Functional, I-Structure, and M-Structure Implementations of NAS Benchmark FT

S. Sur and W. Böhm

Department of Computer Science
Colorado State University
Ft. Collins, CO 80523

Abstract:

We implement the NAS parallel benchmark FT, which numerically solves a three dimensional partial differential equation using forward and inverse FFTs, in the dataflow language Id and run it on a one node monsoon machine. Id is a layered language with a purely functional kernel, a deterministic layer with I-structures, and a non-deterministic layer with M-structures. We compare the performance of versions of our code written in these three layers of Id. We measure instruction counts and critical path length using the Monsoon Interpreter Mint. We measure the space requirements of our codes by determining the largest possible problem size fitting on a one node monsoon machine. The purely functional code provides the highest average parallelism, but this parallelism turns out to be superfluous. The I-structure code executes the minimal number of instructions and as it has a similar critical path length as the functional code, runs the fastest. The M-structure code allows the largest problem sizes to be run at the cost of about 20% increase in instruction count, and 75% to 100% increase in critical path length, compared to the I-structure code.

Keyword Codes: D.1.1.; D.3.3.; E.2
Keywords: Applicative Programming, Language Constructs and Features, Data Storage Representations

1 Introduction

In this paper we study the design of efficient declarative programs by implementing the NAS three dimensional FFT PDE benchmark FT [2] in Id [8]. This study is part of a larger project where we try to assess which declarative language features are of importance to write efficient scientific codes. A declarative programming language allows expressing *what is to be done*, without specifying too much of *how it is to be done*. As an example, declarative programming languages are implicitly parallel, i.e., allocation of tasks and data on processors are not expressed in the program. This frees the programmer from this level of complexity in the design of parallel programs.

The declarative programming language Id has a functional kernel. Arrays in this functional kernel are created using monolithic array constructors called *array comprehensions*. In [1], Arvind and others argue that these array comprehensions lack expressiveness, and that for certain problems a lower level of constructs, manipulating the elements of I-structures, is necessary. An I-structure is a single assignment array with element-level synchronization for reads and writes. Because of their single assignment nature, Id programs with I-structures are deterministic, even though pure functional referential transparency has been lost. In [3] it is shown that for certain problems I-structures are again not powerful enough and that the more expressive M-structures are needed. M-structures are also arrays with element-level synchronization, but do not have the single assignment property anymore. M-structures allow "put" operations to write in an empty array slot, and "get" operations read and empty an array slot. Therefore, with interleaved puts and gets, M-structures allow destructive updates to express potentially non-deterministic producer-consumer relationships.

The above papers argue convincingly that the Id language gets increasingly more expressive when I-structures and M-structures are added. In this paper, we are interested in the time and space efficiency of realistic programs, when written in these various layers of the language. We therefore analyse three Id implementations of the NAS FT benchmark: a functional version, a version using I-structures, and a version using M-structures. We measure the time complexity of our programs by running small problems on the Monsoon Interpreter MINT, which reports on the number of instructions executed and the critical path length and provides parallelism profiles. We measure the space complexity of our programs by determining the maximal problem size that fits in our one node Monsoon machine [7], which has a 4 Megaword data memory. In this case, the goal is to run a $64 \times 64 \times 64$ problem, given in the NAS benchmark specification as the minimal sized problem, on a one node Monsoon machine. One 3-D 64^3 object contains half a Megaword of floating point numbers.

It turns out that the purely functional code provides the highest parallelism, but at the cost of high instruction counts and high space usage. The I-structure code executes the minimal number of instructions and runs the fastest on the one node Monsoon machine, which provides 8-fold parallelism. The M-structure code allows the largest problem sizes to be run and turns out to be the only code that allows us to run the $64 \times 64 \times 64$ problem.

The rest of this paper is organized as follows. Section 2 defines the FT NAS benchmark solver. Section 3 first discusses the representation of 3-D objects, and then highlights the differences in programming styles in the functional, I-structure, and M-structure codes. In section 4 we analyse the time and space performance of our three codes. Section 5 mentions related work and concludes.

