Invited Paper

A Performance Comparison of Shared-Memory OR- and AND-Parallel Logic Programming Architectures for a Common Benchmark

EVAN TICK*

This paper compares the design and execution performance of two parallel logic programming architectures. Aurora is an OR-parallel Prolog system retaining the full semantics of Horn Clause logic (i.e., backtracking non-determinism). KL1-PS is an AND-parallel FGHC system that is a committed-choice architecture (i.e., no backtracking). This paper characterizes the various trade-offs made in these two systems: ease of programming vs. exploitation of parallelism, user-control over parallelism vs. available inherent parallelism, ease of programming vs. execution efficiency. Both systems have been implemented by various research groups on shared memory multiprocessors, and performance statistics collected on the Sequent Symmetry are presented in this paper. Several versions of the N-Queens problem are described and analyzed, for both systems.

1. Introduction

The main purpose of this paper is to compare the design and execution performance of OR- and AND-parallel logic programming architectures, as well as both uncommitted-choice and committed-choice languages. A secondary purpose is to describe parallel logic programming techniques, giving performance measurements as verification of the utility of the techniques. Measurements are not only necessary to present evidence of attainable speedups, but to compare algorithms in different languages and executing on different systems. Linear speedup of a poor algorithm, or within an inherently slow system, should be of little interest. The most important goal of this research is determine how to exploit parallelism to achieve *absolute* performance improvements.

This paper compares the design and execution performance of two parallel logic programming architectures, both of which have been implemented by independent groups [13, 11] on shared-memory multiprocessors. Aurora is an OR-parallel Prolog system retaining the full semantics of Horn Clause logic. KL1-PS is an AND-parallel FGHC system that is a committed-choice architecture. Although the performance measurements presented here do not compare favorably with *C*—we doubt very much that large parallel programs can be written in *C* with the same ease (see for instance Lusk [10]). Compiler technology is expected to bridge some of this gap, i.e., current logic programming compilers still lag behind procedural language compilers.

This paper extends and clarifies preliminary studies discussed in Tick [16, 17]. The contents are organized as follows. Section 2 discusses the Aurora and KL1-PS architectures. Section 3 describes and compares several algorithms solving the *N*-Queens problem. Section 4 presents raw-timing measurements of these programs executing on a Sequent Symmetry [14]. The relative speeds and speedups of the programs are analyzed. Conclusions are discussed in Section 5.

2. Aurora and KL1-PS Architectures

An architecture is an instruction set and storage model implementing a language. The OR-Parallel Prolog architecture (Aurora [20]) and FGHC architecture (KL1-PS [7]) are "high-level" because their instruction sets are more abstract than those of conventional computers. Both of these architectures have been implemented on the same general-purpose host (the Sequent Symmetry multiprocessor [14]) via emulation. Although the Symmetry implementations of both systems are preliminary and not of commercial quality, they among the *first true-parallel, high-level language implementations.*

The most important design considerations in these systems concerning performance are listed below. These issues are all interrelated and quite difficult to analyze individually.

- **engine architecture**—the instruction set design is important with respect to the instruction execution times, the memory bandwidth required and the compiler optimizations allowed. The Aurora system uses Carlsson's (SICStus) version [3] of the Warren Abstract Machine (WAM) [19] instruction set. Modifications were made to implement binding, dereferencing, and trailing with respect to binding arrays (see next item). KL1-PS system uses Kimura's version of the WAM, called KL1-B [9]. KL1-B is both simpler than the WAM because

*Research Center for Advanced Science and Technology, University of Tokyo, 4-6-1 Komaba, Meguro-ku, Tokyo 153.

backtracking has been removed, but also more complex than the WAM because both suspension and locking mechanisms have been integrated. Both systems use the compilation technique of clause indexing.

• **binding mechanism**—in these parallel systems, bindings are the means by which processes communicate among themselves and with the outside world. In Aurora, parallel processes executing a non-determinate procedure produce independent solutions, i.e., they can potentially produce conflicting, but valid, bindings. To implement this, the Aurora system uses a *binding array* per processor wherein bindings to variables shared among branches reside (i.e., bindings to variables that may potentially differ among the processors). This is related to the scheduling mechanism because the overhead of "spawning a process" is the work required changing the values in the binding array to reflect the new process's location (in the process tree). In FGHC, AND-parallel execution implies that all processes have equal authority to bind any variable at any time. Thus the binding problem becomes a locking problem. The binding (in the active part of a clause) of variables passed from the passive part of the clause must be locked. This is related to the instruction set because to reduce the locking penalty, somewhat roundabout code is generated to minimize locking times.

• **scheduler**—the process scheduler must be efficient in two major respects. First, the work must be evenly distributed among the processors (good load balancing). Second, the overhead of process spawning/suspending/resuming must be low. If only large-granularity goals are spawned on different processors, both of these criteria will be met. Whereas goals are stored in a tree structure in Aurora, in KL1-PS, all goals are treated equally, and stored in *goal-queues* local to each process. Aurora scheduling (as studied here) is performed locally in the process with a distributed "tree-walking" algorithm [2]. An idle process (one that succeeded or failed) traverses the OR-tree, constrained by several heuristics, searching for work. In KL1-PS, scheduling is performed in a semi-distributed manner. An idle process (one with an empty goal-queue) requests work from a busy process, via message passing.

• **storage model**—memory management is important to retain the spatial locality needed to make efficient use of local caches. In addition, efficient memory management creates less garbage and therefore garbage collection (GC) will be incurred less often. In Aurora, groups of intertwined stacks are used, where a stack-group, similar to that of the WAM, is allocated to each processor. In KL1-PS, each processor has a group consisting of a heap stack, goal and suspension record lists. These lists are allocated from a larger group of free-lists, split among the processors to avoid contention. In general, KL1-PS storage management is simpler than Aurora's, but the FGHC model creates garbage at a significantly faster rate.

