ACM *Lisp Pointers* **V**,3 (July-Sept. 1992), 15-17.

The Gabriel 'Triangle' Benchmark at Warp Speed

Henry G. Baker

Nimble Computer Corporation, 16231 Meadow Ridge Way, Encino, CA 91436
(818) 986-1436 (818) 986-1360 (FAX)

We show how to speed up the Gabriel 'Triangle' Benchmark by more than two orders of magnitude (206 X faster than the Cray-1) on a Common Lisp system running on the 80860-based OKIstation, by using better programming techniques. The resulting program fits nicely within next generation on-chip caches and kills almost all potential parallelism, thus becoming worthless as a general-purpose Lisp benchmark.

A. INTRODUCTION

The "Triangle" benchmark is typically the longest of the Gabriel Lisp benchmarks, taking between 14.44 seconds on the Cray-1 and 1510 seconds on the Xerox Dolphin [Gabriel85].[1] Triangle performs a classical AI task, in that it solves a puzzle game using a combinatorial search. Unfortunately, the standard implementation of the Triangle benchmark is an embarrassment to the AI and Lisp communities, because it does not utilize the best algorithm, and therefore continues to perpetuate the myth that "Lisp is slow".

We show how to speed up Triangle through a combination of memoizing [Bird80] and bit-vector techniques [Baker90], to yield a time of 0.07 seconds on a modern RISC workstation (the 80860-based OKIstation™), which is *206 times faster than the Cray-1* speed in [Gabriel85]. Our techniques eliminate redundancy from the computation, which has the effect of eliminating potential parallelism, as well.

B. THE STANDARD TRIANGLE BENCHMARK

The code for Triangle given in [Gabriel85] has a glaring error: it attempts to `map` the `quote` special form over a sequence! We suggest using `(coerce *sequence* 'list)`, instead. The standard Triangle code is otherwise correct.

There are some straight-forward improvements in programming style that would be welcome in a Common Lisp version of Triangle. The most obvious is to use zero-origin addressing for the elements of the `*board*` vector instead of one-origin addressing; this change has the effect of renumbering the holes in the puzzle board to one smaller than the numbers shown in the text [Gabriel85,p.218]. The second is to realize that the last "move" ("6 5 4") in the arrays has been duplicated; i.e., there are only 36 distinct possible moves. A third change in the name of "structured programming" would utilize a single 36-element vector of "moves", each of which would be a 3-element sequence.

A truly "structured programming" form of this program would hide the representation of the board, the set of possible moves, and the moves themselves. Such a form would allow for quick changes of representation to find the best one.

C. SPEEDING UP TRIANGLE A "BIT"

Since the vector representing the Triangle "board" consists of exactly 15 locations, each of which contains either zero or one, one should immediately consider representing this "board" using a 15-bit bit-vector. Since a length-15 bit-vector is quite short, we should consider the use of a `fixnum` integer, so that the entire vector can be manipulated in parallel by a single machine instruction (word-parallelism is about the only parallelism that can be effectively utilized in Triangle). Once the "board" can be so succinctly represented, it can be more efficiently passed as a parameter to `try`, rather than remaining a global variable. Passing the "board" as an argument in this way has the additional advantage that the board does not have to be explicitly restored to its previous configuration, because such restoration is already an integral part of the normal calling sequence for Lisp.

Once the "board" is a bit-vector, then references to the board must be made with `ldb` or `logbitp` instead of `aref`. However, a little thought will show that the representation of the "moves" is sub-optimal, since we must shift a 1-bit each time a board position is to be queried. A better solution is to perform the shifts once during initialization, by changing `*a*`, `*b*`, and `*c*` into vectors of bit-*masks*, rather than vectors of bit-*positions*. Further inspection of the `try` function reveals the fact that `*a*` and `*b*` are always used together, so a single vector of masks containing both the "a" and "b" bits will suffice.

[1]Modern RISC architecture performance numbers are closing in on the Cray-1, and better Cray-1 numbers are available [Anderson87], but these do not invalidate our conclusions.

With these changes, `try` now looks like this:

```
(defun try (board i depth)
  (if (= depth 14) (progn (pushnew (1- (integer-length board)) *final*)
                          (push (cdr (coerce *sequence* 'list)) *answer*)
                          t)
    (let* ((ab (aref *ab* i)) (c (aref *c* i)))
      (if (and (= (logand board ab) ab) (zerop (logand board c)))
        (progn (setf (aref *sequence* depth) i)
               (do ((j 0 (1+ j))
                    (nboard (logxor board ab c))
                    (ndepth (1+ depth)))
                   ((or (= j 36) (try nboard j ndepth)) nil)))))))
```

D. MEMOIZING TRIANGLE

The use of bit-twiddling instead of array-hacking can produce an order of magnitude improvement in the Triangle benchmark, but it doesn't attack the real problem of Triangle—its redundancy. Like most combinatorial searching programs, Triangle spends an inordinate amount of time re-investigating the same board positions. This can easily be seen if one compares the number of calls to `try` in the standard benchmark (5,802,572) with the total number of possible board positions (2^{15} = 32,768). The key to eliminating this redundancy is to keep a table of board positions already considered. The size of this table is bounded by the number of possible board positions (32,768), so we can utilize either a hash table or a simple vector indexed by the "board" itself.