2 Problem specification

In the NAS benchmark *FT*, the following three dimensional heat equation is solved numerically:

$$\frac{\delta u(x,t)}{\delta t} = \alpha \nabla^2 u(x,t)$$

where x is a position in three dimensional space and α a constant describing conductivity. When a Fourier transform is applied to each side, this equation becomes:

$$\frac{\delta v(z,t)}{\delta t} = -4\alpha\pi^2 |z|^2 v(z,t)$$

where $v(z,t)$ is the Fourier transform of $u(x,t)$. This equation has the solution:

$$v(z,t) = e^{-4\alpha\pi^2|z|^2 t} v(z,0)$$

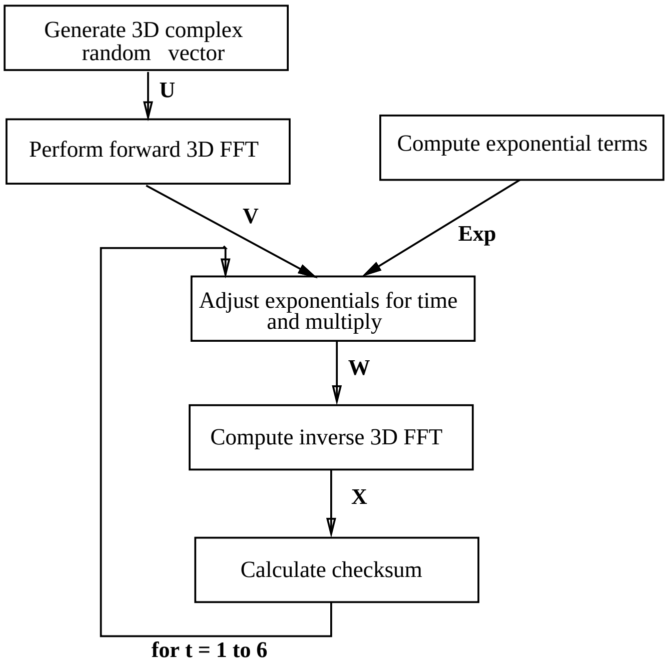

Figure 1: Flow diagram of top level of NAS benchmark FT

The discrete version of the above problem can be solved using Discrete Fourier Transforms (DFT) instead of continuous ones. First, a 3-D DFT is performed on the original state array $u(x, 0)$, then the results are multiplied by certain exponentials and lastly an inverse 3-D DFT is performed (see Figure 1.) The forward and inverse DFTs of the n_1 x n_2 x n_3 array u are defined respectively as:

$$F_{q,r,s}(u) = \sum_{l=0}^{n_3-1} \sum_{k=0}^{n_2-1} \sum_{j=0}^{n_1-1} u_{j,k,l} e^{-2\pi ijq/n_1} e^{-2\pi ikr/n_2} e^{-2\pi ils/n_3}$$

$$F^{-1}_{q,r,s}(u) = \frac{1}{n_1 n_2 n_3} \sum_{l=0}^{n_3-1} \sum_{k=0}^{n_2-1} \sum_{j=0}^{n_1-1} u_{j,k,l} e^{2\pi ijq/n_1} e^{2\pi ikr/n_2} e^{2\pi ils/n_3}$$

In the FT benchmark, the complex array U is initialized using a pseudo-random number generator. Setting V equal to the 3-D DFT of U, $\alpha = 10^{-6}$ and $t = 1$, the intermediate value W is computed:

$$W_{j,k,l} = e^{-4\alpha\pi^2(\bar{j}^2+\bar{k}^2+\bar{l}^2)t} V_{j,k,l}$$

where $\bar{j}$ is defined as j for $0 \leq j < n_1/2$ and $j - n_1$ for $n_1/2 \leq j < n_1$. The indices $\bar{k}$ and $\bar{l}$ are similarly defined with n_2 and n_3. X, the 3-D inverse DFT of W, is then computed. Finally, a checksum $\sum_{j=0}^{1023} X_{q,r,s}$ is computed where $q = j \ (mod\ n_1)$, $r = 3j \ (mod\ n_2)$ and $s = 5j \ (mod\ n_3)$. The computation of W, X and the checksum, is repeated for values t from 2 to 6. V needs only to be computed once. The array of exponential terms for $t > 1$ can be obtained as the t-th power of the array for $t = 1$.

The benchmark allows any algorithm be used for the computation of the 3-D FFTs. The algorithm we implement takes a complex array of size $n_1 \times n_2 \times n_3$ and performs $n_2 \times n_3$ n_1-point 1-D FFTs in the $n1$ direction, followed by $n_3 \times n_1$ n_2-point 1-D FFTs in the $n2$ direction, followed by $n_1 \times n_2$ n_3-point 1-D FFTs in the $n3$ direction yielding the final result. The benchmark also allows any algorithm to be used for the individual 1-D complex FFTs. We use a straight-forward iterative algorithm which reorders the array using bit-reversal of the index, and performs butterfly group recombinations, where the smallest group is 4, and the group size doubles in each iteration. Using a bottom case of 4 instead of 1 or 2 cuts out the bottom-most branches of the FFT tree making it much more space and time efficient. It also makes resource management simpler as pointers and objects are not interchanged [4]. We use iterative FFTs instead of recursive ones, because in the iterative codes all the intermediate arrays can be deallocated immediately after they are no longer required. Details about this can also be found in [4].

