workers and a local engine pool. Several *distributed* and *de-centralised* scheduling components make the model more *scalable*.

Engine allocation and team generation are *dynamic* and *flexible* in the sense that each farmer may have a different number of workers during program execution to *adjust* to the distribution of tasks. Workers may also become farmers *on demand* to process parallel tasks more effectively. Distributed components *communicate infrequently* to reduce certain overheads. Farmers do not interfere with the workers (and their tasks) of other farmers at the same or different level of the hierarchy. Engines communicate only through their parent farmer.

The hierarchy helps the farmers to schedule for a *small number* of workers to improve the efficiency and minimise any bottleneck situations, while reasonable control of task and engine migration is achieved at little cost. The *maintenance* of a distributed pool does not consume much of the engine resources. Farmers perform useful program computation as well. The hierarchy reduces task switches (of schedulers used in Aurora) while preserving (to some extent) the usual Prolog execution strategy and including some of the attractive characteristics of MUSE at little extra cost. To help the scheduler reduce speculative work ADEPT also incorporates an *abort & failure* mechanism.[61)]

Best-first scheduling provides a good degree of *fair engine distribution* and *work-load balancing.* In contrast models that use "depth-first" scheduling (for example Andorra-I) do not always guarantee fair allocation of available system resources. Engines under PAN are sorted in a descending order of their processing capabilities to ensure that farmers (which both schedule remote distribution and perform local computation at the same time) employ "better" engines than their workers. The model distributes both AND and OR parallel tasks *uniformly* without the need for separate schedulers (e.g. Andorra-I).

This task-driven[*7] *farmer-worker* model of best-first engine re-allocation and task re-distribution aims to improve performance by reducing the complexity of interfacing among the scheduler and a large number of engines, to manage synchronisation better and also to keep communication overheads low to perform better on distributed platforms. Scheduling control is done at the Prolog level and its implementation does not re-engineer the WAM (in contrast to distributed systems like PDP and OPERA) which complies with PAN's design choices.

The model imposes some overheads which relate mainly to the frequency of communication between a farmer and its workers and depend mainly on the characteristics of the platform. The bigger the frequency the bigger the overheads. The best frequency to use can be experimentally determined.

§7 Performance

7.1 The Nature of the Experiments

All programs were run under PAN using SICStus Prolog 3.5 on a variety of heterogeneous Sun, Dec and Silicon Graphics workstations on the same LAN. The PAN session consisted of engines with different processing capabilities distributed over three different sub-networks of the LAN, that are connected through gateway hosts. Large input sizes were used in most benchmarks to provide long running non-trivial problems to push the controls and the platform to their limits. PAN's communication overheads generate large granularity thresholds, so only large input sizes can illustrate the performance of the proposed mechanisms. Each benchmark was run 20 consecutive times under the same PAN configuration. However, only the best three runs were taken into account. Time was always measured in seconds on the same workstation. Full details can be found in Reference.[61)]

7.2 Granularity Control

The numbers in the following tables represent the performance improvement (PI) due to the use of the granularity controls presented in Section 5.

[*7] The scheduler initiates only when parallel tasks are generated. Otherwise the Prolog engine performs all computations.

Table 1 Quick Sort Granularity Control

	List Input Size			
	750	*1000*	*2000*	*3000*
4	25.3	40.7	45.7	49.3
8	31.0	43.0	46.5	50.6
12	46.8	47.5	53.0	52.3
16	48.0	48.7	53.6	56.0

Table 2 Merge Sort Granularity Control

	List Input Size			
	750	*1000*	*2000*	*3000*
4	−3.9	−1.5	−4.1	−10.8
8	18.2	18.9	22.8	29.8
12	33.3	35.5	37.8	43.9
16	37.2	43.2	46.7	50.9

Given a program, an input goal and a platform configuration, let T_{WC} be the average time of the best three out of twenty consecutive runs, required by PAN to process that goal with the granularity controls enabled. Similarly, let T_{NC} be the average time of the best three out of twenty consecutive runs, required by PAN to process the same goal with the granularity controls disabled. Performance improvement (PI) due to the use of the controls is calculated using the expression $\mathsf{PI} = \frac{\mathsf{T}_{\mathsf{NC}} - \mathsf{T}_{\mathsf{WC}}}{\mathsf{T}_{\mathsf{NC}}} * 100\%$. The same PAN configuration was used in all experiments.