• **garbage collection**—all languages that dynamically create structures require some form of GC. In Aurora, the WAM automatically recovers memory upon backtracking i.e., when searching for all solutions to a non-determinate problem, memory used to explore bad paths is easily recovered. However, the determinate portions of programs can still produce garbage (in the form of temporary data structures needed to get from one intermediate point to another, and then discarded). FGHC generates more garbage than Prolog because OR-parallel search is *simulated* by the architecture which therefore cannot automatically recover memory upon backtracking. The KL1-PS system studied here has stop-and-copy GC only and the Aurora system has no GC. For the benchmarks measured GC is not a significant factor.

3. *N*-Queens

The *N*-Queens problem is to place *N*-queens on an $N \times N$ chess board so that no queen can attack another queen. *N*-Queens has taken in the role as the classic parallel logic programming language benchmark. This choice is unfortunate because *N*-Queens does not share the characteristics of real parallel applications. One major flaw is that data representation requires only lists of integers, thus the garbage production costs and intergoal communication costs of realistic programs are not adequately reflected. Another major flaw is that *N*-Queens has a symmetric proof tree, facilitating load balancing. However, *N*-Queens *is* useful for the purposes of describing the different programming methodologies used in OR- and AND-parallel architectures, and in committed and non-committed choice languages. We have nullified GC issues in this study by executing all programs with a large memory space.

3.1 Prolog Versions

N-Bruynooghe's, the first Prolog program discussed (Fig. 1) is a classic generate and test solution using permutations for the generator [4]. **perm** is the generator and **check** is the tester. In *all* of the prolog *N*-Queens, **del** is used as the source of OR-parallelism (in the following program listings, refer to previous listings for identical procedure definitions). ***N*-Pereira's** (Fig. 2) is a variant of the first program in which the generator and tester are fused [4]. It is well-known that the fused version is significantly faster than the naive version in sequential Prolog. We add here that the fused version retains the full amount of OR-parallelism in the naive version.

N-Bratko's, Fig. 3, is based upon a pseudo-constraint method [1], although modified here to use structures. Each queen is characterized by a unique integer that represents its row, column, and two diagonals. The latter two sets of constraints are implicitly represented as structure indicies. When a queen is placed, the logical variable at that index is bound, disallowing subsequent conflicts. The algorithm is elegant, parallel, and fast

```
:-parallel del/3.

go(N, A):-gen(N, L), findall(X, queen(X, L), A).

gen(0, [ ]):-!.
gen(N, [N|X]):-M is N-1, gen(M, X).

queen(X, L):-perm(L, X), check(X).

perm([ ],[ ]).
perm([H|T], [A|P]):-del([H|T], A, L), perm(L, P).

del([X|T], X, T).
del([H|T], X, [H|R]):-del(T, X, R).

check([ ]).
check([H|T]):-safe(T, H, 1), check(T).

safe([ ], _, _).
safe([H|T], U, N):-H+N=\=U, H-N=\=U, M is N+1, safe(T, U, M).
```

Fig. 1 Bruynooghe's Naive *N*-Queens (Prolog).

```
queen(X, L):-queen(L, [ ], X).

queen([ ], P, P).
queen([H|T], R, P):-del([H|T], A, L), safe(R, A, 1), queen(L, [A|R], P).
```

Fig. 2 Pereira's Fused *N*-Queens (Prolog).

```
go(N, A):-findall(X, queen(N, X), A).

queen(N, S):-
  gen(N, Dxy),
  N2 is (N*2)-1
  functor(Du, du, N2),
  functor(Dv, dv, N2),
  sol(Dxy, S, Dxy, Du, Dv, N).

sol([ ], [ ], _, _, _, _).
sol([X|Dx1], [Y|Ylist], Dy, Du, Dv, N):-
  del(Dy, Y, Dy1),
  U is X-Y+N, arg(U, Du, X),
  V is X+Y-1, arg(V, Dv, X),
  sol(Dx1, Ylist, Dy1, Du, Dv, N).
```

Fig. 3 Bratko's *N*-Queens with Structures (Prolog).

(the fastest among those tested). Note that the original program (not shown) uses lists to represent all the constraints, making constraint removal inefficient.

N-Kondo's, Fig. 4, uses logical variables to implement a "blackboard" scheme. Closely related to Bratko's method, here a square on the board is represented by two logical variables linking all squares on its two diagonals. The **queen** procedure nondeterminately places a queen on each row (using **arg** to avoid placement of two queens in the same column). The placement binds the diagonal variables, making future attempts to place a diagonally attacking queen fail. **queen** backtracks until a queen has been successfully placed in each row. Note that *N-Kondo's* and *N-Bratko's* cannot

```
go(N, A):-findall(X, queen(N, X), A).

queen(N, Q):-gen(N, L), board(N, P), queen(P, L, [ ], Q).

queen([ ], _, Y, Y).
queen([H|T], L, Y, Z):-
  del(L, E, L1),
  arg(E, H, a(E, E)),
  queen(T, L1, [E|Y], Z).

board(N, B):-M is N*2-1, functor(X, x, M), functor(Y, y, M),
  make(N, M, N, N, B, X, Y).

make(0, _, _, _, [ ], _, _):-!.
make(M, X, Y, N, [S|Ss], Nx, Ny):-
  functor(S, b, N),
  next(N, S, X, Y, Nx, Ny),
  X1 is X-1, Y1 is Y+1, M1 is M-1,
  make(M1, X1, Y1, N, Ss, Nx, Ny).

next(0, _, _, _, _, _):-!.
next(N, S, X, Y, Nx, Ny):-
  X1 is X-1, Y1 is Y-1, N1 is N-1,
  arg(X, Nx, A), arg(Y, Ny, B), arg(N, S, a(A, B)),
  next(N1, S, X1, Y1, Nx, Ny).
```

