SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 19(3), 273–281 (MARCH 1989)

Comparative Timings of Three Different Set Implementations in occam

G. A. WILSON
Department of Computer Science/Department of Information Studies, University of Sheffield, Sheffield S10 2TN, U.K.

SUMMARY

Three different occam implementations of a Set **datatype have been investigated, using arrays of bits, booleans (bytes) and full integers, and the performance of each compared. Execution times and code/data requirements are recorded, and surprisingly the best implementation is not as originally expected.**

KEY WORDS occam Sets Programming

INTRODUCTION

The author's research, involving a number of algorithms for chemical substructure searching, required the use of initially simple, and subsequently more comprehensive, set manipulation routines. The occam language provides no direct support for a number of features present in other languages; instead libraries of functions, procedures and constant definitions are used to provide the required extra functionality. Set support is one such feature not included in the standard set of libraries issued with the Inmos TDS (transputer development system), and so a simple library was developed containing routines such as

Fill	— fill a set with all members
Empty	— empty a set of all members
Insert	— insert a member into a set
Remove	— remove a member from a set
TestIn	— test if a member is present in a set
TestEmpty	— test if the set has no members
Compare	— test if two sets contain the same members
Count	— count the number of members in a set
And	— form new set of members common to two sets (set intersection)
Or	— form new set of members in either of two sets (set union)
Not	— form new set of members not present in a set (complement)
List	— identify numerically all members present in set
FindFirstMember	— find first member present in a set

0038–0644/89/030273–09$05.00

Received 12 May 1988
Revised 8 August 1988

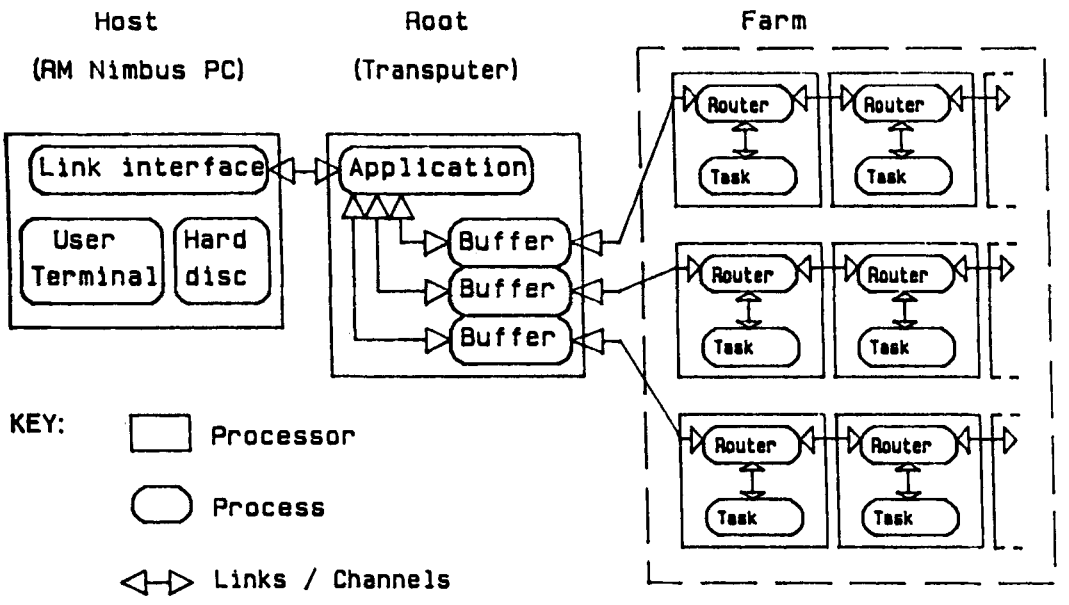

Figure 1. Processor farm environment and topology

The initial serial searching algorithms developed represented sets using boolean arrays (which occam currently implements as arrays of bytes), where each boolean represented one member of the set. Thus a set of set.size items is declared [set.size]BOOL Set, member i is present if Set[i] = TRUE, and absent if Set[i] = FALSE. The set routines were then all short and simple to develop, and the whole package was considered suitable at the time.

The research then required the distribution of the searching algorithms over arrays of processor 'farms', in which packets of work are formed in a *root* processor, and farmed out to some network of *worker* processors for concurrent processing. Figure 1 illustrates a typical processor farm environment, using a triple-chain topology for maximum communications bandwidth between the root and the farm. The root processor runs the *Application* process, which sends data messages into, and collects result messages out of, three identical chains (of any length) of worker processors. The *Buffer* processes enable concurrent communication with all three chains. Each worker processor concurrently runs a *Router* process, which routes messages around the farm, and a *Task* process, which processes data messages into result messages. The root and worker processors used were the INMOS transputer, and the root processor was connected to a PC host machine providing terminal I/O and disk storage. However, it was soon discovered that the fairly large granularity of the problem (the typical ratio of communications time of a work packet to the amount of processing necessary for that packet) restricted the maximum number of nodes possible in the farms, and overall algorithm speed-up, to low numbers.