3 Implementation

3.1 Data Representation

The first choice for the data representation that comes to mind is a 3-D array of complex numbers represented by tuples of two real numbers. However, Id does not treat, for instance, a 1-D sub-array of such an array (e.g. A[i,j,*]) as an independent data structure. Ideally we would like a 1-D FFT function to be able to generate its result directly in a 1-D sub-array of our 3-D array in any of the 3 directions. As this is impossible, it is just as simple and more efficient to have a linear data structure representing the 3-D object. Selecting a vector in a certain direction now becomes stepping through the array with the appropriate stride. Having the complex numbers represented by tuples introduces considerable inefficiency because of the extra indirection introduced by the tuples. Moreover, deallocating an array of tuples can cause complications, if the array elements are sometimes copies (in which case only a new pointer is created) and sometimes new values (in which case a new tuple and a new pointer is created) [4]. We therefore opt for the simplest data representation possible: a linear array of $2n_1n_2n_3$ floating point numbers. To get the input array, $2n_1n_2n_3$ pseudo-random floating point values are generated as specified in the FT benchmark, and then used to fill the complex array $U_{j,k,l}$, $0 \leq j < n_1$, $0 \leq k < n_2$, $0 \leq l < n_3$, where the first dimension varies most rapidly as in the ordering of a 3-D Fortran array. A single complex entry of U consists of two consecutive pseudorandomly generated results and is stored such that the real and imaginary parts are a distance of $n_1n_2n_3$ apart.

3.2 Purely functional implementation

In the functional version of the program, arrays are created using *array comprehensions*. An array comprehension creating an n-dimensional array consisting of m regions is of the form:

$$
\begin{array}{l}
nD\_array((l_1, u_1), \ldots, (l_n, u_n)) \text{ of} \\
\quad | \, [f_1^1(I_1), \ldots, f_1^n(I_1)] = expr_1 \;\; || \; gen_1^1 \; ; \ldots ; \; gen_1^{d_1} \\
\quad \vdots \\
\quad | \, [f_m^1(I_m), \ldots, f_m^n(I_m)] = expr_m \;\; || \; gen_m^1 \; ; \ldots ; \; gen_m^{d_m}
\end{array}
$$

where each generator gen_j^k , is of the form: $i_j^k \leftarrow l_j^k \ to \ u_j^k$. Zero or more generator expressions define a region using a cross product of nested loops. I_j is the vector of loop variables for region j. Array comprehensions allow for elegant concise definitions of arrays. There are, however, a number of problems:

- **No Sharing.** When the computation of a number of array elements can share sub-computations, this cannot be expressed in an array comprehension, as each array element is defined independently.
- **No sub-array target.** We cannot create a substructure and scatter it over a larger whole array as array comprehensions work on element level.
- **More intermediate arrays.** As an array comprehension does not allow the expression of loop carried dependencies, extra intermediate arrays often need to be created.

The consequence of this lack of expressiveness of purely functional code is a high instruction count as well as a high storage use, as we shall see in the Results and Analysis section.

3.3 I-structure Implementation

An I-structure can be created empty somewhere in the code, and partly or completely filled throughout the program. An I-structure allows array elements to be defined by array element assignments, but still retains determinacy by allowing each element to be defined at most once. The I-structure implementation of this benchmark is the closest to the problem definition. Also almost nowhere were we forced to over-specify intermediate array details introduced by the purely functional style. This makes the I-structure approach in certain circumstances more declarative than the purely functional approach [3]. Array elements are now defined in loop constructs and nothing prevents us from defining more than one element in one loop body, thus avoiding the sharing problem of array comprehensions. In I-stucture code we can use loop carried dependencies to avoid the creation of unnecessary intermediate structures. As we shall see, this makes I-structure codes more efficient in instruction counts as well as space usage.

3.4 M-structure Implementation

The problem with the I-structure implementation still is that, when performing a 1-D FFT on a sub-array of the 3-D object, we need two versions of the object: the one that is read, and the one that is written. When inspecting the FFT algorithm, it is clear that exactly the same elements that are being read, need to be rewritten. An imperative algorithm would use only one data structure. M-structures allow us to do the same thing in a declarative context. It should be noted that M-structures are not needed here for expressiveness reasons: the algorithm can be expressed very naturally using I-structures. The only reason to go to an M-structure implementation of FT is space efficiency.