Fig. 4 Kondo's *N*-Queens (Prolog).

be automatically translated into FGHC because they pass unbound variables between procedures. As the measurements later presented indicate, *N-Kondo's* and *N-Bratko's* are about 60% faster, on a single Symmetry PE, than the fastest FGHC algorithm studied. This result illustrates the power of full unification and

```
go(N, A):-gen(N, L), queen(L, [ ], [ ], A, [ ]).

queen([C|Cs], NCs, L, S0, S2):-
  check(L, C, 1, NCs, Cs, L, S0, S1),
  queen(Cs, [C|NCs], L, S1, S2).
queen([ ], [ ], L, S0, S1):-S0=[L|S1].
queen([ ], [_|_], _, S0, S1):-S0=S1.

check([ ], C, _, NCs, Cs, L, S0, S1):-
  append(NCs, Cs, Ps),
  queen(Ps, [ ], [C|L], S0, S1).
check([P|Ps], C, D, NCs, Cs, L, S0, S1):-
  P-C=\=D, C-P=\=D|
  D1 is D+1,
  check(Ps, C, D1, NCs, Cs, L, S0, S1).
check(_, _, _, _, _, _, S0, S1):-otherwise|S0=S1.

gen(0, X):-X=[ ].
gen(N, X):-N>0|N1 is N-1, X=[N|Xs], gen(N1, Xs).

append([ ], Y, Z):-Z=Y.
append([A|X], Y, Z):-Z=[A|Z1], append(X, Y, Z1).
```

Fig. 5 Kumon's Candidates/Noncandidates *N*-Queens (FGHC).

backtracking in solving problems such as this.

3.2 FGHC Versions

Sato [13] reports that one of the major overheads in FGHC execution is caused by suspension of goals. *N-Kumon's* (Fig. 5) is efficient because it *rarely suspends*, even in parallel execution. In the entire program, there is only one direct producer-consumer relationship between body goals, in the first clause of **check**. With depth-first scheduling however, **ps** is produced before the recursive call to **queen** is executed.

N-Kumon's creates a binary process tree. **queen** is the main procedure, representing a process wherein the next queen is to be placed. The first argument of **queen** is a list of candidate columns for queen placement (the associated row is implicit—it could be calculated from the length of the candidate list). **C** represents the column placement of the current queen. The remaining two **queen** clauses deal with the cases when the candidate list is empty. When choosing **C**, two child processes are forked: **check** and another **queen**. The "process reading" is that **check** generates all solutions *including* **C** whereas **queen** generates all solutions *not* including **C**. Of course **C** may be a faulty choice, in which case **check** generates an empty set of solutions.

The second argument of **queen** is a non-candidate list, i.e., a list of columns that cannot be used for placement in the current row. The non-candidate list is necessary to prevent the forked **queen** child process from choosing the same placement as the one being explored by **check**. **check** must create a third list, **Ps**, by concatenating the remainder of the candidate list with the non-candidate list, i.e., **Ps** is the list of remaining unused columns. If the safety check passes, **Ps** will be passed to **queen** as a new first argument for candidates.

The third argument of **queen** and first argument of **check** is the partial solution of queens. This list is examined during the check: each queen is compared with the current queen placement to determine if the new queen is legal. The partial solution is also passed as the fifth argument, L, to be either discarded (if the new queen is bad), or to be used to compose the new partial solution for the call to **queen** in the first clause of **check**.

The final two arguments of **queen** form a difference list (D-list) for the output stream of solutions. Note how this D-list is wired between the two children of **queen**. A complete solution is bound to the stream in

```
go(N, A):-gen(N, L), queen(L, [ ], A, [ ]).

queen([ ], L, S0, S1):-S0=[L|S1].
queen(Q, L, S0, S2):-otherwise|
  d1(Q, [ ], L, S0, S1),
  d2(Q, [ ], L, S1, S2).

d1([ ], _, _, S0, S1):-S0=S1.
d1([P|N], C, L, S0, S1):-check(L, P, 1, N, L, S0, S1, C).

d2([ ], _, _, S0, S1):-S0=S1.
d2([P|U], C, L, S0, S1):-
  d1(U, [P|C], L, S0, S1),
  d2(U, [P|C], L, S1, S2).

check([ ], _, D, N, L, NCs, S0, S1):-b(NCs, L, D, N, S0, S1).
check([P|Ps], C, D, N, L, NCs, S0, S1):-C-P=\=D, P-C=\=D|
  D1 is D+1,
  check(Ps, C, D1, N, L, NCs, S0, S1).
check(_, _, _, _, _, _, S0, S1):-otherwise|S0=S1.

b([ ], B, D, N, S0, S1):-queen(N, [D|B], S0, S1).
b([C|Cs], B, D, N, S0, S1):-b(Cs, B, D, [C|N], S0, S1).
```

Fig. 6 Ueda's *N*-Queens (FGHC): Translated from Fused Prolog and Hand Optimized.

```
go(N, A):-queen(1, N, [[ ]], A).

queen(I, N, In, Out):-I=<N|
  I1 is I+1,
  q(1, N, In, In1),
  queen(I1, N, In1, Out).
queen(I, N, In, Out):-I>N|Out=In.

q(I, N, In, Out):-I=<N|
  I1 is I+1,
  filter(In, I, Out, Out1),
  q(I1, N, In, Out1).
q(I, N, _, Out):-I>N|Out=[ ].

filter(In, I, S0, S1):-filter(In, I, 1, S0, S1).

filter([ ], _, _, S0, S1):-S0=S1.
filter([P|Ps], I, K, S0, S2):-
  filter(Ps, I, K, S1, S2),
  check(P, I, K, P, S0, S1).

check([ ], I, _, P, S0, S1):-S0=[[I|P]|S1].
check([J|Js], I, D, P, S0, S1):-
  J=\=I, D=\=I-J, D=\=J-I|
  D1 is D+1,
  check(Js, I, D1, P, S0, S1).
check(_, _, _, _, S0, S1):-otherwise|S0=S1.
```

