International Journal of Parallel Programming, Vol. 17, No. 1, 1988

Performance of an OR-Parallel Logic Programming System[1]

Peter A. Tinker[2]

Received December 1987; revised August 1988

The research focus in parallel logic programming is shifting rapidly from theoretical considerations and simulation on uniprocessors to implementation on true multiprocessors. This report presents performance figures from such a system, *Boplog*, for OR-parallel Horn clause logic programs on the BBN Butterfly Parallel Processor. Boplog is designed expressly for a large scale shared memory multiprocessor with an Omega interconnect. The target machine and underlying execution model are described briefly. The primary focus of the paper is on detailed statistics taken from the execution of benchmark programs to assess the performance of the model and clarify the impact of design and architecture decisions. They show that while speedup of this implementation on highly OR-parallel problems is very good, overall performance is poor. Despite its speed drawback, many aspcts of the implementation and its performance can prove useful in designing future systems for similar machines. A binding model that prohibits constant time access to bindings, and the inability of the machine to support an ambitious use of machine memory appear to be the most damaging factors.

KEY WORDS: Logic Programming; Parallel programming; Butterfly.

[1] This work was carried out at the University of Utah, Salt Lake City, Utah. It was supported by a University of Utah Graduate Research Fellowship, the National Science Foundation under Grant DCR-856000, and by an unrestricted gift from L. M. Ericsson Telefon AB, Stockholm, Sweden. Production of the document was supported by the Rockwell International Science Center.

[2] Current address: Rockwell International Science Center, 1049 Camino Dos Rios, P.O. Box 1085, Thousand Oaks, California, 91360 U.S.A.

0885-7458/88/0200-0059$06.00/0

1. INTRODUCTION

This article reports experience implementating pure OR-parallel Horn clause logic programming on the BBN Butterfly™ Parallel Processor. The Butterfly is a large-scale shared-memory homogeneous multiprocessor that can comprise several hundred processing elements, and provides communication facilities that scale with the number of processors. It can contain a large amount of physical memory, shared but distributed throughout the system. It is a general-purpose machine, intended to perform well on a wide variety of computational problems with varying degrees of interaction between the processors. The Butterfly's architecture demands that any substantial application be tailored to it. Simply "parallelizing" an application designed for a different type of machine may result in a poor match with the underlying architecture.

Accurate performance predictions for applications on the Butterfly are difficult if not impossible because of the many interacting parameters. An accurate assessment of performance can only be obtained through direct measurement of an actual implementation. The implementation described here is not designed to wring the last bit of speed from the machine (in fact, it is rather slow), but to determine which aspects of this specific architecture and the programming model are compatible and what can be done about incompatibilities. Detailed statistics on critical performance parameters have been taken by direct measurement of the implementation on the Butterfly.

The Butterfly is the subject of Section 2. Section 3 presents the design for *Boplog*, the OR-parallel logic programming system. Section 4 is devoted to assessing Boplog's performance and presents statistics that both judge the effectiveness of the design and clarify the importance of certain difficult problems. Section 5 concludes and summarizes the report.

2. THE BBN BUTTERFLY PARALLEL PROCESSOR

This section describes the Butterfly used for the implementation on which this report is based. It has been superseded in large measure by the Butterfly 1000 product family, headed by the GP1000 announced in October 1987. Some of the architectural details described here do not apply to the GP1000. Later sections examine, qualitatively, the impact of the architecture changes of the new machine.

2.1. Hardware

The hardware for both the mchine described here and the GP1000 is composed of two subsystems. The *processor nodes* are the processing

elements that form the computational engine of the machine, and the *Butterfly switch* forms the communication system that conects them.

A Butterfly comprises one to 256 processor nodes, and can grow in single-node increments. A typical Butterfly node consists of an MC68020 microprocessor, one megabyte of memory, and a *processor node controller* (PNC) that incorporates the switch interface. Nodes may also be configured with four megabytes, yielding a total potential physical memory of one gigabyte. Figure 1 shows the components of a Buterfly node.

The Butterfly used in this report has 18 nodes, of which a maximum of 16 are generally used. Each node uses an MC68020 microprocessor with an MC68881 floating point coprocessor, and some nodes have four megabytes of memory. The 68020 has the approximate power (at 16 MHz.) of a DEC VAX™ 11/780: about one MIPS when coupled with memory of an appropriate speed. The 68020 has an on-chip instruction cache of 256 bytes, and instructions are pipelined.

Figure 1 illustrates the central position occupied by the PNC. All accesses to memory, including memory local to the processor accessing it, are routed through the PNC on each node. The PNC performs all memory operations, using the switch if the reference is to remote memory. In addition to providing basic memory functions, the PNC microcode also implements a variety of atomic functions. These functions enhance the utility of the Butterfly for parallel operations such as queuing, semaphores and locks, and basic message-passing services.

Butterfly nodes are connected through the Butterfly switch, which is a nonblocking Omega network. The number of switching elements used grows as $N \log N$ where N is the number of nodes, while the bandwidth of the network grows approximately linearly with N. The switch uses bit routing, i.e., the destination address uniquely determines the path through the switch from the source to the destination. Data transfers are bit-serial. The raw speed of the network is 32 megabits per second per path. Our

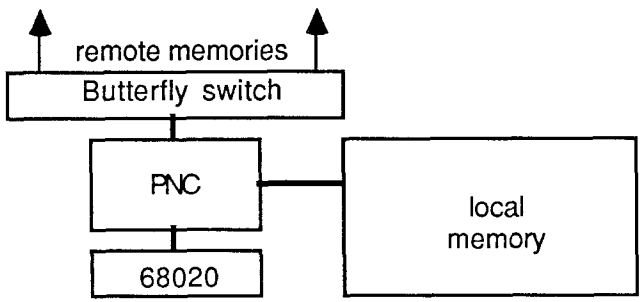

Fig. 1. A Butterfly Processor Node.

benchmarks indicate that 28 megabits per second is realizable under ideal conditions, with the remaining switch capacity dissipated through operations at the source and destination nodes.

Benchmarks indicate that a local 16-bit fetch to a 68020 register takes about 1.35 microseconds and a remote fetch takes 6.3 microseconds. [These figures are the result of timing 4000 consecutive in-line 16-bit fetch instructions to a machine register. As with all subsequent benchmark figures presented here, these instructions were executed with interrupts disabled, and (local) instruction-fetch time is included in the timing results.] The ratio of times for remote and local accesses is not very large: another way of viewing this situation is that local accesses are rather slow, but remote accesses are not very much slower. All 16-bit operations are performed synchronously.

2.2. Software

An executable piece of code on the Butterfly is called a *process*. Each process is composed of a substantial amount of code, and the cost for process creation is very high compared with most other fundamental Butterfly operations. In general, a Butterfly process is a complete program, capable of being executed on just a single processor node. Processes cannot be migrated from one node to another once they begin running.

Processes are run under the supervision of the Chrysalis™ operating system. A copy of the operating system kernel is resident on each node. Many processes may run on each node, with a multitasking scheduler to multiplex the execution of all processes on a node. It uses a prioritized timeslice algorithm to implement a round-robin schedule of context switches among the competing processes.

2.3. The Butterfly Memory System

An understanding of the Butterfly's memory management system is critical to the design of any application that hopes to make effective use of the machine resources. The first Butterfly machines used the MC68000 processor, which supports only 24-bit addresses. This aspect of the original design persists in the Butterfly described here: only 24-bit addresses can be used, even though the 68020 supports full 32-bit addresses. To access the potential gigabyte of physical memory on a fully configured Butterfly, 24-bit virtual addresses are mapped to 32-bit physical addresses. More detail on Butterfly memory management can be found in Refs. 1–4.

Using physical addresses, any byte of memory on the machine can be identified. The physical address encodes the processor node number and

local offset of the indicated memory. The 8-bit node number and 22-bit offset combine to give a total address width of 30 bits, or a maximum of four megabytes on each of 256 nodes. Physical addresses cannot be used directly on the Butterfly because the PNC accepts only virtual addresses, which it maps into physical addresses. The 24-bit virtual address occupies a full 32-bit word: the most significant 8 bits are unused, and the least significant 16 bits specify an offset of the memory from a base address. The base address is indicated indirectly by the second 8 bits, the *SAR number* field. This field specifies a *Segment Attribute Register* (SAR) to use in performing the virtual-to-physical address mapping. A SAR is a writable mapping register in the PNC.

The memory associated with a SAR can be accessed by calling an operating system procedure, *Map_Obj*, that obtains the SAR value for the memory segment, loads it into a free SAR in the PNC, and returns a virtual address corresponding to the first byte of the segment. The virtual address of any byte in the memory segment can be computed by additions to this base address. Any reference to such a virtual address is passed to the PNC by the 68020, where it is converted to its corresponding physical address and the memory access is performed.

