Proceedings of PASCO'94

Implementation 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
email: bmtong@cs.cuhk.hk, lhf@cs.cuhk.hk

Abstract: In [12], we proposed Firebird, a new concurrent constraint programming language [9] which supports both concurrency and data-parallelism. Committed-choice indeterminism and don't know nondeterminism are integrated, but in a manner different from the Andorra Model [16]. Instead of nondeterminate clauses, finite domain variables form the basis of don't know nondeterminism in Firebird. After a domain variable is *labeled*, all possible values in its domain are attempted in an or-parallel manner. A data-parallel implementation of Firebird has been built. The aim of this paper is to present: 1, the scheduler and 2, the parallel backtracking scheme of Firebird, which are not described in [12]. Our *concurrent process scheduler* uses simple data-parallel algorithms for process resumption and deadlock detection. On the other hand, when the number of processor elements is not enough for exploiting or-parallelism, the system resorts to *parallel backtracking* automatically. We present the data structures necessary for maintaining a vector trail stack and methods for restoring all process queues to their original states upon backtracking.

Keywords: Data-Parallelism, Finite Domain Constraint, Concurrent Constraint Programming

1 Introduction

The aim of this paper is to describe the design of: 1, a *concurrent process scheduler* and 2, a *parallel backtracking* scheme for the data-parallel execution of a concurrent constraint programming language [9].

In our last paper [12], we presented, to the best of our knowledge, the first data-parallel concurrent constraint programming system. We found that finite domain constraint languages can be implemented efficiently on massively parallel SIMD computer systems. A new language called *Firebird* has been designed. Its syntax is similar to mainstream logic programming languages, in particular, *flat GHC* [13]. 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. In an *indeterministic derivation step*, execution consists of guard tests, commitment and spawning in the same manner as committed-choice concurrent constraint programming 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. In [12], we also described the data-parallel execution model of Firebird.

*Present address: Department of Computer Science, University of Arizona, Tucson, AZ 85721, U.S.A. email: bmtong@cs.arizona.edu

1.1 Motivation

1. In our last paper [12], the *concurrent process scheduler* is not described. However, some simple data-parallel algorithms for the resumption of processes and deadlock detection are necessary for the data-parallel implementation. These problems will be discussed in this paper.

2. Many real-world problems are combinatorial in nature. Even thousands of processor elements may be depleted when or-parallelism is exploited. This problem has not been solved in [12]. In this paper, we propose to use backtracking. Methods are needed for restoring the heap and the process queues maintained by the concurrent process scheduler to their original states upon backtracking.

1.2 Related Work

Nilsson and Tanaka [7, 8] describe implementations of the committed-choice, flat concurrent logic programming *Fleng* on a vector supercomputer and a data-parallel computer. They attack the problem of scheduling and-parallel processes in a language without constraints. Barklund and Millroth [3] propose a scheme called *Bounded Quantifications* to compile recursive constructs in Prolog to iterative ones suitable for data-parallel execution. An abstract machine and its performance results on a Connection Machine simulator are presented in [2]. Smith [10] describes a data or-parallel logic programming language MultiLog and its implementation on a variety of architectures.

2 Firebird: A Review

In this section, we shall give a brief review of [12]. The reader is referred to [15, 5] for an introduction to the use of finite domain constraints in logic programming.

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 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 is said to be *labeled* [14] 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 principle functor and arity of each atom is the same. In the 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 takes such vectors as arguments and operates on the elements of the 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])`. After the program has been executed, the constraint store has the following constraints. All the goals have been reduced, and all 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 on a SIMD machine. A second nondeterministic derivation will occur. If each branch chooses to create a choice point on `X2`, there will be 3+2+2+2+3 =

