We describe a simple or-parallel execution algorithm for PROLOG that naturally collects all solutions to a goal. For a large class of programs the algorithm has $O(\log n)$ overhead and exhibits $O(n/(\log n)^2)$ parallel speedup over the standard sequential algorithm. Its constituent parallel processes are independent, and hence the algorithm is suitable for implementation on non-shared-memory parallel computers. The algorithm can be implemented directly in Flat Concurrent PROLOG. We describe a simple interpreter-based FCP implementation of the algorithm, analyze its performance under Logix, and include initial measurements of its speedup on the parallel implementation of FCP. The implementation is easily extended. We show an extension that performs parallel demand-driven search. We define two parallel variants of cut, cut-clause and cut-goal, and describe their implementation. We discuss the execution of the algorithm on a parallel computer, and describe implementations of it that perform centralized and distributed dynamic load balancing. Since the FCP implementation of the algorithm relies on full test unification, the algorithm does not seem to have a similarly natural implementation in GHC or PARLOG.

1. INTRODUCTION

Concurrent PROLOG was developed with the goal of being a general-purpose high-level concurrent programming language. Being well aware of the advantages of PROLOG as a high-level programming language, one of our design goals was to properly include PROLOG as a sublanguage of Concurrent PROLOG. Since the
initial design of the language did not seem to fulfill this design goal, the reported
language was called “A Subset of Concurrent PROLOG” [16]. The same design goal
was shared by PARLOG [3]. Not seeing a method for tightly integrating the capabili-
ties of PROLOG with the concurrent component of PARLOG, the designers simply
defined two sublanguages—the single-solution sublanguage, and the all-solutions
sublanguage—which interact with each other at arm’s length. An interface was
defined which allows a single-solution process to invoke an all-solutions process,
and obtain from it a stream of solutions.

Although applicable to Concurrent PROLOG as well, we have not adopted this
solution. We wanted to view Concurrent PROLOG as a future high-level parallel
machine language. Thus all systems programs, higher-level languages, and applica-
tions should be implemented in terms of its basic computational model. Simply
putting two sublanguages side by side, each with its own properties, machine
requirements, and implementation problems, seemed to us to defer the problem of
identifying a uniform fundamental computational model, rather than to solve it.

Further investigation into the computational model of (the subset of) Concurrent
PROLOG revealed that the initial assumption—that this language is not capable of
implementing PROLOG rather directly—was wrong. Kahn (see [16]) showed how
an or-parallel PROLOG interprete can be easily expressed in Concurrent PROLOG.
His interpreter used deep recursion into or-parallel guards in Concurrent PROLOG
to simulate the parallel search in or-parallel PROLOG. This mechanism exploited
the full unification and multiple-environment mechanism of Concurrent PROLOG,
and hence could not be implemented in PARLOG (or in the more recent language
GHC).

The simplicity of the interpreter attested to the expressive power of Concurrent
PROLOG. However, since it implied that implementing Concurrent PROLOG was
even more complex than implementing or parallel PROLOG, it was a sort of
pragmatic counterexample to the implementability of Concurrent PROLOG, espe-
cially to its multiple-environment mechanism.

Indeed, the main focus of our research turned into Flat Concurrent PROLOG, a
subset of Concurrent PROLOG with simple guards, and we were left with the
original problem: How to embed PROLOG in a machine whose language is Flat
Concurrent PROLOG?

One approach to the problem was developed by Ueda [24]. He showed how,
given sufficient information on the mode of unification of a PROLOG program, it
can be converted into a Flat GHC program. His solution is applicable also to FCP.
Codish and Shapiro [5] describe a different approach to the problem, with similar
results: if sufficient mode information of a PROLOG program can be determined
statically, then it can be transformed into an FCP program.

These results are encouraging, since they seem to give a practical solution for a
large class of PROLOG programs. Nevertheless, they do not solve the entire
problem.

This paper describes an or-parallel execution algorithm for PROLOG that has a
very simple realization in Flat Concurrent PROLOG. Unlike the previous propos-
als, the algorithm does not require mode information. Unlike Kahn’s or-parallel
interpreter in Concurrent PROLOG or the standard all-solutions predicates in
PROLOG, the FCP implementation does not require side effects to collect all
solutions. For a large class of PROLOG programs the algorithm has constant or
logarithmic overhead over the standard sequential execution algorithm, and in these cases its parallel execution with $O(n)$ processors exhibits an $O(n/(\log n)^2)$ speedup over a sequential PROLOG computation that requires $n$ steps. In the worst case its overhead is linear and it shows no speedup over the standard algorithm.

Several similar algorithms have been recently proposed, independently and with a different motivation [1, 2, 4].

These algorithms are different from most proposals for implementing or-parallel PROLOG, which attempt to share as much information as possible between the or-parallel processes. The mainstream approach concentrates on providing an efficient representation of the binding environment to allow sharing when possible and support splitting when necessary [6, 17]. By contrast, in our algorithm, as well as in the similar algorithms mentioned, there is no sharing of environments or logical variables between the constituent parallel processes. As a result, the algorithm is very simple, and can use the standard structure-sharing or structure-copying representation of terms. Furthermore, since the constituent parallel processes are independent, the algorithm is most suitable for implementation on non-shared-memory parallel computers.

The algorithm is simple enough to be implemented rather directly in any conventional language. We estimate the effort in constructing an interpreter-based implementation of the algorithm to be comparable to constructing a standard sequential interpreter for PROLOG. However, the algorithm can be implemented even more easily in Flat Concurrent PROLOG, using read-only unification to implement PROLOG’s unification and goal reduction, and using AND-parallel FCP processes to implement the or-parallel PROLOG search processes. Such an implementation has the advantage of flexibility: it can be extended in many different ways quite easily. It also has the advantage of being embedded in a sophisticated development environment for concurrent programming [18], and in an operating system with powerful tools for implementing process and code mapping algorithms [21], as well as dynamic load balancing.

If this algorithm and its FCP implementation prove to be practical, as our present results suggest, it will solve one of the remaining questions about the general applicability of a Flat Concurrent PROLOG-based system: how to embed PROLOG in it. Since its implementation requires full test unification, I do not see, at present, how to implement it directly in GHC or PARLOG.

2. THE ALGORITHM AND ITS COMPLEXITY

Abstractly, the algorithm searches all paths in the PROLOG search tree of the given goal in parallel. Each path is explored by a separate process that has its own copy of the goal and does not share logical variables with other processes. If the path explored by a process is successful, the process adds its (appropriately instantiated) copy of the goal to the set of solutions. The key problem is how to organize the search so that every path in the search tree is explored exactly once.

Assume that the clauses in the program are indexed. There is a one-to-one correspondence between a path in the PROLOG search tree for a goal, and the list of indices of clauses used in this path. Hence a path in the search tree can be represented by a list of indices.
**The Algorithm.** The algorithm is composed of a dynamically changing set of parallel processes, each with its own resolvent and copy of the goal. Each process is invoked with the resolvent initialized to its copy of the goal and with some prefix of a path in the tree, and is responsible for finding all solutions obtained from paths which extend this prefix. There are two types of processes: a *tracer* and an *explorer*. The algorithm begins by invoking an explorer with the empty prefix.

