PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Self-organizing Task Scheduling for Parallel Execution of Logic Programs

Zheng Lin*

Department of Computer Science
University of Maryland
College Park, Maryland 20742
zlin@cs.umd.edu

Abstract

A new scheduling scheme is proposed which directs processors to share the search space according to universal task distribution rules obeyed by all processors involved. Load balancing is achieved by altering the shape of a search tree to remove the so-called structural imbalance, and following a statistically even distribution rule. A condition for task distribution is derived which minimizes the average parallel runtime. We present data showing the effectiveness of the proposed scheme. Simulation results from benchmark programs that can be found in literature demonstrate that the method is able to efficiently treat programs that render mostly fine-grained parallel tasks under a typical existing scheduler. The peak speed-up factors with the proposed technique exceed by a substantial margin that achieved by Aurora Parallel Prolog on the same set of benchmarks.

Key Words: Efficiency, Logic programming, Load balancing, Parallel execution, Scheduling, Speed-up.

1 Introduction

Load balancing is the key to obtaining maximum utilization of a multiprocessor system. Parallel execution of a logic program creates many tasks that need to be assigned to processors at run time. Detecting available tasks at run time and migrating tasks among processors is expensive. This is particularly acute for systems in which communication overhead is high due either to architectural reasons, or to a large number of processors being used, because a traditional task scheduler relies heavily on shared resources, shared memory or interconnection network, to perform its functions. As the scale of a multiprocessor system grows, and the speed of implementing resolution in local processor improves[1], task scheduling becomes increasingly frequent. However, the speed of the scheduler cannot be expected to increase proportionally if the scheduler continues to operate on resources shared by all processors. This motivates us to search for schemes that are less reliant on resources subject to competition by all processors in a multiprocessor system.

In this paper we discuss a scheduling scheme called *self-organizing scheduling* which directs processors to share the search space, the search tree defined implicitly by a program, according to task distribution rules followed by all processors. We discuss methods, including program restructuring and a new interpretation of so-called choice predicates, that help to alter the shape of the search tree so as to facilitate maintaining load balance with a probabilistic task distribution rule. We derive a condition for task distribution that minimizes the parallel runtime. Experimental data are presented showing the effectiveness of the methods. Empirically, many programs that were frequently used as Or-parallelism benchmarks in the literature can be restructured to effectively exploit the advantage provided by the proposed scheduling method. For problems with fine-grained parallelism (e.g. a tightly written 8-queens, zebra, turtles programs, running on 30 or more processors) whose speed-up factors reach peaks at less than 30 processors on a typical Or-parallel Prolog system, we found that the peak speed-up factors can be doubled or tripled using the self-organizing scheduling method *even without* resorting to communication.

The paper is organized as follows: section 2 provides background on parallel execution of logic programs; section 3 discusses the proposed methods; section 4 presents the experimental results, and comparison with existing systems; section 5 discusses advantage and limitation of the proposed method, and possible solutions; section 6 describes related work; section 7 concludes our work.

2 Background

We consider a logic program to be a set of Horn clauses written as,

$$H : -B_1, B_2, ..., B_n$$

*This work is supported by AFOSR grant AFOSR-91-0350 and NSF grand IRI-89-16059.

[1]Speed of sequential Prolog implementation has been improved drastically over the past several years. New developments have been reported [VR90] which could lead to improvement in speed in the order of several times that of the current best Prolog implementations.

where H, the head of the clause, is a positive literal and the B_i's, the body of the clause, are conjunction of either positive of negated literals (possibly empty). The intuitive interpretation of the above rule is *if all B_i's are solved then H is considered solved.*

A query is written as : $-Q$, where Q is a conjunction of literals. Evaluation of Q starts with clause : $-Q$, using resolution [Lloyd84] to derive an empty clause should one exist. There may be multiple selection of rules at each resolution step. All solutions can be found by exhausting every possible alternative in the program. The resolution process can be visualized as the construction of a search tree (backtracking tree, proof tree) for the given query. Given a program and a query, the tree is implicitly defined.

We define a *partition* of the tree as a part of the tree that consists of a set of nodes reachable from the root of the tree. Two partition are *disjoint* if there is no common leaf node in the partitions. We note that a *partition* always contains a path from the root.

2.1 Or-Parallel Execution of a Logic Program

Or-parallel execution of a logic program can be viewed as having multiple processors (resolution engine, workers) simultaneously exploring different parts of a search tree defined implicitly by the program. Execution starts with the original goal sent to one of the workers. The goal is expanded by resolving one of its atoms (the leftmost one in the case of Prolog) with clauses which have matching heads. If more then one potential subgoal is generated, and if there are idle workers, the extra subgoals are made available to the idle workers. Any unsolved subgoal that remains is solved upon backtracking. The procedure repeats until all workers finish their tasks. In this paper, we are concerned only with the situation in which the tree is finite and all solutions need to be found. In other words, the entire search tree is explored.

Task scheduling consists of searching for available tasks (or processors) and transferring a task. Transferring a task from one processor to another means migrating the state (variable bindings, control information, etc.) of one processor corresponding to the task to another processor. Different execution models handle task migration differently [Ali90, But88, Mud91, Kale85, Lusk, Clock88, Giul90], with the objective of balancing load distribution with as little communication as possible. A common characteristic of existing methods is that processors cope with the dynamically changing search space by interchanging messages to detect where a task is available and migrate to the task. While this approach has an obvious advantage of automatically adapting to the shape of the search tree, the overhead of scheduling can be unnecessarily high especially for fine-grained tasks. This will become clear when performance data is presented from a typical Or-parallel system later in this paper.