Consider QuickSort (presented in Reference,[44] pp. 56) and Table 1. The controls adapt well to the characteristics of the platform and the nature of the benchmark producing parallel tasks of different sizes. The more engines that participate in a PAN session, the more AND-tasks are processed remotely. As a result the controls prove beneficial ranging up to 56%. The controls are able to cope adequately with large test sizes (2000 or 3000 elements) and impose small run-time overheads as well. Debray's model provides a speed up of 3% under the ROLOG[30] system using 4 processors and a speedup of 16.2% under the &-Prolog system also using 4 processors for the benchmark QuickSort(75). Under the PDP system performance does not improve at all when 3 and 15 processors are used. Performance improves by 17.8% only when 8 processors are employed to process QuickSort(700). Garcia's model improves the performance of QuickSort(1000) by 21% running on a hierarchical[*8] implementation of &-Prolog with 4 processors.

Consider the MergeSort benchmark (presented in Reference,[5] pp. 578) and Table 2. In this benchmark the controls perform less adequately (but still effectively) than in Quicksort. When MergeSort is run, the controls perform non-profitable[*9] run-time tests since the input size becomes small enough not to satisfy the granularity tests only in the last recursions. On the other hand,

[*8] Basically it's an &-Prolog implementation with arbitrary overheads added to task creation
[*9] Granularity controls impose overhead without restricting parallelism.

Table 3 Perfect Numbers

	Integer Input Size			
	75	*100*	*300*	*500*
4	17.0	21.5	23.7	24.5
8	6.5	9.5	12.2	15.2
12	8.2	13.2	15.5	17.2
16	14.1	20.1	20.4	21.6

Table 4 Fibonacci Numbers

	Integer Input Size		
	10	*15*	*20*
4	1.3	7.2	15.2
8	7.0	12.8	21.8
12	11.4	18.0	27.4
16	17.0	22.7	31.5

the table indicates that the granularity mechanism is able to improve performance when many engines are being used. When many engines are being used, many parallel candidates are actually processed remotely which balances better the non-profitable (unnecessary) run-time tests. Debray's models improves performance of MergeSort(128) by 14.1% under the ROLOG system using 4 processors. The PDP system also using 4 processors improves the performance of MergeSort(500) by 1.44%

Consider the Perfect Numbers benchmark (presented in Reference[50]) and Table 3. In contrast to QuickSort but similarly to MergeSort, the granularity constraints succeed in most recursions because the input size is larger than the threshold of the granularity constraint. Only in the last few recursions do the input sizes become small enough not to satisfy the granularity tests. In fact, the granularity control mechanism in this case, performs more non-profitable (unnecessary) run-time tests in comparison to MergeSort. Additionally, the overall number of parallel tasks generated by this benchmark is much less than the overall number of parallel tasks generated by MergeSort for similar input sizes. As a result Perfect Numbers performs less adequately, but still effectively as the experiment results indicate. In this particular benchmark, controls provide a best-case performance improvement when PAN employs 4 engines. This is because when there are few engines (and many potential parallel tasks) the scheduler does not allow the controls to test the run-time conditions. Instead tasks are processed locally, without imposing non-profitable overheads. When PAN employs more engines, the controls test more run-time conditions, but in practice they impose non-profitable overheads. Due to the nature of this benchmark, only in the last few recursions do the input sizes become small enough not to satisfy the granularity tests.

The Fibonacci Numbers benchmark (as presented in Reference[5]) and Table 4) is 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. Therefore the controls perform

Table 5 Matrix Multiplication

	NxN Matrix Input Size			
	15	*30*	*45*	*60*
4	16.9	19.7	23.4	27.0
8	16.1	19.2	20.7	28.0
12	21.9	22.4	26.7	33.3
16	23.8	24.5	28.7	36.9

Table 6 Permutations

	List Input Size		
	6	*7*	*8*
4	0.0	−5.9	−3.8
8	0.6	34.1	39.0
12	6.1	34.1	40.8
16	22.7	32.6	42.6

less non-profitable tests. Debray's model provides a performance improvement for fib(15) of 27.3% under the ROLOG system and 29.2% under the &-Prolog system both using 4 processors. This is the only 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.

Consider the Integer Matrix Multiplication benchmark (as presented in Reference,[61] pp. 333) and Table 5. This benchmark generates 4 parallel tasks in a single recursion, which is twice as many as QuickSort. Extra parallel tasks require extra granularity tests, which of course impose extra run-time overhead. Debray tests an 8x8 matrix. But the large granularity threshold of this benchmark in PAN makes the 8x8 matrix multiplication without any practical interest. 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.