An explorer operates as follows. If its resolvent is empty, then it returns its instantiated goal as a solution, and halts. If the resolvent is the special constant *fail*, then it halts. Otherwise, it computes the set of indices $C_1, C_2, \ldots, C_n$ of clauses that potentially unify with the left most goal in the resolvent. It picks one clause, say with index $C_1$, and for every remaining alternative clause index $C_i$, $i = 2, \ldots, n$, $n \geq 1$, it spawns a tracer with the prefix $(P.C_i)$ and a new copy of the goal, where $P$ is the prefix explored by the process so far. It reduces the resolvent using the clause with index $C_1$, and iterates with the new resolvent and the prefix $(P.C_1)$.

A tracer reduces its resolvent sequentially, using the clauses specified by its prefix, and then becomes an explorer. If a tracer or an explorer attempts a reduction that fails, it replaces its resolvent by the constant *fail*.

It is easy to see that every path in the search tree is explored by the algorithm exactly once. In addition, observe that only the last reduction of a tracer can fail, and this is handled once the process becomes an explorer.

In the following we analyze the overhead and parallel speedup of the algorithm. To measure the overhead, we compare a standard sequential implementation of PROLOG, with a sequential (timesharing) implementation of the OR-parallel execution algorithm. For this analysis we assume that in a concurrent implementation a deterministic goal reduction can be implemented with the same efficiency as in sequential PROLOG, and that the cost of spawning a process is similar to the cost of allocating a choice point in sequential PROLOG. Both assumptions are substantiated by the performance of FCP implementations [9]. To measure parallel speedup, we assume sufficient processors. If there are not enough processors, then each processor timeshares between several processes; the algorithm's performance with one processor is according to the overhead calculated.

Let $G$ be a goal with a search tree $T$, whose size is $n$, height is $h$, and number of leaves is $l$. Let $p$ be the sum of the lengths of the paths in $T$. From the abstract description of the algorithm it should be evident that the total number of reductions performed by the algorithm is $p$, and that given $l$ processors, the algorithm can compute all solutions in $O(h^2)$ parallel reductions. Since the sequential execution algorithm of PROLOG performs only $O(n)$ reductions (ignoring the overhead of creating choice points and backtracking), the overhead of the parallel algorithm over the sequential one, in terms of the number of reductions performed, is $O(p/n)$. Since the sequential algorithm executes in time $O(n)$, the speedup of the parallel algorithm over the sequential one using $l$ processors is $O(n/h^2)$.

This implies that for thick search tree, i.e., search trees whose height is logarithmic in their size, the overhead of the algorithm is $O(\log n)$, and its parallel speedup is $O(n/(\log n)^2)$. For deterministic (linear) trees both the overhead and the parallel speedup are $O(1)$. The worst performance is obtained for nondeterministic thin trees (nonlinear trees whose height is linear in their size): the overhead is $O(n)$, and the speedup is $O(1)$. 
The parallel-speedup analysis ignores communication costs. These are nil in theoretical shared-memory machine models, such as the PRAM [8]. Below we describe implementations of the algorithm for non-shared-memory models; an analysis of the communication costs of these implementations for concrete non-shared-memory architectures requires further study.

The analysis above ignores the additional work required by the sequential PROLOG algorithm to construct multiple solutions, which may reduce the overhead of the parallel algorithm to $O(1)$ even for thin nondeterministic trees. An example is the goal \text{append}(X, Y, [1, 2, \ldots, n]) whose overhead is $O(1)$, and whose speedup is $O(n^{1/2})$ in case the \text{append} clauses are ordered with the recursive clause first.

The algorithm involves two major operations besides reductions: generating alternative prefixes, and creating new copies of the original goal for every such prefix. A prefix can be generated in time linear in its length. The work to generate it can be apportioned to the first process that explores this prefix, increasing its time by a constant factor.

If the original goal is frozen [11], one can create a new melted copy of a frozen goal $G$ in constant time if structure sharing is used, and in time linear in $\sum \text{depth}(V, G)$, if structure copying is used (where $V$ is a variable occurrence in $G$ and $\text{depth}(V, G)$ is the distance of the occurrence $V$ from the root in the tree representing the term $G$). Most PROLOG programs can be written so that this number is bound by a small constant for all useful top-level goals.

Most PROLOG programs do not perform general unifications of unbounded input data structures. Hence for the majority of programs the number of primitive operations required to perform one reduction can be bound by a constant, which is program-dependent, and thus “reductions” can be replaced by “operations” in the analysis above.

3. AN FCP IMPLEMENTATION

The FCP implementation assumes that the program is accessible via two predicates. One is `clauses(G, Cs)`, which returns the list of indices $Cs$ of the clauses applicable to the reduction of a goal $G$; `clauses` can use sophisticated indexing mechanisms to reduce the size of this list. The other is `clause(C, A, Bs \setminus As)`, which returns in $Bs \setminus As$ a difference list of the goals of the body of the clause $C$, after unifying its head with $A$. If the clause does not unify, then $Bs = \text{fail}$. It also assumes a procedure called `melt(Frozen, Melted)` that can efficiently melt goals for that program.

For example, Program 2 includes the FCP clausal representation of the PROLOG program for generating a permutation, shown in Program 1 of the algorithm. For

```prolog
perm(Xs, [X|Ys]) ←
    select(X, Xs, Xs1), perm(Xs1, Ys).
perm([], []). 
select(X, [X|Xs], Xs).
select(X, [X1|Xs], [X1|Ys]) ←
    select(X, Xs, Ys).
```

PROGRAM 1. A PROLOG permutation program.
the purpose of this paper, the FCP guard predicate `unknown` can be considered similar to PROLOG's `var`.

The FCP implementation of the algorithm is shown in Program 3. It directly matches the description above. The PROLOG proof tree is searched by a pool of FCP processes. Each process is invoked with some prefix, and is responsible for exploring all solutions, whose proof is found by first traversing the prefix. The first process is invoked with the empty prefix. Each process carries a path, a trail, a frozen copy of the goal, a melted copy of it, and an output channel.

There are two types of processes: a tracer and an explorer. The initial process is an explorer with an empty path and trail, a frozen goal, a freshly melted copy of the goal, and a resolvent.

An explorer inspects its resolvent. If its resolvent is `fail` then it halts. If its resolvent is empty, then it sends the instantiated melted goal on its output channel and halts. If its resolvent is not empty, then it computes the set of applicable clauses for the first goal in the resolvent, and becomes `explorel`. If the set of applicable clauses is empty, then `explorel` terminates. Otherwise, it reduces its resolvent using the first clause, adds the clause index to the trail, and forks into two parallel processes: `explorel`, with the updated resolvent and trail, and `explore.rest`, with the rest of the applicable clauses. For every clause index `C` in the list of clause indices, the `explore.rest` process appends `C` to the path traversed so far [represented for efficiency reasons by `append(Path, reverse(Tail))`], and spawns a `trace` process with this path and a freshly melted copy of the goal.

A tracer performs the proof according to the given path, and then becomes an explorer.