There are 512 SARs in each PNC, of which up to 256 (in powers of two) may be used by a process, using buddy system allocation. The total addressing capacity of a process is determined by the 256 SARs, each of which may indicate the start of a 64-kilobyte object. Thus, at any time a process may only address

$$256 \text{ SARs} \times 64 \text{ kilobytes per SAR} = 16 \text{ megabytes}$$

On a fully configured Butterfly with 4 megabytes on each of 256 nodes, this accounts for only 1/64 of the available memory. Furthermore, the limit of 512 SARs per node makes it impossible to have several processes with large address spaces on each node.

If more than 16 megabytes are to be addressed, the values stored in the SARs must be changed to remap the memory system. This can be accomplished using *Map_Obj*, described previously, and *Unmap_Obj*, which frees the SAR associated with it by the corresponding *Map_Obj* call. As shown in Table I, these procedures are rather show. [Results in Table I

Table I. Memory Mapping Using Map_Obj and Unmap_Obj

Local Object	827 μsec.
Remote Object	1028 μsec

are for 500 consecutive in-line *Map_Obj* and *Unmap_Obj* in alternation, using an explicit SAR number, with the Object ID and virtual address in machine registers, and with interrupts inhibited. The time for instruction fetches from local memory is included.] Using them to remap memory dynamically is clearly impractical.

3. A DESIGN FOR OR-PARALLEL LOGIC PROGRAMMING ON THE BUTTERFLY

This section summarizes the design of an OR-parallel logic programming implementation called *Boplog* (*B*utterfly *O*R-*P*arallel *L*ogic). More detailed descriptions can be found in Refs. 5 and 6. Boplog is the first logic programming system designed for and implemented on a large-scale shared-memory multiprocessor. (The system described in Ref. 7 is designed for medium scale multiprocessors with several tens of processors, rather than the several hundred accommodated by the Butterfly.) Its design emphasizes shared data structures, effective use of large memories, locality, scalability, fast task migration and largely sequential execution. It is targeted for an unenhanced commercially available multiprocessor. Boplog is expressly designed to accommodate the Butterfly's drawbacks and capitalize on its strengths. It is targeted for Butterfly systems with many processor nodes, and is designed to allow access to a correspondingly large physical memory. The ability to use the potentially huge physical memory of the Butterfly is important for investigating the performance of the implementation on large problems on large machines.

To run Boplog, programs written in Prolog are compiled into Extended Warren Abstract Machine (EWAM[6]) instructions. In general, the instructions have the same meaning as the corresponding WAM instructions.[8,9] The exceptions corcern the allocation and reclamation of shared data areas, dereferencing of variables, and support for low-level memory management. EWAM code supports only 'pure' OR-parallel clauses: there is no 'sequential OR' construct, no *cut*, *assert*, or *retract*, and a limited variety of evaluable predicates. The EWAM code is then compiled into C code,[10] then into 68020 machine instructions. The resulting code is linked with the Boplog runtine system to form a complete compiled system.

The EWAM instructions introduce parallelism at the choice point level: alternate choices for a clause may be tried on different processors. Each alternative clause of a choice point represents a task that may be done in parallel with other tasks. Parallelism is exploited by having idle processors migrate choices away from choice points on busy processors. Migration involves moving enough information from the busy processor

that the idle processor can explore the search tree implied by the migrated choice.

Boplog's overall structure is *multisequential*:[11] identical copies of the Boplog code reside on each processor. Execution of the code is essentially sequential except where parallelism is introduced at choice points. Boplog program execution is carried out by one or more *Boplog processes*, each of which is capable of completing the problem itself. Each Boplog process is a complete compiled program, which works in concert with other Boplog processes. All of the Boplog processes are identical; they differ only in where they begin executing. Each Boplog process executes the finite state machine indicated by the transition diagram of Fig. 2. All Boplog processes begin in the *start2* state except for one, the *originating process*, which begins in the *start1* state. The originating process is the one designated to begin the execution. It executes EWAM instructions, possibly generating choice points that cause some of the other processes to enter the *active: migrating* state. In Fig. 2, a process in the *active: migrating* state moves work from an active process. In the *active: executing* state, it executes EWAM instructions; in the *active: failing* state it backtracks (perhaps repeatedly). When a process is in the *idle: searching* state, it looks for tasks to migrate.

When the originating process detects that all choices emanating from the program's first choice point have been completed, it sets a termination flag that is read periodically by each Boplog process. In the *idle: checking* state, a process checks this flag to see if it may terminate. If the flag is set, the process halts. If the flag has not been set, the process attempts to recover its stack and heap space. In doing so, it can detect when all work emanating from each choice point has completed, and can inform other processes that work has completed on their choice points.

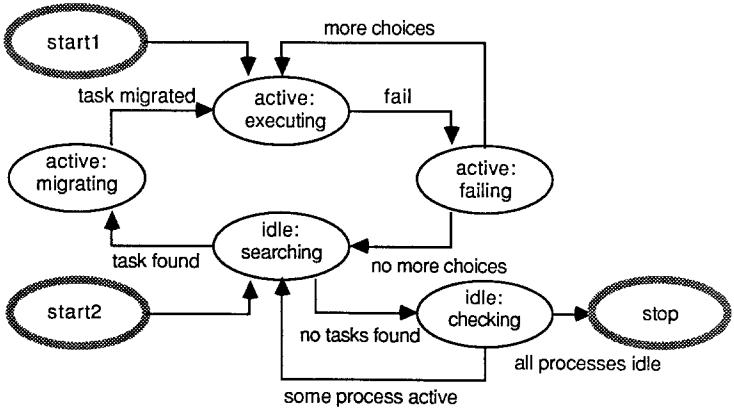

Fig. 2. Boplog Process Finite State Machine.

3.1. Principal Process Data Structures

Logic programming systems based on the WAM are said to use a *three-stack model*[12]: the *stack*, the *heap*, and the *trail* (used to record retractable variable bindings). Boplog uses a stack and heap just as in the WAM, but does not use a trail stack in the WAM sense. The stack and heap are trees, with the trunk shared by all processes. As tasks are migrated, each process forms a new top to the shared heap and stack. Boplog uses two important additional data areas. The *ancestor stack* is the primary data structure used in determining the correct binding for a variable (see Section 3.2). A small *information block* contains all of the data for a process that are necessary for interprocess communication and synchronization.

3.2. Boplog's Binding Environment

The way multiple bindings for variables are stored and accessed is critical to the efficient execution of any OR-parallel implementation. Many different methods have been proposed, e.g., Refs. 11–19. Boplog uses a *time-stamped linked-list* method,[5,6] which uses shared rather than copied data structures. It appears to be competitive with other binding methods that stress shared data structures, but a careful comparative study of these approaches (like that described in Ref. 20) is needed to justify this claim. The Boplog binding method seems well suited to a Butterfly implementation, since it exhibits a high degree of locality and scalability and makes effective use of the low relative overhead for remote accesses. Memory for bindings is allocated only on demand and has little waste or redundancy. Overhead for task migration is limited to copying the ancestor stack, which grows slowly compated with the auxiliary structures of other shared binding methods.

In Boplog, all bindings for a variable are stored as nodes of a distributed linked list called a *binding list;* there is a separate binding list for each variable. Each new binding for the variable is paired with a *timestamp*, forming a *value cell* that is stored as a node on the binding list. A timestamp counter is maintained independently by each process, and is incremented each time a choice is taken from a choice point. The timestamp of a value cell is the value of the counter at the time the binding is made, and indicates when the binding was made relative to other bindings for the same variable.

When a variable is accessed, it must be *dereferenced* to find its current value. In Boplog, dereferencing consists of two phases. In the first phase, the binding list and the ancestor stack are used to *disambiguate* the bindings to determine which, if any, represents the correct value in the

current context. The ancestor stack summarizes the ancestry of the thread of execution currently being pursued by a process. Each item on the ancestor stack contains a process number and a timestamp *binding span* that denotes when that process was investigating a choice that eventually led to the current execution thread. The span's lower bound is the timestamp at which either a choice was migrated to that ancesor process, or a choice was taken from an existing choice point of that ancestor. The upper bound is the timestamp at which the next choice point was created on the ancestor. These bounds indicate the span of timestamps during which the process could possibly have supplied a value used in the current thread. If a value cell, created by a specific process, has a timestamp that falls within the bounds of a span associated with the process, then the binding in the value cell was created by that ancestor while investigating the current execution thread. A variable is disambiguated by comparing the timestamp of each value cell on the binding list with the binding spans on the ancestor stack. If a cell is found whose timestamp and location indicates that its value was supplied by an ancestor, that cell is used as the variable's value in the current context. The value obtained by disambiguation is than used in the second dereferencing phase to chain backward through variable bindings to find the ultimate value of the variable.

3.3. Backtracking and Task Migration

Backtracking in the WAM involves freeing obsolete parts of the environment and stack and resetting various machine registers to restore the state of the computation to that of an earlier time. In Boplog, backtracking and task migration are closely linked, since both deal with the manipulation of choice points and their contents. The relevant information for backtracking and migration is stored in choice point data structures on the stack. They are linked together so that backtracking may chain backwards as deeper backtracking occurs, and so that idle processes can determine the earliest unprocessed choice for migration. Each choice point contains a count of the number of choices, a count of the number of choices already begun, a count of the number of choices completed, a level number that indicates the depth of the search tree at that point, a pointer to a list of code entry points that represent the unification code, and a copy of the ancestor stack of the process at the time the choice point was created. When the number of choices remaining is zero, the choice point is exhausted and no further migrations can take place from it.