Work packets into the farm contained a substantial quantity of set data, and to improve granularity the packet sizes, and hence communication costs, were reduced. This was achieved using a second implementation of the set datatype, where a single

bit represented each set member. A set was represented as an array of (32-bit) integers, each integer containing up to 32 members of the set, and individual members were accessed using the bit masking and shifting operations present in occam. Again, a set of set.size items is declared [(set.size+31)/32]INT Set (i.e. rounded up to the next multiple of 32), and member i is present or absent depending on whether the (i REM 32)th bit of Set[i/32] is set or not.

The performance of the processor farms was seen to improve; typically one application which could previously use a maximum farm size of two nodes to achieve near unity speed-up in execution time over the serial case (i.e. the multiprocessor version actually ran slower!) was now able to use farms of four nodes to achieve 2 to 3 times speed-up. The new set implementation was introduced into the serial algorithms, and surprisingly performance was also seen to improve; again the same (serial) algorithm now executed in 180 s compared to 230 s previously.

COMPARATIVE TIMINGS

Timings of individual routines were carried out on four different sizes of sets, namely 5, 20, 40 and 60 members, and results averaged over 100 calls. Many of the routines have similar internal forms, so only a subset was considered for timing purposes, and the results extrapolated according to the classification shown in Table I.

The test program harness is shown in Figure 2, the actual tests performed can be seen in more detail in Figure 3 and the results are reported in Table II for the bit implementation and Table III for the boolean implementation. The code size and data required for each test program are also reported. The FindFirstMember routine is tested twice, once with only the first member present (headed Find0), and once with only the last member present (headed FindN). A traditional mainframe-style implementation of sets, using one word for each set member (declared [set.size]INT Set, Set[i] = 1 for member i present, Set[i] = 0 for member i absent) was also considered for completeness, and these results are reported in Table IV.

Care was taken to ensure that the same test harness could be used for each implementation, and that the set libraries contained the same routines in the same order. The transputer procedure call mechanism can take different times according to how far away the routine is, i.e. according to the order in which the library routines are linked

Table I

Routine tested	Representative of
Empty Set	
FillSet	
InsertSet	RemoveSet, TestInSet
CountSet	
NotSet	
AndSet	OrSet
CompareSet	TestEmptySet
FindFirstMember	

```
PROC exe (CHAN OF ANY Keyboard, Screen)
  #USE "l:\tdsiolib\userio.tsr"
  #USE "a:\lib\bitset.tsr"              * pick one of *
  --#USE "a:\lib\boolset.tsr"           * these three *
  --#USE "a:\lib\intset.tsr"            * libraries   *
  VAL repeat IS 100:        -- average over this many calls
  INT timenow, timebefore:
  TIMER clock:
  [2]INT32 big.set:    -- bits          * pick one of *
  --[60]BOOL big.set:    -- bools       * these three *
  --[60]INT big.set:     -- words       * declarations*
  PROC timeit ()
    write.int (Screen, (timenow - timebefore) / repeat, 7)
  :
  SEQ
    SEQ i = 0 FOR 4              -- test four different sizes
      VAL set.size IS [5,20,40,60][i]:
      --set IS [big.set FROM 0 FOR set.size]:
      --                              bool and int sets only
      VAL bit.set.size IS (set.size + 31) / 32:
      set IS [big.set FROM 0 FOR bit.set.size]:
      --                              bit sets only
      SEQ
        write.full.string (Screen, "*c*nSize:")
        write.int (Screen, set.size, 3)
        ... do the tests
    Keyboard ? timenow           -- pause
:
```

Figure 2. Test program harness

Table II. Timing results — bit implementation

Size	Empty	Fill	Insert	Count	Not	And	Compare	Find0	FindN
5	12	28	74	43	32	19	18	18	38
20	12	26	291	119	30	19	18	18	116
40	16	32	586	221	39	28	25	18	59
60	16	30	880	321	37	28	25	18	169

Code space: 448 bytes, data space: 188 words

Table III. Timing results — boolean implementation

Size	Empty	Fill	Insert	Count	Not	And	Compare	Find0	FindN
5	30	30	50	43	43	55	49	13	34
20	93	99	191	138	142	186	162	13	113
40	176	190	375	266	275	359	313	13	217
60	259	280	561	394	407	532	463	13	320

Code space: 500 bytes, data space: 200 words

Table IV. Timing results — word implementation

Size	Empty	Fill	Insert	Count	Not	And	Compare	Find0	FindN
5	29	29	52	42	43	53	48	13	35
20	96	97	200	140	142	179	156	13	117
40	183	186	395	272	275	346	301	13	227
60	270	274	591	403	407	513	445	13	335