Program#Goal is the Logix notation for solving Goal using the axioms in the program module named Program.
solve( Program, FrozenGoal, Sols ) ←

Sols are the provable instances of the melted FrozenGoal.

solve( Program, FrozenGoal, Sols ) ←

Program ≠ melt(FrozenGoal?, Goal),
explore([Goal?], [], [], FrozenGoal? Goal?, Sols \ [], Program).

explore( Resolvent, Path, Trail, FG, G, Sols, P ) ←

Sols are the instances of FG whose proof has a prefix
append(Path, reverse(Trail)), given that Resolvent and G
are the resolvent and the goal obtained by reducing melt(FG)
using the clauses specified by that prefix.

explore(fail, Path, Trail, FG, G, Sols \ Sols, P).
explore([A|As], Path, Trail, FG, G, Sols, P) ←
P ≠ clauses(A?, Cs),
explore1(Cs?, [A|As], Path, Trail, FG, G, Sols, P).

explore1([], As, Path, Trail, FG, G, Sols \ Sols, P).
explore1([C|Cs], [A|As], Path, Trail, FG, G, Sols \ Sols2, P) ←
P ≠ clause(C?, A?, Bs \ As),
explote Bs?, Path, [C|Trail], FG, G, Sols \ Sols1, P,
explote_rest(Cs?, Path, Trail, FG?, Sols1 \ Sols2, P).

explore_rest([], Path, Trail, FG, Sols \ Sols, P).
explore_rest([C|Cs], Path, Trail, FG, Sols \ Sols2, P) ←
append_reverse(Path?, [C|Trail], NewPath),
P ≠ melt(FG?, G),
trace(NewPath?, [G?], NewPath?, FG, G, Sols \ Sols1, P),
explote_rest(Cs?, Path, Trail, FG?, Sols1 \ Sols2, P).

trace(Cs, Resolvent, Path, FG, G, Sols) ←
Sols are the instances of FG whose proof has a prefix Path,
given that Resolvent and G are the resolvent
and the goal obtained by reducing melt(FG)
using the clauses specified by Path \ Cs.

trace([C|Cs], [A|As], Path, FG, G, Sols, P) ←
P ≠ clause(C?, A?, Bs \ As),
trace(Cs?, Bs?, Path, FG, G, Sols, P).
trace([], As, Path, FG, G, Sols, P) ←

append_reverse([X|Xs], Ys, [X|Zs]) ←
append_reverse(Xs?, Ys, Zs).

append_reverse([], Ys, Zs) ←
reverse(Ys?, [], Zs).

reverse([], Ys, Ys).
reverse([X|Xs], Ys, Zs) ←
reverse(Xs?, [X|Ys], Zs).

PROGRAM 3. An or-parallel PROLOG interpreter.
4. PERFORMANCE ANALYSIS OF OVERHEAD AND POTENTIAL PARALLEL SPEEDUP

The interpreter shown is not the most efficient way to implement the algorithm in FCP. The algorithm is best implemented by a compiler, which can either be derived from the interpreter using techniques of partial evaluation [13] or be constructed manually. Furthermore, programs (or subprograms) whose mode of unification can be determined at compile time are better implemented using Ueda's OR-to-AND transformation [24]. Such an implementation can be easily interfaced to our execution algorithm, in case only some self-contained subsets of the program are amenable to such analysis. The construction of a compiler-based implementation of the algorithm is a subject of further research, as well as the execution of the algorithm on the parallel implementation of FCP on the iPSC hypercube [22].

To measure the overhead of the parallel algorithm over the standard sequential execution algorithm for PROLOG, one can compare the uniprocessor performance of the FCP implementation with the performance of a good PROLOG implementation, e.g., Quintus PROLOG. Quite accurate figures can be obtained, if the differences in implementation technologies are taken into account (remember we are trying to compare the algorithms, not their particular implementations). First, the PROLOG programs should be run under a PROLOG meta-interpreter, to factor out the FCP interpretation overhead. Second, the difference between the implementation technologies should be factored out. For example, when comparing Quintus PROLOG with Logix on the same machine, the Logix time should be divided by about 3 or 4, to factor the difference between the highly optimized assembly-language emulator of Quintus, and the C-based emulator of Logix.

An alternative is to run the PROLOG programs under a normal interpreter. We have taken the second approach, and used the C-PROLOG interpreter [12] for benchmarking.

Logix contains a crude mechanism to measure the amount of parallelism available in a computation: the number of reductions performed in one queue cycle. Since several dependent reductions might be performed in a single cycle, this measure should be considered only an upper bound on the available parallelism. However, since most of the parallel processes in our algorithm are independent, the effect of these "hidden pipelines" should be marginal.

We have performed several simple benchmarks, and obtained the following results. C-PROLOG is about three to four times faster on combinatorial problems, such as generating all permutations or the N-queens problem. It is twice as fast for deterministic naive reverse. It is about twice as slow for nondeterministic append.

The amount of parallelism found was abundant. The number of processors and the number of parallel machine cycles required to solve the problem were of the same order of magnitude.

The results are summarized in Appendix 1. We have also measured the actual parallel speedup of a load-balancing variant of the interpreter, explained in Section 8. The results are summarized in Appendix 2.

5. DEMAND-DRIVEN PARALLEL SEARCH

The algorithm as described performs an all-out parallel search for all solutions. Sometimes only a few solutions to a goal are needed, and in some cases the need for additional solutions is determined by the previous solutions.
We describe a variant of the basic algorithm that performs demand-driven search. The modified algorithm is composed of a centralized monitor and several search processes. The monitor maintains two queues: one of unsatisfied requests for solutions, the other of unexplored prefixes. At any time one queue at the most is not empty. As in the classical solutions to demand-driven computations [20], unsatisfied requests are represented by terms containing uninstantiated logical variables, and requests are satisfied by instantiating these variables.

If the monitor receives a request for a new solution, and the prefixes queue is not empty, it dequeues a prefix and starts a search process with the request and the prefix. Otherwise it enqueues the request to the requests queue.

If the monitor receives a new prefix from some of the search processes, and the requests queue is not empty, it dequeues a request and starts a search process with the request and the prefix. Otherwise it enqueues the prefix in the prefixes queue.

The monitor is initialized with the prefixes queue containing the empty prefix. It terminates when all search processes terminate and the prefixes queue is empty.

```prolog
solve(Bootstrap, FrozenGoal, Sols) ←
    stream # merger([path([], merge(Sols?), merge(ToMerger?)], FromMerger),
        monitor(FromMerger?, [], [], ToMerger, FrozenGoal, Bootstrap).

monitor(In, Paths, Sols, ToSelf, FrozenGoal, P) ←
    In is a stream of:
        solution(X) - get another solution,
        path(X) - X is an unexplored path.
    Paths is a list of unexplored paths.
    Sols is a list of unsatisfied requests for solutions.
    ToSelf is a stream to the merger to self.

monitor([solution(Sol) | In?], [Path | Paths], Sols, [merge(ToMonitor) | ToSelf], FG, P) ←
    P # melt(FG, G),
    trace(Path?, [G?], Path, G, Sol, ToMonitor, P),

monitor([solution(Sol) | In?], [], Sols, ToSelf, FG, P) ←
    monitor(In, [], Sols, ToSelf, FG, P).

monitor([path(Path) | In?], Paths, Sols, [merge(ToMonitor?) | ToSelf], FG, P) ←
    P # melt(FG, G),
    trace(Path?, [G?], Path, G, Sol, ToMonitor, P),

trace([C | Cs], [A | As], Path, G, Sol, ToMonitor, P) ←
    P # clause(C?, A?, Bs \ As?),
    trace(Cs?, Bs?, Path?, G, Sol, ToMonitor, P).
trace([], As, Path, G, Sol, ToMonitor, P) ←
    explore([], As, Path, [G, G, ToMonitor, P]).

explore([], Path, Trail, G, G, [], P).
explore(fail, Path, Trail, G, [solution(Sol)], P).
explore([A | As], Path, Trail, G, Sol, [solution(Sol)], P) ←
    P # clauses(A?, Cs),
    explore([Cs? | As], Path, Trail, G, Sol, ToMonitor, P).
```