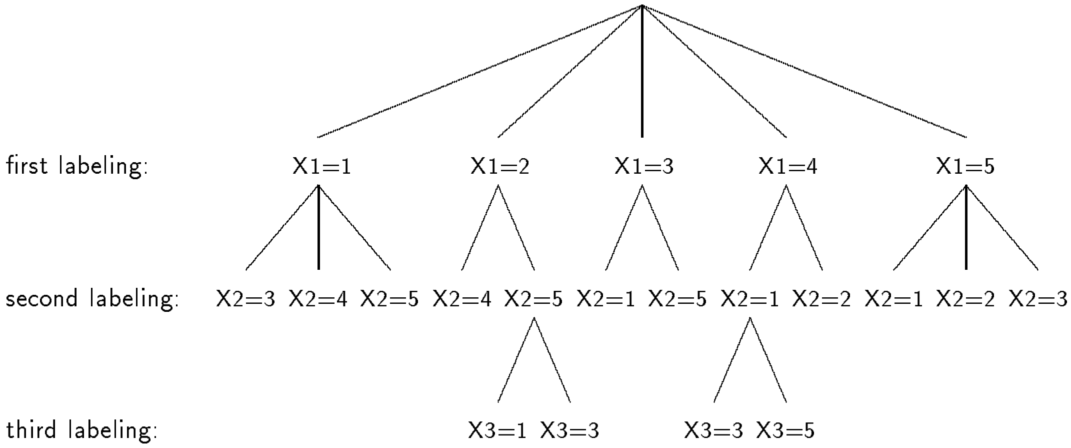


Figure 2: Or-parallel branches in the 5-queens example

12 branches (see Figure 2). Thus the next 9 constraints can be solved with 12 times speedup. `X3` will also be labeled, giving rise to a total of 14 branches. Therefore, the remaining 9 constraints can be evaluated with 14 times speedup. 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 shall 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 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 [10], our model has the advantage that interprocessor communication is not necessary.

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 $H > 0$ will commit to the first clause and those with constraint stores entailing $H = 0$ will 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.

However, 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 detected this at compile time using a *join algorithm*. At present our compiler joins all 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.

2.3 Data-Parallel Abstract Machine

The *Data-Parallel Abstract Machine* (DPAM)[1] is the basis of the implementation of Firebird. The knowledge of WAM [1] is assumed.

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

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

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 *mask* bit to turn a processor element on/off. A processor element executes instructions if and only if its mask bit is set, except that there are special instructions which move a bit to the mask bit uncontingently.

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 [6] 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*.

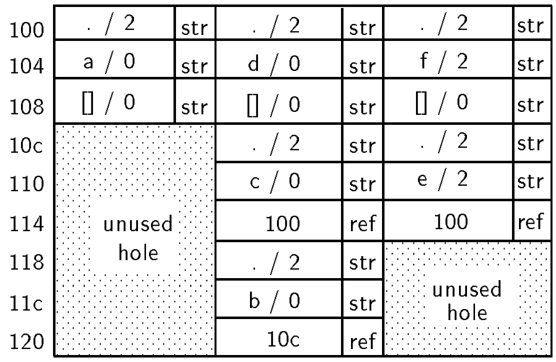
2.3.3 Heap Frames

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[2] *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 the 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 leads to faster execution.



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

Figure 3: Example: a heap frame

3 Concurrent Process Scheduler

3.1 Procedure Calling Convention And Process Creation

In DPAM, a *procedure* is defined to be an executable subroutine. An n-ary goal or tell constraint is compiled to a procedure which takes n input arguments, which are stored in general purpose vector registers `p1` to `pn`. The *mask* bit vector defines the set of physical partitions in which the procedure is applicable.

In WAM [1], failure is like a fault or an exception. When a failure occurs execution continues

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

directly at the next clause. This is not possible in a data-parallel system because the failure of a physical partition does not imply the failure of all physical partitions. Therefore, upon return, the procedure signifies that a physical partition has succeeded by setting the mask bit and that a physical partition has failed by clearing the mask bit.

In Figure 4, we illustrate how the mask bit vector is changed during the execution of 3 call instructions on a 8-processor machine. Note that the mask shrinks monotonically.

procedure call	input mask	output mask
`call p/0`	11111111	11011111
`call q/0`	11011111	01011001
`call r/0`	01011001	00011000

Figure 4: Example: mask bit vector

The *concurrent process scheduler* is responsible for executing procedures. The scheduler treats all procedures as black boxes. It simply performs a subroutine call to the procedure after loading the input arguments. A procedure may call other procedures in the same manner.