As was mentioned in the introduction, M-structures allow elements to be "put" into an empty location: $A![i] = expression$. A *put* cannot overwrite a value, i.e. a *put* to a full location will result in a run-time error. Elements can be extracted, i.e. read and emptied, by a "get" operation: $x = A![i]$. A third operation $x = A!![i]$ reads (or extracts and puts back) an M-structure element. A fourth operation $A!![i] = x$ replaces (extracts and puts a new value) an element of an M-structure. Notice that the operations with one ! change the full/empty state of the elements, whereas the operations with the double !! take a full element and leave it full.

M-structures in a parallel model of computation, such as the Id model of computation, give rise to non-determinism: if a number of threads try to get an M-structure element, only one can have it, and which one that will be depends on the arrival time of the particular get operation.

We found that there are two styles of M-structure programming. The *imperative* style uses reads and replaces to ensure that M-structure elements are always full, and provides explicit synchronization using barriers (notation - - -). This style mimics imperative programming closely. Of the M-structure implementations of the FT benchmark, this is the easiest to write, simply because use of replace operations on M-arrays can closely imitate imperative programming style. Use of barriers for explicit synchronization can also emulate close to sequential programming style, extending the ease of programming (to people who are more familiar with sequential thought process). This style gives us the maximum space efficiency we are after. However, the use of barriers combined with the reads and replaces, which are more expensive than gets and puts, makes this style almost as inefficient as the purely functional style!

The second style of M-structure programming, the *data driven style*, uses *puts* and *gets* for both communication and synchronization and avoids barriers as much as possible. Now the *puts* and *gets* need to be perfectly balanced, which makes this style much harder and more error prone. However, inspecting the behavior of FFT algorithms, it is clear that the butterfly data dependences in the recombination phases allow extracting two elements, performing some shared computation on these and putting two elements back in exactly the same place. Therefore, *puts* and *gets* without extra barriers work for FFTs.

3.5 Code Example

We consider the function that performs $n2$ one dimensional FFTs each of size $n1$ from an array which is $n1 \times n2$ long. This essentially requires partitioning the input vector in slices, performing a 1-D FFT on each such slice and gluing the resulting together to obtain the resulting array. An I-structure implementation of the function is as follows:

```
def cffts IS  n1 n2 x ro = {
stride = n1*n2; size = n1*2; typeof v = I_vector(F); v= I_vector(1, 2*n1*n2);
  {for j<-1 to n2 do
      y = slice j n1 stride x;  z = fft y ro IS;
      {for i<-1 to n1 do
          v[(j-1)*n1 +i] = z[i];
          v[(j-1)*n1 +i +stride] = z[n1+i];  } }
in v};
%            1-D FFT on a slice:   I-Structure Code
```

In the above code the function *slice* copies a slice of values from x, and the vector *ro* contains roots of unity. When the function is implemented in imperative M-structure style, we need to sequentialize the problem using explicit barriers before every stage, because the arrays y and z will be reused. The function *read_slice* reads a slice of values from x and updates y with these values. The function *fft* updates z. Again, *ro* contains roots of unity.

```
def cffts IS  n1 n2 x ro = {
typeof x = M_vector(F); typeof ro = I_vector(F);
stride = n1*n2; size = n1*2;
z = {M_array (1,size) of |   [j] = 0.0 || j<- 1 to size};
y = {M_array (1,size) of |   [j] = 0.0 || j<- 1 to size};
---
  {for j<-1 to n2 do
     _ = read_slice j n1 stride x y;
     ---
     _ = fft y ro IS z;
     ---
     {for i<-1 to n1 do
        x!![(j-1)*n1 +i] = z!![i];
        x!![(j-1)*n1 +i +stride] = z!![n1+i];  } }
in x};
%            1-D FFT on a slice:  Imperative M-Structure Code
```

When writing the same code in a data-driven M-structure style, we get rid off all the explicit barriers except one. Lack of this barrier causes writing on the same location of x (within the i loop), before it is extracted (by function *get_y*). The function *get_slice* extracts values out of x and puts them in y. The function fft extracts the values out of y and puts them in z, which is emptied again in the code that rewrites the slice of x that was emptied by *get_slice*.

```
def cffts IS  n1 n2 x ro = {
typeof x = M_vector(F); typeof ro = I_vector(F);
stride = n1*n2; size = n1*2;
z = 1d_X_array(1,size); y = 1d_X_array(1,size);
  {for j<-1 to n2 do
% fn get_y fills y and empties a section of x
     _ = get_y j n1 stride x y;
     _ = fft y ro IS z;
     ---
     {for i<-1 to n1 do
        x![(j-1)*n1 +i] = z![i];
        x![(j-1)*n1 +i +stride] = z![n1+i];  } }
in x};
%            1-D FFT on a slice:  Data-driven M-Structure Code
```