**PROGRAM 4.** A demand-driven or-parallel PROLOG interpreter.
A search process operates like the tracer and explorer of the basic algorithm, with two major differences. If it reaches the end of a path and the resolvent is empty, it satisfies the request by instantiating it to the current goal and halts. If no solution was found, it returns the unsatisfied request to the monitor and halts. If several extensions to the prefix are found, the process chooses one to explore by itself and returns the remaining prefixes to the monitor.

Note that as long as there are at least $n$ unsatisfied requests and $n$ unexplored prefixes, there will be at least $n$ search processes operating in parallel.

The FCP implementation of the algorithm is shown in Program 4. For simplicity, it does not detect termination of the search processes, and it ends in deadlock rather than in proper termination. Distributed termination can easily be incorporated in it using the standard short-circuit technique [19]. Also, the program maintains stacks rather than queues.

6. PARALLEL CUT

The standard definition of cut is asymmetric and relies on the sequential execution model of PROLOG. Cut still makes sense and can be useful in a parallel context. However, its semantics should be made symmetric, lest its correct implementation should require a considerable reduction in the available parallelism.

To simplify both the definition and implementation of cut, we break its functionality into two control constructs, `cut-clause` and `cut-goal`. Cut-clause, denoted `!c`, cuts alternative clauses, and is a symmetric version of what has been called `soft-cut`. Cut-goal, denoted `!g`, cuts alternative solutions from goals preceding the cut in the clause, and has sometimes been called `snip`. The full effect of (parallel) cut can be achieved by the conjunction of cut-goal and cut-clause: the symmetric version of the standard cut, denoted `!`, is defined to be the result of textually substituting it by the conjunction (`!g`, `!c`).

More precisely, the operational semantics of cut-clause and cut-goal is as follows. Let $G$ be a node in the search tree with path $P$ from the root, with clauses indexed $C_1, C_2, \ldots, C_n$ as immediate extensions. Let $G'$ be a goal in the clause $C_i$. If $G'$ is a cut-clause goal, the execution of $G'$ aborts all computations that search a path that is an extension of $(P, C_j)$, $j \neq i$. If $G'$ is a cut-goal goal, the execution of $G'$ aborts
all computations that search a path that is an extension of \((P.C_i)\) except the one in which \(G'\) participates.

The effect of cut-clause and cut-goal is shown in Figure 1.

Given this operational semantics, one can distinguish between red cuts and green cuts. Green cuts abort computations that may not contribute additional solutions. Red cuts abort computations that include additional solutions. Green cuts do not change the logical semantics of the program, and the or-parallel execution algorithm remains a complete and correct deduction mechanism. Red cuts reduce the completeness of the algorithm. Nevertheless, their use may be justifiable in certain cases on practical grounds. Note that both cut-clause and cut-goal can be either red or green.

Note also that both cut-clause and cut-goal are subject to race conditions, and the time in which a cut is executed determines whether alternative solutions are found or not. Hence, unlike sequential cut, the parallel cuts cannot be used to implement negation as failure, and in this sense are closer to the commit operator of concurrent logic programming languages.

Our implementation of the parallel cuts requires no modification to the interpreter and puts the burden on the program representation. (We do not have an explanation of why this mixture of compilation and interpretation turns out to be the simplest.) We implement the cuts as follows. We associate with every goal an interrupt variable, which is used by the goal to sense if the path it is involved in has been aborted.
6.1. Cut-clause

A nondeterministic goal $G$, whose potentially unifiable clauses $C_1, C_2, \ldots, C_n$ contain cut-clause, is reduced as follows. For each clause $C_i$ a new abort variable $AbortC_i$ is allocated. A $Cut$ variable is allocated, which is shared by all clauses, and a cut_clause/3 process is spawned connecting the abort variable of $G$, $ParentAbort$, with $Cut$ and the $AbortC_i$ variables, as shown in Figure 2.

In addition, the cut-clause goal in the $i$th clause is replaced by a goal that unifies the variable $Cut$ with the integer $i$.

The cut_goal process operates as follows. If it receives an abort signal from its parent (i.e., $ParentAbort$ is instantiated to $abort$), it aborts all its children processes by instantiating the clause abort variables $AbortC_i$, $i = 1, \ldots, n$, to $abort$ and terminates. If a cut was executed, that is, $Cut$ was instantiated to some clause index $i$, then the cut_clause process instantiates all other $AbortC_j$ variables, $j \neq i$, to $abort$, unifies $ParentAbort$ with $AbortC_i$, and terminates. Its implementation is shown in Program 5.

Computations are aborted by the clauses/2 procedure, which returns an empty list of alternatives when an abort interrupt is sensed.

Program 6 is a PROLOG program for computing whether one of two lists has an element in common with a third list.

The FCP representation of Program 6 is shown as Program 7. Note that melt/2 is defined to share the abort variable of the original goal between all computation paths. This is necessary for implementing cut-clause in the clauses unifiable with the top-level goal. It has the added benefit that the user can abort the computation at will by instantiating that variable to $abort$.

```
cut_clause (ParentAbort, Cut, Children) ←
    known(Cut) | abort(Cut?, ParentAbort, Children?).
cut_clause(ParentAbort, Cut, Children) ←
    ParentAbort? = abort | abort(0, abort, Children?).
abort(C, Abort, [[C, Abort]|Cs]) ←
    abort(C, Abort, Cs?).
abort(C, Abort, [[Cl, abort]|Cs]) ←
    C = \ = Cl | abort(C, Abort, Cs?).
abort(C, Abort, []).
```

PROGRAM 5. An implementation of the cut-clause process.
\[\text{intersect}(X, L_1, L_2, L_3) \leftarrow \text{member}(X, L_1), \text{member}(X, L_3), \text{!}.\]
\[\text{intersect}(X, L_1, L_2, L_3) \leftarrow \text{member}(X, X_2), \text{member}(X, L_3), \text{!}.\]
\[\text{member}(X, [X|Xs]).\]
\[\text{member}(X, [Y|Ys]) \leftarrow \text{member}(X, Ys).\]