If a procedure (a compiled goal or tell constraint) wants to suspend, it calls a system library subroutine `suspend_process` which saves all the argument registers and other status information (*e.g.* continuation pointer) in a *process structure*. The procedure must tell `suspend_process` the set of physical partitions which need suspension by setting the mask bit vector. Then the procedure returns control to the caller with the mask bit set to the set of physical partitions which have either succeeded or suspended. In other words, the caller cannot distinguish whether the callee has succeeded or suspended (the scheduler can, of course).

The evaluation of tell constraints is left to the particular implementation. DPAM defines only the interface—the constraint must obey the abovementioned calling convention, and it must call `suspend_process` for a suspension.

In DPAM, ask constraints never appear in any process queues. They do not enter any parts of the scheduling system. An ask constraint is *not* a process, but is treated as a function. For instance, an arithmetic ask constraint is compiled to a function which takes n arguments in `p1` to `pn` and returns one of the following values: $<, \leq, >, \geq, =, \neq$, *suspend*. The return value must be put in `p0`. The implementation of the function is left unspecified. The decision graph compiler generates code which calls the function and branches according to the return value.

3.2 Process Structure

A *process structure* must contain enough information for the reinvocation of a process when the process is resumed. In DPAM, the process structure is scalar and resides on the host computer. The arguments are stored separately in the vector argument stack. The process structure consists of the following fields.

- *process-id*
- continuation pointer
- number of arguments
- pointer to the argument vectors on the argument stack
- pointers to the next and the previous process structures
- pointers to the next and the previous processes in a suspension queue (to be discussed in Section 4.2.1)

To save storage, the mask of a process is indicated by its first argument vector. A physical partition is outside the mask of a process if it has a zero first argument. A pseudo-argument vector which stores the mask will be added to 0-ary processes.

3.3 Overview of System Queues

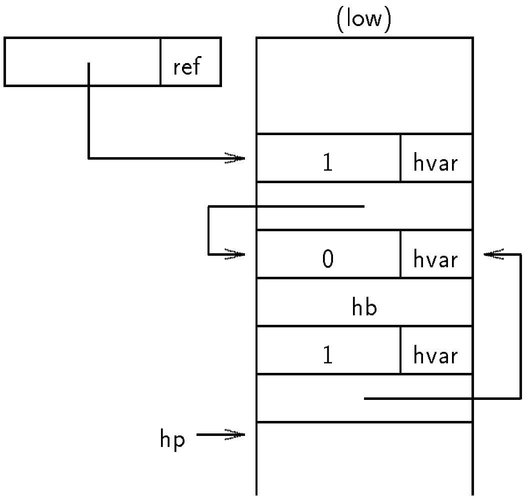
The system maintains *hash queues*, the *ready queue*, the *resumption queue* and *free lists* for the creation and scheduling of processes. Both the next and previous pointer fields are used in *hash queues*, which are doubly linked circular lists used to locate a process with a given *process-id* in a resumption operation. Alternatively, the next and previous pointers can be used as *primary* and *secondary linkage pointers* so that a process can be in two singly linked lists simultaneously. The *ready queue*, the *resumption queue* and *free lists* use the primary linkage pointer and the *labeling queue* uses the secondary one.

Each process in the ready queue is executed and then the process structure is placed back in one of the free lists. When a process is created, DPAM tries to reuse process structures in the free lists first before it allocates new space on the process stack and the argument stack. When a process is resumed it is first placed in the resumption queue. After the ready queue is exhausted the processes in the resumption queue will be moved to the ready queue.

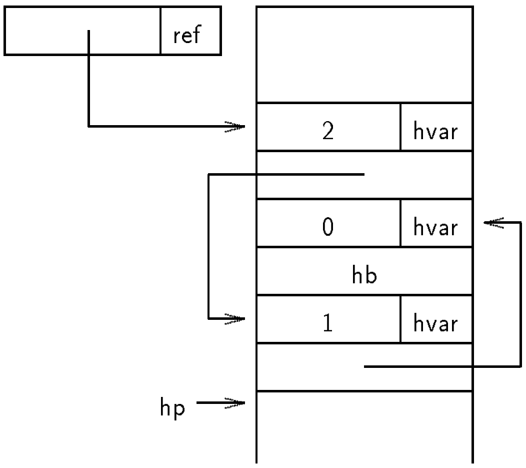
3.4 Suspension and Resumption

There are two possible ways of implementing suspension and resumption.