Each Boplog process maintain three choice point references to control backtracking, task migration, and stack reclamation.

- *lcp*: The *local choice point* reference identifies the choice point most recently created by this process.
- *rcp*: The *remote choice point* reference identifies the earliest (least recently created) choice point of this process that still has choices remaining to be tried.
- *scp*: The *shared choice point* reference identifies the most recent choice point of this process from which choices may have been migrated by other processes.

lcp is the Boplog analog of the WAM *B* register. It moves later in the stack when new choice points are created, and falls back again on failure. *rcp* is used by idle processes when they attempt to migrate choices from a busy process: idle processes take choices from *rcp* and increment the number of choices taken from the choice point. When the *rcp* is exhausted, *rcp* is moved forward to the next choice point of that process. *scp* indicates the lastest potentially shared stack area of this process. Any part of the stack or heap beyond the *scp* choice point's creation is guaranteed not to be shared by any other process. *scp* moves foward in the stack when *rcp* moves foward, and falls back when work on all choices in its choice point have completed. The part of a process's stack that lies beyond *scp* is treated exactly like the stack in the sequential WAM model, and is much more efficient; *scp* is also critical to the recovery of stack space.

When *lcp* and *rcp* reference the same choice point, and that choice point is exhausted of all choices, the process becomes idle. Idle processes (processes that have never done work or have completed execution of their current branch) are responsible for obtaining more work from busy processes. An idle process attempts to migrate a choice from a choice point that is near the root of the entire search tree so that task granularity can be maximized. The assumption made here, of course, is that branches spawned near the root of the entire tree are the longest. Determining which choice to migrate involves both determining a process as the parent and determining which of the parent's choices to migrate. An idle process examines the *rcp* of all of the other processes to find that process with the smallest level number in its *rcp* choice point. If the *rcp* choice point of the indicated process is exhausted of all choices, its *rcp* and *scp* are reset to reference its next choice point, and the search for new work repeats.

Because data structures are shared extensively, migration of choices from one Boplog process to another is straightforward and involves little transfer of data between nodes. To migrate a choice, an idle process increments a count of the number of choices taken from the choice point, copies the choice point into its own memory, and begins execution of the clause indicated by the choice point. The ancestor stack stored with the

choice point now becomes the ancestor stack of the descendant process. The speed with which work can be transferred is particularly important near the end of problem execution, when the length of sequential segments becomes small and processes compete more frequently for less work.[21]

3.4. Memory Management

Attributes of the Butterfly's memory-management system described in Section 2.3 conflict with the desire to use all of the Butterfly's potentially huge physical memory. Two problems are most daunting: the limited virtual address space and the high cost of system calls for dynamic memory map changes. As Table I showed, Chrysalis system calls to support dynamic memory map changes are unacceptably slow. With care, however, it is possible to make use of all of the Butterfly memory without undue concern for the number of SARs available or the time-consuming use of calls to *Map_Obj* and *Unmap_Obj* to change the memory map. Boplog uses a scheme called *SAR-smashing*,[22] with which all of the available memory can be accessed by any process, as few as one SAR can be used to access all of the physical memory, and changing the memory map can be accomplished some 50 times faster than by using *Map_Obj* and *Unmap_Obj*.

Benchmarks indicate that the memory map can be changed in about 18.9 microseconds using SAR-smashing. [The benchmarks involved 100 executions of a loop with 5000 consecutive in-line SAR smashes, with all relevant data machine registers and with interrupts inhibited. This time includes the time for instruction fetches (instructions are not in the 68020 cache) and for the calculation of the address of the location holding the SAR value.] It can be integrated easily with the standard Butterfly memory management system, so that access to often-used memory is as fast as the Butterfly will allow, while access to less-used memory encounters the SAR-smashing overhead. The price paid for these features is increased user responsibility for managing memory references, and a higher per-access cost for some accesses. These costs are assumed by the Boplog runtime system, not by the Prolog programmer.

With dynamic memory mapping, vitual addresses no longer have a one-to-one correspondence with physical memory. Since virtual addresses cannot be used to refer to unique memory locations, Boplog uses *references* that are independent of both process and processor node. These 32-bit references are capable of locating any 32-bit item in a one-gigabyte physical memory. Table II summarizes some benchmark results of using references. The first time appearing in Table II indicates the time taken to construct a virtual address from a given reference; the second time gives the time taken to change the memory map based on information in the reference.

Table II. Memory Operations Using References

Creation of a virtual address	6.4 μsec.
Memory map change	23.5 μsec.

3.5. Locks and Critical Sections

Because Boplog is designed to run as sequentially as possible, actual interprocess interaction occurs rather infrequently, although provision for such interaction is pervasive. Resource locking is frequent, but processes seldom wait for locks to clear. The points of process interaction occur during variable binding and dereferencing, searching for more work by idle processes, backtracking, and task migration.

Although dereferencing of variables is one of the most common of Boplog operations, there is little overhead from the enforcement of critical sections during dereferencing. Locking is required at three points. First, each value cell must be locked during disambiguation to ensure that its timestamp is not changed while determining if the binding was made during any ancestor time span. Second, if a new value cell needs to be created and linked into the binding list for the variable, some locking is needed to prevent conflict when the link fields of the value cells are updated. Thus, a value cell is also locked when the timestamp of a value cell is updated for a new binding. This locking operation is the complement of the locking of a value cell during disambiguation. Finally, if a value cell is local to this processor node, it may be unshared. To determine this case, the *scp* of the process is examined. To ensure that the *scp* is not updated by another process during this time, this process's information block must be locked.

4. PERFORMANCE AND ASSESSMENT

The intent of this section is to give detailed statistics for the performance of Boplog's binding method and migration strategy; show how some performance aspects change as the number of processes in increased, and provide data for comparison with those from other implementations as they become available. It does not show how the implementation performs on a wide variety of programs: the results are based on just a few programs and executions, which are of course not enough to give a complete picture of Boplog's performance. Where times are reported for activities of short duration, they are generally the averages of many (often several million) timings of separate instances. Hopefully, these results provide some insight into which aspects of the programming paradigm, execution model, architecture, and implementation deserve more attention in the future.

Statistics have been gathered for Boplog running several simple programs. The benchmark programs were selected because they are common in the literature and because, with one exception, they exhibit a high degree of OR-parallelism. The exception is a program that generates no such parallelism, and is used to evaluate sequential performance and the impact of idle processes on the performance of active ones. Statistics were collected on a per-process basis and then aggregated to smooth differences between individual process. These programs are trivial and too regular to validate the design decisions for Boplog, but they do offer some insight into the Boplog execution model, and the impact of various sequential and parallel overheads. Where times for various operations are given, the reported times are somewhat longer than the actual times, since there is some overhead for gathering the statistics. This overhead is generally about 20% of the total time reported.

4.1. Benchmark Programs

The first benchmark is the standard 'naive list reversal' program, for which Boplog code is given in Fig. 3. This program is deterministic in the WAM model; because of clause indexing, no choice points are created. The naive reverse program is used to evaluate the sequential efficiency of Boplog and to assess the impact of idle processes on execution times. The second benchmark program finds all permutations of a list of elements, and is of interest here because some OR-parallel benchmark results have already been reported elsewhere.[20] The program here differs from its usual form in that instead of reporting all of the permutations it filters out all but one. Code for the Boplog permutation program is given in Fig. 4. The third benchmark program is a naive 'generate-and-test' solution to the *n*-queens

```
?- list30(L), nreverse(L, X).

nreverse([X | L0], L) :-
   nreverse(L0, L1), concatenate(L1, [X], L).
nreverse([], []).

concatenate([X | L1], L2, [X | L3]) :-
    concatenate(L1, L2, L3).
concatenate([], L, L).

list30([1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
    11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
   21, 22, 23, 24, 25, 26, 27, 28, 29, 30]).
```

Fig. 3. Naive Reverse.

```
?- range(1, n, X), nreverse(X, Y), perm(X, Z), Z=Y.

perm([], []).
perm(X, [V | Z]) :- delete(V, X, Y), perm(Y, Z).

range(First, Last, [First | L]) :-
    less(First, Last), plus(First, 1, First1),
    range(First1, Last, L).
range(Last, Last, [Last]).
```

Fig. 4. List Permutation.

problem from Sterling and Shapiro's book,[23] rewritten to avoid the use of *not*. It is used to generate large search spaces, and is shown in Fig. 5.

