# Performance of a Data-Parallel Concurrent Constraint Programming System

Bo-Ming Tong* and Ho-Fung Leung†
Department of Computer Science
The Chinese University of Hong Kong
Shatin, New Territories
Hong Kong

**Abstract:** Firebird [12] is a data-parallel concurrent constraint programming system. It is shown that finite domain constraint languages can be implemented on massively parallel SIMD machines. Firebird has been further enhanced with *parallel backtracking* capability and the design of the *Data-Parallel Abstract Machine*, the basis of Firebird's implementation, has been presented in [13]. An almost complete implementation has been built on a DECmpp 12000 Sx-100‡ massively parallel computer with 8,192 processor elements and some preliminary performance results are given in this paper. A speedup of 2 orders of magnitude is possible when we compare the performance using 8,192 processor elements and the performance using a single processor element of the same machine. Furthermore, the speedup is scalable, provided that the problem size is large enough for effective exploitation of or-parallelism. On the other hand, we measure the effects of several control strategies and optimizations on execution time and memory consumption in a data-parallel context.

**Keywords:** performance evaluation, parallelism, finite domain constraints, concurrent constraint programming

## 1 Introduction

To the best of our knowledge, Firebird [12] is the first data-parallel concurrent constraint programming system. It is shown that *finite domain constraint languages* [15] can be implemented efficiently on massively parallel SIMD computer systems. The syntax of Firebird is similar to mainstream concurrent logic programming languages, in particular, *flat GHC* [14]. The most distinguishing semantic feature of this language is that committed-choice indeterminism is integrated with don't know nondeterminism by adopting the notion of domain-variable-based choice points. Alternatives of such choice points can be attempted in parallel on a data-parallel machine. We have enhanced Firebird with *parallel backtracking* capability and we have presented the *Data-Parallel Abstract Machine* which forms the basis of Firebird's implementation [13].

In this paper, we present some preliminary performance results of our implementation. we also measure the effects of several control strategies and optimizations on execution time and memory consumption in a data-parallel context. In the next section, we give a brief review of [12, 13]. The reader is referred to [15] for an introduction to the use of finite domain constraints in logic programming, and to [6, 7, 8] for concurrent constraint programming. Section 3 is the

*Present address: Department of Computer Science, University of Arizona, Tucson, AZ 85721, U.S.A. email: `bmtong@cs.arizona.edu`
†email: `lhf@cs.cuhk.hk`
‡DECmpp 12000 Sx-100 is equivalent to MasPar MP-1.

performance data and section 4 is a comparison between SIMD MultiLog [9] and Firebird. We conclude with section 5.

# 2 Firebird: A Review

## 2.1 The Firebird Computation Model

In Firebird, the notions of committed-choice indeterminism and don't know nondeterminism are integrated using the *Firebird Computation Model*. Every clause is divided into a *guard* part and a *body* part by a commit operator. In an *indeterministic derivation step*, execution consists of guard tests, commitment, output unification and spawning in the same manner as committed-choice languages. In a *nondeterministic derivation step*, a choice point based on one of the domain variables in the system is set up and all possible values in its domain are attempted in an or-parallel manner. The domain variable used in a nondeterministic derivation step is said to be *labeled*[1] [15] and each or-parallel branch is called a *partition*. Execution consists of alternating indeterministic and nondeterministic derivations.

## 2.2 Exploitation of Data-Parallelism in Firebird

The partitions can be executed in parallel on SIMD computers if, in an indeterministic derivation step, we choose an atom from each partition such that the predicate symbol and arity of each atom is the same. In other words, the "same" goal is evaluated in all partitions, but with different sets of arguments. An argument can be represented by a vector, each element of which comes from some partition. Every predicate is compiled to a procedure which manipulates all the elements of the argument vectors in parallel. Constraints can also be handled using the same principle.

### 2.2.1 An Illustrative Example

To illustrate how Firebird exploits data-parallelism, it is helpful to trace the execution of 5-queens and the query `queen(5,[X1,X2,X3,X4,X5])`. The constraints in the system just before the first nondeterministic derivation are shown in Figure 1. All goals have been reduced, and all domain variables have the same domain {1,2,3,4,5}.

| | | |
|---|---|---|
| X1 $\neq$ X2 | X1 $\neq$ X2 + 1 | X1 $\neq$ X2 - 1 |
| X1 $\neq$ X3 | X1 $\neq$ X3 + 2 | X1 $\neq$ X3 - 2 |
| X1 $\neq$ X4 | X1 $\neq$ X4 + 3 | X1 $\neq$ X4 - 3 |
| X1 $\neq$ X5 | X1 $\neq$ X5 + 4 | X1 $\neq$ X5 - 4 |
| X2 $\neq$ X3 | X2 $\neq$ X3 + 1 | X2 $\neq$ X3 - 1 |
| X2 $\neq$ X4 | X2 $\neq$ X4 + 2 | X2 $\neq$ X4 - 2 |
| X2 $\neq$ X5 | X2 $\neq$ X5 + 3 | X2 $\neq$ X5 - 3 |
| X3 $\neq$ X4 | X3 $\neq$ X4 + 1 | X3 $\neq$ X4 - 1 |
| X3 $\neq$ X5 | X3 $\neq$ X5 + 2 | X3 $\neq$ X5 - 2 |
| X4 $\neq$ X5 | X4 $\neq$ X5 + 1 | X4 $\neq$ X5 - 1 |

Figure 1: Constraints remaining after first deadlock

If we label `X1` using nondeterministic derivation and try the 5 possible values in a data-parallel fashion, we can evaluate the first 12 constraints with an ideal 5 times speedup[2] on a SIMD machine. A second nondeterministic derivation will occur. If every branch chooses to create a choice point on `X2`, there will be $3+2+2+2+3=12$ branches (see Figure 2). Thus the next 9 constraints can be solved with an ideal speedup of 12. `X3` will also be labeled, giving rise to a total of 14 branches. Therefore, the remaining 9 constraints can be evaluated with an ideal speedup of 14. Thousands of processor elements can be fully utilized easily in this way because many problems are combinatorial in nature.

### 2.2.2 Mapping Partitions to Processor Elements

A *logical partition*, or simply a *partition*, is mapped to a number of identical *physical partitions*. Each physical partition corresponds to a single processor element, and we use the two terms interchangeably. Initially, all processor elements execute exactly the same initial logical partition. If the data-parallel computer has $N$ processor elements and a choice point with 5 alternatives is created, $\frac{N}{5}$ processor elements

[1] *Labeling* a domain-variable means instantiating a domain-variable by attempting each value in its domain one by one or in parallel.

[2] This is just a rough approximation. The Amdahl's Law dictates that such an "ideal" speedup cannot be obtained in practice.