Method 1 If there is a suspension, a new process structure will be created, and a process-id will be assigned to it. The new process will be placed in one of the hash queues. Its *process-id* will be stored in a suspension list node which becomes the head of the suspension list of the unbound variable. New suspension list nodes can be inserted at the head of the suspension list, by copying the old list head to the top of the heap and replacing the old list head by the newly created node (see Figure 5).



1. suspension list head copied to top of heap



2. old suspension list head is overwritten

Figure 5: Suspension

When a variable is assigned, each *process-id* in the suspension lists is fetched. If a process can be found in one of the hash queues, it is removed from the hash queue and added to the resumption queue. Otherwise, the process has already been resumed by another assignment and can be ignored.

Naturally, each processor element may have a suspension list with different processes. To resume them the following algorithm is devised.

1. Select an arbitrary processor element p.
2. Each processor element fetch the *process-id* in the head of the suspension list, unless the list is empty.
3. Let the *process-id* as obtained by processor element p be i. Resume the process with *process-id* i.
4. Every processor element whose *process-id* in the suspension list node is equal to i may proceed to the next suspension list node (unless the list is empty).
5. Repeat steps 3–4 until the entire suspension list of processor element p has been traversed.
6. Select another processor element p and repeat steps 3–5 until the suspension list of every processor element has been traversed.

See Figure 6 for an example of the above algorithm at work.

Method 2 The traditional scheme using no hash queues can be used. Instead of a *process-id*, the head of each suspension list node contains the pointer to a scalar hanger [4] residing on the host. The hanger will point to the process structure. When a process is resumed the hanger will be zeroed to prevent the process from being resumed more than once. An algorithm similar to that of method 1 can be used to resume a vector of suspension lists.

The first method will be slower because of searching but it makes simpler memory management possible. This is because it is not necessary to write a separate garbage collector to clear obsolete hangers from the host's memory. Both schemes allow old process structures to be freed and reused.

To resume a process, we can do either of the following.

1. The process is resumed in all physical partitions even if the variable is assigned in some physical partitions only.

2. The process is split into two. One of the processes, with the mask set to consist of the physical partitions in which the resumption has taken place, will be placed in the resumption queue. The other process, whose mask consists of physical partitions in which the resumption has not taken place, will be left in the suspension list.

We took the first method because it is simpler and does not have the overhead of splitting. On a SIMD computer, the time to execute a process is unrelated to the number of active processor elements. The falsely resumed physical partitions will execute the process alongside with the others without any additional cost. If these physical partitions commit we actually save execution time. Otherwise, they will just re-suspend.

Unlike *Parallel Parlog* [4], we do not make any distinction between *single-suspension* and *multi-suspension* because

1. The Parallel Parlog scheme improves the efficiency of *single-suspension*, but for finite domain constraints, multi-suspension is more the rule than the exception. All unary constraints in the form X *op* c, where *op* is an equality, inequality or disequality relation and c is an integer, can be solved immediately without suspension. Other constraints suspend on two or more variables (*multi-suspend*).

2. In Parallel Parlog, the pointer to the next process structure resides in the process structure itself in a single-suspension. However, we have a vector of variables in a variable assignment, resulting in several threads of processes to be resumed in general. Thus the *next* field of the DPAM process structure, which is scalar, cannot be used.