Code space: 484 bytes, data space: 245 words

```
clock ? timebefore
SEQ i = 0 FOR repeat
  EmptySet (set)
clock ? timenow
timeit ()

clock ? timebefore
SEQ i = 0 FOR repeat
  FillSet (set, set.size)   -- bit sets only
  --FillSet (set)           -- others
clock ? timenow
timeit ()

clock ? timebefore
SEQ i = 0 FOR repeat
  SEQ i = 0 FOR set.size
    InsertSet (set, i)
clock ? timenow
timeit ()

FillSet (set, set.size)     -- bit sets only
--FillSet (set)             -- others
clock ? timebefore
SEQ i = 0 FOR repeat
  INT members:
  CountSet (set, set.size, members) -- bit sets only
  --CountSet (set, members)         -- others
clock ? timenow
timeit ()

clock ? timebefore
SEQ i = 0 FOR repeat
  NotSet (set, set, set.size)   -- bit sets only
  --NotSet (set, set)           -- others
clock ? timenow
timeit ()

clock ? timebefore
SEQ i = 0 FOR repeat
  AndSets (set, set, set)
clock ? timenow
timeit ()

clock ? timebefore
SEQ i = 0 FOR repeat
  BOOL same:
  CompareSets (set, set, same)
clock ? timenow
timeit ()

EmptySet (set)
InsertSet (set, 0)
clock ? timebefore
SEQ i = 0 FOR repeat
  INT member:
  FindFirstMember (set, member)
clock ? timenow
timeit ()

EmptySet (set)
InsertSet (set, set.size-1)  -- last member
clock ? timebefore
SEQ i = 0 FOR repeat
  INT member:
  FindFirstMember (set, member)
clock ? timenow
timeit ()
```

Figure 3. Tests performed

```
-- this implementation maintains any unused bits in the
-- last word of a set at 0.
--
VAL empty.word IS 0(INT32), bit.one IS 1(INT32):
PROC EmptySet ([]INT32 Set)
  SEQ Word = 0 FOR (SIZE Set)
    Set[Word] := empty.word
:
PROC CountSet (VAL []INT32 Set, VAL INT max.no.members,
               INT Members)
  INT32 Membs, temp:
  VAL remainder IS max.no.members \ 32:
  INT full:
  SEQ
    full := (SIZE Set) - 1    -- full words to count
    Membs := 0(INT32)
    IF
      remainder = 0
        full := full + 1      -- last word also full
      TRUE
        SEQ
          temp := Set[full]   -- last word not full
          SEQ i = 0 FOR remainder
            SEQ
              Membs := Membs PLUS (temp /\ bit.one)
              temp := temp >> 1
    SEQ Word = 0 FOR full     -- count the full words
      SEQ
        temp := Set[Word]
        SEQ i = 0 FOR 32      -- each of 32 members
          SEQ
            Membs := Membs PLUS (temp /\ bit.one)
            temp := temp >> 1
    Members := (INT Membs)
:
PROC NotSet ([]INT32 SetA, VAL []INT32 SetB,
             VAL INT max.no.members)
  VAL size IS SIZE SetA:
  VAL remainder IS max.no.members \ 32:
  SEQ
    SEQ i = 0 FOR size
      SetA[i] := BITNOT SetB[i]
    IF
      remainder = 0
        SKIP
      TRUE
        VAL shift IS 32 MINUS remainder:
        lastword IS SetA[size-1]:
        SEQ
          lastword := lastword << shift    -- clear
          lastword := lastword >> shift    -- unused bits
:
PROC AndSets ([]INT32 SetA, VAL []INT32 SetB, SetC)
  SEQ i = 0 FOR SIZE SetA
    SetA[i] := SetB[i] /\ SetC[i]
:
PROC FillSet ([]INT32 Set, VAL INT max.no.members)
  VAL size IS SIZE Set, Full IS BITNOT empty.word:
  VAL remainder IS max.no.members \ 32:
  SEQ
    SEQ Word = 0 FOR size
      Set[Word] := Full
    IF
      remainder = 0
        SKIP
      TRUE
      VAL shift IS 32 MINUS remainder:
      lastword IS Set[size-1]:
```

```
        SEQ
          lastword := lastword << shift    -- clear
          lastword := lastword >> shift    -- unused bits
:
PROC InsertSet ([]INT32 Set, VAL INT Member)
  Word IS Set[(Member >> 5)]:
  VAL Index IS Member /\ 31:
  Word := Word \/ (bit.one << Index)
:
PROC FindFirstMember (VAL []INT32 Set,
                      VAL INT max.no.members, INT Member)
  IF
    IF i = 0 FOR SIZE Set
      VAL Word IS Set[i]:
      Word <> empty.word        -- word not empty
        IF j = 0 FOR 32         -- find first set bit
          (Word /\ (bit.one << j)) <> empty.word -- improve?
            Member := (i << 5) PLUS j
    TRUE
      Member := -1              -- no members
:
PROC CompareSets (VAL []INT32 SetA, SetB, BOOL Same)
  IF
    IF i = 0 FOR SIZE SetA
      SetA[i] <> SetB[i]
        Same := FALSE
    TRUE
      Same := TRUE
:
PROC RemoveSet ([]INT32 Set, VAL INT Member)
  Word IS Set[Member >> 5]:
  VAL Index IS Member /\ 31:
  Word := Word /\  (BITNOT (bit.one << Index))
:
PROC TestInSet (VAL []INT32 Set, VAL INT Member,
                BOOL YesOrNo)
  VAL Word IS Set[Member >> 5]:
  VAL Index IS Member /\ 31:
  YesOrNo := (Word /\ (bit.one << Index)) <> empty.word
:
PROC TestEmptySet (VAL []INT32 Set, BOOL Empty)
  IF
    IF word = 0 FOR SIZE Set:
      Set[word] <> empty.word
        Empty := FALSE
    TRUE
      Empty := TRUE
:
PROC ListSet (CHAN OF ANY Sink, VAL []INT32 Set)
  #USE "l:\tdsiolib\userio.tsr"
  SEQ Word = 0 FOR (SIZE Set)
    VAL PartMember IS Word * 32:
    INT32 temp:
    SEQ
      temp := Set[Word]
      SEQ i = 0 FOR 32
        SEQ
          IF
            (temp /\ bit.one) = bit.one
              SEQ
                write.int (Sink, PartMember + (i + 1), 0)
                write.char (Sink, ' ')
            TRUE
              SKIP
          temp := temp >> 1
:
```