**PROGRAM 6.** A PROLOG program with cut-clause.

\[\text{melt}(\text{intersect}(X, L_1, L_2, L_3, \text{Abort}), \text{intersect}(\_1, L_1, L_2, L_3, \text{Abort})) \leftarrow \text{unknown}(X) \mid \text{true}.\]
\[\text{melt}(\text{intersect}(X, L_1, L_2, L_3, \text{Abort}), \text{intersect}(X, L_1, L_2, L_3, \text{Abort})) \leftarrow \text{known}(X) \mid \text{true}.\]
\[\text{clauses}(\text{member}(X, L, \text{Abort})), \text{clause}(\text{member}(X, L, \text{Abort}), \{1, \text{Abort}\}, \{2, \text{Abort}\}) \leftarrow \text{unknown}(\text{Abort}) \mid \text{true}.\]
\[\text{clauses}(\text{intersect}(X, L_1, L_2, L_3, \text{Abort})), \text{clause}(\text{intersect}(X, L_1, L_2, L_3, \text{Abort}), \{3, \text{Cut}, \text{Abort}\}, \{4, \text{Abort}\}) \leftarrow \text{unknown}(\text{Abort}) \mid \text{cut_clause}((\text{Abort}?, \text{Cut}?, \{3, \text{Abort}\}, \{4, \text{Abort}\}).\]
\[\text{clauses}(\text{member}(X, L, \text{Abort}), \{\text{member}(X, L, \text{Abort}), \text{true}\}) \leftarrow \text{unknown}(\text{Abort}) \mid \text{true}.\]
\[\text{clauses}(\text{Goal}, []) \leftarrow \text{otherwise} \mid \text{true}.\]
\[\text{clause}(1, \text{Abort}, \text{member}(X, [X|Xs], \text{Abort}), \text{As} \setminus \text{As}).\]
\[\text{clause}(2, \text{Abort}, \text{member}(X, [Y|Ys], \text{Abort}), \text{member}(X, Ys, \text{Abort})\setminus \text{As}).\]
\[\text{clause}(3, \text{Cut}, \text{Abort}), \text{intersect}(X, L_1, L_2, L_3, \text{Abort}), \text{member}(X, L_1, \text{Abort}), \text{member}(X, L_3, \text{Abort}), \text{member}(X, L_1, L_2, L_3, \text{Abort}), \text{unknown}(\text{Abort}) \mid \text{true}.\]
\[\text{clause}(4, \text{Cut}, \text{Abort}), \text{intersect}(X, L_1, L_2, L_3, \text{Abort}), \text{member}(X, L_1, \text{Abort}), \text{member}(X, L_3, \text{Abort}), \text{member}(X, L_1, L_2, \text{Abort}), \text{unknown}(\text{Abort}) \mid \text{true}.\]
\[\text{clause}(\text{true}, A, \text{As} \setminus \text{As}).\]
\[\text{clause}(N, A, \text{fail} \setminus \text{As}) \leftarrow \text{otherwise} \mid \text{true}.\]

**PROGRAM 7.** An FCP representation of a program with cut-clause.

6.2. **Cut-goal**

Cut-goal is implemented as follows. Call the part of the clause to the left of cut-goal the *prunable* part of the clause. When a goal is reduced with a clause that contains cut-goal, a new *Abort* variable is allocated for aborting amongst the computations attempting to solve the prunable part of the clause. A *cut_goal/3* process is spawned for passing the abort signal from the parent goal to the prunable part of the body. In addition, the cut-goal goal is replaced by a goal that unifies the abort variable of the prunable part with *abort*.

The implementation of the *cut_goal* process is shown in Program 8.

The PROLOG Program 9 for computing intersection has the FCP representation shown in Program 10.

One of the consequences of this implementation is that an abort interrupt may “chase” a computation that is exploring a path, without ever reaching its frontier. To prevent this, processes that propagate the abort signal need to be given higher priority than normal computations.
cut_goal(From, To) ←
    From? = abort | To = abort.
cut_goal(From, To) ←
    To? = abort | true.

**PROGRAM 8.** An implementation of the `cut_goal` process.

intersect(X, L1, L2) ←
    member(X, L1), member(X, L2), !g.

**PROGRAM 9.** A PROLOG program with cut-goal.

melt(intersect(X, L1, L2, Abort), intersect(_, L1, L2, Abort)) ←
    unknown(X) | true.
melt(intersect(X, L1, L2, Abort), intersect(X, L1, L2, Abort)) ←
    known(X) | true.

clauses(intersect(X, L1, L2, Abort), [{5, Abort}]) ←
    unknown(Abort) | true.
clauses(member(X, L, Abort), [{1, Abort}, {2, Abort}]) ←
    unknown(Abort) | true.
clauses(`='(X, X, Abort), [true]) ←
    unknown(Abort) | true.
clauses(Goal, []) ←
    otherwise | true.

clause({5, Abort}, intersect(X, L1, L2, Abort),
    [member(X, L1, Abort1),
     member(X, L2, Abort1),
     `='(Abort1, abort, Abort)|As\As
    ] ←
    cut_goal(Abort?, Abort1).
clause({1, Abort}, member(X, [X|Xs], Abort), As\As).
clause({2, Abort}, member(X, [Y|Ys], Abort), [member(X, Ys, Abort)|As\As).
clause(true, A, As \ As).
clause(N, A, fail \ As) ←
    otherwise | true.

**PROGRAM 10.** An FCP representation of a PROLOG program with cut-goal.

Several straightforward optimizations are applicable. If there are no cuts in the set of clauses used to reduce a goal, then neither new processes nor new variables need to be allocated. Neck cuts can be handled specially by first attempting to reduce using clauses with neck cuts, and only if these reductions have failed then trying the other paths. Our implementation of cut easily accommodates such optimizations.
7. MIXING TRACING WITH ENVIRONMENT FREEZING

For every node \( N \) in the search tree, define \( \text{environment}(N) \) to be the pair containing the resolvent and the instantiated goal in that node, and \( \text{depth}(N) \) to be the depth of that node in the tree.

Using the standard \( \text{DAG} \) representation of terms, there is a program-dependent constant \( c_1 \) such that

\[
|\text{environment}(N')| - |\text{environment}(N)| < c_1
\]

for any two consecutive nodes \( N \) and \( N' \) in the tree.

Also, there is a program-dependent constant \( c_2 > 0 \) such that every program reduction takes at least \( c_2 \) steps. Hence there is a constant \( c \) such that for every node \( N \) for which

\[
|\text{environment}(N)| < c \text{depth}(N),
\]

it is cheaper to copy the environment by freezing it and melting it than to trace the computation of \( N \) from the root of the tree. Furthermore, if, when freezing the environment, a multiset of subterms whose total size is \( g \) is discovered to be ground, the cost of melting the environment can be reduced to \(|\text{environment}(N)| - g\). In addition, if subterms of the environment are known before freezing to be ground, the cost of freezing can be reduced similarly. This can be done, for example, by recording such information at the time the environment was last frozen and/or using additional marking propagation techniques [7].

Another factor to be considered is communication costs. In non-shared-memory implementations of the algorithm, such as the ones discussed below, the cost of communicating environments would be greater than the cost of communicating prefixes, since the latter have a very compact representation. Finding a good strategy for mixing tracing with freezing, and the effect of communication costs on such a strategy, is a subject of further research. When found, the interpreter shown above can be enhanced to accommodate such a strategy. Each process, instead of maintaining just a frozen copy of the goal, would maintain a frozen copy of an environment. Its path would be the path from the node in the search tree where the environment was frozen, rather than from the root. An explorer that encounters a branch in the search tree can either melt its inherited frozen environment and spawn a tracer that traces the alternative path from the point in which the environment was frozen, or freeze its own environment and spawn an explorer with a melted copy of that environment and a path containing just the alternative clause, whichever is dictated by the strategy chosen.