With increasingly fast implementation of sequential resolution engines, and even larger scale multiprocessor systems available, the issue of scheduling has added a new element of how to keep up with the speed of the resolution engine which operates primarily on local and private resources. The computing power of fast local resolution engines can be utilized fully *only when the scheduler is able to allocate tasks for them efficiently.*

We investigate a method that divides the search space and coordinates the search by following universal rules rather than via communication among processors. We describe the method and present performance results in following sections.

3 Self-organizing Scheduling

The idea of the method is to allow each processor to decide, locally, a partition (defined in above section) in the search tree to explore, according to universal rules agreed on by the whole system. It works as follows: every processor obtains a copy of the original goal (the root of the search tree), and performs a depth-first search on the tree. At each node, a processor expands *all* children of the node and claims those belonging to it according to rules agreed on by every other processor in the system, then processes them *independently.* The decision of which path(s) to pursue is made locally by each processor. No dialogue among processors is necessary until the first processor completes the task it claims.

An ideal situation would be that each processor obtains a partition of equal size. However, this is unlikely unless the granularity of a task is predictable. We propose a program restructuring method which alters the shape of the search tree so as to facilitate a probabilist task distribution rule, which will be discussed later.

Or-parallel branches in the search tree are created by the selected literal (for expansion) unifying the heads of multiple rules. Imbalances of the tree are results of either 1) terminated branches (cut-offs) or 2) syntactic characteristic of the program which results in an imbalanced search tree, as will be referred to as *structural imbalance* in the rest of the paper.

An important class of programs written in a logic programming language are the generate-and-test programs, where the generating phrase produces candidates, stored in a structure, and the testing phrase retrieves and tests a candidate from the structure. Generating and testing can be interwaiving. While we do not know yet how to speculate on cut-offs, structural imbalances can be cured by changing the way the candidates are retrieved.

To illustrate the idea, we examine a Prolog predicate, the *member* predicate. This predicate, and its variation, can be found in many normal style generate-and-test programs as a mean to create alternative choices. The predicate is usually defined as,

```
member(X,[X|Y]).
member(X,[H|Y]) :- member(X, Y).
```

Given a list as the second argument, *member* returns an element from the list in the first argument of the predicate. All element can be retrieved eventually by exhausting, recursively, all alternatives.

Predicates which represent multiple choices in the resolution are refered to as *choice* predicates, as oppose to *determinate* predicates which has only one valid choice. The *member* predicate defined above is a choice predicate when called with an instantiated second argument and uninstantiated first argument. Notice that whether or not a predicate is a choice predicate is contingent on not only the way it is written but also the argument pattern it is called with. A recursive choice predicate and a recursive determinate predicate is not distinguishable syntactically in Prolog. We assume choice predicate are explicitly identified with annotation suplied by users. This assumption is consistent with practice in many existing parallel Prolog systems[But88, Ali90], which require explicitly distinction between predicates to be evaluated sequentially or in parallel.

At run time the normal style *member* predicate defined above produces a search tree "biased" to the right: the left child of a node in the tree corresponds to the first rule and the right subtree of a node corresponds to the second recursive rule of the definition. When this predicate is embeded in a program, a left branch so generated represents **one** element of the given list to be processed, and a right branch represents **the rest** elements to be processed. *The difference cannot be observed by the resolution engines being at the parent node of the branches.* Furthermore, the degree of bias is magnified if the predicate is called from inside a loop.

3.1 Flattening Choice Predicates

Program Restructuring: Retrieving members of a given structure can be written in a non-recursive form. For instance, the *member* predicate can be defined as,

```
member(X,[X|Y]).
member(X,[_,X|Y]).
member(X,[_,_,X|Y]).
member(X,[_,_,_,X|Y]).
member(X,[_,_,_,_,X|Y]).
member(X,[_,_,_,_,_,X|Y]).
member(X,[_,_,_,_,_,_,X|Y]).
```

if the number of elements the predicate will be called with is known at compile time. Otherwise a recursive rule has to be added to ensure the correctness of the definition,

```
member(X,[_,_,_,_,_,_,_|Y]) :- member(X,Y).
```

We consider this approach a partial solution to the problem because it is not sufficient for all programs in general. It is a useful program pre-processing technique until a new interpreter is built that takes care of general cases as suggested below.

Flattening Choice Predicates at Run Time: We propose that the evaluation of a choice predicate be separated from normal resolution so that choices can be represented in the search tree in a flatten form regardless how the choice predicate is written. A choice predicate is compiled into a special structure distinguishable from the rest of the code and is evaluated at run time separately. We identify such a structure as a *choice graph*. The choice graph is intended to help expand all possible alternatives at run time. Ideally it should also provide mechanism to recognize "bogus" choices, i.e. choices that quickly lead to failure. For this purpose a *guard* can be incorporated to validate an alternative. A predicate will then be defined as

$$Head : -Guard : Body$$

At run time, the *Guard* is evaluated before a branch is actually expanded in the search tree. A choice graph is constructed at compile-time as follows:

- the choice predicate forms a node called the root;
- the right-hand-side of each alternative definition is a child node of the root. There is a directed arc from the root to every child. A child node has two part, the *guard* and the *body*. The body can contain an arc, in position of the recursive call to the choice predicate, leading to the root of the graph, representing recursion. We limit our discussion to direct recursion in this paper.