Fig. 7 Naive Pipelined Filters *N*-Queens (FGHC).

```
go(N, A):-queen(1, N, begin, LS), fromLStoL(LS, A).

queen(I, N, In, Out):-I=<N|
  I1 is I+1,
  q(1, N, In, In1),
  queen(I1, N, In1, Out).
queen(I, N, In, Out):-I>N|Out=In.

q(I, N, In, Out):-I=<N|
  I1 is I+1,
  filter(In, I, Out, Out1),
  q(I1, N, In, Out1).
q(I, N, _, Out):-I>N|Out=[ ].

filter(In, I, S0, S1):-S0=[I*In1|S1], filter1(In, I, 1, In1).

filter1(begin, _, _, Out):-Out=begin.
filter1([ ], _, _, Out):-Out=[ ].
filter1([J*In|Ins], I, D, Out):-
  J=\=I, D=\=I-J, D=\=J-I|
  D1 is D+1,
  Out=[J*NewIn|Out1],
  filter1(In, I, D1, NewIn),
  filter1(Ins, I, D, Out1).
filter1([_|Ins], I, D, Out):-otherwise|filter1(Ins, I, D, Out).

fromLStoL(LS, L):-fromLStoS(LS, [ ], L, [ ]).
fromLStoS(begin, Stack, L0, L1):-L0=[Stack|L1].
fromLStoS([ ], _, L0, L1):-L0=L1.
fromLStoS([A*LS1|Rest], Stack, L0, L2):-
  fromLStoS(LS1, [A|Stack], L0, L1),
  fromLStoS(Rest, Stack, L1, L2).
```

Fig. 8 Okumura's Layered-Streams *N*-Queens (FGHC).

the second clause of **queen** when both the candidate and non-candidate lists are empty, i.e., all the queens have been placed. The final clause of **queen** represents the case when no more candidates remain, yet there are still non-candidates, i.e., the partial solution is bad. In this case, that branch in the OR-tree is abandoned and its output stream is closed.

N-Ueda's is a translation of the fused *N-Pereira's* into FGHC by the continuation method [18]. The translated program was then hand-optimized (Fig. 6) with folding/unfolding rules [6]. It is interesting to note that a continuation-based translation of the naive *N-Bruynooghe's* into FGHC is only about 40% slower (for 9-Queens) than the fused version. Yet the speed difference between the original Prolog programs is over 30 times. The increase in efficiency of the naive program is due to the co-routining introduced by Ueda's method [18]. Also note that raw translation of fused *N-Pereira's* is about 17% slower than the hand-optimized version. Speedup is due to fusing procedures together to avoid reductions, using lists to implement simple continuations, and removing unnecessary continuations. These optimizations appear too complex to be automated. Note that optimized *N-Ueda's* is essentially *N-Kumon's*.

The FGHC program given in Fig. 7 uses a linear pipeline of **filter** processes to incrementally create and discard partial solutions. Initially a column of N **filter/4** processes is spawned for every column in the board. Thus there are N^2 filters spawned. Filters belonging to the same column share a single input stream (broadcast to all filters). The filters of the column have their outputs linked together in an output stream. Initially a seed input of [ ] is given to the filter process structure. The job of a filter is to determine if the incoming message, representing a partial solution, is compatible with the queen represented by that filter. Every filter has two state elements: its row (I) and its depth (D) (its column number is implicit in the process tree). The depth is used identically to that of *N-Kumon's* to perform the check. If the incoming partial solution is *not* compatible (clauses 2-4 of **check**) then the solution is discarded. if the incoming partial solution *is* compatible (clause 1 of **check**) then the new queen is prepended to the partial solution. The output stream from the final column of filters will contain only complete, valid solutions.

Pipeline parallelism is the concurrent computation of different filters in the pipeline. Initially the computation "spreads" through the pipe from the leftmost column of the board (given the seed). The "spreading" is a natural result of process scheduling and automatic producer-consumer stream communication. The parallelism is severely limited by the times to fill and empty the pipeline and the serialization caused by using an output *stream* for each column of filters. When **check** starts to check partial solution **P** against queen I, no output is made until the check has completed. Notice that later columns will also have to check **P**, but cannot begin until the previous check is finished. We can envision a type of *speculative* parallelism wherein

```
go(N, A):-gen(N, L), top(S), queen(L, L, [ ], S, A, [ ]).

queen([C|Cs], [R|Rs], NCs, Snd, S0, S2):-
  Snd=[check(R, C, Reply)|SndT],
  check(Reply, Rs, R, C, NCs, Cs, SndR, SndT, S0, S1),
  queen(Cs, [R|Rs], [C|NCs], SndR, S1, S2).
queen([ ], _, [_|_], Snd, S0, S1):-Snd=[ ], S0=S1.
queen([ ], _, [ ], Snd, S0, S1):-Snd=[echo(L, [ ])], S0=[L|S1].

check(no, _, _, _, _, _, SndR, Snd, S0, S1):-SndR=Snd, S0=S1.
check(yes, Rs, R, C, NCs, Cs, SndR, Snd, S0, S1):-
  append(NCs, Cs, N),
  merge(SndL, SndR, Snd),
  piece(Snd1, SndL, R, C),
  queen(N, Rs, [ ], Snd1, S0, S1).

piece([ ], Snd, _, _):-Snd=[ ].
piece([echo(A, B)|Rcv], Snd, X, Y):-
  A=[Y|C], % add queen to reply
  Snd=[echo(C, B)|Snd0], % send request up
  piece(Rcv, Snd0, X, Y).
piece([Check|Rcv], Snd, X2, Y2):-
  Check=check(X1, Y1, Answer),
  X2-X1=\=Y2-Y1, X2-X1=\=Y1-Y2|
  Snd=[Check|Snd0], % check succeeds . . .
  piece(Rcv, Snd0, X2, Y2).
piece([check(_, _, A)|Rcv], Snd, X, Y):-otherwise|
  A=no, % check fails . . .
  piece(Rcv, Snd, X, Y).

top([ ]).
top([check(_, _, A)|Rcv]):-A=yes, top(Rcv).
top([echo(A, B)|Rcv]):-A=B, top(Rcv).

merge([X|Xs], Ys, Z):-Z=[X|Zs], merge(Xs, Ys, Zs).
merge(Xs, [Y|Ys], Z):-Z=[Y|Zs], merge(Xs, Ys, Zs).
merge([ ], Y, Z):-Y=Z.
merge(X, [ ], Z):-X=Z.
```