8. DYNAMIC LOAD BALANCING IN A PARALLEL IMPLEMENTATION

The algorithm as described does not address issues of process and code mapping. One possible realization of it on a non-shared-memory computer is as follows. Initialize each processor to contain a copy of the program and a frozen copy of the goal. Connect all processors to a centralized queue. Initialize the queue to contain the empty prefix.
Each processor performs the following cycle: Dequeue a prefix from the queue, explore it to completion, returning unexplored branches to the queue. If a solution is found, return it. Iterate.

The only data structures communicated at runtime between processors (besides solutions) are prefixes. Prefixes can be encoded very efficiently as vectors of integers, whose range can be restricted to the number of clauses in the program.

This implementation of the algorithm can be realized on a parallel computer directly, using its native operating system and programming language. One would have to implement the many-to-one and one-to-many communication mechanism required by the queue, and implement unification and indexing on the processors. It seems that developing such an interpreter-based implementation in a low-level language (e.g., C) should not be more difficult than developing a sequential PROLOG interpreter, since the mechanisms associated with backtracking (trail, choice points) are not needed.

In addition, one can simply connect multiple PROLOG implementations (whether interpreter or compiler based), and let each execute code similar to Program 11. This approach is a bit of an overkill, since the backtracing capabilities of PROLOG
solve \( (N, \text{Program}, \text{FrozenGoal}, \text{Sols}) \leftarrow \)

\( N \) is the number of processors to be used.

solve(\(N, \text{Program}, \text{FrozenGoal}, \text{Sols}) \leftarrow \)

queue([enqueue([\(1, \_\)\FromProcessors?], Q \setminus \text{Q}, \text{Sols}),
    processors(N, \text{FrozenGoal}, \text{Program}, \text{ToQueue}),
    stream \# \text{merger(ToQueue?, \FromProcessors)}).

queue([dequeue(X, \_\text{In}], [X\text{H}] \setminus \text{T}, \text{Sols}) \leftarrow
    queue(\text{In?}, \text{H} \setminus \text{T}, \text{Sols}).

queue([enqueue(X, \_\text{In}], \text{H} \setminus [\text{X}\text{\mid}], \text{Sols}) \leftarrow
    queue(\text{In?}, \text{H} \setminus \text{T}, \text{Sols}).

queue([solution(X)\text{In}, \text{H} \setminus \text{T}, [\text{X}\text{\mid}\text{Sols}) \leftarrow
    queue(\text{In?}, \text{H} \setminus \text{T}, \text{Sols}).

processors(0, \text{FG}, \text{P}, [])\).

processors(N, \text{FG}, \text{P}, [merge(Out)\text{Out1}]) \leftarrow
\text{N} > 0

    processor(\text{Out}, \text{FG}, \text{P}), \quad \% @\text{here}
    \text{N1} := \text{N} - 1,
    processors(\text{N1?}, \text{FG}, \text{P}, \text{Out1})@\text{next}.

\text{PROGRAM 12. Mapping processes to processors, and connecting them to a centralized queue.}

would not be exploited. However, it might be easier to construct, given a \text{PROLOG}
implementation with interface to a low-level language such as \text{C}.

An alternative approach is to implement the algorithm on top of a parallel
implementation of \text{FCP}, such as the iPSC implementation developed at the Weiz-
mann Institute [22]. The benefits of this approach, of course, are that in addition to
using \text{FCP} unification to implement the \text{PROLOG} unification, it can also rely on the
powerful communication mechanism of \text{FCP} to implement the queue and the
communication between it and the processors, as well as use its process and code
mapping mechanisms to map the processes implementing the algorithm, the
\text{PROLOG} program, and the frozen goal onto the physical processors. Such an
implementation is described below.

Program 11 contains the code of the process executed within each processor. The
basic cycle of this process is: dequeue a prefix from the queue and trace it. Explore
one extension of it, returning unexplored branches to the queue. When the end of
the explored path is reached, iterate.

\text{8.1. Centralized Dynamic Load Balancing}

Assume that processors are ordered and that the mapping annotation \( P@\text{next} \) is
interpreted by a processor as a request to execute the process \( P \) on the next
processor. Using such a mapping notation, the processes executing in each processor
can be mapped and connected to a centralized queue using the \text{FCP} Program 12.

Similar to the previous implementation, Program 12 deadlocks instead of detecting
termination. It can be enhanced to terminate properly by testing that the
top-level queue has \( n \) unsatisfied dequeue requests, where \( n \) is the number of processors spawned.

It is interesting to observe that even though the FCP interpreter performs dynamic load balancing of the underlying PROLOG execution, its constituent processes are mapped statically. In addition, once the process network is in place, no FCP remote process spawn is performed during the entire computation. This corroborates an earlier claim [15] that one of the better ways to implement dynamic load balancing is via statically mapped interpreters.

Note that copying of the PROLOG program and the frozen goal to each processor is specified implicitly by the FCP program, and is performed by the underlying FCP implementation on a demand-driven basis. Furthermore, the prefixes are not really sent to the queue; rather, a reference to the prefix is sent by the enqueuing processor and obtained by the dequeuing processor. The actual prefix is transmitted directly between the enqueuer and dequeuer, not via the queue.

Note that extending this implementation to support a mixed strategy that combines tracing with environment freezing increases the communication overhead: the frozen environment has to be sent together with a (partial) prefix, in contrast to just the (full) prefix in the pure algorithm. More detailed analysis is required to determine the exact tradeoffs between a mixed and a pure strategy.

8.2. Parallel Performance of Centralized Dynamic Load Balancing

We have ported Programs 11 and 12 to the iPSC-based parallel implementation of Flat Concurrent PROLOG [23], with a slight variation: Each processor was attached to a local queue. The local queue serves as a buffer between the processor and the central queue. It maintains up to a bounded number of enqueue or dequeue requests, and serves the matching requests if it can. If it overflows, it passes the request to the central queue manager.

We tried the scheme with the permutation PROLOG program, for an input list of length five, with one processor serving as the central queue and the remaining fifteen processors as workers, each with a processor process and a local queue process. For comparison, we have also measured the case where there is one worker which communicates with one queue (wall-clock time was about 10 minutes, depending on the size of the local queue). We have experimented with several local buffer sizes. When the buffer size was small, work was spread very quickly to all workers, and all workers were busy all the time, but there was a lot of communication overhead. Time was 2:10 for a zero-size buffer, and 1:08 for a buffer of size 3. When the buffer size was larger, it took longer to spread the work, but once the work was spread all processors were pretty much busy all the time. Time was 0:58 for a buffer of size six. When the buffer size was too large, it took very long till work was spread to all workers. There was very little communication, but towards the end of the computation there were periods when only a few workers were busy, and only occasionally a burst of work would be spread to other processors. Time was 1:08 for a buffer of size 10, and 1:31 for a buffer of size 16.