4.2. Sequential Performance

The performance of Boplog running on one processor node (one Boplog process) gives a measure of its speed relative to other implementations. Table III shows execution times for various instances of the benchmark programs. Table III also shows the execution times for the same programs run on a DEC MicroVAX™-II workstation using CProlog1-4, a moderately optimized Prolog interpreter.[24] The MicroVAX has roughly the speed of a single 68020 running at 16 MHz. Although the equivalence of a Butterfly processor node and a MicroVAX is questionable, the dramatic difference in speeds clearly indicates that Boplog, running compiled EWAM code, is much slower than a reasonable sequential Prolog interpreter.

```
?- queen(n, Qs).

queen(N, Qs) :- range(1, N, Ns), perm(Ns, Qs), safe(Qs).

noattack(_, _, []).
noattack(X, N, [Y | Ys]) :-
    plus(Y, N, X1), nequal(X1, X),
    minus(Y, N, X2), nequal(X2, X),
    plus(N, 1, N1), noattack(X, N1, Ys).

safe([Q | Qs]) :- safe(Qs), notattack(Q, Qs).
safe([]).

notattack(X, Xs) :- noattack(X, 1, Xs).
```

Fig. 5. *n*-Queens.

Table III. Single-Process Benchmark Execution Times

Benchmark	Boplog	CProlog
naive reverse	1,299 msec	343 msec.
4-element list permutation	1,453 msec.	197 msec.
5-element list permutation	6,802 msec.	903 msec.
6-element list permutation	39,912 msec.	5,280 msec.
4-queens	2,879 msec.	593 msec.
5-queens	16,720 msec.	3,547 msec.
6-queens	109,239 msec.	23,043 msec.

Because the sequential performance of a Boplog process is poor, the combined performance of many Boplog processes cooperating on a single problem cannot be expcted to be very good. Speed-up (the performance enhancement due to parallel execution) may be impressive, but this result is misleading because of the poor single-processor performance. The overhead of memory map changes, the inability to compare virtual addresses, fixed overheads for disambiguating bindings, and the slow memory of the Butterfly all constrain the maximum speed.

With the single-processor speed of the implementation in mind, the statistics for sequential and parallel execution are still valid as a means of evaluating the environment model, task migration strategy, memory mapping overhead, scalability, and locality. The following sections present these statistics and comment on their interpretation. Except where otherwise noted, the statistics were gathered from the 8-*queens* program (*n-queens* with $n = 8$) running on 16 nodes. The size of the problem ensures that a large number of samples are combined in the statistics. For comparison with other times reported, the execution time of the 8-*queens* program on 16 nodes is about 494 seconds.

4.3. Memory Management

Memory management costs reflect overheads incurred in any Butterfly implementation that tries to use all available memory; these in turn impact the performance of model-specific aspects of the implementation. Boplog's concern for using all available memory results in considerable overhead for memory map changes, recomputation of virtual addresses, and the inability to compare data locations using virtual addresses. Time spent for these overheads may have a dramatic effect on the impact of communication patterns during an execution. SAR-smashing has by far the highest cost. Time spent in remapping the Butterfly's memory is time during which the process

Table IV. SAR-Smashing (8-Queens)

SAR smashes (total)	38,216,764
SAR smashes with a known SAR value	9,154,774
SAR smashes from a reference	29,062,990
Virtual address calculations	13,185,587

doing the mapping cannot put any load on the interconnect. If this overhead is too high it will mask communication overhead. Table IV presents SAR-smashing statistics for the 8-*queens* program on 16 nodes. Each SAR smash results in a memory map change. If the SAR value is known, the memory map change is faster than if it must be obtained from a reference; Boplog is careful to use known SAR values whenever possible. It also tries to avoid recomputing virtual addresses, which can be preserved across SAR smashes.

Using SAR-smashing, Boplog asserts direct control over the memory mapping process, using kernel mode privileges to write directly into the hardware mapping registers on the PNC. By doing so, memory map changes are made in the minimum possible time. Each change involves only a 32-bit write to a fixed location in the PNC. This takes just a few microseconds, and the remaining time for each SAR-smashing operation is spent obtaining the necessary information for determining the correct SAR value. These operations involve shifting and logical operations on the reference. If this information could be found more easily, each SAR-smashing operation would be correspondingly faster. Several alternatives are discussed in Ref. 6.

4.4. The Environment Model

The environment model encompasses both variable dereferencing (including disambiguation) and task migration. Its performance appears to be the most critical element in determining overall performance.

4.4.1. Dereferencing

The performance of Boplog's environment method is determined primarily by the number of binding list links traversed to find a variable binding, the number of ancestor spans examined, and the relative numbers of remote and local value cells examined. Table V presents statistics for dereferencing in the 8-*queens* program running on 16 nodes.

Table V indicates that some ten million dereferences were performed during the course of program execution. Of these, some six million were

Table V. Environment Performance (8-Queens)

dereferences	10,320,354
nontrivial dereferences	5,959,490
maximum number of ancestors on stack	51

operation	total	per nontrivial dereference
time	3,335 sec.	559 μsec.
nodes traversed	9,615,865	1.61
links traversed	6,125,328	1.02
total value cells examined	10,163,262	1.70
local value cells examined	9,763,202	1.63
remote value cells examined	400,060	.06
dereference SAR-smashes	12,626,841	2.11
from a known SAR value	1,638,793	.28
from a reference	10,988,048	1.84
virtual address calculations	10,849,688	1.82
ancestors examined	14,313,507	2.40
ancestor spans examined	8,503,793	1.42

nontrivial, requiring variable disambiguation and dereferencing. On average, each nontrivial dereference required the examination of cells on fewer than two processor nodes, and each variable chained through just one value cell to find its ultimate value. (A link in the chain was counted when following a reference from one value cell to another reference. In most cases, the second reference was not to a variable but to a ground term.)

On average, each dereference resulted in the examination of between one and two value cells. Of these cells, more than 96% were located in memory on the processor node of the process performing the dereference. An average nontrivial dereference required that between two and three items on the ancestor stack be examined. This figure is low in light of the fact that there were often many items on the stack (a maximum of 51). Between one and two time spans were examined for each nontrivial dereference.

These figures may be misleading because of the small size and regularity of the problem, but they are encouraging since they indicate that very little search is required to find the correct value cell. The time per dereference, however, is rather discouraging. The dereference time tends to dominate the execution, and accounts for nearly half of the total execution

time. A large number of dereferences occurred, and each nontrivial dereference took nearly 600 μsecs. Clearly, this time is too long—one could hope that the average time would be at least an order of magnitude less.

Where is the time spent? Table V seems to show that the model performs well, but the implementation fails to capitalize on its strengths. The implementation adds hidden costs to dereferencing, which are not reflected in the numbers from the table. An average of between two and three memory map changes are made per dereference, which could account for some 75–100 μsecs., according to benchmarks presented earlier. Also, each value cell examined is locked during dereferencing to ensure that the time stamp does not change during dereferencing. The impact of the locking is difficult to determine; Table VI summarizes some information, showing that little lock contention occurs, but that each lock attempt takes a relatively long time.

All Boplog locks are spin-locks and are implemented by an atomic inclusive-OR operation. A locking attempt consists of OR'ing the current value of the lock with a fixed number and determining if the state of the lock changed. A locking failure occurs if the state did not change, indicating that the value had already been set by another process. Many locking attempts were made, but few failed. However, because the locking operation requires interaction with the PNC, it is rather slow. [The time reported as 'time per attempt' is the result of performing the atomic inclusive-OR operation on a memory location on a remote node. Interrupts were disabled, and 100 loops of 100 in-line operations were timed.] A more efficient hardware locking mechanism might reduce the time spent.

4.4.2. Task Migration

Table VII presents statistics for task migration. Overall, only 229 migrations were performed, just a very small fraction of the total number of choices generated. Each migration took less than a millisecond, and each copied an average of about 240 bytes. The size of data copied includes the ancestor stack, which averaged about 17 ancestors per migration. These

Table VI. Locking Statistics (8-Queens)

total locking attempts	17,005,346
failures	41,650
value cell locks	12,746,084
info block locks	3,189,989
choice point locks	1,065,945
time per attempt	20.1 μsec.

Table VII. Migration Statistics (8-Queens)

total choices generated	1,065,278
migrations	229
total time	209 msec.
time per migration	913 μsec.
ancestors per migration	17
migration size	240 bytes
migration attempts	3,099
total time	5,238 msec.
time per attempt	1,690 μsec.

figures support a claim of fast task migration for Boplog, relative to other costs.

A problem with the current task migration strategy is made apparent by noting that several thousand migration attempts were performed. A migration attempt involves an idle process checking to see if any choices can be migrated away from their parent, and requires time-consuming locking and SAR-smashing as each remote process's information block is examined for pending work. More than 90% of these checks fail, but each attempt incurs a time penalty of more than one millisecond. This problem is discussed further in Section 4.6.

The number of ancestors copied on migration was small, indicating that a series of 16-bit transfers might perform even better than the block transfers used in this implementation. A small number of ancestors in the choice point is also an indication that tasks are migrated from points early in the execution, resulting in more sequential execution in each process. For larger programs, it is likely that migrations near the end of task execution would involve many more ancestors. For these tasks, block copies would be preferred. A run-time examination of the number of ancestors could determine which would be better for each migration.