At run time, a choice predicate is evaluated according to its choice graph. Choices generated by the evaluation become *immediate children to the node at which the choice predicate is calleld.* The tree so generated will be as if the choice predicates in the program were flattened syntactically, achieving the same effect of removing structural imbalances in the search tree while keeping the original program intact. We note that to create a choice branch in the search tree, it is sufficient to evaluate only the guard and predicates positioned to the left of the recursive call (the recursive arc in the choice graph).

3.2 Task Distribution Rules

Effectiveness of the self-organizing scheduling approach lies in whether a balanced load distribution can be obtained. By removing structural imbalance of a program, cut-offs are the only factor that remains causing imbalanced load distribution. Cut-offs exhibit high degree of

Figure 1: Sample Probability Density Functions

Strategy	L_{alloc}	N_{alloc}	C_{rd}
eager-splitting	$log_d\ n$	$< d$	0
lazy-splitting	$log_2\ n$	$(\frac{d}{2})^{log_2\ n}$	$\simeq (\frac{1}{1-2/d}) * (\frac{d}{2})^{log_2\ n}$

Table 1: Comparison of the Two Splitting Strategies. See Text for Further Explanation.

uncertainty, or randomness. Here we investigate task distribution rules that minimizes *average* parallel runtime in theory. In the next section, we study the effectiveness of these rules on benchmark programs.

Until now, we have been using the term task without formally defining it. A task is a sequence of consecutive resolution steps including backtracking performed by a processor. A task is a basic unit of work to be assigned to one or more processor(s). It can be represented by a node or several nodes in the search tree. Task is created dynamically at run time.

We assume that runtime is proportional to the number of nodes traversed. Runtime of a parallel execution is the longest runtime of all processors. In the following discussion, runtime *is* measured by number of nodes traversed, as to simplify the description.

We prove, in the Appendix, the following result:

Theorem: Let N be the number of processors, let m ($\frac{N}{m}$ is an integer) be the number of tasks whose sizes are *statistically identical* and exhibits the following property:

1. the probability density function is non-increasing, or
2. the probability density function is symmetric with respect to a positive central point.

then the average parallel runtime is minimized iff identical number of processors are assigned to each of the tasks.

The conditions in the theorem are satisfied by distribution of shapes illustrated in 1, including, but not limited to, *uniform*, *exponential*, and *normal* distributions.

Statistical identicality of tasks can be guaranteed by enforcing fairness in creating a task, that is, a particular node from a pool of available nodes has equal chance to be included in any task.

Problem remains as to how many tasks are to be created under any particular node. We could create as many tasks as the number of processors being present the node, evenly dividing them among processors, or create only one task, assigning it to all processors. In the former case the search space is divided among processors in the fast possible way. In the latter case the search space is not divided at the current node of the search tree. Redundant computation is incurred, but the ability to adapt to the shape of the search tree can be improved as will be explained later.

Here, we focused on the following task distribution rules, both satisfying the statistical identicality condition:

1. the eager-splitting strategy: at each choice point where m processors are present, assume there is n valid choices. m tasks are created and assigned evenly to m processors. If $n \geq m$, each task contains $\frac{n}{m}$ choices, the left-over choices are randomly included in some of the tasks. If $n < m$, each choice constitute $\frac{m}{n}$ tasks, the rest tasks are formed by randomly picking one choice for each.
2. the lazy-splitting strategy: at each choice point, two tasks are created and assigned to each of the half of the processors. In case of choices being not evenly dividable, left-overs are treated in a way similar to that in the eager-splitting rule.

With the eager-splitting strategy, the search tree is divided among processors in the fastest possible way. The lazy-splitting strategy is the opposite, trading computational overhead for better adaptability.

Assuming that there are $n = 2^k$ processors, and the search tree is balanced and is of degree d (i.e. every node has d branches). Under these conditions, the two task distribution rules are compared in terms of parameters described below:

- allocation level, L_{alloc}: the depth (from the root, level 0) in the search tree where an individual processor commits itself to one or more nodes exclusively.
- number of nodes allocated, N_{alloc}: the number of nodes a processor commits to at the allocation level.
- redundant computation C_{rd}: redundant node expansion compared to the eager-splitting rule, which is considered 0.

Table 1 summarizes results comparing the two splitting strategies. We expect that the eager-splitting strategy minimizes redundant computation, but it is not very adaptive to the shape of the search tree, in the sense that some processors may quickly be out of work due to encountering cut-offs in the tree. This strategy is suitable for a shallow search tree. On the other hand, the lazy-splitting strategy introduces redundant computation but it commits a processor to much more nodes in the search tree compared to the eager-splitting strategy. It is expected to be more adaptive because there are more "alternative" tasks for a processor. For a deep (i.e. the height of the tree is much greater that *log* n) and bushy search tree, the lazy-splitting strategy is expected to perform better since it is more adaptive to the shape of the tree and the redundant computation is relative insignificant in such a case.