The results are summarized in Appendix 2.

These are very preliminary benchmarks. Our main conclusion from this short experiment is that we are quite far from understanding the dynamics of parallel
computations. Until such an understanding is gained, flexible tools which allow experimenting with various strategies and parameters are essential.

8.3. Distributed Dynamic Load Balancing

The centralized queue shown above could be a communication bottleneck in a parallel computer with a local-connections interconnection network, such as a mesh or a hypercube. Even in a globally connected computer network the centralized queue is far from ideal, since it does not give priority to satisfying requests locally, within a processor.

We describe below a distributed queue, which alleviates these problems somewhat. The distributed queue is composed of a tree of queue processes. It can be mapped on a tree- or hypercube-connected computer using standard techniques. It can be mapped on a mesh using the N-tree mapping technique [15].

The result of serving \textit{Req} with respect to \textit{Q} and \textit{Out} is \textit{Q1} and \textit{Out1}. \textit{Q} is the queue of unserved enqueue and dequeue requests. There are three basic cases:
1. The \textit{Request} was already served, so ignore it (clause 1).
2. The \textit{Request} can be served locally, so mark the \textit{Request} and the matching queue entry as served, and update the queue (clause 2).

\textbf{PROGRAM 13.} Spawning a distributed queue.
3. The Request cannot be served locally: add it to the local queue, and also send it to the parent queue (clauses 3, 4). In addition, if served requests are found in the queue, then remove them (clause 5).

Queue invariant: the queue always contains requests of one type only. Hence if the first queue entry is of the wrong type (same as the request), then conclude that the request cannot be served locally.

serve(\{Type, Path, Served\}, Q, Q, Out, Out) ← Served? = served\(_\) | true. %1
serve(\{Type, Path, served\(_\)\}), [(\{Type1, Path, served\(_\)\})\{Q\}, Q, Out, Out
) ←
Type = \_ = Type1|true. %2
serve(\{Type, Path, Served\}, \[\], [(\{Type, Path, Served\}]], [(\{Type, Path, Served\})\{Out\}], Out
) ← unknown(Served) | true. %3
serve(\{Type, Path, Served\}, [Req\{Q\}], [(\{Type, Path, Served\}]), Req\{Q\]. [(\{Type, Path, Served\})\{Out\}], Out
) ← Req = (Type, ., Served1), unknown(Served1), unknown(Served) | true. %4
serve(Req, [(\{Type, Path, Served\})\{Q\}], Q1, Out, Out1) ← Served? = served\(_\) | serve(Req, Q?, Q1, Out, Out1). %5

filter([\(\ldots\), \ldots])\{In\}, Out) ←
filter(In?, Out). filter([solution(Sol)]\{In\}, [solution(Sol)]\{Out\}) ← filter(In?, Out).
filter([\], []).

merge([X[Xs], Ys,[X[Zs]]) ←
merge(Ys?, Xs?, Zs). merge([Xs],Y[Ys]], [Y[Zs)]) ←
unknown(Xs)) merge(Ys?, Xs?, Zs).
merge([\], Xs, Xs).

PROGRAM 13. Continued.

The leaves of the queue tree are connected to the processors. Processors send to the queues enqueue and dequeue requests as well as solutions. Each queue process operates as follows. When it receives an enqueue or dequeue request, it attempts to satisfy it using a matching unsatisfied request from its local queue. If it cannot satisfy the request locally, it enqueues it, and sends a copy of it to its parent. Solutions are simply forwarded to the parent. The root of the queue tree is connected to a filter process that ignores enqueue and dequeue requests and forwards the solutions it receives to the user.
The key problem in this approach is to ensure that even though multiple copies of the same request are spread throughout the tree, a request will be satisfied once at the most.

This is guaranteed using a mutual exclusion mechanism that relies on the atomicity of unification in FCP, and on the property that the unification of two distinct read-only variables suspends. The idea is as follows. Each enqueue and dequeue request is sent with an additional argument, which is initially uninstantiated. A queue that wants to serve a request using a matching request, attempts atomically to instantiate the mutual-exclusion variables in both requests to the term served(_?), where _? is an anonymous read-only variable. If it succeeds, then no other queue can serve either of these requests, since the unification of their mutual exclusion variable with another such term will not succeed. If a queue process ever discovers that a mutual exclusion variable of a request is instantiated to served( X), for any X, then it can conclude that the request has already been served and discard it from the queue.

This idea is implemented in Program 13. The process-to-processor mapping notation is omitted.

9. CONCLUSIONS

We have presented a simple OR-parallel execution algorithm for PROLOG, and have analyzed its overhead and amenability to parallel execution. The theoretical analysis indicates that the algorithm may be practical for a large class of PROLOG programs. The algorithm can be easily interfaced to more specialized execution schemes for PROLOG, such as the one proposed by Ueda [24], which are applicable in case additional mode information can be derived.

We have demonstrated an FCP implementation of the algorithm, which preserves its complexity. The implementation uses FCP read-only unification to implement PROLOG's unification. Languages that have a weaker notion of unification, such as GHC and PARLOG, might not have such a direct implementation of the algorithm. The FCP implementation can be extended quite easily in several useful ways, and in this sense is preferable to a direct implementation of the algorithm. Its performance on a uniprocessor indicates that the approach has acceptable execution overhead.

Further research, both theoretical and experimental, is required to investigate the parallel speedup and communication overhead of non-shared-memory parallel implementations of the algorithm. Other research directions include the development of an effective strategy for mixing tracing and freezing, and the construction of an OR-parallel PROLOG-FCP compiler, based on the interpreters shown.

The paper has not addressed the issue of side effects in PROLOG. We feel that in the scheme we have proposed of embedding OR-parallel PROLOG in FCP, problems whose implementation in PROLOG requires side effects, i.e., interaction with the environment and maintenance of updatable data, are best left to FCP. FCP can express their solution in a much cleaner way, using pure logic problems without side effects. Which capabilities of the underlying FCP system should be provided to PROLOG and how, are yet to be determined. We suspect they may be quite different from the ones provided by current PROLOG systems.

The algorithm described is suitable for small programs with a lot of search. An extension, or a different algorithm, may be required for handling large PROLOG
### APPENDIX 1. PRELIMINARY UNIPROCESSOR PERFORMANCE ANALYSIS

The benchmarks were run under Logix and C-PROLOG on a VAX 8650. The results are shown in Table 1.

"PROLOG" is the C-PROLOG time in milliseconds. The C-PROLOG collected all solutions using bagof. "OrpP" is the time, in milliseconds, of the OR-parallel PROLOG interpreter in FCP. The interpreter in Program 2 was compiled with the clause representation in the same module, to eliminate the overhead of Logix's remote procedure calls. Trust-mode compilation was used.

"Ln" represents the list \([l, 2, \ldots, n]\). Time is in milliseconds. "Creations" is the numbers of processes created during the computation; "Reductions" is the number of process reductions performed. "RPC" stands for reductions per queue cycle and is a crude measure of the available parallelism. "Cycles" (when available) is the total number of queue cycles. They were obtained by setting the process time slice of Logix to 1 (breadth-first scheduling). The line marked "(FCP)" indicates the performance of naive reverse in FCP. The programs used are shown in Appendix 3.