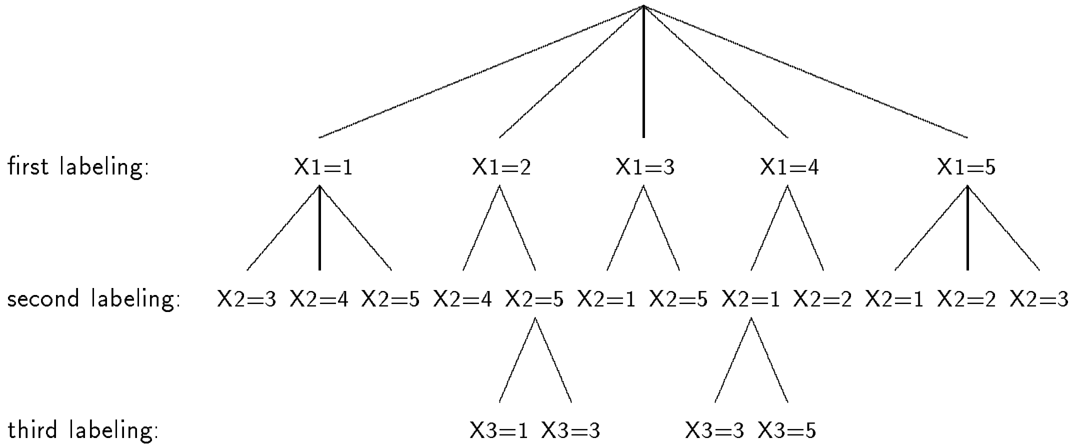



Figure 2: Or-parallel branches in the 5-queens example

will be allocated to each alternative. Since a *processor-id* is associated with each processor element, each processor element can compute which alternative it should take autonomously. Compared to *SIMD MultiLog* [9], our model has the advantage that interprocessor communication is not necessary. We expect a performance gain in SIMD MultiLog if our mapping technique is applied.

#### 2.2.3 Masks

Consider the following example.

```
p([H|T]) :- H > 0 | q(H), p(T).
p([H|T]) :- H = 0 | r(H), p(T).
```

When `p` is executed, the physical partitions with constraint stores entailing $H > 0$ commit to the first clause and those with constraint stores entailing $H = 0$ commit to the second. When `q(H)` is spawned, it runs only in those physical partitions which have committed to the first clause. Similarly, `r(H)` executes in physical partitions which have committed to the second clause only.

To implement this, a *mask* is associated with each goal. The mask is the set of physical partitions in which the goal is applicable. It is represented as a bit vector distributed over the whole processor element array.

When `p(T)` is spawned, its mask can be set to the set of physical partitions which have committed to either clause rather than spawning two `p(T)` with disjoint masks. We are looking forward to detect this at compile time using a *join algorithm*. At present our compiler joins only tail-recursive calls.

The mask of the original goal `p([H|T])` is changed to the set of physical partitions which suspend. If the set is empty, the original goal can be discarded because it is meaningless to execute a goal with an empty mask.

#### 2.2.4 Parallel Backtracking

Like any or-parallel logic programming system, the number of parallel branches explodes combinatorially in Firebird. For example, more than 2,000 processor elements are required to solve 8-queens. When the processor elements are exhausted, the system resorts to *parallel backtracking*, in which the partitions create choice points and backtrack in parallel.

In general, the number of or-branches is different for each partition because the domain size of the domain variable used to set up the choice point varies from partition to partition. As a result, some partitions will finish attempting all the alternatives earlier than the others.

We adopt the *synchronous backtracking* approach, in which the partitions which have finished wait until all partitions have finished before going back to the last choice point. The advantage of this approach is lower trail and choice-point management overhead. The reader is referred to [13] for the details of parallel backtracking.

## 2.3 Data-Parallel Abstract Machine

The *Data-Parallel Abstract Machine*[3] (DPAM) is the basis of the implementation of Firebird. We assume the knowledge of *Warren's Abstract Machine* (WAM) [1].

### 2.3.1 Hardware

We assume that the implementation platform consists of a sequential front end called the *host computer* and a data-parallel back end called the *processor element array*. The host computer is responsible for dispatching instructions and broadcasting data to the processor element array. Memory is distributed over the processor element array and each processor element has its own local memory. We also assume that there is a *contingent* bit to turn a processor element on/off. A processor element executes instructions if and only if its contingent bit is set, except that there are special instructions which move a bit to the contingent bit unconditionally. The contingent bits are used to implement the mask bit vector (Section 2.2.3).

### 2.3.2 Implementation Overview

A Firebird system consists of a compiler, either an emulator or a native code generator, a *concurrent process scheduler* and a runtime library. The compiler employs the *decision graph* [5] technique to compile a Firebird program to DPAM code. The instruction set is designed in such a way that the same compiler can be used for both sequential and data-parallel implementations. However, the emulator or native code generator, the scheduler and the runtime library must be modified or rewritten for the sequential implementation.

The processor element array is responsible for the execution of goals and the evaluation of constraints. Vector memory areas include the *argument stack*, the *trail stack* and the *heap*. The host computer is responsible for process scheduling and choice point management. Scalar memory areas include the *choice point stack* and the *process stack*.

[3] Also known previously in [12] as the *Firebird Abstract Machine*.

### 2.3.3 Heap Frames

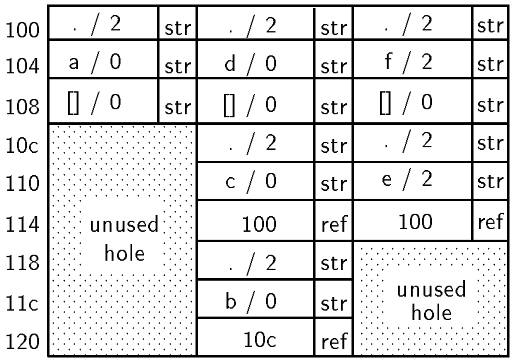
On many data-parallel computers (*e.g.* MasPar MP-1, Connection Machine CM-2 and some vector supercomputers) local indirect addressing (each processor element accesses a different memory location) is slower than direct addressing (each processor element accesses the same memory location). We introduce the concept of *heap frames* for building heap terms in order to improve performance. The amount of heap memory required by a clause is determined at compile time. There is only one scalar[4] *heap pointer* (`hp`) which is used in all processor elements. Each procedure has a *heap frame pointer* (`hf`), also scalar, from which heap terms are built. `hp` is moved to `hf` when the procedure is executed. Recall that each physical partition may commit to no more than one clause because Firebird is a committed-choice language. Each clause of the same predicate is executed in turn, with the mask set to the set of physical partitions committing to that clause. Before each clause is executed, `hp` is set to `hf`. As a result, each clause builds terms starting from the same location. A *heap pointer maximum* (`hm`) keeps the highest heap pointer encountered in the clauses which have been attempted. After all clauses have been tried `hm` is moved to `hp`. `hm` and `hf` are saved across procedure calls. For example, if the following program is executed and the heap pointer is initially 100 (hex),

```
p(X,Y) :- X = 0 | Y = [a].
p(X,Y) :- X = 1 | Y = [b,c,d].
p(X,Y) :- X = 2 | Y = [e,f].
```

the resulting heap is shown in Figure 3. After all clauses in the goal have been executed, `hm` is moved to `hp` and the final value of `hp` will be 124 (hex).

To sum up, more memory will be consumed if the heap frame scheme is used (because of heap fragmentation). However, it would lead to faster execution because the more expensive local indirect addressing operation needs not be used in heap term building.

[4] A scalar data item always resides on the host computer, whereas a vector data item always resides in the processor element array.




hf = 100, hp = 118, hm = 124
final hp = 124

Figure 3: Example: a heap frame

#### 2.3.4 Process Scheduling

For implementation convenience, a constraint is treated as a process, like a goal. A constraint may suspend, resume or perform ask/tell operations on the store. In DPAM, the *ready queue* consists of processes which are ready for immediate execution. A *labeling process* is associated with each domain variable in the system. A labeling process is taken from the *labeling queue* and moved to the ready queue when a nondeterministic derivation step is needed. When a process is resumed, it is first moved to the *resumption queue*. The processes in the resumption queue are moved to the ready queue when the processes in the ready queue are exhausted. Please refer to [13] for a detailed description of DPAM's scheduling subsystem.