Consider the Permutations benchmark (as presented in Reference,[51] pp. 91) and Table 6. Permutations is a typical example of fine-grained OR-parallelism. The clauses generate a search tree with several OR-branches. The proposed model generates coarse-grained parallel tasks that improve performance on distributed platforms like PAN. However, this benchmark does not perform that well when 4 processors are used. The percentage of OR-candidates processed in practice is very small and the effect of granularity controls can not balance the overhead. But performance improves significantly when more processors are used.

The previous example is the core program for other benchmarks like naive N-Queens (presented in Reference,[51] pp. 119). OR-parallelism in N-Queens is generated in a similar way to Permutations and as a result performance figures are very similar. The controls used in the PDP are able to provide a best case performance improvement of 5% (approx.) for queen(8) running on 15

Table 7 Tree Lookup Granularity Control

	Integer Input Size		
	5	*7*	*9*
4	52.5	65.4	77.7
8	57.9	73.1	85.2
12	64.2	77.2	90.4
16	68.6	81.0	93.2

processors.

Consider the Tree Lookup benchmark (presented in Reference,[61] pp. 335) and Table 7. In contrast to previous examples, Tree Lookup may generate more than 2 OR-branches in each recursion (depending on the shape of the tree) and increase the number of potential parallel tasks. 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 to form coarse-grained tasks, which adapt better to the characteristics of the platform. The controls prove very useful when the tree is unbalanced. Debray's model improves performance the performance of tree(8) only by 3% under the ROLOG system using 4 processors.

7.3 Run-time Scheduling

Direct comparison of PAN with parallel Prologs on shared-memory multiprocessors is not always reasonable. They usually perform better than distributed platforms as argued in References.[11,29] It is not always feasible to compare the performance of distributed platforms either, because they have different configurations making it difficult to establish a general and fair comparison metric. The numbers in tables represent the relative performance improvement RPI due to parallel execution in comparison to sequential execution. RPI is calculated using the formula $\mathsf{RPI} = \frac{\mathsf{SE}}{\mathsf{PE}}$. PE is the average parallel execution time (in seconds) of the best 3 out of 20 consecutive runs. SE is the sequential execution time (in seconds). SE is defined as follows:

$$\mathsf{SE} = \left(\sum_{\mathsf{i}=1}^{\mathsf{number\_of\_engines}} \mathsf{SE_i} \right) / \mathsf{number\_of\_engines}$$

SE_i is the average sequential execution time of the best 3 out of 20 consecutive runs, for every engine i that participates in a PAN session. To obtain SE_i, programs were run under pure SICStus without the use and/or intervention of PAN. SE provides a fair comparison metric for this heterogeneous platform since all programs were run on all engines participating in a given PAN session. All benchmarks were run under the same PAN configuration.

7.4 AND-parallel Execution

To illustrate the performance of the model for AND-parallelism the QuickSort program (Table 8), the MergeSort program (Table 9), the Perfect Numbers

Table 8 Quicksort Runtime Scheduling

	List Input Size			
	750	*1000*	*2000*	*3000*
4	2.11	2.18	3.09	3.68
8	2.71	3.27	4.24	4.50
12	3.07	3.66	4.87	5.22
16	3.64	4.44	5.65	5.88

Table 9 MergeSort Runtime Scheduling

	List Input Size			
	750	*1000*	*2000*	*3000*
4	1.57	1.84	2.23	2.59
8	2.09	2.21	2.61	3.12
12	2.53	3.07	4.11	4.57
16	2.71	3.81	4.49	5.07

Table 10 Perfect Nos. Runtime Sched

	Integer Input Size		
	100	*300*	*500*
4	2.159	3.045	3.828
8	4.849	6.372	7.090
12	6.150	9.220	10.195
16	7.050	11.562	12.887

Table 11 NxN Matrix Runtime Sched

	NxN Matrix Input Size		
	30	*45*	*60*
4	1.663	1.629	1.657
8	1.720	1.678	1.740
12	1.859	1.816	1.941
16	2.146	2.122	2.414

program program (Table 10) and the Big Integer Matrix Multiplication (Table 11) were run under PAN.