Program	Size (res. steps)	Description
9-queens	225926	placing 9 queens such that they cannot attack each other
n-square	77217	testing if all but one of the elements of a square grid can be removed using tic-tac-toe like jumps
patten	50520	testing if certain pattern of a list can be otained
8-queens	47483	placing 8 queens such that they cannot attack each other
tree	22676	traversing a tree generated by pruning branches in a quad-tree randomly with probability set equal to 0.5 The height of the tree is 16
turtles	19678	fitting 9 sqaure pieces into a 3 by 3 board so that certain constraints on matching edges are satistied
zebra	17478	solving the puzzle of who owns the zebra

Table 2: Benchmark Programs

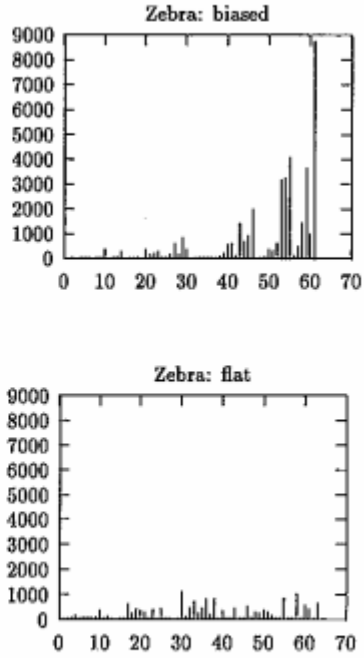

Figure 2: Load Distribution from Running the Zebra Programs on 64 Processors

4 Performance Study

Performance of the self-organize scheduling approach is studied on a set of benchmark programs listed in Table 2. The size of a program is the size of the search constructed during execution of the program. It is the number of resolution steps (logic inferences) during the execution, excluding evaluating Prolog built-in predicates. These programs are pre-processed with the program restructuring method described in the previous section. We note that there is no significant performance changes due to the restructuring in any of the benchmarks running with Sicstus Prolog 0.6.

4.1 Load Distribution

First, we are interested in how effectively the task distribution rules can balance the load, with structural imbalance in a program removed. We defined *balance factor* as,

$$B = \frac{\frac{1}{n}\sum_i^n T_i}{Max(T_i)}$$

where T_i is the total number of nodes in the search tree allocated to processor i, n is the total number of processors. A better balanced load distribution will be reflected in a larger B value. The load balance factor is similar to the efficiency factor e used in other literature [Kumar87], defined as

$$e = \frac{1}{n}\frac{T}{Max(T_i)}$$

where T is the total number of nodes in the tree. $B = e$ if $\sum_i^n T_i = T$, which in many cases is untrue due to that the load on each processor (measured by the number of nodes it possesses) cannot always be $\frac{T}{n}$, because the search tree may not have sufficiently many branches at any particular moment to keep every processor busy. The notion B tries to reflect a realistic load distribution that is possible under a particular load balancing strategy.

The first set of results shows how the balance factor is improved by eliminating the structural imbalance in a program. These results are obtained by extracting the search tree of a program and then exploring the tree with the self-organizing scheduling rule in a simulation with a uniprocessor machine. The eager-splitting rule is used unless specified otherwise.

Figure 2 shows the difference of load (in term of tree nodes) distribution on 64 processors between two versions of a *zebra* program, one with a regular choice predicate and the other with a flattened choice predicate. Load balancing in vastly improved due to program restructuring. It is generally true that flattening the choice predicate results in a better balanced load distribution, though the improvement varies depending on different programs.

We summarize the result by presenting the curves of balance factor for several other benchmarks, shown in Figure 3. The eager-splitting rule is used in this experiment. As can be seen, the balance factor is significantly improved for all but the *n-square* and *tree* programs, which have a deep and bushy search tree that cannot be sufficiently taken care of by the eager-splitting rule. The *n-square* program, and the *tree* program were run

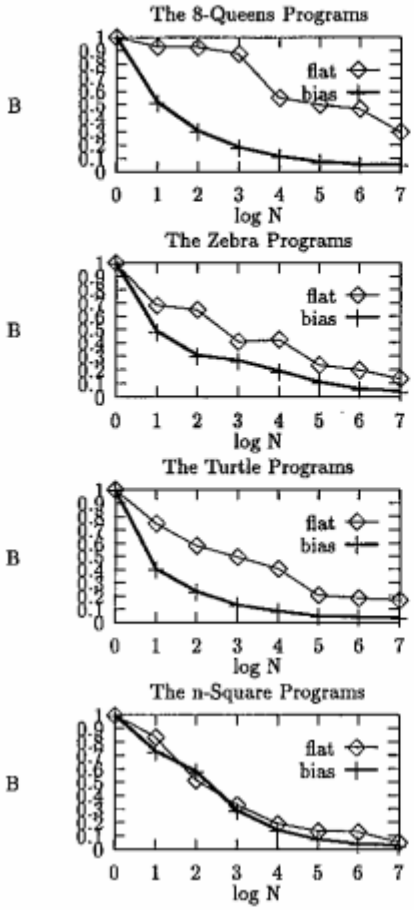

Figure 3: Comparison of Balance Factors (B) between Programs with Flattened Choice Predicates (labeled 'flat' in the figure) and with Normal Style Choice Predicates (labeled 'bias' in the figure)

with the lazy-splitting rule. Results are given Fig. 4. The balance factor is substantially improved (i.e > 100% with 128 processors) since the lazy-splitting is better in coping with irregular shaped tree. However, the overhead of redundant computation makes the lazy-splitting rule unsuitable for a shallow search tree such as that of the *8-queens*, the *zebra*, or the *turtles* program. The height of the search trees for these programs is not sufficiently larger than $log(128)$, the level at which each of the 128 processors commits to its own tasks.