# 3 Performance Results

The aim of this section is to evaluate the performance of our data-parallel implementation and to analyze the effects of a number of design decisions.

In all of the following benchmarks, $t_1$, $t_2$, *etc* are the execution time for all solutions in seconds, neglecting any time spent on input/output. $\#proc$ is the number of processor elements, $DL$ the number of deadlocks, $BT$ the number of backtrackings and $P$ the number of logical partitions. $P$ may change in the course of execution, but it can never exceed the number of processor elements. Only the value of $P$ taken at the end of execution is given. A dash indicates that a benchmark is not available because memory is not enough for its execution. Since many parameters are studied, only one parameter is varied in each benchmark. The other parameters are listed below each table. The benchmark set is shown in Table 1. These benchmark programs are selected because we want to test different kinds of constraints and their combinations. For example, `queen` is a program with disequality constraints only. `send`, `eq10` and `eq20` have an assortment of equality and disequality constraints. `magic` is also included here because entailment constraints are used.

## 3.1 Uniprocessor Performance

We measure the performance of our implementation on a DECmpp 12000 Sx-100 massively parallel computer, but using only one of the processor elements. The result is compared with CHIP version 3.2 on a DECstation 3100. The CHIP benchmark is obtained using generalized forward checking but no first fail heuristics. The Firebird system is modified to apply nondeterministic derivation (*i.e.* labeling) to domain variables in the order they are created. These ensure that the order of labeling of CHIP is the same as that of Firebird. The result is given in Table 2. Except for the magic series problem, the results indicate that our implementation has very poor performance compared to CHIP. The magic series problem has good performance because Firebird is a concurrent language which allows a different formulation of the problem using *entailment* (ask) constraints [2, 8].

We attribute this to the deficiency of our implementation platform. We profile our execution and find that an average machine instruction requires about 10 machine cycles to execute on our 12.5 MHz DECmpp 12000 Sx-100. In general, most sequential instructions require only 1 machine cycle to execute, while some parallel instructions take several hundred cycles. Table 3 lists the average execution time, in machine cycles, of a number of machine instructions. All operands of the instructions tested are registers. Each machine in-

| | |
|---|---|
| `send` | SEND + MORE = MONEY |
| `eq10` | 10 simultaneous linear equations over 7 variables |
| `eq20` | 20 simultaneous linear equations over 7 variables |
| `queen` | $n$-queens problem |
| `magic` | magic series problem |
| `magich` | magic series problem with redundant constraint $\sum_{i=0}^{n} s_i = n$. |

Table 1: Benchmark set

| | Firebird | | | CHIP | |
|---|---|---|---|---|---|
| benchmark | $t_1$ | $DL$ | $BT$ | $t_2$ | $t_2/t_1$ |
| `send` | .028 | 2 | 3 | .007 | .25 |
| `eq10` | 4.720 | 21 | 163 | .660 | .14 |
| `eq20` | 5.153 | 13 | 115 | .848 | .16 |
| `queen(4)` | .028 | 3 | 5 | .007 | .25 |
| `queen(6)` | .239 | 27 | 39 | .047 | .20 |
| `queen(8)` | 3.172 | 265 | 415 | .512 | .16 |
| `queen(10)` | 60.002 | 4229 | 6665 | 8.547 | .14 |
| `queen(12)` | - | - | - | 211.948 | - |
| `magic(3)` | .110 | 2 | 7 | .051 | .46 |
| `magic(6)` | 2.128 | 5 | 31 | 6.378 | 3.00 |
| `magic(8)` | 8.634 | 7 | 57 | 212.303 | 24.59 |
| `magich(3)` | .127 | 2 | 6 | .058 | .46 |
| `magich(6)` | 1.112 | 4 | 19 | 2.422 | 2.18 |
| `magich(9)` | 4.747 | 7 | 41 | 140.886 | 29.68 |

Test conditions: #*proc*=1, eager bit vector creation, eager nondeterministic derivation, no solitary memory access, no priority scheduling.

Table 2: Benchmark: uniprocessor performance

struction in DECmpp has only 2 operands. Sometimes two 2-operand instructions are required to perform the same task as a single 3-operand instruction typical of modern RISC architectures. Furthermore, we found that some other instruction sequences can be replaced by a single instruction on a RISC computer[5]. For these reasons, we estimate that a single processor element of our implementation platform is about 50 times slower than an average workstation. The same result is observed in the implementation of *SIMD MultiLog* [9].

One could always enhance the speed of individual processor elements to get around this problem. We test an earlier version of Firebird on a newer model, MasPar MP-2, which is binary compatible with the DECmpp 12000 Sx-100 we are currently using, and found that it is about 2 times faster on the $n$-queens problem[6]. See [12] for the benchmark.

## 3.2 Solitary Mode

On our implementation platform, DECmpp, 16 processor elements form a cluster and share a single 8-bit memory port. We would

[5] For example, a conditional branch. One instruction is used to move each processor's flag to the contingent bit. The next instruction obtains the global or-ing of all contingent bits and stores the result in the carry flag of the *host*. The last instruction is the actual branch.

[6] The manufacturer claims a speedup of up to 4.5, without any modifications to the program. The exact reason why we obtain only a speedup of 2 is unknown, but we suspected that this is because memory throughput was made only 2 times faster, and our system is very memory intensive.

| Parallel instructions | | | | Sequential instructions | |
|---|---|---|---|---|---|
| ld32 | 79.0 | mov32 | 18.9 | cmov32 | 1.0 |
| ld32 (indirect) | 224.6 | add32 | 24.9 | cadd32 | 1.0 |
| st32 | 74.3 | mul32 | 244.6 | clea32 | 1.0 |
| st32 (indirect) | 197.0 | div32 | 422.9 | cld32 | 5.2 |
| ldsol32 | 18.6 | mod32 | 469.3 | cst32 | 6.3 |
| ldsol32 (indirect) | 43.5 | shll32 | 63.9 | cjmp | 3.2 |

Table 3: Execution time of machine instructions (in machine cycles)

| | solitary (#*proc*=512) | | | | normal (#*proc*=8,192) | | | | |
|---|---|---|---|---|---|---|---|---|---|
| benchmark | $t_1$ | *DL* | *P* | *BT* | $t_2$ | *DL* | *P* | *BT* | $t_2/t_1$ |
| `send` | .014 | 2 | 4 | 0 | .019 | 2 | 4 | 0 | 1.36 |
| `eq10` | .701 | 6 | 76 | 28 | .334 | 6 | 76 | 9 | .48 |
| `eq20` | .757 | 3 | 58 | 10 | .519 | 3 | 140 | 0 | .69 |
| `queen(4)` | .009 | 2 | 6 | 0 | .014 | 2 | 6 | 0 | 1.56 |
| `queen(6)` | .028 | 4 | 40 | 0 | .042 | 4 | 40 | 0 | 1.50 |
| `queen(8)` | .157 | 11 | 182 | 4 | .101 | 7 | 548 | 0 | .64 |
| `queen(10)` | 3.957 | 133 | 366 | 191 | 1.400 | 27 | 2399 | 17 | .35 |
| `queen(12)` | - | - | - | - | 92.183 | 971 | 4165 | 1191 | - |
| `magic(3)` | .051 | 2 | 11 | 0 | .074 | 2 | 11 | 0 | 1.45 |
| `magic(6)` | .514 | 2 | 38 | 0 | .791 | 2 | 38 | 0 | 1.54 |
| `magic(9)` | 1.957 | 2 | 83 | 0 | 2.909 | 2 | 83 | 0 | 1.49 |
| `magich(3)` | .060 | 2 | 7 | 0 | .086 | 2 | 7 | 0 | 1.43 |
| `magich(6)` | .282 | 2 | 21 | 0 | .411 | 2 | 21 | 0 | 1.46 |
| `magich(9)` | .833 | 2 | 45 | 0 | 1.210 | 2 | 45 | 0 | 1.45 |
| `magich(12)` | 1.787 | 2 | 78 | 0 | 2.544 | 2 | 78 | 0 | 1.42 |

Test conditions: eager bit vector creation, eager nondeterministic derivation, no priority scheduling.