Fig. 9 Tick's Distributed *N*-Queens (FGHC).

later columns can begin checking **P** even *before* previous columns have completed their check. A possible correction to some of these deficiencies is a **merge** tree between columns of filters, but this will incur a high suspension overhead for communication (as we shall see in *N-Tick's* discussed later). Instead we fix the naive pipelined filters program with a data structure called a *layered stream*. In essence, layered streams reduce pipeline startup and wind-down times and exploit the speculative parallelism mentioned.

N-Okumura's (Fig. 8) uses the layered-steam methodology [12]. A layered stream is a list of structures H*Ts, where H is a head of a list and Ts is a set of all possible tails of H. Ts is itself represented with a layered stream. This style of programming exploits a greater amount of parallelism than either normal stream-based programming (*N-Kumon's*) or continuation-based programming (*N-Ueda's*), because a producer can send an element through a layered stream before its tail has been constructed. Essentially, *streams are to lists as layered streams are to streams.*

A primary filter is spawned for each row and column of the board in the exact same manner as in the naive pipelined filters program. Initially a seed of the atom **begin** is input to the first column of filters. Initially each column also independently seeds its successor column with partial solution, **I*In1** in **filter/4. filter1** encapsulates the queen check. For an input layered stream **I*In**, if queen J at depth D is compatible with the filter's queen I, then the partial solution **J*NewIn** is issued down the output stream (clause 3). Two filters are then spawned. The first filter checks the sub-layers of solution J and creates **NewIn**. The second filter checks remaining solutions at the current level, **Ins** and creates **Out1**. If the check fails then the remaining partial solutions are checked (clause 4). Note that **fromLStoL** converts the layered stream back into a list of solutions.

In the FGHC *N*-Queens implementations previously discussed, the programmers' goal was of course to make *N*-Queens run as quickly as possible. When such programs are used as performance benchmarks, the analyst may lose sight of the fact that *N*-Queens execution entails passing a trivially simple data structure around a symmetric proof tree. As a result, information gathered about *N*-Queens execution does not have direct bearing on more realistic applications. With this in mind, *N-Tick's* (Fig. 9) was written with a general, distributed process tree so that the framework could be mapped onto other applications.

N-Tick's process tree is too complex for the trivial composition of leaves (the check in **piece**). As a result, it is no surprise that *N-Tick's* performs significantly worse than the other FGHC programs. Although based on the *N-Kumon's* process structure, *N-Tick's* is fully distributed, i.e., the partial solution is represented solely by a group of **piece** processes. Each **piece** process can respond to the following commands: **echo (I, O)**—return D-list **I, O** of the queens (to the root), and **check (X, Y, A)**—check pair **X, Y** with the internal queen for safety: if safe, return **A=yes**, otherwise **A=no**.

A **queen** process first selects a queen **X, Y**, for the next placement. The selection algorithm will never choose **X=Y** so subsequent checking need not check for this. The queen is sent to a checker which checks for correctness (i.e., consistency with previous choices). If the queen is consistent, the checker will spawn a **queen** process to find all solutions including this queen. Whether the queen is correct or incorrect, another **queen** process is spawned to find all solutions without the queen.

The **piece, top** and **merge** processes are perpetual, i.e., they act as objects, receiving a message, acting on it, and then calling themselves in anticipation of another message. Eventually, they kill themselves when a [ ] message instructs them to do so. These AND-parallel processes form a virtual OR-tree that is incrementally pruned as good solutions are found and bad partial solutions are discarded. As we shall see, the largest execution overhead of *N-Tick's* are frequent suspensions of these objects, 30 times *N-Okumura's* and over 2700 times *N-Kumon's*!

4. Performance Measurements

We now present performance measurements of the 9-Queens programs executing on a Sequent Symmetry (all execution times are given in seconds). Table 1 summarizes the dynamic characteristics of the programs on eight PEs: thousands of instructions, number of reductions, number of backtracks (suspensions for FGHC), their sum ("entries"), thousands of entries per second (KEPS), and instructions per reduction. Table 2 gives raw execution time and Table 3 gives two measures of speedup, S_n/S_r. S_n is the naive speedup with respect to the algorithm itself running on a single PE. S_r is the real speedup with respect to the *fastest* algorithm (in the

Table 1 9-Queens High-Level Characteristics (Eight PEs Symmetry).