4.2 Speed-up Factors

Speed-up factor is defined as sequential runtime divided by the parallel run time. It is a generally accepted indication of how well a parallel system is able to improve the runtime of a program. Next, we present data showing speed-up factors of the proposed approach on the selected benchmark programs.

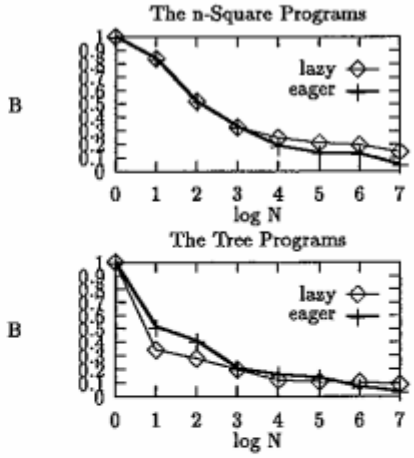

Figure 4: Comparison of the Eager-splitting Rule (labeled 'eager' in the figure) and the Lazy-splitting Rule (labeled 'lazy' in the figure), With Flattened Choice Predicates in Both Programs

Table 3 lists speed-up factors from a simulation study running on a uniprocessors. In this simulation, the run time is measured by the number of resolutions performed in the execution (number of nodes traversed in the proof tree).

Proc. / Prog.	4	8	16	32	64	128
	Eager-splitting					
8-queens	3.9	7.5	9.4	17.0	31.9	**40.1**
9-queens	2.9	4.5	8.6	16.7	22.7	**42.2**
zebra	3.2	4.0	8.3	9.1	15.3	**20.6**
turtles	3.0	5.2	8.6	8.6	15.3	**27.6**
pattern	2.8	5.5	6.2	12.5	21.7	**21.7**
n-square	2.6	2.8	3.2	3.7	3.7	**6.7**
tree	2.4	2.4	3.7	6.5	6.8	**6.8**
	Lazy-splitting					
n-square	2.2	2.8	4.3	7.2	13.0	**17.7**
tree	1.6	2.3	2.7	4.9	9.0	**14.2**

Table 3: Speed-up from Simulation Study. Speed-up is defined as sequential runtime divided by parallel runtime.

Table 4 lists speed-up factors from a parallel emulation study running on a BBN Butterfly TC2000 with 32 processors. The run time is measured by the physical clock. We assume that each resolution step takes constant time. Cost of a real resolution step varies in general. However, here we are merely interested in the total time of a task which consists of a large number of resolution steps. The total time (the sum of the time by all resolution steps) can be considered as the *average cost of each resolution step* times the *number of resolutions*. In other words, the difference of time spent on each resolution step is immaterial. For a given program, the constant can be regarded as the *average cost* of each resolution step.

In order to observe the real overhead of task allocation, which is the time to compute the partition of tasks, the resolution speed must be realistic. In the emulation, resolution engine speed is set equal to that of Aurora Parallel Prolog[2], a well known parallel Prolog implementation, running on one Butterfly processor. Both the eager and the lazy scheduling strategies are implemented in the emulator. The eager-splitting rule was used for the programs *n-queens*, *zebra patten* and *turtles*. The lazy-splitting rule was used for the programs *n-square* and *tree*. From the emulation study, we are able to verified that the sequential simulation, which measures run time by the number of resolution steps performed, accurately reflects the speed-up result by the parallel emulation, which measures run time by the real clock, for up to 32 processor. The overhead of calculating the task distribution, the only overhead not considered in the simulation, is nearly invisible in the emulation, given that the speed-up factors are almost identical to that from the sequential simulation. Notice that there is no communication involved here.

Program	1	4 proc	8 proc	16 proc	32 proc
	Eager-splitting				
8-queens	1	4.0	7.6	9.3	16.5
9-queens	1	3.0	4.6	8.4	16.4
zebra	1	3.2	4.0	8.0	8.9
turtles	1	3.1	5.2	8.2	8.2
patten	1	2.8	5.5	6.1	12.1
	Lazy-splitting				
n-square	1	2.2	2.8	4.2	6.9
tree	1	1.6	2.3	2.7	4.6

Table 4: Speed-up from Emulation Study.

4.3 Performance Comparison with Aurora Parallel Prolog

The same set of benchmarks were run with Aurora Parallel Prolog on the Butterfly machine. Runtime and speed-up factors (the best out of 10 runs) are listed in table 5.