Table 4: Benchmark: solitary mode performance

prefer each processor element to have its own 32-bit memory port, giving 64 times of memory bandwidth, in order to be competitive with a workstation. Although we use the *overlapping feature* of DECmpp so that memory access can be performed in parallel with other machine instructions whenever possible, a concurrent constraint programming system is so memory intensive that performance is completely dominated by memory access time.

We introduce a *solitary mode* to alleviate this memory bottleneck. Only 1 processor element in each cluster is used, so that the processor element has exclusive access to the memory port. Each `ld` (load) and `st` (store) instruction is replaced by the corresponding solitary equivalent (`ldsol` and `stsol`). The performance of 512 processor elements in solitary mode is compared with 8,192 processor elements in normal mode (Table 4). From Table 3, a solitary memory instruction is 4–5 times faster. By improving the speed of memory access alone a speedup of 1.5 is attained (despite some memory instructions are executed in background). The advantage of solitary mode is lost when 512 processor elements are not enough and backtracking is required, as in `eq10`, `eq20`, `queen(8)` and `queen(10)`.

| | eager creation | | | lazy creation | | | unoptimized | | |
|---|---|---|---|---|---|---|---|---|---|
| benchmark | $t_1$ | heap | trail | $t_2$ | heap | trail | $t_3$ | heap | trail |
| `send` | .019 | 460 | 0 | .019 | 460 | 0 | .019 | 472 | 0 |
| `eq10` | .334 | 1728 | 162 | .334 | 1728 | 162 | .345 | 1756 | 162 |
| `eq20` | .519 | 3636 | 0 | .519 | 3636 | 0 | .528 | 3664 | 0 |
| `queen(4)` | .014 | 440 | 0 | .014 | 452 | 0 | .013 | 444 | 0 |
| `queen(6)` | .042 | 964 | 0 | .042 | 996 | 0 | .042 | 968 | 0 |
| `queen(8)` | .101 | 1656 | 0 | .101 | 1708 | 0 | .101 | 1660 | 0 |
| `queen(10)` | 1.400 | 3208 | 780 | 1.400 | 3276 | 780 | 1.397 | 3212 | 780 |
| `queen(12)` | 92.183 | 5404 | 1764 | 92.177 | 5484 | 1764 | 92.062 | 5408 | 1764 |
| `magic(3)` | .074 | 2044 | 0 | .074 | 2044 | 0 | .077 | 2252 | 0 |
| `magic(6)` | .791 | 13512 | 0 | .791 | 13512 | 0 | .822 | 14100 | 0 |
| `magic(9)` | 2.909 | 41476 | 0 | 2.908 | 41480 | 0 | 3.006 | 42644 | 0 |
| `magich(3)` | .086 | 2324 | 0 | .086 | 2324 | 0 | .090 | 2512 | 0 |
| `magich(6)` | .411 | 9276 | 0 | .411 | 9276 | 0 | .429 | 9800 | 0 |
| `magich(9)` | 1.210 | 24280 | 0 | 1.210 | 24280 | 0 | 1.260 | 25368 | 0 |
| `magich(12)` | 2.544 | 47596 | 0 | 2.543 | 47596 | 0 | 2.644 | 49432 | 0 |

Test conditions: #*proc*=8,192, eager nondeterministic derivation, no solitary memory access, no priority scheduling.

Table 5: Benchmark: on demand creation of bit vectors

## 3.3 Bit Vectors of Domain Variables

The domain of a domain variable is represented by a bit vector. Many newer finite domain constraint programming systems, like *clp(FD)* [3] and *cc(FD)* [16], do not have bit vectors for continuous domains. A bit vector is created on demand only when the domain is broken into two parts because one of the invalid values is removed. For example, if $X \in \{1 \ldots 5\}$, $X \neq 1$, then $X \in \{2 \ldots 5\}$ and a bit vector is unnecessary. However, if $X \in \{1 \ldots 5\}$, $X \neq 3$, then $X \in \{1, 2, 4, 5\}$ and a bit vector representing the domain is created. We test the effect of this optimization in a data-parallel context.

The optimization leads to a slight reduction of both heap consumption and execution time, except for the $n$-queens problem, where both execution time and memory consumption are made worse. We find that several bit vectors may be created for a single domain variable. For example, suppose $X \in \{1 \ldots 5\}$ and $X \neq Y$, where $Y$ is 1 in partition 1, 3 in partition 2 and 5 in partition 3. As a result, a bit vector is created for the $X$ in partition 2 only. If there is another constraint $X \neq Z$, where $Z$ is 3 in partition 1, 4 in partition 2 and 5 in partition 3, a bit vector will be created for partition 1. Two bit vectors have been created, leading to slower execution. Under the heap frame scheme both bit vectors consume heap memory.

We devise an *eager creation* scheme to get around this problem. Note that a bit vector can be created only when a disequality constraint is encountered. In processing a disequality constraint, if any of the physical partitions needs a bit vector, bit vectors are created for all physical partitions. *Lazy creation* refers to the scheme in which bit vectors are created only for partitions in need. The 3 schemes are compared in Table 5. The heap and trail usages are given in bytes.

The eager creation scheme is slightly better than the unoptimized version on average, and is preferred because sometimes very large continuous domains may appear in users' programs. The eager bit vector creation scheme has been used to obtain the performance results in previous sections.

## 3.4 Heap Consumption of the Heap Frame Scheme

The heap frame scheme (Section 2.3.3) is aimed to improve memory access time when building heap terms. However, it has the drawback that sometimes heap fragmentation occurs. Let $H_1$ be the heap consumption (in bytes) when the heap frame scheme is used, and $H_2$ be the heap consumption when the heap frame scheme is not used (each physical partition may have a different value, but we take the highest value). We define *fragmentation* as

$$frag = \frac{H_1 - H_2}{H_1}$$

Our results indicate that fragmentation occurs only in some of our benchmark programs (Table 6). From Table 3, we find that a direct addressing 32-bit store instruction (`st32`) is 2.7 times faster than its local indirect addressing counterpart. We have also shown in Section 3.2 (page 7) that memory access time has a great impact on system performance. Therefore, we believe that the slight memory overhead of the heap frame scheme is acceptable.

| benchmark | *frag* | benchmark | *frag* |
|---|---|---|---|
| `send` | 0 | `magic(3)` | .020 |
| `eq10` | 0 | `magic(6)` | .060 |
| `eq20` | 0 | `magic(9)` | .077 |
| `queen(4)` | 0 | `magich(3)` | .010 |
| `queen(6)` | 0 | `magich(6)` | .012 |
| `queen(8)` | .015 | `magich(9)` | .053 |
| `queen(10)` | .109 | `magich(12)` | .062 |
| `queen(12)` | .140 | | |

Test conditions: #*proc*=8,192, eager bit vector creation, eager nondeterministic derivation, no solitary memory access, no priority scheduling.