4.4.3. Task Selection Strategy

The average time spent searching a particular branch of the search tree following task migration is long compared to other activities. Periods of sequential execution are therefore long, which promotes the multisequential execution model. The frequency of task migration increases dramatically near the end of program execution. Figure 6 shows the times at which task migrations occurred. The circles on the line indicate the average time (across all processes) at which the migration occurred that began the ith task for each process. The bars around each circle indicate the minimum

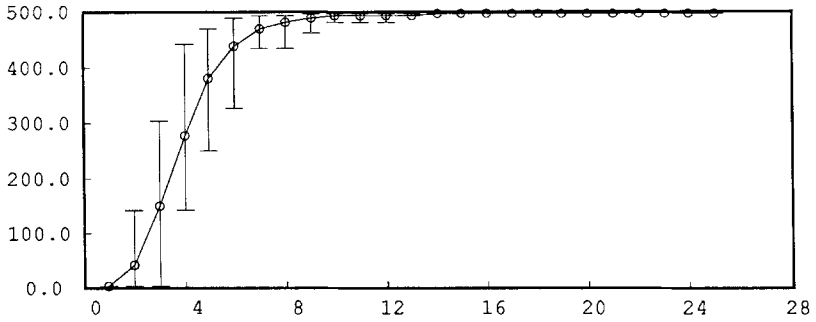

Fig. 6. Time of Migration (seconds) vs. Task Number (8-Queens).

and maximum migration times across all processes. Figure 6 shows that nearly all migrations occurred very late in the execution—halfway through the total execution, most processes had started fewer than four tasks. Tasks that were begun early tended to last a long time relative to the total execution time. This observation is confirmed by Fig. 7, which shows the duration of the ith task executed by each process. Figure 7 shows that nearly all of the work in the program is done by the early tasks. It also shows that there is a dramatic difference in the time spent on any task. For most processes, tasks beyond their eighth task were very short.

Long periods of sequential execution are encouraged by migrating the available choice that has the lowest level number, where the level number is incremented each time a process makes an EWAM procedure call. This simple heuristic appears to work well. As an additional benefit, this migration strategy results in migration of a choice from a choice point with fewer ancestors than choice points created later in the execution. Figure 8 summarizes this effect, showing the number of ancestors in the choice point

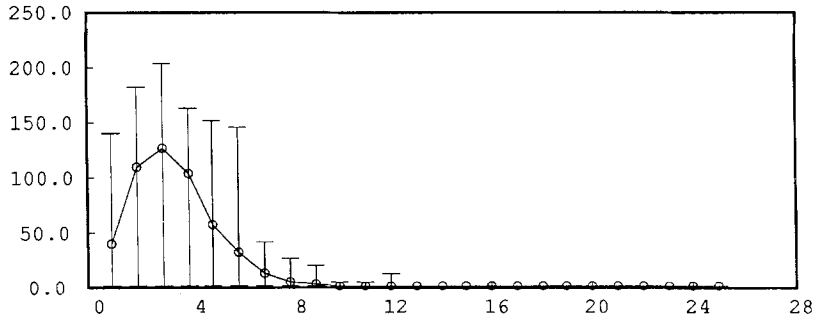

Fig. 7. Task Duration (seconds) vs. Task Number (8-Queens).

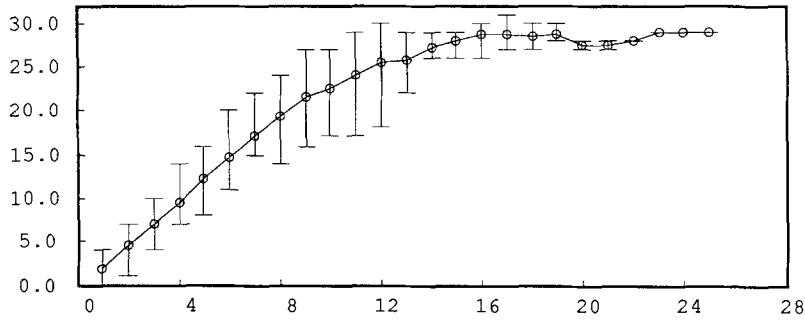

Fig. 8. Number of Ancestors vs. Task Number (8-Queens).

for the ith migration performed by each process. The number of ancestors migrated with the early, long-lived tasks is less than half the number migrated by short later tasks.

4.5. Locality

With a multisequential implementation on a machine with a penalty for remote accesses, locality is of great concern. When disambiguating variables may involve traversing a distributed binding list, locality becomes even more important. The statistics for variable dereferencing in Table V show that remote value cells account for only a small fraction of the total number of value cells examined during dereferencing. Although the value cells themselves are almost entirely local, it is possible that many references to nonvariable values are nonlocal. Such references do not require the examination of value cells to determined their value.

4.6. Processor Utilization

Since there is one Boplog process per Butterfly processor node, processor utilization is equivalent to process 'busy time.' In an ideal parallel implementation, each process would spend all of its cycles performing useful work: executing EWAM instructions, deteferencing variables, migrating choices, and so on. In practice this is never possible, since there are always sequential execution segments that force some processes to await the availability of work. For example, execution must begin with some single process, which runs sequentially until other processes can migrate work. Similarly, when program execution is nearly complete, no further work is generated, so some processes will be unable to perform useful functions. These 'ramp-up' and 'ramp-down' times are short for the OR-parallel programs for which Boplog has been used so far.

Table VIII. Idle Time Statistics (8-Queens)

migration attempts	3,099
termination attempts	2,662
attempts to recover shared memory	3,099

operation	total time	time per attempt	%
migration attempts	5,238 msec.	1,690 μsec.	79.7%
termination attempts	144 msec.	54 μsec.	2.2%
attempts to recover shared memory	1,193 msec.	385 μsec.	18.1%

When a process is idle, it executes no EWAM instructions. It spends its cycles searching for more work, testing for termination, and attempting to recover shared data areas. Table VIII shows the amount of time spent performing each of these tasks and the percentage of time they take collectively. Even though migration attempts are costly, the total idle time accounts for only about 1% of the total exacution time.

4.7. Scaling Considerations

How well the model scales as processor nodes are added can be assessed by running the same program on different numbers of nodes (different numbers of processes) and noting changes in performance statistics. So far, Boplog has only been run on as many as 16 nodes. Whether the performance continues to change in the same way when there are many nodes is unknown.

The following figures present changes in several performance statistics as the number of processes is increased. The data are for both the *7-queens* and *8-queens* programs. Memory limitations currently prevent *8-queens* being executed on fewer than five processor nodes. The first set of figures deals with variable disambiguation and dereferencing. Figure 9 shows that the ratio of local to remote value cells examined during dereferencing remains roughly constant and at a very high level. Locality drops slowly as processes are added. Under the assumption that value cells for any given variable are distributed uniformly among all processes, this result is expected, since the probability that a cell is on a specific processor node drops as nodes are added. Figure 10 shows that more ancestors are examined for each dereference as the number of processes is increased. This result is consistent with an increase in the number of ancestors copied with each

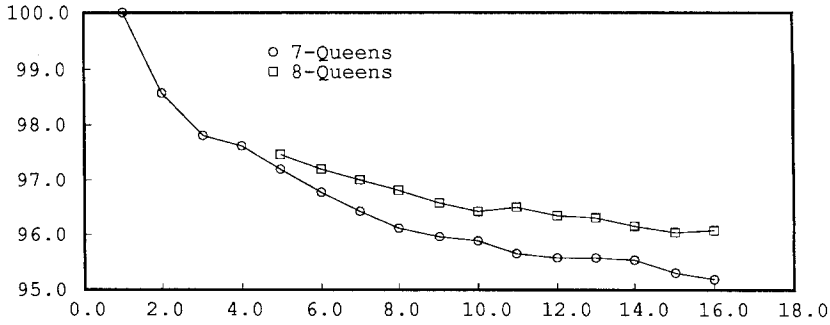

Fig. 9. Percentage Local Value Cells per Dereference vs. Processes (8-Queens).

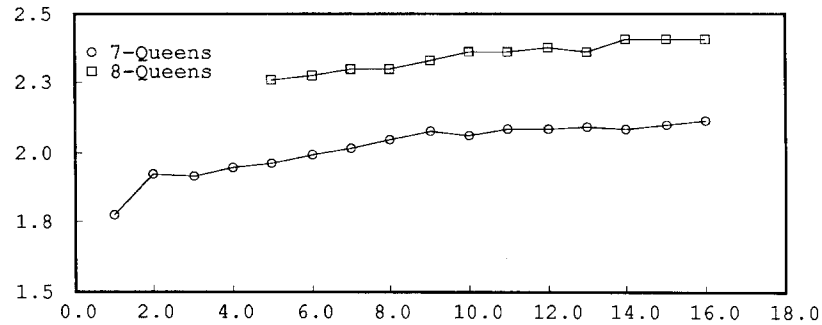

Fig. 10. Ancestors Examined per Dereference vs. Processes (8-Queens).