The Peak Speed-up Factors: The speed-up curves for all benchmark programs either have reached the peak (bold face numbers) or at least level off with Aurora Parallel Prolog on 32 processors, as shown in Table 5. Using the self-organizing scheduling approach, simulation results (Table 3) on up to 128 processors showed that:

- the peak speed-up factors for the *8-queens*, *zebra* and *turtles* programs (with fine grain parallelism) exceed, by a margin of at least 200%, experimental results on Aurora;
- the peak speed-up factors for the *9-queens* program is twice as that on Aurora;
- the peak speed-up factors for the *n-square* program (with a very bushy search tree) is about 30% faster that that on Aurora.

[2] Aurora 0.6/Foxtrot, patch #8, with the Manchester Scheduler.

Program	1 proc	16 proc	24 proc	32 proc
8-queens	1,620	141/11.5	**122/13.3**	123/13.2
9-queens	7,500	533/14.1	367/20.4	**350/21.4**
zebra	2,600	490/5.3	**500/5.2**	525/4.9
turtles	4,300	**550/7.8**	580/7.4	569/7.5
pattern	1,084	**130/8.3**	160/6.8	240/4.5
n-square	2,230	190/11.7	**170/13.1**	178/12.6

Table 5: Runtime (ms.) / Speed-up factors with Aurora Parallel Prolog

Speed-up Comparison: Given the number of processors, the speed-ups achieved by self-organizing scheduling appears to be comparable to that of Aurora, but somewhat lower when the number of processors is small (e.g < 16). Note that these results are obtained without communication. The same speed-up result is expected to hold regardless of the speed at which resolution engine is running. Therefore, absolute speed comparison will favor the self-organizing scheduling scheme.

5 Discussion

In the above experiment we studied the behavior of the proposed technique without communication among processors. We demonstrated that the scheme is able to effectively deal with problems which render mostly fine-grained parallel tasks under a traditional scheduler. The loss of processor utilization due to the unevenness in load distribution can be more than covered by the benefit of reduced scheduling overhead. The advantage of the proposed technique is its non-communicating nature, as frees it from possible constraints such as communication bandwidth among processors that could otherwise limit the ability of a scheduler to function effectively. The limitation, however, is its unable to re-use processors that complete tasks they allocated before the termination of the (parallel) execution. We have shown, in the above simulation study, that this would not necessarily compromise performance of programs specially those that generate mostly fine-grained tasks at run time under a traditional scheduler. But the worse case scenario could happen despite the effort to obtain a better balanced load distribution by removing structural imbalance of the search tree and using a statistically even distribution rule. Below, we discuss options to deal with the problem.

One possible solution to the problem is to resort to dynamic task redistribution as existing schedulers do. As we know, the overhead of dynamic task redistribution is relatively small for medium to large-grained tasks, and it provides us with the adaptiveness necessary to deal

with some extraordinary shape search space. On the other hand, the self-organizing scheduling approach introduces low overhead and thus ensures that when it does not help improve performance it is not expect degrade it either. When the two methods are careful integrated, it can be a combination that takes advantage of what the two methods are best at. The issue is when and how dynamic task redistribution should be invoked to achieve the best result. Preliminary research has been conducted in this direction and we will present results in a separate paper. Another option that alleviates the problem is to have idle processors collected by a higher level scheduler (e.g. the operating system) and assigned to other queries. The idea is to use dynamic scheduling only at the level of user queries which usually offer larger granule. In a multi-user environment, this approach can yield a high system throughput given sufficient queries. Global load balancing is involved here. It appears an interesting subject for future investigation.

Static program analysis that provides probability of cut-offs according to given query patterns will be very helpful to guide task distribution. More research is yet to be done before this becomes a feasible alternative to the currently used statistical distribution rule.

Finally, we note that an interesting feature of the self-organizing scheduling approach is that it establishes linkage between processor mapping and the syntax of a program. This feature provides user a mean to influence the mapping of processors to tasks, as would be particularly helpful for applications in which tasks are clearly defined and dynamic task redistribution is known to be not beneficial (there are many such applications). Again, dynamic task redistribution can be used to guard against abuse of this feature.

6 Conclusion and Future Work

A task scheduling technique, self-organizing scheduling, is proposed in this paper. The method directs processors to share the search space, a search tree defined implicitly by the program, according to universal rules followed by every processor in the system. Load balance is achieved by altering the shape of the search tree to remove the so-called structural imbalance (see section 3), and imposing a statistically even task distribution rule to deal with the randomness in cut-offs in the tree. Resolution engines only share the program and the original query. A condition for task distribution that minimizes the average parallel runtime is given and proved. An advantage of the method is that it allows all processors to operate independently on private resources both for resolution and task allocation, while being able to maintain a fairly balanced load distribution among processors. The effectiveness of the self-organizing scheduling scheme is independent of the speed of the resolution engine, and architectural characteristics of the multiprocessor.

We presented data showing the effectiveness of the proposed methods on programs that belong to the generate-and-test category. By removing structural imbalances in a program, it was found that a reasonably balanced load distribution can be obtained by following a statistically even distribution rule. We discussed two distinct task distribution rules, the eager-splitting rule and lazy-splitting rule and examed their effectiveness. We showed that the peak speed-up factors with self-organizing scheduling for a set of benchmark programs exceeds, by a substantial margin, results achieved on the same programs by Aurora Parallel Prolog, a well-known parallel Prolog implementation. Given a fixed number of processors, the speed-up factors by the self-organizing scheduling scheme are competitive. By experimenting with the two near-extreme case task distribution rules we also demonstrated that adaptability can be gained on the cost of redundant computation within this framework.