Table 6: Benchmark: heap fragmentation

## 3.5 Lazy Nondeterministic Derivation vs Eager Nondeterministic Derivation

Sometimes it is not necessary or even desirable to wait for a deadlock before a nondeterministic derivation step is applied. With *eager nondeterministic derivation*, a labeling process is moved to the ready queue whenever deadlock of any physical partition is detected. *Lazy nondeterministic derivation* refers to the control strategy in which a labeling process is moved to the ready queue only after the deadlock of all physical partitions is detected. *Eager nondeterministic derivation* is chosen over *lazy nondeterministic derivation* for DPAM and we justify this choice using empirical results (Table 7).

From the results, we find that lazy/eager nondeterministic derivation is basically an execution time/degree of parallelism tradeoff. Eager nondeterministic derivation creates partitions more aggressively. As a result more parallelism can be exploited, leading to better performance. It is worth noting that although lazy nondeterministic derivation reduces the number of backtrackings, it does not lead to any performance gain because of a lower degree of parallelism.

## 3.6 Priority Scheduling

One way to increase the degree of parallelism is to schedule those processes resumed by a labeling process ahead of all the others. We have implemented a prototype of this *priority scheduling* scheme on top of our system and its effect on performance is measured (Table 8).

Priority scheduling is faster on average, with best performance on the $n$-queens problem. A possible explanation is that priority scheduling is more suitable when disequality constraints are predominant.

## 3.7 Execution Profile

We measure the time spent in nondeterministic derivation (*ND*), constraint solving (*C*), backtracking (*BT*) and the execution time up to the first deadlock (*FDL*). The backtracking time includes the time to update the choice point, unwind the trail and restore the process queues. The percentage time is shown in Table 9.

It is evident that constraint solving dominates execution time for large problems. This implies that compiler optimization may not be as useful as an efficient constraint solver. In large problems, very little time is spent before the first nondeterministic derivation step, which is sequential.

| | lazy nondet. derivation | | | | eager nondet. derivation | | | | |
|---|---|---|---|---|---|---|---|---|---|
| benchmark | $t_1$ | $DL$ | $P$ | $BT$ | $t_2$ | $DL$ | $P$ | $BT$ | $t_2/t_1$ |
| `send` | .019 | 2 | 4 | 0 | .019 | 2 | 4 | 0 | 1.00 |
| `eq10` | .537 | 4 | 164 | 0 | .334 | 4 | 181 | 8 | .62 |
| `eq20` | .791 | 3 | 116 | 0 | .519 | 3 | 140 | 0 | .66 |
| `queen(4)` | .016 | 2 | 6 | 0 | .014 | 2 | 6 | 0 | .88 |
| `queen(6)` | .054 | 4 | 40 | 0 | .042 | 4 | 40 | 0 | .78 |
| `queen(8)` | .201 | 7 | 416 | 0 | .101 | 7 | 548 | 0 | .50 |
| `queen(10)` | 3.542 | 27 | 2397 | 11 | 1.400 | 27 | 2399 | 17 | .40 |
| `queen(12)` | 209.056 | 952 | 4171 | 1027 | 92.183 | 971 | 4165 | 1191 | .44 |
| `magic(3)` | .082 | 2 | 8 | 0 | .074 | 2 | 11 | 0 | .90 |
| `magic(6)` | .814 | 2 | 32 | 0 | .791 | 2 | 38 | 0 | .97 |
| `magic(9)` | 3.379 | 2 | 74 | 0 | 2.909 | 2 | 83 | 0 | .86 |
| `magich(3)` | .086 | 2 | 7 | 0 | .086 | 2 | 7 | 0 | 1.00 |
| `magich(6)` | .425 | 2 | 20 | 0 | .411 | 2 | 21 | 0 | .97 |
| `magich(9)` | 1.284 | 2 | 42 | 0 | 1.210 | 2 | 45 | 0 | .94 |
| `magich(12)` | 2.721 | 2 | 74 | 0 | 2.544 | 2 | 78 | 0 | .93 |

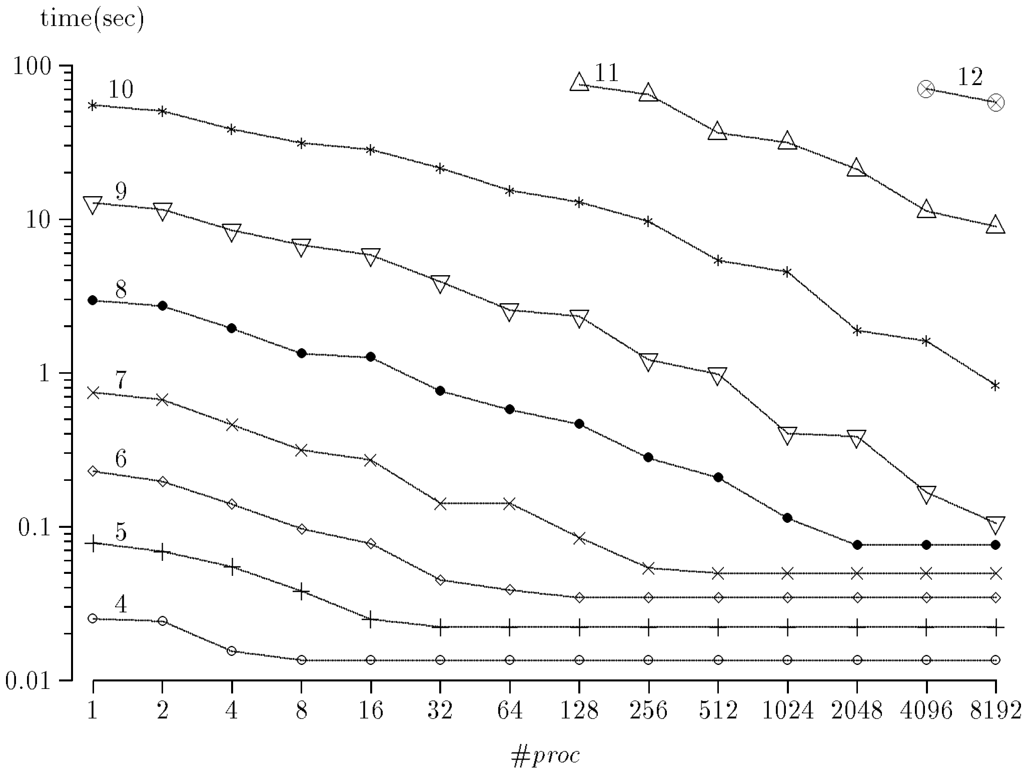
Test conditions: #*proc*=8,192, eager bit vector creation, no solitary memory access, no priority scheduling.