The Matrix Multiplication program generates 4 medium-grained parallel tasks on each recursion. However this program generates AND-tasks very fast (in the sense that these AND-tasks have no LHS goals to delay their generation) and much faster than PAN can effectively handle them (in the sense that the distribution rate of PAN is much lower than the generation rate of AND-tasks). This incurs extra run-time overheads because it schedules potential parallel tasks, which are processed locally instead. The overhead imposed by the scheduler in addition to the considerable communication cost, can not be balanced as effectively as with Quicksort by the gains of parallel execution. Several parallel tasks are not actually processed in parallel but wait locally for an engine to become available. Such programs perform better on shared-memory multiprocessors. In contrast, the rate that the QuickSort and MergeSort programs generate coarse-grained parallel tasks is reasonably close to the rate that

PAN can effectively process them. As a result the scheduler imposes less overhead in the latter case, because more potential AND-tasks are actually processed remotely.

The Perfect Numbers program provides the best speed up, indicating that for non-trivial and coarse-grained applications this model distributes tasks effectively to engines while controlling the communication overheads and exploiting good degrees of parallelism. Especially when the rate of generation of AND-tasks is reasonably close to the rate, PAN can process them remotely.

The proposed scheduling scheme is able to improve the performance of PAN as the input size of tasks and the number of engines participating in a PAN session increases indicating that the distributed scheduling components can effectively partition the work load and can also adapt to the changing configuration of the platform. &-Prolog provides a speed-up of 4.9 for QuickSort(1000) running on 10 nodes of a shared-memory multiprocessor. The AND-OR-parallel distributed Prolog executor[54)] improves the performance of QuickSort(2000) by 2.7 on 30 processors and the PDP system improves the performance of QuickSort(700) 2.9 times running on 15 processors. &-Prolog provides a linear speed up of 10 for Matrix(50) running on 10 processors, but distributed platforms like PDP provide a speed up of 1.85 for Matrix(75) on 15 processors. Finally the PDP system provides a speed up of 2.6 for MergeSort(500) on 12 processors.

7.5 OR-parallel Execution

Analysis gets more complicated when it comes to OR-parallel execution. Programs like Permutations or naive N-Queens usually do not perform that well under platforms with considerable communication costs as argued in References.[29,51)] Preliminary results showed that PAN is not an exception. The Permutations program and especially the *select/3* goal is the main source of OR-parallelism for other programs like Naive N-Queens. However *select/3* generates OR-parallel tasks very fast (in the sense that the OR-task has no LHS goals to delay its generation) and much faster than PAN can effectively handle them (in the sense that the distribution rate of PAN is much lower than the generation rate of OR-tasks by *select/3*). The overhead imposed by the scheduler and by the OR-interpreter of these tasks in addition to the considerable communication cost, can not be balanced by the gains of parallel execution because most of parallel tasks are not actually processed in parallel but wait locally for an engine to become available. PAN pays the penalty of keeping the control of OR-parallelism at the Prolog level (using an interpreter) instead of pushing most of the control to the WAM level to optimise the OR-parallel execution. However, the flexible configuration of ADEPT, allows PAN to incorporate an improved version of the current OR-interpreter, or even to employ a new one.

For the 10-Queens benchmark OPERA provides a best case performance improvement of (approx.) 2 on 16 processors, but to achieve that the usual WAM-based engine is re-engineered. ROLOG and PDP on the other hand improve performance further, however they are based on an execution model that differs significantly from PAN and a direct comparison would not be fair.

Table 12 OR-Tree Runtime Sched

	Integer Input Size			
	1000	*3000*	*5000*	*7000*
4	1.37	2.74	3.22	3.91
8	3.59	5.85	6.87	7.82
12	6.08	9.06	10.29	10.97
16	6.52	9.80	12.69	14.03

Table 13 Deep Fail Runtime Sched

	Integer Input Size			
	1000	*3000*	*5000*	*7000*
4	1.84	2.63	3.07	3.78
8	2.38	3.68	4.67	5.50
12	3.00	5.26	6.10	7.12
16	3.85	6.36	8.29	9.03

Table 14 AND-OR Parallelism

Eng.	Synthetic-1	Synthetic-2	Synthetic-3
4	2.537	2.032	3.132
8	3.583	3.079	4.180
12	4.302	3.705	5.125
16	4.423	3.819	6.404

Alternative benchmark programs have to be used to illustrate the performance of distributed platforms. The OR-Tree and Deep Fail programs (as presented in Reference,[61] pp. 338, 339) are variations of benchmarks used in the performance analysis of several distributed systems in Reference.[29] Tables 12 and 13 illustrate that PAN can perform adequately for a certain class of non-trivial and large scale applications adapting well to the changing nature of the platform and the characteristics of each program. The main characteristic of this class of applications is that the rate of generation of OR-tasks is reasonably close to the rate that PAN can process them effectively.