We believe that the condition for task distribution derived in the paper can be useful for other scheduling schemes. Also, the idea of removing structural imbalances in a program will help with a tree-based scheduler that employs the top-most dispatching strategy [But88, Cald88].

We are currently investigating incorporating traditional task redistribution techniques in order to handle large but highly uneven shaped search trees. Preliminary results indicate that allowing limited communication among processors one can substantially improve the efficiency of the execution. Global load balancing, aimed at maximizing throughput of a system that supports multiple user and multiple queries, is an interesting topic for future research.

References

[Ali90] Ali, K. and Karlsson, R., "The Muse Or-Parallel Prolog Model and its Performance", *Proceeding of the North American Conference of Logic Programming, 1990*, MIT press, 1990.

[Ali91] Ali, K. and Karlsson, R., "Scheduling Or-Parallelism in Muse", *Proceeding of the 8th International Conference on Logic Programming*, MIT Press, 1991.

[But88] Butler, R., Disz, T., Lusk, E., Overbeek, R., and Stevens, R., "Scheduling OR-Parallelism: an Argonne perspective", *Logic Programming, Proceedings of the Fifth International Conference and Symposium on Logic Programming*, MIT press, 1988.

[Cald88] Calderwood, A., Szeredi P., "Scheduling Or-parallelism in Aurora - the Manchester Scheduler", *Proceedings of the Sixth International Conference on*

Logic Programming, pages 419-435, MIT Press, Jun. 1989.

[Clock88] Clocksin, W. F. and Alshawi, H., "A Method for Efficiently Executing Hown Clause Programs Using Multiple Processors", *New Generation Computing*, 5, 1988 p 361-376 OHMSHA, Ltd. and Springer Veriag.

[Giul90] Giuliano, M., Kohli, M., Minker, J., Durand, I., "Prism: A Testbed for Parallel Control", *Parallel Algorithms for Machine Intelligence*, edited by Kanal, L., and Kumar, V., to appear.

[Kale85] Kale, L. V., "Parallel Architectures for Problem Solving", Technical report No. UIUCDCS-R-85-1237, Department of Computer Science, University of Illinois at Urbana-Champaign.

[Kumar87] Kumar V. and Nageshwara Rao V. "Parallel Depth First Search. Part II. Analysis" *International Journal of Parallel Programming*, Vol. 16, No.6, 1987.

[Lloyd84] Lloyd, J. W. "Foundations of Logic Programming", Springer-Verlag, 1984.

[Lusk] Lusk, E., Warren H. D., Haridi, S., *et al.* "The Aurora Or-Parallel Prolog System", Argonne internal technical report.

[Mud91] Mudambi, S., "Performance of Aurora on NUMA Machines", *Proceeding of the 8th International Conference on Logic Programming*, MIT Press, 1991.

[VR90] Van Roy, P. L., "Can Logic Programming Execute as Fast as Imperative Programming", Univ. of California, Berkeley Technical Report UCB/CSD 90/600, Dec., 1990.

Appendix

We prove the following theorem:

Theorem: Let N be the number of processors, let m ($\frac{N}{m}$ is an integer) be the number of tasks whose sizes are *statistically identical* and exhibits the following property:

1. the probability density function is non-increasing, or
2. the probability density function is symmetric with respect to a positive central point.

then the average parallel runtime is minimized iff identical number of processors are assigned to each of the tasks.

Before the proof, we describe some basic terminology and notations to be used.

Capital letters X, Y, Z are used for random variables. The probability density function for X is $f_X(x)$, the cumulative probability distribution function for X is $F_X(x)$, we have $F_X(x) = \int_{-\infty}^{x} f_X(t)dt$ by definition. Or in other words, $f_X(x) = F'_X(x)$. In addition, $f_X(x) \geq 0$ and $0 \leq F_X(x) \leq 1$. $F_X(x)$ is non-decreasing since $f_X(x) \geq 0$.

Runtime of a parallel execution is the longest runtime of all processors. Runtime is measured by the size of a task, in our case, the number of nodes to be traversed in a search tree.

N is the number of processor available. $T_1, T_2, ..., T_m$ are random variables denoting the size of m tasks which are statistically identical, that is, with an identical probability distribution function $f(x)$ and $F(x)$. Let $k_1, k_2, ..., k_m$ be the number of processors assigned to $T_1, ..., T_m$, respectively. $k_1 + k_2 + ... + k_m = N$.

We illustrate the proof with a special case when $m = 2$.

Proof:

Let Z be a random variable denoting the runtime by assigning k_1 to task T_1 and k_2 to task k_2. We assume that T_1 is processed in time $\frac{T_1}{k_1}$ and T_2 is processed in time $\frac{T_2}{k_2}$.

$$Z = max(\frac{T_1}{k_1}, \frac{T_2}{k_2})$$

The cumulative distribution function for Z is $F_z(x)$,

$$\begin{aligned} F_z(x) &= \textit{probability that } Z \leq x \\ &= \textit{probability that } (\frac{T_1}{k_1} \leq x) \textbf{ AND } (\frac{T_2}{k_2} \leq x) \\ &= \textit{probability that } (T_1 \leq k_1 x) \textbf{ AND } (T_2 \leq k_2 x) \\ &= F(k_1 x)F(k_2 x) \end{aligned}$$

Average runtime is the mean of Z,

$$\overline{Z} = \int_{-\infty}^{\infty} (1 - F_z(x))dx$$

We need to show that $\overline{Z}$ is minimized when $k_1 = k_2$, given that $k_1 + k_2 = N$, a constant.