Table 7: Benchmark: lazy nondeterministic derivation vs eager nondeterministic derivation

| | priority scheduling | | | | w/o priority scheduling | | | | |
|---|---|---|---|---|---|---|---|---|---|
| benchmark | $t_1$ | $DL$ | $P$ | $BT$ | $t_2$ | $DL$ | $P$ | $BT$ | $t_2/t_1$ |
| `send` | .019 | 2 | 4 | 0 | .019 | 2 | 4 | 0 | 1.00 |
| `eq10` | .299 | 4 | 226 | 0 | .334 | 4 | 181 | 8 | 1.12 |
| `eq20` | .429 | 3 | 138 | 0 | .519 | 3 | 140 | 0 | 1.21 |
| `queen(4)` | .013 | 2 | 6 | 0 | .014 | 2 | 6 | 0 | 1.08 |
| `queen(6)` | .034 | 4 | 46 | 0 | .042 | 4 | 40 | 0 | 1.24 |
| `queen(8)` | .075 | 7 | 564 | 0 | .101 | 7 | 548 | 0 | 1.35 |
| `queen(10)` | .827 | 27 | 2530 | 15 | 1.400 | 27 | 2399 | 17 | 1.69 |
| `queen(12)` | 57.264 | 952 | 4178 | 1027 | 92.183 | 971 | 4165 | 1191 | 1.61 |
| `magic(3)` | .076 | 2 | 8 | 0 | .074 | 2 | 11 | 0 | .97 |
| `magic(6)` | .698 | 2 | 38 | 0 | .791 | 2 | 38 | 0 | 1.13 |
| `magic(9)` | 3.204 | 2 | 83 | 0 | 2.909 | 2 | 83 | 0 | .91 |
| `magich(3)` | .090 | 2 | 7 | 0 | .086 | 2 | 7 | 0 | .96 |
| `magich(6)` | .463 | 2 | 22 | 0 | .411 | 2 | 21 | 0 | .89 |
| `magich(9)` | 1.315 | 2 | 45 | 0 | 1.210 | 2 | 45 | 0 | .92 |
| `magich(12)` | 2.729 | 2 | 78 | 0 | 2.544 | 2 | 78 | 0 | .93 |

Test conditions: #*proc*=8,192, eager bit vector creation, eager nondeterministic derivation, no solitary memory access.

Table 8: Benchmark: priority scheduling




Test conditions: eager bit vector creation, eager nondeterministic derivation, no solitary memory access, priority scheduling.

Figure 4: Benchmark: run time of $n$-queens program

| benchmark | *FDL* | *C* | *ND* | *BT* |
|---|---|---|---|---|
| `send` | 75.0 | 78.0 | 2.5 | 0 |
| `eq10` | 7.0 | 95.8 | .7 | 1.9 |
| `eq20` | 16.2 | 98.1 | .9 | 0 |
| `queen(4)` | 52.3 | 59.7 | .06 | 0 |
| `queen(6)` | 35.8 | 72.7 | .06 | 0 |
| `queen(8)` | 25.8 | 80.6 | .05 | 0 |
| `queen(10)` | 2.9 | 91.3 | 3.4 | 2.7 |
| `queen(12)` | .08 | 90.9 | 2.8 | 3.7 |
| `magic(3)` | 40.6 | 57.1 | .9 | 0 |
| `magic(6)` | 11.0 | 63.2 | .1 | 0 |
| `magic(9)` | 6.0 | 60.4 | .04 | 0 |
| `magich(3)` | 36.5 | 61.2 | .9 | 0 |
| `magich(6)` | 19.7 | 66.6 | .3 | 0 |
| `magich(9)` | 14.5 | 68.3 | .1 | 0 |
| `magich(12)` | 11.3 | 68.5 | .06 | 0 |

Test conditions: *#proc*=8,192, eager bit vector creation, eager nondeterministic derivation, no solitary memory access, no priority scheduling.

Table 9: Benchmark: execution profile (in %)

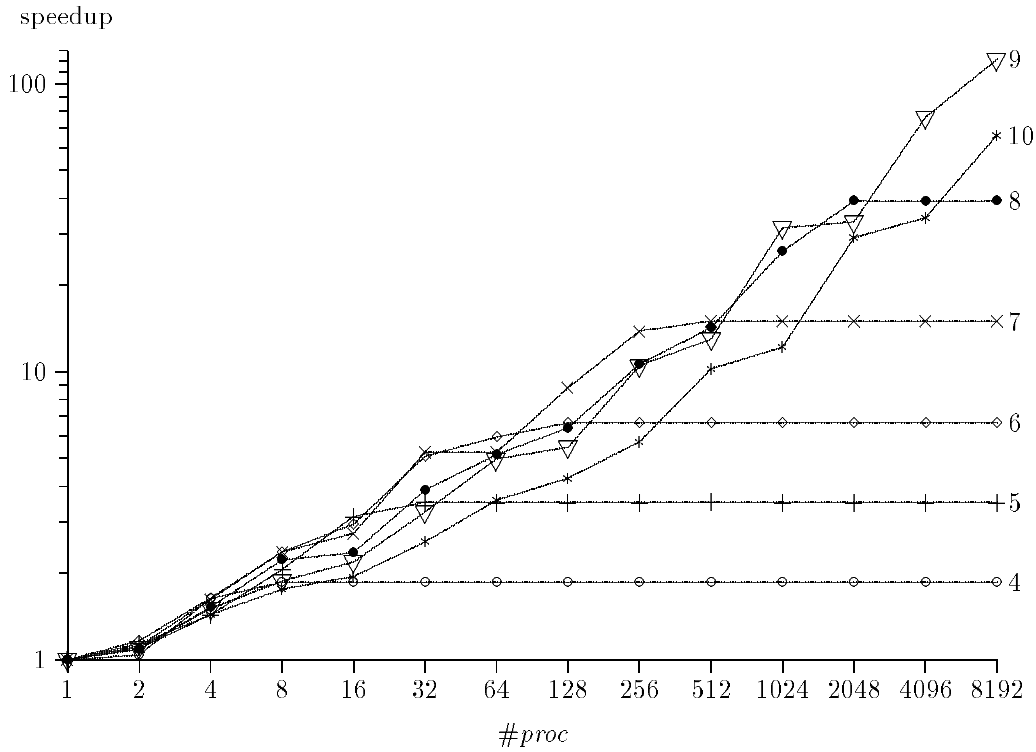
Hence most of the time is spent on parallel execution. Nondeterministic derivation and backtracking overhead are almost negligible.

### 3.8 Effect of the Number of Processor Elements on Performance

Figure 4 shows the execution time of $n$-queens, for $4 \leq n \leq 12$, with 1 to 8,192 processor elements. We use priority scheduling because it is particularly suitable for the $n$-queens problem.

We do not have enough memory (each processor element has only 64K bytes) to run 13-queens. Some data points are missing from the above graph for the same reason. This is only the limitation of the particular implementation platform we are using, but not a limitation of Firebird or DPAM.

Figure 5 shows the speed up with respect to the execution time using 1 processor element as the number of processor elements is varied.




Test conditions: eager bit vector creation, eager nondeterministic derivation, no solitary memory access, priority scheduling.

Figure 5: Benchmark: speedup of N-Queens Program

The speed up value continues to increase until there are too many processor elements to be utilized. Otherwise, the speed up value is fairly independent on problem size and fairly constant for a given number of processor elements. Speed up is scalable but sublinear. For instance, the speed up of 7-queens levels at about 15, but 512 processor elements are required to obtain this speed up. The reason for this is that the processor elements are divided evenly in a nondeterministic derivation step, which may not be the optimal processor allocation strategy. Furthermore, a processor element will remain idle after failure until the system backtracks.