7.6 AND-OR-Parallel Execution

Implementation schemes combining AND and OR parallelism typically pay a penalty in the form of a higher control overhead. Table 14 presents the speed up numbers of the *synthetic* benchmarks used in the performance analysis of PDP (Reference[2] pp. 73, 74). They generate AND-under-OR and OR-under-AND parallelism respectively. It indicates that PAN is able to control to a certain extent the extra control overhead. PAN better exploits AND-under-OR parallelism of the synthetic-1 benchmark because it requires fewer task switches between the OR-interpreter and the Prolog engine in comparison to OR-under-AND parallelism generated by the synthetic-2. This indicates that PAN favours the use of the OR-interpreter at the top levels of the execution tree while the tasks in lower levels can be processed either sequentially or in AND-parallel. Synthetic-3 is a variation of synthetic-1 which generates twice as many AND

and OR tasks.

For the synthetic-1 benchmark PDP provides a speed up of up to 4.5, and for the synthetic-2 benchmark a speed up of up to 4.6. The latter benchmark performs better when it runs under the PDP system because OR-parallel execution is realised by extending the WAM, which imposes no task switches between the Prolog engine and the OR-mechanism which is the case in PAN. But PAN also performs reasonably well using mainstream Prolog technology. As the scale of parallelism grows (synthetic-3) performance improves indicating that PAN performs better running large scale applications.

§8 Conclusion

PAN combines SICStus Prolog, PVM and Tcl/Tk technology to create a message passing parallel system running on a virtual multiprocessor under a script controlled X Window interface. To exploit parallelism efficiently PAN uses suitable compile-time techniques to detect potential parallel tasks. Determinacy, Freeness and Side-Effects Flow analyses impose sequentiality constraints to maintain standard Prolog semantics. The use of strict and well informed granularity controls improve the performance of the platform by coalescing fine-grained tasks to form coarse-grained goals adding little (if any) overhead. Tasks are distributed to engines using a flexible task-driven hierarchy of distributed scheduling components. The performance results show that PAN performs better on large scale and non-trivial applications rather than fine-grained parallel tasks.

References

1) Ali, K. A. M. and Karlsson, R., Scheduling OR-parallelism in MUSE. in *Proc. of 8th Int. Conf. on Logic Prog.* (Furukawa, K., ed.), pp. 807–821, 1991.

2) Araujo, J. and Ruz, J.J., "A Parallel Prolog System for Distributed Memory," *Int. Journal of Logic Prog., 33, 1*, pp. 49-79, 1997.

3) Baron, U., de Kergommeaux J.C., Hailperin M., Ratcliffe M., Robert P., Syre J.C., and Westphal H., "The parallel ECRC Prolog system PEPSys: An overview and evaluation results," in *Int. Conf. on FCGS* (ICOT, ed.), pp. 841–850, 1988.

4) Beaumont, T. and Warren, D. H. D., "Scheduling Speculative Work in OR-parallel Prolog Systems," in *Proc. of 10th Int. Conf. on Logic Prog.* (Warren, D. S., ed.), pp. 135–149, 1993.

5) Bratko, I., *Prolog: Prog. for Art. Intelligence,* 2nd ed. Addison Wesley, 1991.

6) Briat, J., Favre, M., Geyer, C., and de Kergommeaux, J. C., "OPERA: Or-parallel Prolog system on Supernode," *Implementations of Distributed Prolog* (Kacsuk, P. and Wise M. J, eds.), John Wiley, pp. 45–64, 1992.

7) Bruynooghe M., Deomoen N., Boulanger D., Denecker M., and Mulkers A., "A Freeness and Sharing Analysis of Logic Programs Based on Pre-interpretation," in *Proc. of 3rd Int. Symp. on Static Analysis* (R. Cousot and D. A. Schmidt, eds.), pp. 128–142, 1996.

8) Carlton M. and Van Roy P., *A Distributed Prolog System with AND-Parallelism*, IEEE Software, pp. 43–51, Jan., 1988.

9) Clocksin W. F., "The Delphi Multiprocessor Inference Machine," in *Proc. of ICSLP '92–Work. on Concurrent and Parallel Implementations* (Apt, K., ed.), 1992.