Author	Method	instr	reduct	back†	entries	sec	KEPS	inst/red
				Aurora/Prolog				
Bruynooghe	naive gen & test	59088K	5776886	986411	6763297	113.0	60	10.2
Bratko	constraints	3781K	312168	481020	793188	13.6	58	12.1
Pereira	fused gen & test	1708K	138214	24013	162227	5.3	31	12.4
Kondo	blackboard	518K	45152	32055	77223	2.7	29	11.5
Bratko	with structures	604K	45067	24022	69089	2.6	27	13.4
				Panda/FGHC				
Tick	distributed	5772K	319767	222057	541824	13.9	39	18.0
Tick	pipelined filters	4274K	287757	559	288316	15.6	18	14.9
Ueda	trans. Pereira	3456K	218458	52	218510	6.6	33	15.8
Ueda	optimized	2311K	186405	79	186484	4.2	44	12.4
Kumon	candidates	2112K	144918	82	145000	4.2	35	14.6
Okumura	layered streams	1480K	98731	7523	106254	3.2	33	15.0

†suspensions in FGHC

Table 2 9-Queens: Total Execution Time (Seconds) on Symmetry.

Author	Method	1 PE	2 PE	4 PE	8 PE	12 PE	15 PE
				Aurora/Prolog			
Bruynooghe	naive gen & test	850.0	424.3	217.4	113.0	74.2	59.7
Bratko	constraints	88.6	46.3	23.6	13.6	7.0	6.2
Pereira	fused gen & test	24.4	12.4	6.8	5.3	3.0	2.7
Kondo	blackboard	15.0	7.4	4.2	2.7	2.2	2.0
Bratko	with structures	13.4	7.3	4.2	2.6	2.1	2.0
				Panda/FGHC			
Tick	distributed	95.8	49.5	26.7	13.9	10.2	7.9
Tick	pipelined filters	58.1	30.6	19.3	15.6	21.7	16.6
Ueda	trans. Pereira	49.7	25.2	12.8	6.4	4.3	3.5
Ueda	optimized	31.8	16.2	8.3	4.2	2.8	2.3
Kumon	candidates	28.3	14.3	7.3	3.7	2.5	2.0
Okumura	layered streams	21.5	11.7	6.1	3.2	2.4	1.9

Table 3 9-Queens Speedups (S_n/S_r) on Symmetry.

Author	Method	1 PE	2 PE	4 PE	8 PE	12 PE	15 PE
				Aurora/Prolog			
Bruynooghe	naive gen & test	1.00/0.02	2.00/0.03	3.91/0.06	7.52/0.12	11.46/0.18	14.24/0.22
Bratko	constraints	1.00/0.15	1.91/0.29	3.75/0.57	6.51/0.99	12.66/1.91	14.29/2.16
Pereira	fused gen & test	1.00/0.55	1.97/1.08	3.59/1.97	4.60/2.53	8.13/4.47	9.04/4.96
Kondo	blackboard	1.00/0.89	2.03/1.81	3.57/3.18	5.56/4.96	6.82/6.09	7.50/6.70
Bratko	with structures	1.00	1.84	3.19	5.15	6.38	6.70
				Panda/FGHC			
Tick	distributed	1.00/0.22	1.94/0.43	3.59/0.81	6.89/1.55	9.39/2.11	12.13/2.72
Tick	pipelined filters	1.00/0.37	1.90/0.70	3.01/1.11	3.72/1.38	2.68/0.99	3.50/1.30
Ueda	trans. Pereira	1.00/0.43	1.97/0.85	3.88/1.68	7.77/3.36	11.56/5.00	14.20/6.14
Ueda	optimized	1.00/0.68	1.96/1.33	3.83/2.59	7.57/5.12	11.36/7.68	13.83/9.35
Kumon	candidates	1.00/0.76	1.97/1.50	3.88/2.95	7.65/5.81	11.32/8.60	14.15/10.75
Okumura	layered streams	1.00	1.84	3.52	6.72	8.96	11.32

same language) running on a single PE. The fastest algorithms measured were *N-Bratko's* and *N-Okumura's* for Prolog and FGHC respectively.

To calibrate the two systems, *N-Kumon's* was translated from FGHC into Prolog. Both the Prolog and FGHC versions of the algorithm were executed for 8-Queens on one PE, resulting in 13.0 sec (KL1-PS) and 13.8 sec (Aurora). SICStus Prolog V0.5 ran in 8.0 sec, showing Aurora overheads degraded performance over 40%. Larger benchmarks measured by Lusk [11] indicate an average 20% overhead. These overheads are due in part to an earlier version of the compiler. Note that other committed-choice architectures will probably differ in performance from KL1-PS. For example JAM-Parlog executes *N-Kumon's* 25% faster than KL1-PS and other programs from 20%–40% faster [5]. Thus both the Aurora and KL1-PS systems may be hamstringed, but still they are calibrated in their present state.

In FGHC, although *N-Kumon's* causes almost no suspensions, *N-Okumura's* has better performance because of data-sharing in the layered-stream method. All of the algorithms except *N-Okumura's* perform redundant checks when constructing (in the OR-tree) independent solutions that contain identical queens. In *N-Okumura's*, since the OR-tree is represented by the layered stream and the process structure has one main filter per board square, no redundant checks are performed. For 9-Queens, *N-Okumura's* is 32% faster than *N-Kumon's* on a single PE. As the number of PEs increase, so do suspensions in *N-Okumura's*, and its advantage narrows. On 15 PEs, *N-Okumura's* is only 5% faster. The 10-Queens OR-tree is larger and therefore the effect of data-sharing is more pronounced: *N-Okumura's* is 26%–45% faster. The version of *N-Okumura's* measured, however, differs slightly from that in Fig. 8—a layered stream is represented as [H|Ts]. Without this optimization, the program generates significantly more memory references and runs 8% slower for 9-Queens.