Figure 4. The bit-set library

in to the program ; and variable access times are similarly affected by relative positions in the code. It is assumed that this is the reason for the slight difference in timings between the Fill and Empty routines, despite their having almost identical code, for the boolean and integer implementations.

ANALYSIS OF RESULTS

Clearly, the bit-set routines outperform all others, except for the following cases: InsertSet, where there is a similar order of performance; CountSet, where performance is worse only for small sets; and FindFirstMember, where performance is worse only when finding earlier first members.

The data-space figures progress as expected; however, the code-space figures are unusual. Despite being (textually) the most complex, the bit sets require the least code space. We also notice the unusual timings for FindFirstMember using the bit sets. To find member 20, 20 bit tests are necessary; whereas to find member 40, 32 members are tested concurrently as a single-word test, and then only eight further bit tests are necessary.

EXAMPLE LIBRARIES

The source of the bit-set Library is shown in Figure 4, whereas Figure 5 compares the code of the InsertSet routine using all three implementations. All of the code is simple and illustrative only, and omits error checking for example. Much use is made of the replicated IF construct to produce concise and elegant code. The routines will manipulate sets of any size, limited only by data-storage capacity.

All tests are run on a revision A processor. The compiler used is that supplied with the Inmos beta2 release of the transputer development system, which does not yet support functions. A number of routines, namely FindFirstMember, CountSet, TestInSet, CompareSets and TestEmptySet, could be converted usefully to functions. Unfortunately, an implementation restriction in the impending product release compiler is that array values may not be returned by functions, and so routines such as AndSets currently cannot be converted to functions.

Notice that the bit-array library is slightly less convenient to use, owing to the necessary redefinition of the set size. In the other two implementations, any routine requiring to know the set size may determine it simply using the occam SIZE operator, whereas the bit array routines require this value to be passed explicitly. It is possible to write a modified implementation suitable for set sizes in multiples of 32 only, which would not only be more efficient, but conveniently would not require the extra set-size parameter.

CONCLUSIONS

The bit implementation of the set library is more efficient, not only for data storage, but surprisingly also in code size, and in execution time for the majority of cases. For applications containing a majority of insert/remove operations the boolean library may be considered. Finally, the full word implementation seems to have nothing to recommend it to any applications.

```
VAL bit.one IS 1 (INT32):
PROC InsertSet ([]INT32 Set, VAL INT Member)
  Word IS Set[(Member >> 5)]:
  VAL Index IS Member /\ 31:
  Word := Word \/ (bit.one << Index)
:

PROC InsertSet ([]BOOL set, VAL INT member)
  set[member] := TRUE
:

PROC InsertSet ([]INT set, VAL INT member)
  set[member] := 1
:
```

Figure 5. Code comparison of all three implementations

ACKNOWLEDGEMENTS

The author is currently researching into 'The application of reconfigurable microprocessors to information retrieval problems'. This work is funded by the British Library and the U.K. Department of Trade and Industry. Thanks are due to the supervisors of the project, Professor M. F. Lynch, Dr G. A. Manson and Dr P. Willet.