#### NOTE.
These numbers should be taken as indicative figures and are not the result of a serious performance analysis. Several system factors were taken into account slightly differently in different columns, and as a result some of the numbers don’t add (e.g., rpc \times Cycles = Reductions does not always hold).

#### NOTE.
The code in the paper was cleaned up a bit after benchmarking was finished. This resulted in about 10–20% performance improvement.

<table>
<thead>
<tr>
<th>Goal</th>
<th>Prolog</th>
<th>OrpP</th>
<th>Creations</th>
<th>Reductions</th>
<th>RPC</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>reverse(L30, _)</td>
<td>166</td>
<td>350</td>
<td>1500</td>
<td>2500</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>reverse(L60, _)</td>
<td>416</td>
<td>1150</td>
<td>5700</td>
<td>9700</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>append(X, Y, L30)</td>
<td>466</td>
<td>190</td>
<td>670</td>
<td>2055</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>append(X, Y, L60)</td>
<td>1150</td>
<td>690</td>
<td>2550</td>
<td>6200</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>perm(L4, _)</td>
<td>100</td>
<td>320</td>
<td>1200</td>
<td>2800</td>
<td>19</td>
<td>157</td>
</tr>
<tr>
<td>perm(L5, _)</td>
<td>550</td>
<td>2250</td>
<td>7300</td>
<td>17800</td>
<td>70</td>
<td>257</td>
</tr>
<tr>
<td>perm(L6, _)</td>
<td>4000</td>
<td>16500</td>
<td>52600</td>
<td>133100</td>
<td>313</td>
<td>425</td>
</tr>
<tr>
<td>queens(L4, _)</td>
<td>80</td>
<td>260</td>
<td>750</td>
<td>1700</td>
<td>12</td>
<td>154</td>
</tr>
<tr>
<td>queens(L5, _)</td>
<td>250</td>
<td>1100</td>
<td>3100</td>
<td>7150</td>
<td>29</td>
<td>256</td>
</tr>
<tr>
<td>queens(L6, _)</td>
<td>800</td>
<td>5100</td>
<td>13600</td>
<td>33000</td>
<td>73</td>
<td>453</td>
</tr>
<tr>
<td>queens(L7, _)</td>
<td>3800</td>
<td>24700</td>
<td>63250</td>
<td>158100</td>
<td>214</td>
<td>738</td>
</tr>
<tr>
<td>queens(L8, _)</td>
<td>17600</td>
<td>123700</td>
<td>313000</td>
<td>802000</td>
<td>746</td>
<td>1074</td>
</tr>
</tbody>
</table>

Programs whose code can be distributed across several processors, e.g., database applications.
APPENDIX 2. PRELIMINARY PARALLEL PERFORMANCE ANALYSIS

Programs 11 and 12 were augmented with a local queue process, which serves as a buffer between the processor process and the central queue process. Its definition is as follows:

\[
\text{buffer}(\text{Int}, \text{Ext}) \leftarrow \\
\text{buffer}(\text{Int}, \text{Ext}, X, 0, 6).
\]

\[
\text{buffer}([\text{solution}(X) | \text{In}], \text{On}, H \setminus T, \text{Size}, \text{Maxsize}) \leftarrow \\
\text{buffer}(\text{In}, \text{On}, H \setminus T, \text{Size}, \text{Maxsize}).
\]

\[
\text{buffer}([\text{enqueue}(X, _) | \text{In}], [\text{enqueue}(X, _) | \text{On}], H \setminus T, \text{Maxsize}, \text{Maxsize}) \leftarrow \\
\text{buffer}(\text{In}, \text{On}, H \setminus T, \text{Maxsize}, \text{Maxsize}).
\]

\[
\text{buffer}([\text{enqueue}(X, _) | \text{In}], \text{Export}, H \setminus [X | T], \text{Size}, \text{Maxsize}) \leftarrow \\
\begin{align*}
\text{Size} &= \text{Maxsize}, \text{Size1} \leftarrow \text{Size} + 1 \\
\text{buffer}(\text{In}, \text{Export}, H \setminus T, \text{Size1}, \text{Maxsize}).
\end{align*}
\]

\[
\text{buffer}([\text{dequeue}(X, _) | \text{In}], [\text{dequeue}(X, _) | \text{On}], H \setminus T, \text{Maxsize}) \leftarrow \\
\begin{align*}
\text{Size} &= 0, \text{Size1} \leftarrow \text{Size} - 1 \\
\text{buffer}(\text{In}, \text{Export}, H \setminus T, \text{Size1}, \text{Maxsize}).
\end{align*}
\]

With a queue process and a processor process on two adjacent processors, and a local buffer of size 6, time was 10:27 minutes (1:23 minutes for garbage collection). With four processor processes on the four processors adjacent to the queue processor, time was 2:48 (0:12 for garbage collection).

With fifteen processors executing the processor process, the following timings were obtained, depending on the size of the local buffer:

<table>
<thead>
<tr>
<th>Buffer</th>
<th>0</th>
<th>1</th>
<th>3</th>
<th>6</th>
<th>10</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>2:10</td>
<td>1:19</td>
<td>1:08</td>
<td>0:58</td>
<td>1:08</td>
<td>1:31</td>
</tr>
</tbody>
</table>

About 10 to 15 seconds was spent in garbage collection in these computations.

APPENDIX 3. BENCHMARK PROGRAMS

\[
\text{rev}([X | Xs], Ys) \leftarrow \text{rev}(Xs, Zs), \text{append}(Zs, [X], Ys).
\]

\[
\text{rev}([], []).
\]

\[
\text{append}([X | Xs], Ys, [X | Zs]) \leftarrow \text{append}(Xs, Ys, Zs).
\]

\[
\text{append}([], Xs, Xs).
\]

\[
\text{perm}(Xs, [X | Ys]) \leftarrow \\
\text{select}(Xs, Xs1), \text{perm}(Xs1, Ys).
\]

\[
\text{perm}([], []).
\]

\[
\text{select}(X, [X | Xs], Xs).
\]

\[
\text{select}(X, [X | Xs], [X | Ys]) \leftarrow \\
\text{select}(X, Xs, Ys).
\]

\[
\text{queens}(Ns, Qs) \leftarrow \\
\text{queens}(Ns, [ ], Qs).
\]

\[
\text{queens}(Ns, [Ss], Qs) \leftarrow
\]
select(Q, Ns, Ns1),
safe(Q, 1, Ss),
queens(Ns1, [Q|Ss], Qs).
queens([], Ss, Ss).
safe(Q, N, [X|Xs]) ←
Y1 is Q + N, Y2 is Q − N, X \=\= Y1, X \=\= Y2, N1 is N + 1,
safe(Q, N1, Xs).
safe(Q, N, []).

The development of the algorithm was inspired by an enumeration technique for FCP computations, developed jointly with Yossi Lichtenstein and Michael Codish [10]. Thanks to John Gallagher, Ken Kahn, Vinet Singh, Jeff Ullman, Moshe Vardi, and the anonymous referees for their comments. Rony Shapiro helped with the iPSC port and measurements.

Part of this research was carried out while the author was at Stanford University, Department of Computer Science.

REFERENCES