Considering the effect of algorithm on performance, we first note that a *good algorithm* is not necessarily a *parallel algorithm*. For example the Prolog version of *N-Kumon's* is one of the faster Prolog algorithms, but it is not OR-parallel. On the other hand, we note that a *poor algorithm* often cannot overcome its inefficiency, even if it is a *parallel algorithm*. For example, naive *N-Bruynooghe's*, even though it gets linear speedup, has a higher complexity order than the other algorithms and cannot compete with even sequential *N-Kumon's*. However, as a third point, we note, as did Ueda [18] and others, that a simple *change in algorithm* can drastically improve naive *N-Bruynooghe's*—we measured a factor of 21–35 speedup.

Implementation technology and data structures also has an impact on program performance. Noting that *N-Tick's* and *N-Kumon's* have the same program structure, the vast difference in reductions and suspensions is due to the distributed check, i.e., the overhead of **merge**. This problem may be ameliorated with special system support for **merge**. The pipelined filters program and *N-Okumura's* have the same process structure but use different data structures: streams and layered streams respectively. Streams actually slowdown on more than eight PEs because of the serialized method in which the streams are constructed, and the large amount of garbage created as a byproduct of stream communication. Layered streams however continue to speedup through 15 PEs. We also note that although *N-Bratko's* is an elegant algorithm, it should ideally use arrays, not lists. By simulating write-once arrays with structures in *N-Bratko's*, performance improved by a factor of 5–7 on symmetry. In general, the Prolog algorithms exploiting backtrackable unification out-perform the other programs by a significant margin. Thus the algorithm must be well matched to the language technology.

The measurements presented here concerning automatic translation from Prolog to FGHC agree with Ueda's [18]—the translation speeds up the naive Prolog program by a factor of 8–9. However, the FGHC translation of the fused algorithm runs (on a single PE) only about half as fast as the Prolog version, and only about half as fast as the fastest (hand-written) FGHC program (*N-Okumura's*). On 15 PEs the superior speedup of 14.2 achieved by *N-Ueda's* closes the gap with Prolog and FGHC layered streams. In addition, source-to-source optimizations can essentially convert Ueda's program into *N-Kumon's*.

Concerning speedups, *N-Bratko's* displays the *worst* naive speedup, yet the best real speedup, of the Prolog programs. We believe this is because of the high overhead for maintaining multiple bindings, i.e., solutions make many private bindings to the 34 constraint variables (in **Du** and **Dv** for 9-Queens). *N-Kondo's* has the same problem, with 34 variables in the 9×9 board. Of the FGHC programs, naive pipelined filters with streams is the only program to display a slowdown—the algorithm does not simulate an OR-tree as does *N-Tick's*, *N-Ueda's* and *N-Kumon's*. *N-Tick's* also displays low speedup because of suspension overheads in the message-passing model. The fastest FGHC program, *N-Okumura's*, displays poor naive speedup (due to both suspensions and the use of a pipeline instead of a tree), but the best real speedup. *N-Kumon's* displays greater naive speedup, indicating that the increased parallelism afforded by (lazy) layered streams is in reality decreased by their suspension rate. Furthermore the reduced complexity order of the layered-stream algorithm depreciates with increasing numbers of PEs because a tree-structured algorithm, such as *N-Kumon's*, can consistently achieve higher speedup.

5. Conclusions

This paper presents a performance comparison of two parallel logic programming systems (Aurora and

KL1-PS) executing a family of algorithms implementing the same application (*N*-Queens). Measurements given of "real" speedups on 15 Symmetry PEs indicate that the best 9-Queens speedup attained was only 6.7 and 11.3, for Prolog and FGHC respectively. Increasing the problem size (from 9 to 10 queens) increased the speedup on Aurora by 22% and on KL1-PS by 9%. This indicates that even with the large, regular tree of 9-Queens, further increasing granularity (by increasing problem size) helps the schedulers, i.e., the current schedulers in the systems are not fully efficient. The Aurora scheduler particularly has trouble exploiting fine-grain parallelism near the leaves of the 9-Queens OR-tree. This problem becomes most apparent on more than eight PEs. Note that the systems measured use low-performance emulators, each executing at about four KEPS. We expect custom processors to achieve 50 times this rate, giving shared-bus multiprocessors a much tougher job attaining similar speedups.

Compared to the superior algorithms of each language, other algorithms are at best 50% efficient i.e., their real speedup on N PEs is about $N/2$ (credit must be given to *N-Kumon's* and its derivatives which are slightly better). All of the algorithms presented are parallel, and many display close to linear (naive) speedup. These characteristics indicate that designing *fast* sequential algorithms that can be parallelized (relatively easy in languages such as Prolog and FGHC) is more important than designing massively parallel (yet stupid) algorithms. The superior Aurora algorithms achieve speedups of less than seven on 15 PEs, and for KL1-PS we achieve speedup of less than 12 on 15 PEs, yet the sequential speed of these algorithms proves their superiority. In general, the best Prolog algorithms have faster sequential speed than the FGHC algorithms and KL1-PS achieves higher speedups than does Aurora. For this benchmark problem, the equilibrium point of the two systems appears to be near 15 PEs.

In FGHC, it was shown that frequent suspensions due to message routing can ruin an otherwise reasonable algorithm (*N-Tick's* compared to *N-Kumon's*). The detrimental effect of suspensions on layered-streams programs (not due to message passing, but rather to lazy evaluation) is more than balanced by the avoidance of redundant computation afforded by the data-sharing nature of the layered stream. *N-Kumon's* illustrates how an OR-parallel program can be written in FGHC, incurring almost no suspensions, by a candidate/noncandidate paradigm. Another method of OR-parallel search in FGHC, automatic translation (*N-Ueda's*), was shown to be rather inefficient. Hand-optimizing the translation helped bring *N-Ueda's* on par with *N-Kumon's*, but whether such optimizations can be automated is an open question.