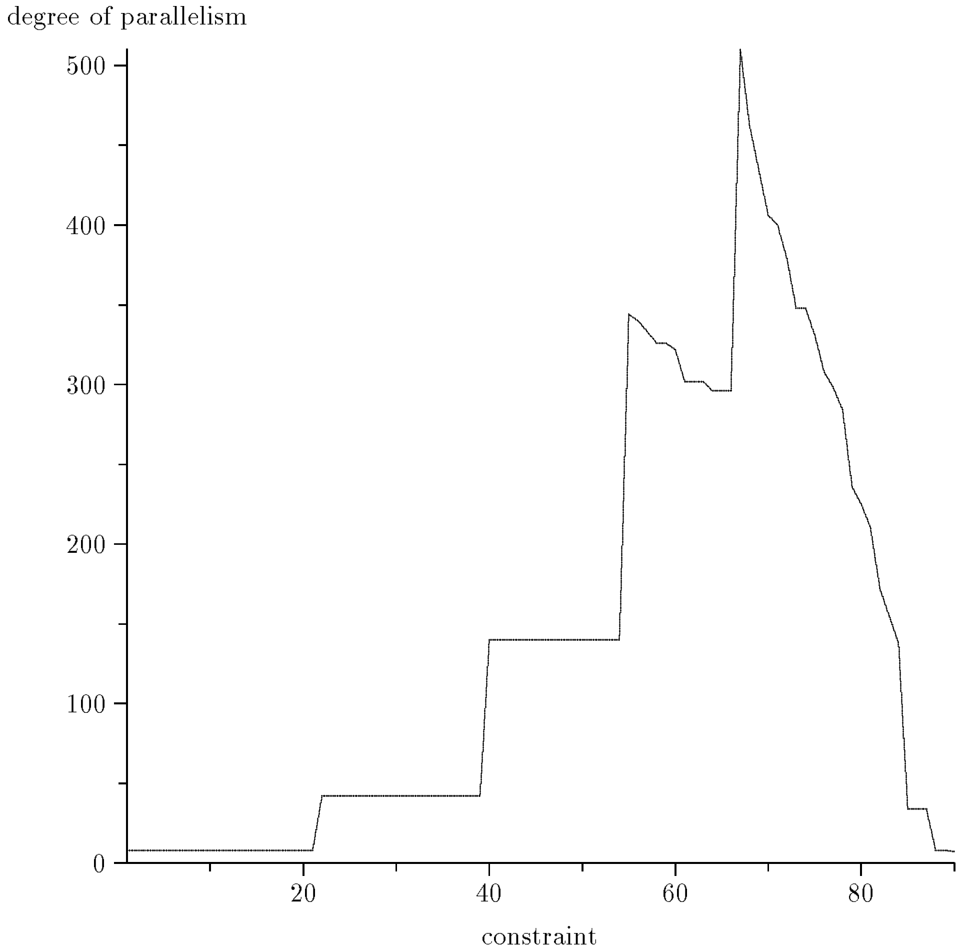
We obtain a maximum speed up of 121 for 9-queens. This may seem disappointing given that 8,192 processor elements are used, but to the best of our knowledge, no other parallel implementation of a concurrent constraint programming language has ever attained a speedup of 2 orders of magnitude. Furthermore, the shared-memory multiprocessor architecture, which is employed by most of the other parallel implementations of concurrent constraint languages, is simply incapable of scaling to that level.

Due to space limitations, we have only analyzed the performance of $n$-queens in detail but not all other programs exhibit a similar behaviour to $n$-queens.

## 3.9 Change of the Degree of Parallelism During Execution

We are interested in the change of the degree of parallelism during execution. We count the number of active partitions when each constraint is executed and plot the graph in Figure 6. We do not include anything before the first nondeterministic derivation step, with the understanding that

1. there can only be a single active partition, and




Test conditions: #*proc*=8,192, eager bit vector creation, eager nondeterministic derivation, no solitary memory access, priority scheduling.

Figure 6: Execution trace of 8-queens, #*proc*=8,192

2. execution time before the first nondeterministic derivation is only a small portion of the total execution time.

Furthermore, since we have found out that priority scheduling can improve the performance of the system, we use it when obtaining the plot. Other parameters remain unchanged.

From the plot we can identify the nondeterministic derivation steps as sudden leaps in the degree of parallelism. A peak of 510 is attained. After that, the degree of parallelism drops because of the failure of some partitions. We are actually approaching the theoretical limit of or-parallelism. Using or-parallelism alone the peak of 510 can never be exceeded, although the more flexible MIMD architecture may be able to exploit higher degrees of parallelism after that peak.

We conclude that the inherent limitation of or-parallelism will show up in any massively parallel implementation. Degree of parallelism rises slowly at the beginning, making full utilization of processor elements impossible. After the peak is reached, some or-branches fail, again limiting the degree of parallelism and hence processor element utilization.

Next, we show that when the number of processor elements is very small when compared to the number of or-branches, reasonably high processor element utilization can be maintained. For the 8-queens problem, 64 processor elements are just enough for the first two nondeterministic derivation steps. The plot is shown in Figure 7.

# 4 A Comparison with SIMD MultiLog

SIMD MultiLog [9] is another or-parallel system implemented on the MasPar MP-1 (equivalent to the DECmpp we are using). A new *disj* operator is introduced. Solutions to a disjunctive goal *disj* $G$ are collected. The solutions form a disjunctive set of environments and goals appearing after $G$ can be executed in these environments in parallel. No special language constructs are needed in Firebird.

[4], SIMD MultiLog and Firebird all execute goals over a disjunctive set of environments[7], exploiting or-parallelism. [4] relies on a vectorizing compiler, MultiLog uses solution aggregation and in Firebird the environments fall out of the labeling operation on domain variables naturally.

Like our approach, MultiLog has the advantage that traditional compilation techniques are applicable. Furthermore, *engine* variables which reside on the host computer are distinguished manually from *multi* variables which reside on the processor elements. This leads to higher time and space efficiency. See [11] for a theoretical analysis of the resulting performance gain. Automatic compilation of the engine/multi distinction is possible and there is no inherent technical difficulty to add it to Firebird.

The current MasPar implementation of SIMD MultiLog has the overhead of environment copying which is not necessary in our Firebird implementation. [9] points out that environment copying is a performance bottleneck. However, there is no technical difficulty to incorporate our processor element mapping technique to SIMD MultiLog, and we would expect a tremendous performance gain. Another drawback of the current SIMD MultiLog implementation is that processor element utilization can be limited when backtracking is used. Furthermore, the user has to specify how many environments should be allocated or how much memory should be used. This is not necessary in our Firebird implementation. Again, we expect a performance gain if our backtracking scheme is incorporated into SIMD MultiLog.

The performance of SIMD MultiLog cannot be directly compared with Firebird because the former is a logic programming system but the latter is a concurrent constraint programming system. However, there is no better system for comparison because as far as we know there are no other data-parallel concurrent constraint programming systems and SIMD MultiLog is implemented on the same platform as Firebird.