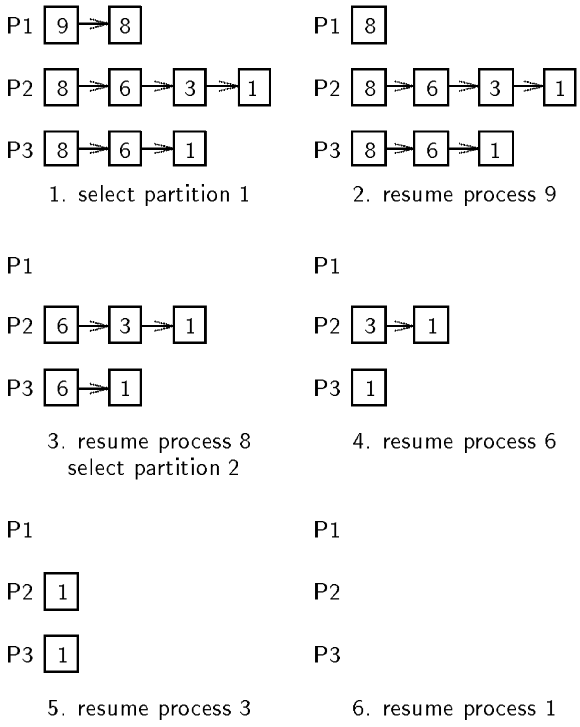


Figure 6: Example: resumption of suspension lists

3.5 Deadlock Detection

A *labeling process* is responsible for labeling a *d*-variable in a nondeterministic derivation step. It is created and placed in the *labeling queue* whenever a process suspends on a *d*-variable for the first time, unless the *d*-variable is originally a *h*-variable whose domain is inferred from an equality constraint. For example, suppose X is an *h*-variable and there is a constraint $X = Y+1$, $Y \in \{1,2\}$, the domain of X is inferred to be $\{2,3\}$. It is sufficient to label Y only but a labeling process for X is never created.

In order to detect deadlock of individual physical partitions, a process count and a deadlock flag are maintained in each physical partition. The process count of a physical partition keeps track of the number of processes in that physical partition. The deadlock flag is initially set. When a process is resumed in some physical partitions, the deadlock flags of those physical partitions are cleared. When the ready queue is exhausted, the physical partitions are checked for deadlock.

1. If the process counts of any physical partitions are zero, those physical partitions have succeeded.

2. If there are any physical partitions which have not succeeded, and any of them have set deadlock flags, a deadlock has occurred. A process is moved from the labeling queue to the ready queue. If the labeling queue is empty, the deadlock is irrecoverable. An irrecoverable deadlock is treated as an error.

3. If the resumption queue is empty, a deadlock has occurred. A process is moved from the labeling queue to the ready queue. If the labeling queue is empty, the deadlock is irrecoverable.

4. After checking, append all processes in the resumption queue to the ready queue. (In other words, if a deadlock has occurred, the ready queue will consist of a labeling process followed by other resumed processes. Otherwise, the ready queue will consist of resumed processes only.) Set deadlock flag for all physical partitions again.

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. If the finished partitions go back to the last choice point immediately, a higher degree of parallelism may result but we have not yet implemented it to evaluate it empirically.

All partitions will resort to backtracking if any of the partitions do not have enough processor elements. In this way, more parallelism can be exploited if there is a nondeterministics derivation step afterwards with domain sizes small enough so that backtracking is unnecessary. In synchronous backtracking, a physical partition will be idle after all its alternatives have been attempted. As a result, even if we exploit or-parallelism in partitions with enough processor elements, such partitions will still have to wait for those partitions using backtracking.

Like WAM [1], a choice point is freed before the last possible value is attempted (since only one choice is left, the system may 'commit' to that choice). Whereas WAM has `try_me_else`, `retry_me_else` and `trust_me` instructions to create, update and remove a choice point, such instructions do not exist in DPAM. A choice point is created by the labeling process if needed and updated/removed by the scheduler.

4.1 Choice Point

A choice point is created when there are not enough processor elements to be divided among the alternatives in a nondeterministic derivation step. The scalar register `b` points to the top of the (scalar) choice point stack. A choice point contains the following fields:

- *process-id* allocation counter at last choice point
- ready queue at last choice point
- labeling queue at last choice point
- trail pointer (`tr`) at last choice point
- `hb` at last choice point
- restore queue
- head and tail of suspension queue

The process count, status word and processor element allocation information are saved on top of the heap rather than the choice point stack because they are vectors. They can be accessed at any time via `hb`. The consumed heap space can be reclaimed when the choice point is freed.

4.2 Trailing

The trailing scheme of WAM [1] is extended for use in a data-parallel context. The trail stack resides in the vector memory of the processor element array. Each entry in the trail is composed of a <*mask, address, old value*> triplet. The scalar `tr` points to the top of the trail stack. The `hb` (heap pointer at last choice point), which is scalar like `hp`, is used to determine whether a variable should be trailed. If trailing is not needed by any physical partition then no trail entry is created on the trail stack. Otherwise the mask field, which is set to the set of physical partitions which require trailing, is recorded together with the address and old value fields on the trail.