10) Costa, V. S., Warren, D. H. D., and Yang, R., "The Andorra-I Preprocessor Full Prolog on the Basic Andorra Model," in *Proc. of 8th Int. Conf. on Logic Prog.* (Furukawa, K., ed.), pp. 599–613, 1991.

11) Coulouris, G. F., and Dollimore, J., *Distributed Systems: Concepts and Design, 2nd ed.*, Addison Wesley, 1994.

12) Cunha, J. C., Medeiros, P. D., Carvalhosa, M. B., and Pereira, L. M., "Delta-Prolog: A Distributed Logic Programming Language and its Implementation on Distributed Memory Multiprocessors," *Implementations of Distributed Prolog* (Kacsuk, P. and Wise, M. J., eds.), John Wiley, pp. 335–356, 1992.

13) Dutra I., "Strategies for Scheduling AND and OR Parallel Work in Parallel Logic Programming Systems," in in *Proc. of 1994 Int. Symp. of Logic Prog.* (M. Bruynooghe, ed.), pp. 289–304, 1994.

14) Debray, S. K., Garcia, P. L., Hermenegildo, M. V. and Lin N. W., "Estimating the Computational Cost of Logic Programs," in *Proc. of Static Analysis Symp. (Charlier, B. L., ed.), 1994*, pp. 255–265, 1994.

15) Debray, S. K. and Lin, N., "Cost Analysis of Logic Programs," *TOPLAS, 15, 5*, pp. 826–875, 1993.

16) Debray, S. K., Lin, N. W. and Hermenegildo, M. V., "Task Granularity Analysis in Logic Programs," in *Proc. of 1990 ACM Conf. on Prog. Lang. Design and Implementation*, pp. 174–188, 1990.

17) DeGroot, D., "Restricted AND-parallelism and Side Effects," in *Proc. of Int. Symp. on Logic Prog.*, pp. 80–89, 1987.

18) Futo, I., "The Real Time Extension of CS-Prolog Professional," *ICLP'94– Work. on Parallel and Data Parallel Execution of Logic Programs* (Barklund, J., Jayaraman, B. and Tanaka, J., eds.), 1994.

19) Garcia, P.L, Hermenegildo, M.V. and Debray, S.K, "A Methodology for Granularity Based Control of Parallelism in Logic Programs," *Journal of Symbolic Computation, 22*, pp. 715–734, 1996.

20) Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V., *PVM User's Guide and Reference Manual*, ORNL, Tennessee, 1995.

21) Gupta, G. and Costa, V. S., "Cuts and Side-effects in AND-OR Parallel Prolog," *Journal of Logic Prog., 27, 1*, 1996.

22) Gupta G, Hermenegildo M, and Costa V., S., "AND-OR Parallel Prolog: Recomputation Based Approach," *New Generation Computing, 1, 3–4*, pp. 770–782, 1993.

23) Hermenegildo, M. V. and Greene, K. J., "The &-Prolog System: Exploiting Independent AND-parallelism," *New Generation Computing, 9, 3–4*, pp. 233–257, 1991.

24) Hermenegildo M. V., Bueno F., Puebla G., and Lopez P., "Program Analysis, Debugging and Optimisation Using the CIAO System Preprocessor," in *Proc. of 1999 IJCSLP* (D. De Schreye, ed.), pp. 52–66, 1999.

25) Hermenegildo, M. V., "An Abstract Machine for Restricted AND-parallel Execution of Logic Programs," *Proc. of 3rd Int. Conf. on Logic Prog.* (Shapiro, E., ed.), pp. 25–40, 1986.

26) Hermenegildo M. V. and Rossi, F., "On the Correctness and Efficiency of Independent AND-parallelism in Logic Programs," in *Proc. of 1989 N. Amer. Conf. on Logic Prog.* (Lusk, E. L. and Overbeek, R. A., eds.), pp. 369–389, 1989.

27) Jacobs D. and A. Langen A., "Accurate and Efficient Approximation of Variable Aliasing in Logic Programs," *Journal of Logic Prog., 13, 2–3*, pp. 291–314, 1992.

28) Kacsuk, P., "OR-parallel Prolog on Distributed Memory Systems," *LNCS, 817*, Springer-Verlag, pp. 543–463, 1994.

29) Kacsuk, P. and Wise, M. J., *Implementations of Distributed Prolog*. John Wiley, Chichester, 1992.