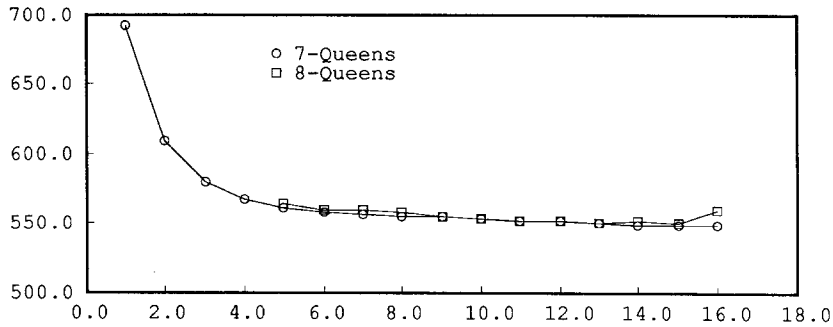

Fig. 11. Dereference Time (μseconds) vs. Processes (8-Queens).

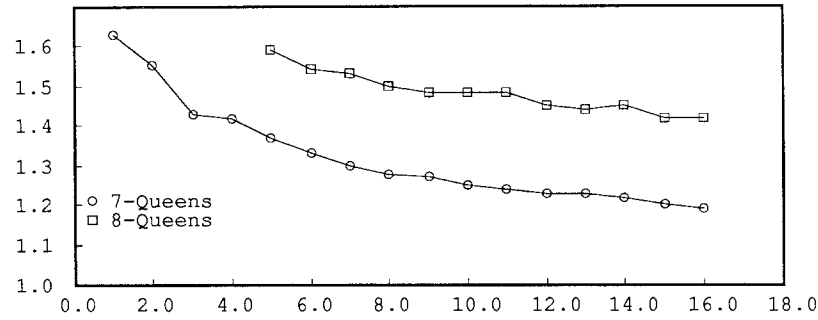

Fig. 12. Binding Spans Examined per Dereference vs. Processes (8-Queens).

migration (noted later). Given the results of Figs. 9 and 10, a degradation in dereference performance might be expected as processes are added. Figure 11, however, shows that the average time for a variable dereference decays slowly after the number of processes increases to about five. A possible explanation is found in Fig. 12. It shows that although the number of ancestors examined increases, the number of ancestor binding spans actually checked against value cell time stamps falls. This decrease occurs because the ancestors are distributed among more processes, so that each process contributes fewer ancestors. Fewer ancestors per process should result in fewer binding spans being examined.

The next set of figures shows how migration statistics change as more nodes are used. Figure 13 shows that the total number of migrations performed over all processes increases nearly linearly, indicating that the number of migrations per process remains roughly constant. The number of migration attempts shows a similar behavior (Fig. 14).

The average time to migrate a task increases slowly as more processes are used (ignoring some large changes when few are used). The increase is

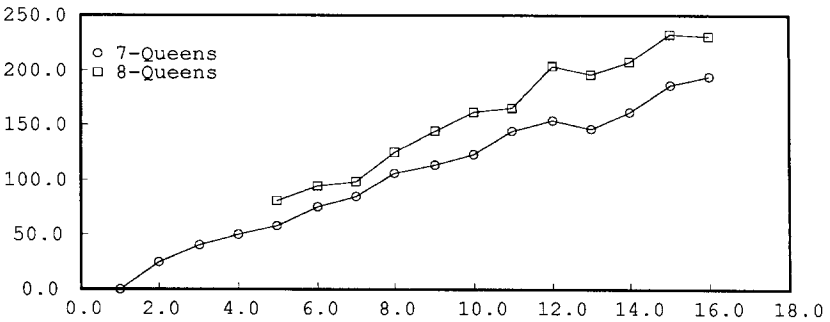

Fig. 13. Number of Migrations vs. Processes (8-Queens).

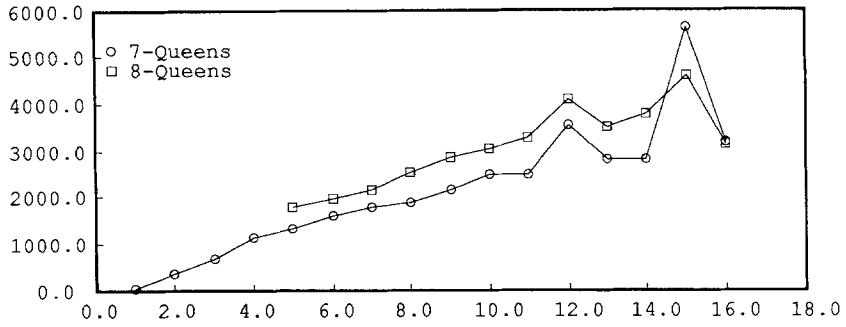

Fig. 14. Number of Migration Attempts vs. Processes (8-Queens).

due to increased numbers of ancestors in the choice points migrated. The increase in ancestors occurs because more processes are participating. A branch of the search tree is more likely to be extended by many processes if there are more processes available. The increases in time and size are rather slow, as shown in Figs. 15 and 16.

Dereferencing and task migration are central to Boplog's performance and are the only aspects of its execution that are intimately concerned with multiple processes. It is not surprising that good scalability of these activities promotes good overall scalability. Total execution times for the *7-Queens* and *8-Queens* programs are shown in Fig. 17. Both programs show a smooth decrease in total execution time as processes are added. Figure 18 shows the ratio of multiprocess execution time to the single-process execution time. These speed-up figures show that the speed increase is generally close to or superior to the "ideal." This phenomenon is not rare in parallel applications and usually reflects poor single-process performance. If the addition of processes makes each process slightly more efficient than in the single-process case, performance will increase faster

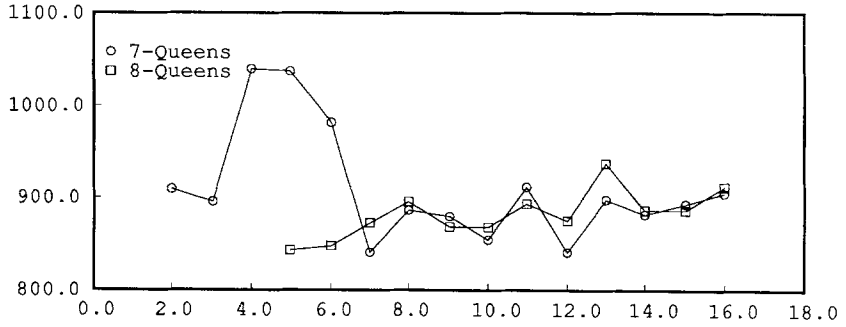

Fig. 15. Migration Time (μseconds) vs. Processes (8-Queens).

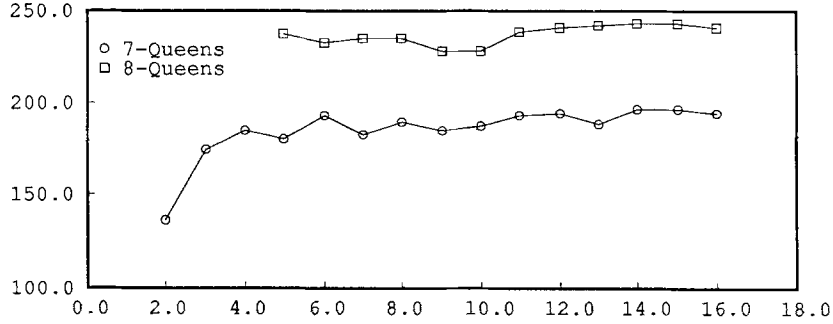

Fig. 16. Migration Data Size (bytes) vs. Processes (8-Queens).

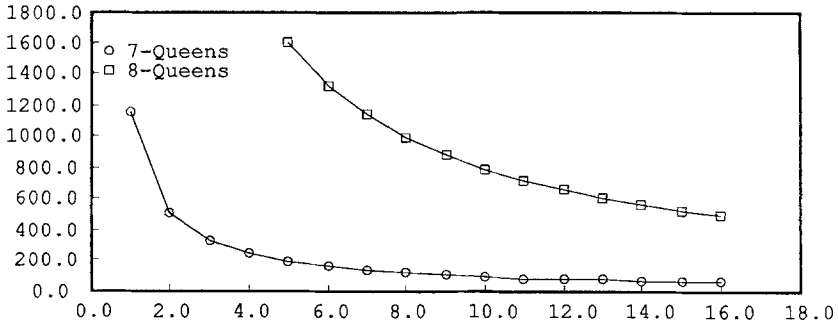

Fig. 17. Total Execution Time (seconds) vs. Processes (8-Queens).

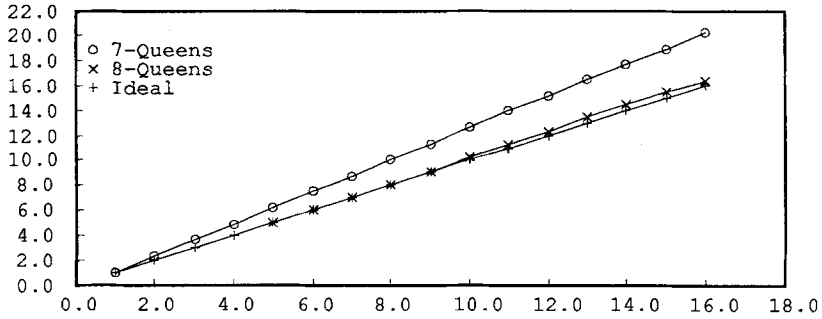

Fig. 18. Speedup vs. Processes (8-Queens).

than the number of processes. This can happen, for example, if there is some operation which needs to be performed less frequently when multiple processes are executing than when only one is executing. In the present instance, this operation might be context switching: context switches take a larger portion of processor time when a single process is executing. Note that in Fig. 18, the single-process execution time used for obtaining the speed-up for 8-*Queens* is assumed to be five times the five-process time.