4.2.1 Trailing of Suspension List Updates

Several processes may suspend on an unbound variable in between two choice points. As a result, the unbound variable may be updated several times, which may lead to *multiple trailing*. Multiple trailing refers to the problem that a memory location is trailed more than once in between two consecutive choice points. This is inefficient because only the first trail entry is sufficient for restoring the heap to its original state when backtracking occurs.

operation	with time stamp	without time stamp
variable creation	read current time stamp create time stamp field create empty suspension list	 create empty suspension list
update list	read suspension list resume read time stamp check time stamp trail time stamp update time stamp copy node to top of heap trail node update node	read suspension list resume check address copy node to top of heap trail node update node
dereferencing	unaffected	unaffected
unwind	copy time stamp and node	copy node only

Table 1: Comparison of suspension list update with and without time stamp

We observe that when a new node is inserted at the head of a suspension list, the first suspension list node is copied to the top of the heap. Then it is overwritten by the new suspension list node (see Figure 5 again). The link field of the new node provides a clue to the age of the variable. At the next time the variable is updated, trailing is required only if it points to an address below `hb`. In other words, when a variable is bound or a new suspension list node is added, it is the link field of the first suspension list node rather than the address of the suspension list node itself which is compared with `hb`. The empty node of a freshly created unbound variable has a zero process-id as head and `hp` at the time of creation as tail. Table 1 compares the operations required by a system using time stamps and one which does not.

It is evident that efficiency can be improved no matter trailing is needed or not. In addition, we save the memory used for storing time stamps. The advantage of the time stamp scheme is that it is suitable for shared-memory implementation where each processor maintains its own heap, but this is not needed in DPAM.

4.2.2 Trailing of Domain Variables

In DPAM, a domain variable is represented by a record with these fields: a suspension list, a touch list (a list of suspended processes waiting for the domain to be changed), minimum/maximum values, a bit vector representing the domain and a pointer to the bit vector.

We do not trail the whole bit vector as in [5], but create a new bit vector on the top of the heap, modify the bit vector pointer field of the domain variable and trail the modification of the pointer instead. Like the trailing of a suspension list update, the bit vector pointer field itself is compared with `hb` to check if trailing is necessary.

In the time-stamp approach, the time-stamp itself must be trailed as well, and the bit vector is copied twice (when it is trailed and when the trail is unwound). In our approach, however, the bit vector is copied only once. We save trail space, but at the expense of heap space. In both cases the memory can be reclaimed upon backtracking.

There are no pointers in the minimum and maximum fields of a *d*-variable which can be used to infer the age. Therefore, unlike suspension list or bit vector updates, time stamps are associated with the minimum and maximum fields in order to avoid multiple trailing. We use the value of `hp` at the last time the field is updated as the time stamp.

4.2.3 Correctness Issues

Consider the following scenario.

1. An invalid value is removed from the domain of a domain variable. As a result, the processes in the touch list of the variable are resumed. After the resumptions the touch list field of the variable is cleared, and replaced by an empty suspension list node with a zero *process-id* and the current `hp` as the link field, as an indication of the time of the latest update.

2. No heap space is consumed afterwards, and a choice point is created. Note that `hp` = `hb`.

3. A process suspends on the touch list of the domain variable. Since the link field of the

first node is equal to `hb`, the update is not trailed, leading to incorrect result.

Therefore, at least one of the following conditions must be met to ensure correct operation of our trailing mechanism.

1. The update is ultimate (*e.g.* binding a domain variable to a constant), or

2. The update consumes heap space (*e.g.* inserting a node at the head of a suspension list (see Figure 5 again) or creating a new bit vector and updating the pointer to bit vector field of a domain variable).

DPAM enforces condition 2 and ensures correct operation by storing processor element allocation information, the status word and the process count on the heap (see Section 4.1 again) when a choice point is created.

4.3 Recovering the Process Queues

The system maintains a *process-id allocation counter*. When a process is created its *process-id* is set to the *process-id allocation counter*, and then the counter is incremented. The *process-id* provides a process with a convenient age. Processes can be trailed like heap terms: at each choice point the *process-id allocation counter* is saved. If a ready process has an *id* greater than *process-id allocation counter at last choice point*, then it is created after the choice point and can be discarded and returned to the free lists after being executed. Otherwise it must be retained.