30) Kale, L. V. and Ramkumar, B., "The REDUCE/OR Process Model for Parallel Logic Programming on Non-shared Memory Machines," *Implementations of Distributed Prolog* (Kacsuk, P. and Wise, M. J., eds.), John Wiley, pp. 187–212, 1992.

31) Kaplan, S., "Algorithmic Complexity of Logic Programs," in *Proc. of 5th Int. Conf. and Symp. on Logic Prog.* (Kowalski, R. and Bowen K, eds.), pp. 780–793, 1988.

32) King A., Shen K., and Benoy F., "Lower-bound Time-complexity Analysis of Logic Programs," in *Int. Symp. on Logic Prog.* (J. Maluszynski, ed.), pp. 261–276, 1997.

33) King, A. and Soper, P., "Heuristics, Thresholding and a New Technique for Controlling the Granularity of Concurrent Logic Programs," Tech. Rep. CSTR 92-08, Dept. of Electronics and Computer Science–Southampton Univ., 1992.

34) King A. and Soper P., "Depth-k Sharing and Freeness," in *Int. Conf. on Logic Prog. 1994* (P. Van Hentenryck, ed.), pp. 553–568, 1994.

35) Lusk, E., Warren, D. H. D. and Haridi, S., "The Aurora OR-parallel System," *New Generation Computing, 7, 2–3*, pp. 243–271, 1990.

36) Morel, E., Briat, J., de Kergommeaux, J. C. and Geyer, C., "Side-effects in PloSys Or-parallel Prolog on Distributed Memory Machines," *ICSLP'96-Compulog Net Meeting* (Maher, M. J., ed.), Bonn, 1996.

37) Mulkers A., Simoens W., Janssens G. and Bruynooghe M., "On the Practicality of Abstract Equation Systems," in *Proc. of 12th Int. Conf. on Logic Prog.* (L. Sterling, ed.), pp. 781–795, 1995.

38) Muthukumar K. and Hermenegildo M. V., "The CDG, UDG and MEL Methods for Automatic Compile-time Parallelization of Logic Programs for Independent AND-parallelism," in *7th Int. Conf. on Logic Prog.* (D. H. D. Warren and P. Szeredi, eds.), pp. 221–236, 1990.

39) Nelson, R. D. and Squillante, M. S., "Modeling and Analysis of Task Migration in Shared-memory Multiprocessor Computer Systems," in *Proc. of 4th Int. Work. on MASCOT, Computer Society*, pp. 262–266, 1996.

40) Palmer, D. and Naish, L., "NUA-Prolog, Extension to WAM for Parallel Andorra," in *Proc. of 8th Int. Conf. on Logic Prog.* (Furukawa, K., ed.), pp. 599–613, 1991.

41) Pontelli, E., Gupta, G. and Hermenegildo, M. V., "&-ACE: a High Performance Parallel Prolog System," in *Proc. of Int. Parallel Processing Symp., IEEE Computer Society*, 1995.

42) Sahlin, D., "Determinacy Analysis for Full Prolog," *ACM Symp. on Partial Evaluation and Semantics Based Program Manipulation*, ACM Press, 1991.

43) Shapiro, E., "An OR-parallel Algorithm for Prolog and its FCP Implementation," in *Proc. of 4th Int. Conf. on Logic Prog.* (Lassez, J.L, ed.), pp. 311–337, 1987.

44) Shapiro, E. and Sterling, L., *The Art of Prolog*, MIT Press, 1988.

45) Shen, K., "Improving the Execution of the Dependent AND-parallel Prolog DDAS," *LNCS, 817*, Spring-Verlag, pp. 438–452, 1994.

46) Shen K., Costa V. S., and King A., "A New Metric for Controlling Granularity for Parallel Execution," in *Work. on Parallelism and Implementation Technology for Constraint Logic Prog. Langs.*, 1997.

47) Shen K., Costa V. S., and King A., "Distance, a New Metric for Controlling Granularity for Parallel Execution," *Functional and Logic Prog.*, 1999.

48) Sindaha R.Y., "Branch-level Scheduling in Aurora: The Dharma Scheduler," in *Int. Symp. on Logic Prog.* (D. Miller, ed.), pp. 403–419, 1993.

49) Takeuchi, A., "*Parallel Logic Prog.*," PhD thesis, Univ. of Tokyo, Japan, 1990.