4.8. Switch Contention

Many Butterfly systems are configured with multiple paths through the switch between each pair of nodes. A 16-node machine, for example, may be paired with enough switch hardware for 32 nodes, allowing two paths between each pair of nodes. In addition to increasing throughput during normal program execution, having alternate paths allows switch contention to be estimated qualitatively. If an execution slows dramatic improvement when alternate paths are enabled, that is evidence that there is much contention on the switch in the single-path execution. This qualitative result is useful when determining the effect of memory reference patterns on overall program efficiency.

The results reported in this paper did not make use of alternate Butterfly switch paths. Enabling these paths causes performance to improve by avout 2%, indicating either that there is little switch contention, or that memory references through the switch are very uniform. Any contention is probably the result of frequent access to the process information blocks. Value cell references could also cause contention, but this is unlikely because such references are scattered widely through memory. In either case, switch contention is not a problem with this implementation. It is likely that the lack of contention can be attributed to the slowness of the implementation: memory references across the switch are infrequent because so much local processing is performed. It may be the case that the amount of switch contention observed is an unavoidable cost of an implementation that relies so heavily on shared data structures. More study is needed to determine the source of contention, and whether some selective copying of data might reduce the total execution time.

4.9. The Impact of Idle Boplog Processes

The naive reverse program is used to evaluate the impact of idle processes on sequential execution. Since the naive reverse program generates no choice points in the WAM model, only the originating Boplog process does any useful work. The other processes spend all of their time

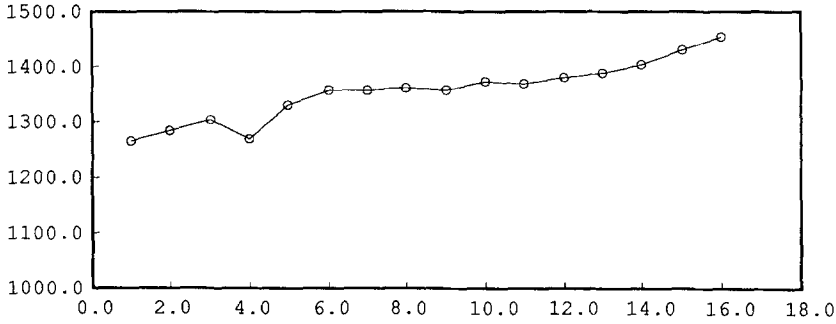

Fig. 19. Total Execution Time (seconds) vs. Processes (Naive Reverse).

searching for work and checking for termination. Figure 19 shows total execution time for increasing numbers of processes. Note that as the number of processes increases, so does the time taken to complete the program. For programs that do not generate enough choices to keep all of the processes busy, Boplog's method of finding work causes a degradation of performance. Speedup is negative, as shown in Fig. 20. Since the degradation is slow, however, the impact of idle processes on overall execution time is small.

4.10. Impact of Alternative Designs and Architectures

Because the current Boplog implementation incorporates design decisions made prior to building the system, no direct comparisons with alternative designs have been made. It is important to note, at least qualitatively, how different design decisions might impact its performance. Furthermore, Boplog was explicitly intended for use on the Butterfly.

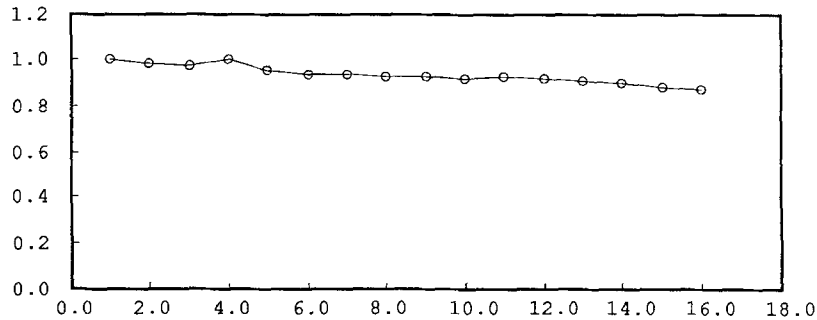

Fig. 20. Speedup vs. Processes (Naive Reverse).

Details of the architecture have a significant effect on Boplog's performance, and it is reasonable to conjecture how Boplog's performance might differ if certain architectural features were changed.

4.10.1. Better Implementation

Preceding sections have noted that the performance suffers from inefficient implementation. Some of the poor performance can be attributed to overhead to support large memory requirements, but the primary fault is probably a result of poor performance of individual procedures executed often in the body of the program. The results reported in Ref. 7 show that an OR-parallel implementation can achieve per-process performance levels near those of sequential Prologs, when a good sequential implementation is used as its foundation. A direct comparison of that implementation with Boplog is misleading, however, since it was designed for a small- to medium-scale multiprocessor using a bus interconnect that imposes no penalty for nonlocal references.

A better implementation would probably perform fewer deferences. The current implementation is conservative and probably dereferences variables unnecessarily on occasion. Since dereferences are expensive, decreasing their number should result in higher performance. It is also the case that the proportion of time spent dereferencing will increase. Dereferencing has been optimized more than other parts of the code, so making the remaining code faster will shorten the overall time while reducing dereference time only slightly.

A better implementation is likely to increase switch contention. If the code for one Boplog process is more efficient, the time between remote accesses will decrease while the number of remote accesses remains fairly stable. More frequent remote accesses means more contention for memory and more load on the switch.

As the implementation becomes more efficient, the statistics that measure its performance become more meaningful. While the proportion of time spent gathering statistics will rise somewhat, those statistics will measure the critical aspects of the model more accurately. In particular, the bandwidth requirements of the switch should become more important as the frequency of remote accesses increases, and the distribution of remote data to minimize hot-spots will be an important metric.

4.10.2. Different Binding Method

On qualitative grounds, supported by the previous statistical results, I believe that the linked-list method can be made roughly equivalent in per-

formance to the other "shared" binding methods, e.g., Refs. 13, 14 and 19, with the probable exception of the versions-vector method[16] and the SRI method.[19] The two methods are similar, and will probably outperform all other methods on small numbers of processors. The versions-vector method does not appear to scale well and lacks locality; both require auxiliary binding structures that may not be effectively used. At this time, only simulation results support their effectiveness, as no results of parallel implementations have yet been reported. Their main atraction is that each process has at most a single binding for each variable, and the time to access any binding is constant.

Implementing either of these methods in Boplog would involve creating some data structures statically, so that the benefits of the constant time variable access could be preserved. The overall effect would be a substantial improvement in memory mapping costs and dereferencing. Since these costs are the most damaging to Boplog's performance, the change would be well worth any concomitant poorer memory usage and restrictive memory allocation.

Other alternatives are also possible. If a trail is maintained and unwinding is acceptable, bindings could be stored in one large hash window on each node. The window would be copied when migrating a task. The original value cell for a variable would be located in some process's stack. If that value cell is unbound, it would indicate that the variable was unbound when the next choice point was created. When disambiguating the variable, a process would then search in its local hash window. If a binding is found there, that binding is used; otherwise, the variable is unbound and its new binding is entered into the local hash window. Although variable accesses would be reasonably fast and the method has good locality and scales well, it requires that the trail and unwinding be used, and that the hash windows be copied when migrating.

SAR-smashing might provide yet another binding method. If each process created its own binding for a variable and stored it at the same *virtual* address locally, then a process would only have to examine that location to find the binding. Searching through a list of ancestor processes to determine if they bound the variable could be done quickly by using SAR-smashing to change the physical location indicated by the variable's unique virtual address.

No concrete decisions about the relative merits of binding methods are possible without implementation. Each competing method should be used by the same runtime system and compared on the basis of the sort of statistics that have been gathered for Boplog. Only then can questions of efficiency, scalability, and locality be assessed adequately. Still, it seems reasonable to trade poorer memory usage for the obvious advantages of constant time variable accesses.

4.10.3. Faster Memory

Because all memory references generated by the 68020 are routed through the Butterfly's PNC, local memory references are rather slow. If local memory were faster, execution speed would probably be better by about the same degree, even if remote memory reference time remained unchanged. This conjecture is based on the ratio between local and remote value cells examined. (The new GP1000 Butterfly executes a local 16-bit fetch about four times faster than the Butterfly used here.) Since faster local memory would improve overall performance, the effect would be the same as that noted earlier for improved program efficiency—switch contention could be expected to be higher, and the statistics more valid.