However, before we can utilize a table of board positions, we must decide what information should be stored into the table. We could store an entire set of moves for each board position, but these sets are quite large themselves, so merely constructing them will dominate the computation. Since most of these sets will never be required, such constructions are a waste of effort. The minimum amount of information required for each board position is a single bit—whether the current board position can be extended into a "winning" position. An efficient algorithm can be constructed which is based on this implementation. We will use a slightly different representation, however. Our table will store for each board position the list of *first* moves which lead to winning positions; this list will be empty if there are no winning moves at all. (Note that the number of first moves which lead to a winning position can be much smaller than the number of moves possible in the current board configuration.) This choice will lead to a slightly simpler algorithm for reconstructing the winning move sequences.

We are thus led to a two-stage algorithm—determine the winning moves starting from the initial board position, and then reconstruct the winning sequences. By deferring the construction of the winning sequences until the second stage, we avoid constructing any but the winning sequences.

```
(defun try (board depth &aux (entry (aref *table* board)))
  (if (not (empty entry)) entry   ;emptiness is an arbitrary non-'list' value.
    (setf (aref *table* board)
      (if (= depth 14) (cons nil nil)
        (mapcan
         #'(lambda (i)
             (let* ((ab (aref *ab* i)) (c (aref *c* i)))
               (when (and (= (logand board ab) ab)
                          (= (logand board c) c)
                          (try (logxor board ab c) (1+ depth)))
                 (list i))))
         *moves*)))))

(defun construct (board depth movelst &aux (entry (aref *table* board)))
  (if (= depth 14) (progn (pushnew (1- (integer-length board)) *final*)
                          (push movelst *answer*)) ;answers are reversed.
    (dolist (i entry nil)
      (construct (logxor board (aref *ab* i) (aref *c* i))
                 (1+ depth)
                 `(,i ,@movelst)))))
```



```
(defun gogogo (i)
  (let* ((*answer* ()) (*final* ())
         (board (logxor (- 32767 16) (aref *ab* i) (aref *c* i))))
    (try board 2)
    (gather board 2 nil)
    *answer*))
```

In the above program, the same 775 solutions are still generated, but in a time of just 0.07 seconds—0.05 seconds for `try` and 0.02 seconds for `construct`. After the search, `*table*` has 1195 entries, of which only 121 are non-null. The non-null entries contain just 246 cons cells, for an average list length of about 2. `construct` uses just 4827 cons cells for representing the winning lists of moves themselves, for an average of about 6 cons cells per list, even though each winning list prints as 12 cons cells, and 775 cons cells for the list of winning lists. Thus, we save about a factor of 2 in consing by sharing as much of the sequence lists as possible.

We also ran Triangle without choosing the first move ("22"); this run produces 1550 solutions, half of them mirror images of the other half. However, this run took only 0.12=0.07+0.05 seconds—less than twice as long as the normal run. The savings become clear when `*table*` is examined; after this run, `*table*` has 1651 entries, of which 190 are non-null, with an average list-length of about 2. The constructed winning lists use 9656 cons cells in addition to the 1550 cells required for the list of winning lists. Thus, we can generate all of the solutions to Triangle 120 times faster than the Cray-1 produces only half of them using the old algorithm.

E. RESULTS

The bit-vector and memoizing optimizations improve Triangle by more than two orders of magnitude. With but a day's work on the 40MHz 80860-based OKIstation, we achieved a Triangle time of 0.07 second, which is 206 X faster than the Cray-1 on the old benchmark. Although our version of Triangle uses a memo table implemented as a simple vector, this table requires 128Kbytes and is only 4% full, so a smaller true hash table might actually be faster so long as the entire table fits in the cache. Unfortunately, our reductions in memory utilization have destroyed the usefulness of this benchmark, because the memoized Triangle benchmark will fit entirely within the on-chip cache of the next generation microprocessors.

The success of memoization shows that almost all of the parallel processes in the parallel implementations of Triangle are performing identically the same computation, hence the Triangle benchmark is worthless for estimating parallelism in real (i.e., highly optimized) applications. In a sense, memoization is the most powerful of all parallel programming techniques, because one processor simulates the execution of many processes with a single execution!

F. REFERENCES

Anderson, J.Wayne, *et al*. "Implementing and Optimizing Lisp for the Cray". *IEEE Software* (July 1987),74-83.

Baker, H.G. "Efficient Implementation of Bit-vector Operations in Common Lisp". ACM *Lisp Pointers 3*,2-3-4 (April-June 1990), 8-22.

Bird, R.S. "Tabulation Techniques for Recursive Programs". *ACM Comp. Surv. 12*,4 (Dec. 1980),403-417.

Gabriel, R.P. *Performance and Evaluation of Lisp Systems*. MIT Press, Camb., MA, 1985.

Keller, R.M., and Sleep, M.R. "Applicative Caching". *ACM TOPLAS 8*,1 (Jan. 1986),88-108.

Steele, Guy L. *Common Lisp, The Language; 2nd Ed*. Digital Press, Bedford, MA, 1990,1029p.