50) Taylor, H., "Assembling a Resolution Multiprocessor from Interface, Programming and Distributed Processing Components," *Computer Languages, 22, 2–3*, pp. 181–192, 1996.

51) Tick, E., *Parallel Logic Prog.*, MIT Press, 1991.

52) Tick, E., "Compile-time Granularity Analysis of Parallel Logic Programming Languages," *New Generation Computing, 7, 2*, 1990.

53) Tick, E. and Zhong, X., "A Compile-time Granularity Analysis Algorithm and its Performance Evaluation," *New Generation Computing, 1, 3–4*, 1993.

54) Verden, A. and Glaser, H., "An AND-OR-parallel Distributed Prolog Executor," *Implementations of Distributed Prolog* (Kacsuk, P. and Wise, M. J., eds.), John Wiley, pp. 143–157, 1992.

55) Warren, D. H. D., "The SRI Model for OR-parallel Execution of Prolog–Abstract Design and Implementation," in *Proc. of Int. Symp. on Logic Prog.* (Warren, D. H. D. and Szeredi, P., eds.), pp. 92–102, 1987.

56) Warren, D. H. D., "The Extended Andorra Model with Implicit Control," *ICLP'90–Work. on Parallel Logic Prog.* (Warren, D. H. D. and Szeredi P, eds.), 1990.

57) Wegbreit, B., "Mechanical program analysis," *CACM, 18, 9*, pp. 528–539, 1975.

58) Winsborough W. and Waern A., "Transparent AND-parallelism in the Presence of Shared Free Variables," in *5th Int. Conf. and Symp. on Logic Prog.* (R. Kowalski and K. Bowen, eds.), pp. 749–764, 1988.

59) Wise, M. J., "Experience with PMS-Prolog," *Software Practice and Experience, 22, 2*, pp. 151–175, 1993.

60) Xirogiannis, G., "Compile-time Analysis of Freeness and Side-effects for Distributed Execution of Prolog Programs," in *Proc. of 6th Hellenic Conf. on Informatics* (Sellis, T. and Pagkalos, G., eds.), pp. 701–722, 1997.

61) Xirogiannis, G., "*Execution of Prolog by Transformations on Distributed Memory Multi-Processors*," *PhD thesis*, Heriot-Watt Univ., Edinburgh, 1998.

62) Xirogiannis, G., "Granularity Control for Distributed Execution of Logic Programs," in *Proc. of 18th Int. Conf. on Distributed Computing Systems* (Papazoglou, M. P., Takizawa, M., Kramer, B. and Chanson, S., eds.), pp. 230–237, 1998.

63) Xirogiannis, G., and Taylor, H., "A Dynamic Task Distribution and Engine Allocation Strategy for Distributed Execution of Logic Programs," in *Proc. of 1998 Int. Conf. on High-Performance Computing & Networking* (Sloot, P., Bubak, M. and Hertzerger, B., eds.), pp. 294–304, 1998.

64) Yang, R., Beaumont, T., Dutra, I., Costa, V. S. and Warren, D. H. D., "Performance of the Compiler-based Andorra-I System," in *Proc. of the Tenth International Conference on Logic Programming* (David S. Warren, ed.), pp. 150–166, Budapest, Hungary, 1993. The MIT Press.

George Xirogiannis, Ph.D.: He received his B.S. in Mathematics from the University of Ioannina, Greece in 1993, his M.S. in Artificial Intelligence from the University of Bristol in 1994 and his Ph.D. in Computer Science from Heriot-Watt University, Edinburgh in 1998. His Ph.D. thesis concerns the automated execution of Prolog on distributed heterogeneous multi-processors. His research interests have progressed from knowledge-based systems to distributed logic programming and data mining. Currently, he is working as a senior IT consultant at Pricewaterhouse Coopers. He is also a Research Associate at the National Technical University of Athens, researching in knowledge and data mining.

Hamish Taylor, Ph.D.: He is a lecturer in Computer Science in the Computing and Electrical Engineering Department of Heriot-Watt University in Edinburgh. He received M.A. and MLitt degrees in philosophy from Cambridge University and an M.S. and a Ph.D. degree in computer science from Heriot-Watt University, Scotland. Since 1985 he has worked on research projects concerned with implementing concurrent logic programming languages, developing formal models for automated reasoning, performance modelling parallel relational database systems, and visualisizing resources in shared web caches. His current research interests are in applications of collaborative virtual environments, parallel logic programming and networked computing technologies.