It is difficult to accurately compare the Aurora and KL1-PS systems because they differ in many implementation details. Moreover it has been said that such a comparison is unfair because the domains of OR-parallel and AND-parallel computation rarely overlap. This aside, it is valuable to compare the systems because it helps quantify the advantages one paradigm holds over the other. If one considers *N*-Queens an OR-parallel problem, then Aurora has an advantage in this study, and as the results show, Prolog algorithms outperformed FGHC algorithms for the most part. Yet Prolog's performance advantage grew thin with increasing numbers of PEs. For example automatic translation from OR-parallelism into AND-parallelism lost a factor of two in speed on a single PE, but on 15 PEs the original Prolog program executed only 30% faster. Given their weakness in all solutions OR-parallel search, the FGHC programs did surprisingly well. Certainly there are countless other problems with no OR-parallelism in which the OR-parallel Prolog programs would get no speedup, and the FGHC programs would.

It is for the reader to decide which of the algorithms are most readable, declarative, easily debugged, and extensible. Layered streams are difficult to utilize in complex problems; however, the general method of candidates/noncandidates (*N-Kumon's*) performs a very close second. Similarly one could claim that the constraints methods used in *N-Kondo's* (and *N-Bratko's*) are also limited. However, the fused generate and test program (*N-Pereira's*) gives fairly reasonable performance. Research is currently underway to compare the Aurora and KL1-PS architectures at the levels of shared memory, bus and cache performance [15, 8], to compare KL1-PS with other committed-choice architectures, such as JAM-Parlog [5], and to develop a more extensive benchmark suite.

6. Acknowledgements

This research was supported by NSF Grant No. IRI-8704576 and conducted at the Institute of New Generation Computer Technology (ICOT). R. Overbeek and E. Lusk of Argonne National Laboratories kindly supplied the Aurora system for Symmetry. M. Sato of ICOT wrote and upgraded the KL1-PS system to allow the measurements presented herein. Discussions with A. Okumura, M. Sato, and K. Ueda provided much insight into this research. The author is currently supported by an Information Sciences Chair at the University of Tokyo endowed by the CSK Corp.

References

1. BRATKO, I. *Prolog Programming for Artificial Intelligence*. Addison-Wesley Ltd., 1986.
2. BUTLER, R. et al. Scheduling OR-Parallelism: an Argonne Perspective. In *Int. Conf. and Symp. on Logic Prog.*, MIT Press, (August 1988), 1565–1577.
3. CARLSSON, M. *SICStus Prolog User's Manual*. PO Box 1263, S-16313 Spanga, Sweden, February 1988.
4. COELHO, H. and COTTA, J. C. *Prolog By Example*. Springer-Verlag, 1988.
5. CRAMMOND, J. and TICK, E. Comparison of Two Shared-Memory Emulators for Flat Committed-Choice Logic Programs. Technical report, RCAST, University of Tokyo, October 1989. to be published.

6. FURUKAWA, K., OKUMURA, A. and MURAKAMI, M. Unfolding Rules for GHC Programs. *New Generation Computing*, 6 (2-3) (1988), 143-157.

7. GOTO, A. Parallel Inference Machine Research in FGCS Project. *In First Japan-U.S. AI Symposium* (December 1987), 21-36.

8. GOTO, A., MATSUMOTO, A. and TICK, E. Design and Performance of a Coherent Cache for Parallel Logic Programming Architectures. In *16th International Symposium on Computing Arch.* Jerusalem, May 1989.

9. KIMURA, Y. and CHIKAYAMA, T. An Abstract KL1 Machine and its Instruction Set. In *International Symposium on Logic Programming*, San Francisco (August 1987), 468-477.

10. LUSK, E. et al. *Portable Programs for Parallel Processors*. Holt, Rinehart & Winston, 1987.

11. LUSK, E. et al. The Aurora Or-Parallel Prolog System. In *International Conference on FGCS*, Tokyo (November 1988), 819-830.

12. OKUMURA, A. and MATSUMOTO, Y. Parallel Programming with Layered Streams. In *International Symposium on Logic Programming*, San Francisco (August 1987), 224-233.

13. SATO, M. and GOTO, A. Evaluation of the KL1 Parallel System on a Shared Memory Multiprocessor. In *IFIP Working Conf. on Parallel Processing*. North Holland, May 1988.

14. Sequent Computer Systems, Inc. *Sequent Guide to Parallel Programming*, 1987.

15. TICK, E. A Performance Comparison of AND- and OR-Parallel Logic Programming Architectures. In *Sixth International Conference on Logic Programming*. Lisbon, MIT Press, June 1989.

16. TICK, E. Comparing Two Parallel Logic-Programming Architectures. *IEEE Software*, 6(4), July 1989.

17. TICK, E. Parallel Logic Programming on Shared-Memory Multiprocessors: A Tale of *N*-Queens. In *Proceedings Japanese Symposium of Parallel Processing*. Atami, February 1989.

18. UEDA, K. Making Exhaustive Search Programs Deterministic: Part II. In *Fourth International Conference on Logic Programming*, MIT Press (May 1987), 356-375.

19. WARREN, D. H. D. An Abstract Prolog Instruction Set. Technical Report 309, SRI International, 1983.

20. WARREN, D. H. D. The SRI Model for OR-Parallel Execution of Prolog—Abstract Design and Implementation. In *Symposium on Logic Programming* (August 1987), 92-102.

(Received May 19, 1989; revised August 28, 1989)