For fixed k_1, k_2, define function $G(x) = \frac{F(k_1 x) + F(k_2 x)}{2}$. We have

$$\int_{-\infty}^{\infty} (1 - F_z(x))dx \geq \int_{-\infty}^{\infty} (1 - G^2(x))dx$$

since $F(k_1 x)F(k_2 x) \leq G^2(x)$, given that $F(x)$ is non-negative. Equality holds when $k_1 = k_2$.

Case I: the probability density function f(x) is non-increasing.

It can be shown that the curve of $F(x)$ is either of an arch shape, or a straight line, as illustrated in figure 5. The curve of $G(x)$ lies below (or on) that of $F(x)$ because the curve of $G(x)$ is composed from center points in lines whose two ends are on curve $F(x)$. $G(x) - F(x) \leq 0$, hence $G^2(x) - F^2(x) = (G(x) - F(x))(G(x) + F(x)) \leq 0$ Therefore,

$$\int_{-\infty}^{\infty} (1 - G^2(x))dx \geq \int_{-\infty}^{\infty} (1 - F^2(x))dx$$

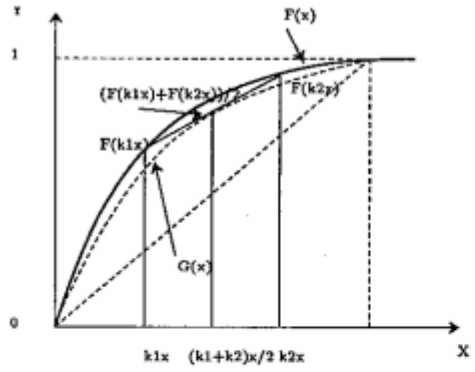

Figure 5: An Arch Shape Distribution

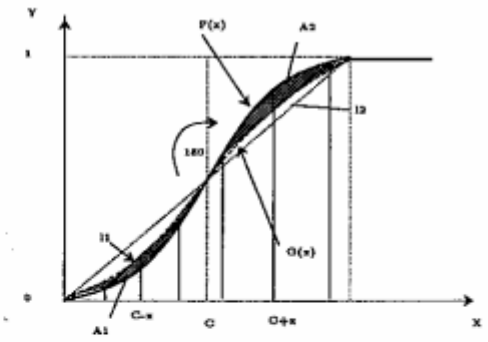

Figure 6: An S Shape Distribution

Equation holds when $k_1 = k_2$.
Thus, we have

$$\int_{-\infty}^{\infty}(1-F_x(x))dx \geq \int_{-\infty}^{\infty}(1-G^2(x))dx \geq \int_{-\infty}^{\infty}(1-F^2(x))dx$$

and equality holds when $k_1 = k_2$. Thus the mean of Z is minimized when $k_1 = k_2$.

Case II: the probability density function f(x) is symmetric with respect to a positive center point, denoted by C.

The curve of $F(x)$ is of the shape an S tilted to the right, as illustrated in figure 6. The curve of $G(x)$ is another S shape curve "contained" in that of $F(x)$. We want to show that

$$\int_{-\infty}^{\infty}(1-G^2(x))dx \geq \int_{-\infty}^{\infty}(1-F^2(x))dx$$

or,

$$\int_{-\infty}^{\infty}(F^2(x)-G^2(x))dx \geq 0$$

This is equivalent to showing

$$\int_{-\infty}^{\infty}(F(x)-G(x))dx \geq 0$$

since

$$\int_{-\infty}^{\infty}(F(x)+G(x))dx > 0$$

Notice that we can no longer have $(F(x)-G(x)) \geq 0$ for all x. However, the integral of (F(x) - G(x)) can still be non-negative if we can proof the shaded areas A_2 is larger or equal to A_1 in figure 6. It suffices to show that for any $(C-x)$ and $(C+x)$ on the X axe, $F(C+x) - G(C+x) \geq G(C-x) - F(C-x)$, and equality holds when $k1 = k2$.

Observe that (C-x,G(C-x)) is the center point of a line, l_1, whose end points are on the curve of $F(x)$. (C+x,G(C+x)) is the center point of another line, l_2, whose end points are on the curve of $F(x)$. Now, rotate the lower part of the S shaped curve of $F(x)$ 180°. The two part of the S matches each other and it can be shown that l_1, after the rotation, completely lies above or on l_2. Thus,

$$F(C+x) - G(C+x) \geq G(C-x) - F(C-x)$$

Equality holds when $k1 = k2$. Proof done for $m = 2$. □

The same idea can be used to prove the general case. A formal proof of the general case will not be presented here, but we note that a property of polygon that is crucial to the proof is that the center of a convex polygon resides *inside* the polygon.

Acknowledgement: I wish to thank Professor Jack Minker for his guidance on this work. Thanks to Dr. Mark Guiliano for his comments on an early draft of this paper. Also, I would like to express my appreciation to Argonne National Laboratory for providing parallel computing facilities.