4.10.4. Adequate Virtual Address Space

The SAR-based address-mapping strategy of the Butterfly has proven to be a serious stumbling block for Boplog. In retrospct, the decision to use the memory effectively through dynamic memory map changes may have been too ambitious. Performance certainly would have been substantially better had a static memory mapping scheme been employed, as in the Uniform System package provided by BBN for the Butterfly.[2]

In a standard uniprocessor WAM implementation, the data areas are laid out carefully so that address comparisons can be used to identify the area in which an item resides, or to determine the relative ordering of two items in the same area. An example of the first use is the ability to use address comparisons to identify a value cell on the stack that must be copied to the heap before applying the last-call optimization[25] (a so-called "unsafe variable"). The second use occurs, for example, when two unbound variables are unified. In this circumstance, the later variable is always bound to the earlier one, where "later" means "having the higher virtual address." If binding is done in this direction, and if the heap is located in lower memory than the stack, then no dangling references result from deallocations of the stack, and no heap item ever references a stack item.

Because SAR-smashing dynamically changes the relationships between virtual addresses and physical memory locations, address comparisons cannot be used in the usual WAM style. Instead, a combination of several SAR fields are used for the same purposes. This scheme requires substantial overhead for unpacking the reference or masking certain fields, and comparing the field values. The overall impact is much higher than the simple unsigned address comparison used in statically mapped systems.

This virtual address problem is reduced on the new GP1000 version of the Butterfly. The impact of this change on Boplog would be that overhead for memory map changes (SAR-smashing) would be eliminated, and

virtual addresses would not need to be constructed. Comparisons of virtual addresses would be valid, since each would have a one-to-one correspondence with a location in physical memory. These comparisons could be used to disambiguate bindings and dereference variables more quickly. Stack maintenance would also be easier than in the current implementation, as address comparisons could be used for stack-overflow detection.

5. CONCLUSION

Boplog represents a preliminary effort to identify and confront problems that will impact the implementation of declarative programming paradigms on future machines similar to the Butterfly. The results presented in Section 4 suggest the following conclusions.

In attempting to make all physical memory available to Boplog, a high performance cost is paid. Memory management presents a tantalizing dilemma: without runtime memory map changes, only a small fraction of a large Butterfly's memory can be used; a dynamic memory map is very expensive and restricts system performance. Until this problem is solved (as it appears to be using new hardware in the Butterfly GP1000 series), implementations should be constrained to operate within the bounds of a static memory map. Similarly, memory for important data areas (stack and heap) should be allocated statically during program initialization, and should perhaps share virtual address spaces. Doing so would allow address comparisons for dereferencing, much as in the WAM and SRI models.

Reference locality can be encouraged with suitable choices of data structures. Because each Boplog process manages its own local stack and heap areas, references to remote memory is reduced. Multiple bindings for variables are also stored in the memory of the process which makes the binding, further enhancing locality. Such attributes may be immaterial on bus-connected architectures, but the time ratio of some four to one for remote and local memory operations on the Butterfly makes locality important. Local operations are not only substantially faster, but also place less burden on the switch interconnect. Faster local operations in the GP1000 raise the ratio of remote to local access time to about 15:1, magnifying the importance of locality.

The impact of some kinds of Boplog operations were surprising. Dereferencing took much longer than expected, but task migration was faster and less common than anticipated. Each locking operation was costly, but lock conflicts were infrequent. Although the policy of migrating from "early" choice points was expected to lengthen tasks, the very small percentage of migrated choices was unexpected. Bottlenecks that might have occurred as a result of examining stacks for migration and

termination did not arise. These results lend support to the value of direct measurement over simulation.

The Butterfly described in this article is difficult to use. Its operating system is primitive and attempts to provide capabilities that sequential systems do not need. Its user interface is frustratingly fragile, cumbersome, and inadequate. To use the Butterfly in more than a very naive fashion requires intricate knowledge of the operating system and memory management system as well as the underlying architecture and specific hardware characteristics. Building simple programs can be challenging; building and debugging large programs is nightmarish. The introduction of more modern and complete Butterfly operating systems derived from CMU's Mach[26] will undoubtably make the Buterfly easier to use, but possibly less efficient.

Finally, although the implementation of Boplog on the Butterfly was quite slow, it allowed sampling of important events during actual execution. The statistics gathered during Boplog runs identified problem areas that simulation alone might not have been able to detect. Direct measurement of performance by a real implementation clarifies the impact of low-level architectural, design, and implementation decisions on overall performance.

REFERENCES

1. BBN Advanced Computers, Inc., *Tutorial for Programming in the C Language* (1986).
2. BBN Advanced Computers Inc., *The Uniform System Approach to Programming the Butterfly Parallel Processor* (1986).
3. BBN Laboratories, Inc., *Chrysalis Programmer's Manual*, 2.3.1 Edition (1986).
4. Bruce Moxon, *The Butterfly RAMFile System, BBN* (1986).
5. Peter Tinker, *Managing Large Address Spaces Effectively on the Butterfly*, Technical Report UUCS-87-012, Computer Science Department, University of Utah (April 1987).
6. Peter A. Tinker, *The Design and Implementation of an OR-Parallel Logic Programming System*, PhD thesis, University of Utah, Salt Lake City, Utah (August 1987).
7. Terry Disz, Ewing Lusk, and Ross Overbeek, Experiments with OR-parallel Logic Programming, In *Proceedings of the Fourth International Conference on Logic Programming*, pp. 576–600, MIT Press, Melbourne, Australia (May 1987).
8. R. P. Gabriel and J. McCarthy, Queue-based Multiprocessing Lisp, In *ACM Symposium on Lisp and Functional Programming, ACM*, pp. 25–43, Austin, Texas (August 1984).
9. David H. D. Warren, *An Abstract Prolog Instruction Set*, Technical Note 309, SRI International, Menlo Park, California (October 1983).
10. Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language. Prentice-Hall Software Series*, Prentice-Hall (1978).
11. A. M. Ali Khayri, OR-Parallel Execution of Prolog on a Multi-sequential Machine, *International Journal of Parallel Programming*, **15**(3):189–214 (June 1987).
12. John S. Conery, *Closed Environments: Partitioned Memory Representation for Parallel Logic Programs*, Technical Report CIS-TR-86-02, Department of Computer and Information Science, University of Oregon (February 1986).

13. Peter Bogwardt, Parallel Prolog Using Stack Segments on Shared-memory Multiprocessors, In *Proceedings of the Symposium on Logic Programming, IEEE*, pp. 2–11 (November 1984).
14. Andrzej Ciepielewski and Seif Haridi, A Formal Model for OR-parallel Execution of Logic Programs, In *Proceedings of IFIP*-83 (1983).
15. J. S. Conery, *The AND/OR Process Model for Parallel Interpretation of Logic Programs*, PhD thesis, University of California, Irvine (1983).
16. Bogumil Hausmann, Andrzej Ciepielewski, and Seif Haridi, *OR-parallel Prolog Made Efficient on Shared Memory Multiprocessors*, Technical Report, Swedish Institute of Computer Science (1987).
17. Kouichi Kumon, Kideo Masuzawa, Akihiro Itashiki, Ken Satoh, and Yukio Soma, Kabuwake: A New Parallel Inference Method and Its Evaluation, In *Proceedings of the CompCon, IEEE*, pp. 168–172 (1986).
18. Gary Lindstrom and Prakash Panangaden, Stream-based Execution of Logic Programs, In *Proc. 1984 Int'l. Symp. on Logic Programming*, pp. 168–176 (February 1984).
19. David H. D. Warren, The SRI Model for Or-parallel Execution of Prolog—Abstract Design and Implementation Issues, In *Proceedings of the Symposium on Logic Programming*, pp. 92–102, Seattle, Washington (1987).
20. Jim Crammon, A Comparative Study of Unification Algorithms for OR-parallel Execution of Logic Languages, In *Proceedings of the International Conference on Parallel Processing, IEEE/ACM*, pp. 131–138 (August 1985).
21. Yukio Sohma, Ken Satoh, Kouichi Kumon, Kideo Masuzawa, and Akihiro Itashiki, A New Parallel Inference Mechanism Based on Sequential Processing, In (ed.), J. V. Woods, *Proceedings of the IFIP TC 10 Working Conference on Fifth Generation Computer Architectures*, pp. 3–14, Elsevier Science Publishers, Manchester, United Kingdom (July 1985).
22. Peter Tinker and Gary Lindstrom, A Performance-oriented Design for OR-parallel Logic Programming, In *Proceedings of the Fourth International Conference on Logic Programming*, Melbourne, Australia (May 1987).
23. Leon Sterling and Ehud Shapiro, *The Art of Prolog*, The MIT Press (1986).
24. Fernando Pereira, David Warren, David Bowen, Lawrence Byrd, and Luis Pereira, *C-Prolog User's Manual* (January 1986).
25. David H. D. Waren, Optimizing Tail Recursion in Prolog, In *Logic Programming and its Applications*, Chapter 4, pp. 77–90, Ablex, Norwood, New Jersey (1986).
26. Richard Rashid, Threads of a New System, *Unix Review*, **4**(8):37–49 (August 1986).