Apart from the linkage pointers no fields in the process structures are modified in DPAM. Therefore to recover the ready queue and the labeling queue upon backtracking we just record the heads of the queues in the choice point. If a process suspended before the last choice point is resumed and removed from a hash queue, it should be restored to the hash queue upon backtracking. The *restore queue* is a singly linked list in each choice point consisting of these processes. When a process which suspended before the last choice point is resumed, it is added to both the resumption queue as usual and the restore queue as well. The restore queue uses the secondary linkage pointer. Hashing on the *process-id*, each process in the restore queue can be returned to the appropriate hash queue.

If a process suspends after the last choice point and has not been resumed, the process should be removed from the hash queue and freed upon backtracking. In each choice point there is a *suspension queue* which is a doubly linked circular list of all processes suspended after the last choice point. When a process suspended after the last choice point is resumed it is removed from the suspension queue.

5 Results

Due to lack of space, we present only the execution time of our implementation in Table 2. A detailed performance evaluation can be found in [11]. *DL* is the number of deadlocks, *P* the number of partitions (measured at the end of execution), and *BT* the number of backtrackings. Results are given in seconds, excluding any input/output time. Performance of CHIP version 3.2 on DECstation 3100 is included for comparison. When interpreting the results, notice that an individual processor on DECmpp (a 4-bit processor running at 12.5MHz) is very slow compared to DECstation 3100. We do not give speedup figures for `magic`. Here a direct comparison is unfair because entailment (*i.e.* ask) constraints (not available in CHIP) are used in the Firebird program.

6 Conclusion

We have presented a concurrent process scheduler and a parallel backtracking scheme for the data-parallel execution of the finite domain concurrent constraint programming language Firebird. An implementation based on DPAM has been built on a DECmpp 12000 Sx-100 computer with 8,192 processor elements and we have reported its performance results.

Acknowledgement

Travel support for the second author to attend the symposium is provided by the Hong Kong Telecom Foundation.

References

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

[2] H. Arro, J. Barklund, and J. Bevemyr. Parallel bounded quantifiers—preliminary results. Presented at JICSLP '92 Workshop on Distributed and Parallel Implementation of Logic Programming Systems, Washington D. C., 1992.

	Firebird				CHIP	
benchmark	t_1	DL	P	BT	t_2	t_2/t_1
`send`	.019	2	4	0	.007	.37
`eq10`	.299	4	226	0	.660	2.21
`eq20`	.429	3	138	0	.848	.37
`queen(8)`	.075	7	564	0	.512	6.83
`queen(9)`	.105	9	2365	2	2.133	20.31
`queen(10)`	.827	27	2530	15	8.547	10.33
`queen(11)`	8.922	157	2797	137	41.501	4.65
`queen(12)`	57.264	952	4178	1027	211.948	3.7
`magic(3)`	.090	2	7	0	.058	-
`magic(6)`	.463	2	22	0	2.422	-
`magic(9)`	1.315	2	45	0	140.886	-
`magic(12)`	2.729	2	78	0	-	-

Table 2: Benchmark results

[3] J. Barklund and H. Millroth. Providing iteration and concurrency in logic programs through bounded quantifications. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 817–824, ICOT, Japan, 1992.

[4] J. Crammond. The abstract machine and implementation of Parallel Parlog. Technical report, Department of Computing, Imperial College, London, July 1990.

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

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

[7] M. Nilsson and H. Tanaka. A flat GHC implementation for supercomputers. In *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, pages 1337–1350, Seatle, 1988. ALP, IEEE, The MIT Press.

[8] M. Nilsson and H. Tanaka. Massively parallel implementation of flat GHC on the Connection Machine. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 1031–1040, Japan, 1988. ICOT.

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

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

[11] B. M. Tong. Data-parallel concurrent constraint programming. M. Phil. thesis, Department of Computer Science, The Chinese University of Hong Kong, Hong Kong, 1994.

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

[13] 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.

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

[15] 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.

[16] D. H. D. Warren. The extended Andorra model with implicit control. Presented at ICLP '90 Workshop on Parallel Logic Programming, Jerusalem, 1990.