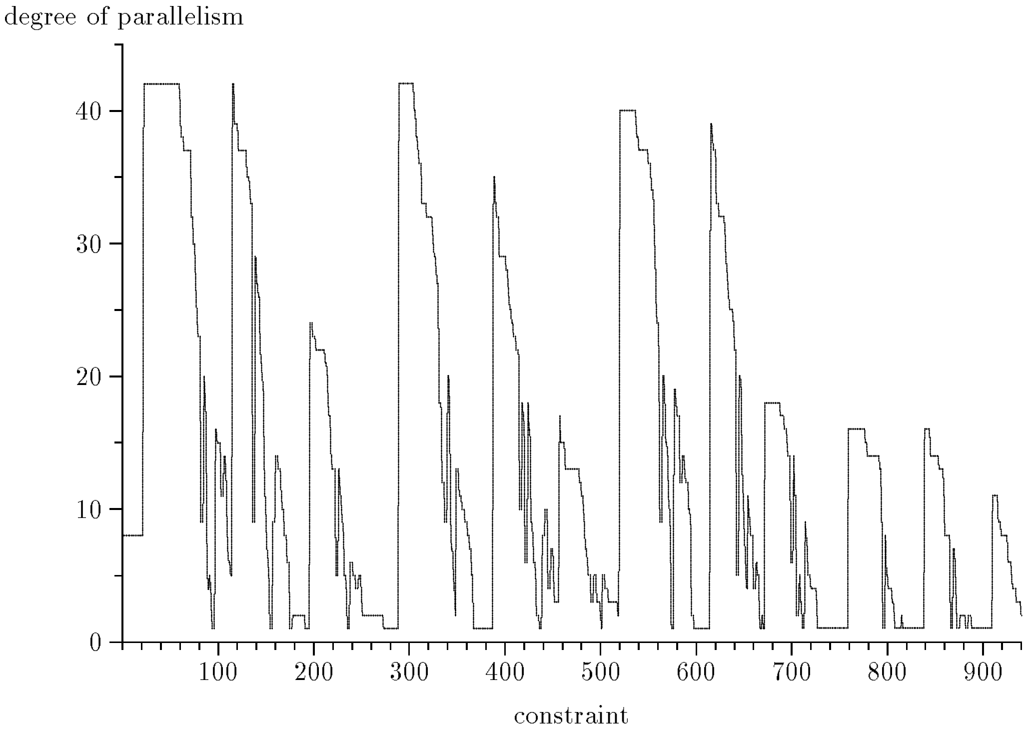
When the execution time of MultiLog using 8,192 processor elements and that of using a workstation are compared, 10 out of the 16 programs in a benchmark suite [10] (*Path, Sat, Tri, WIM, 12-queens, Knight, Waltz, 11-Bratko, Cube, 11-queens*) have a speedup at or below 3.5. The remaining 6 are *WIM-dyn* (8.1), *Costas* (9.6), *Costas-dyn* (30.7), *20 bits-pal* (295.2), *20 bits-pal-n* (1872.7), *24 bits-pal-n* (2089.9). *n bits-pal* is a program which gives all $n$-bit strings which are palindromic (by generate and test, not unification) and *bits-pal-n* is the same program except that naïve reverse is used.

We conclude that SIMD MultiLog attains excellent speedup in some special cases. This is because, for example, the *bits-pal-n* benchmark forks 8,192 branches very quickly using simple generate-and-test and naïve reverse applied in parallel gives rise to a very high speedup. If we omit such generate-and-test programs from the benchmark set, the maximum speedup of SIMD MultiLog is 30.7 (*Costas-dyn*). Firebird, on the other hand, is up to 20.3 times faster than a workstation implementation of CHIP (9-queens). Therefore, the performance of the two systems are roughly equal. Many implementation techniques of one system can be applied to the other and the combination of both SIMD MultiLog and Firebird is probably more powerful than either alone.

# 5 Conclusion

In this paper, we present some preliminary performance results of our massively parallel implementation of Firebird. Our figures indicate that a speedup of 2 orders of magnitude is possible when we compare the performance using 8,192 processor elements

[7] An *environment* in MultiLog is analogous to a *partition* in Firebird




Test conditions: #*proc*=64, eager bit vector creation, eager nondeterministic derivation, no solitary memory access, priority scheduling.

Figure 7: Execution trace of 8-queens, #*proc*=64

and the performance using a single processor element of the same machine. Furthermore, the speedup is scalable, provided that the problem size is large enough for effective exploitation of or-parallelism. It seems at first glance that a speedup of 121 is disappointing given the 8,192 processor elements used to obtain it, but we argue that the shared-memory architecture is simply incapable of scaling to that level. In fact, the design of our Data-Parallel Abstract Machine is optimized for a maximum degree of parallelism, and is most suitable in situations where resources are repleted and top performance is sought after.

## Acknowledgement

We would like to express our thanks to Samuya Debray for proof-reading this manuscript and presenting it. We would also like to thank Donald A. Smith, Edward Tsang, Albert Lai and Jimmy Lee for their invaluable comments and suggestions.

## References

[1] H. Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press, 1991.

[2] B. Carlson, M. Carlsson, and D. Diaz. Entailment of finite domain constraints. In *Logic Programming: Proceedings of the Eleventh International Conference*, S. Margherita Ligure, Italy, 1994. MIT Press.

[3] D. Diaz and P. Codognet. A minimal extension of the WAM for clp(FD). In D. S. Warren, editor, *Logic Programming: Proceedings of the Tenth International Conference*, pages 774–790, Budapest, Hungary, 1993. The MIT Press.

[4] Y. Kanada, K. Kojima, and M. Sugaya. Vectorization techniques for Prolog. In *Proceedings of the ACM International Conference on Supercomputing*, pages 539–549, St. Malo, 1988.

[5] S. Kliger and E. Shapiro. From decision trees to decision graphs. In S. Debray and M. Hermenegildo, editors, *Proceedings of the 1990 North American Conference on Logic Programming*, pages 97–116, Austin, 1990. ALP, The MIT Press.

[6] M. J. Maher. Logic semantics for a class of committed-choice programs. In J.-L. Lassez, editor, *Logic Programming: Proceedings of the Fourth International Conference*, pages 858–876, Melbourne, 1987. The MIT Press.

[7] V. A. Saraswat. A somewhat logical formulation of CLP synchronisation primitives. In R. A. Kowalski and K. A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, pages 1298–1314, Seatle, 1988. ALP, IEEE, The MIT Press.

[8] V. A. Saraswat and M. Rinard. Concurrent constraint programming. In *Proceedings of the 17th Symposium on Principles of Programming Languages*, pages 232–244, San Fransisco, 1990.

[9] D. A. Smith. MultiLog: Data or-parallel logic programming. In D. S. Warren, editor, *Logic Programming: Proceedings of the Tenth International Conference*, pages 314–331, Budapest, Hungary, 1993. The MIT Press.

[10] D. A. Smith, 1994. Personal communication.

[11] D. A. Smith. Why multi-SLD beats SLD (even on a uniprocessor). In *Proceedings of the Sixth International Symposium on Programming Language Implementation and Logic Programming*, Madrid, Spain, 1994. Springer-Verlag.

[12] B. M. Tong and H. F. Leung. Concurrent constraint logic programming on Massively parallel SIMD computers. In D. Miller, editor, *Logic Programming: Proceedings of the 1993 International Symposium*, pages 388–402, Vancouver, Canada, October 1993. The MIT Press.

[13] B. M. Tong and H. F. Leung. Implementation of a data-parallel concurrent constraint programming system. In *Proceedings of the First International Symposium on Parallel Symbolic Computation*, pages 382–393, Linz, Austria, September 1994. World Scientific.

[14] K. Ueda. Guarded horn clauses. In E. Wada, editor, *Logic Programming '85 — Proceedings of the 4th Conference*, Lecture Notes in Computer Science 221, pages 168–179, Tokyo, July 1985. Springer-Verlag.

[15] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

[16] P. Van Hentenryck, V. A. Saraswat, and Y. Deville. Design, implementation and evaluation of the constraint language cc(FD). Technical Report CS-93-02, Department of Computer Science, Brown University, Providence, 1993.