Building an array out of a variable number of variable sized sub-arrays turns out to be quite hard to implement in the purely functional style. The problem here is that the output value of one element of a slice is not known independently, as for one input slice, all the elements of the resulting slice are evaluated. It would be too expensive to evaluate a whole slice of elements just to get the value of one element. To get around this problem, we create an intermediate vector of vectors, each vector representing a slice.

```
def cffts IS  n1 n2 x ro = {
stride = n1*n2; size = n1*2; length = 2*stride;
typeof tv = vector(vector(F));
tv = { vector (1,n2) of
        | [j] = get1d j n1 stride x || j <- 1 to n2};

  defsubst get1d j n1 stride x = {
    y = get_y j n1 stride x;
    z = fft y ro IS;
  in z};

v = { vector (1,length) of
       | [(j-1)*n1 +i] = tv[j][i] || j <- 1 to n2; i <- 1 to n1
       | [(j-1)*n1 +i +stride] = tv[j][i+n1] || j <- 1 to n2;  i <- 1 to n1};
in v};
%            1-D FFT on a slice:  Functional Code
```

4 Results and Analysis

In this section we measure instruction counts and critical path length using the Monsoon Interpreter Mint. Two time related measures are recorded: S_1, the total number of instructions executed, and the *critical path length* S_∞, the number of parallel time steps required, when the availability of an unbounded number of processors is assumed. We measure the space requirements of our codes by determining the largest possible problem size that fits on a one node Monsoon Machine.

4.1 Time Analysis

Table 1 summarizes the instruction counts (S_1) and critical path lengths S_∞ for some problem sizes. The functional code has the maximum instruction count. This is caused by the inability to share computation in array comprehensions, and by the need to create intermediate data structures, as shown in the slice example. The I-structure code has the lowest instruction count, and a critical path length close to the functional code, which indicates that the higher parallelism of the functional code is superfluous. The data-driven M-structure code requires about 20% more instructions than the I-structure code. Also, the M-structure code has a 75% to 100% longer critical path length. This is caused by the need for explicit synchronization, and by the fact that the M-structure X, after the checksum has been computed, needs to be completely emptied before it can be reused in (producer-consumer style) in a next stage. This occurs after the checksum operation. Also, the code performing the checksum needs to use the more expensive *reads* in order to simplify the emptying of the array.

4.2 Space Analysis

To reach the goal of running a $64 \times 64 \times 64$ size problem on the one node Monsoon machine, we first need to determine how many such arrays can be stored before we run out of heap memory. Running just the random vector generator (generating U in Figure 1), we establish that the

Method	Functional		I-structure		M-structure	
	S_1	S_∞	S_1	S_∞	S_1	S_∞
4x4x4	3,241	220	1,025	230	1,257	340
8x4x4	6,403	300	1,556	300	1,902	520
8x8x4	13,729	480	2,665	480	3,211	840
8x8x8	32,303	800	4,973	800	5,877	1,600

Table 1: S_1 and S_∞ for FT benchmark ($\times$ 1000)

maximum size of a 3-D object that can be generated on our machine is $128 \times 128 \times 80$, and that at most 4 $64 \times 64 \times 64$ arrays can exist at the same time. According to the benchmark specification (see Figure 1), Exp needs to be created once and remains needed throughout the program. In the first stage of the program U and V coexist. After that, only V will be needed. In the cycle for t from 1 to 6, W and X coexist. Therefore at least 4 3-D objects must coexist. In the M-structure implementations we can over-write V on U and X on W, which brings the requirement down to 3 3-D objects. From this we conclude that on a one node Monsoon, we can never run any larger than 64^3 problem.

To establish the space usage of our 1-D FFT codes, we ran these independently of the rest of the FT benchmark. The functional code without resource management allows problem sizes of up to 2^{16}, whereas with resource management it allows problem sizes of up to 2^{18}. The corresponding numbers for the I-structure implementation are 2^{17} and 2^{18}. The M-structure code does not require explicit resource management as it reuses its data structures. It allows problem sizes up to 2^{19}. The double capacity of the M-structure code is explained by the fact that it writes the resulting fft back on its input array.

The maximal FT benchmark problem sizes we could run were 16^3 for the resource managed functional code, 32^3 for the I-structure code, and 64^3 for the M-structure code. This phenomenon corroborates our discussion in the implementation section. The functional implementation creates, even when resource managed, intermediate structures. The I-structure implementation requires independent structures for input and output, whereas the M-structure implementation can write results back into the input structure.

5 Conclusion

I-Structures and M-structures are introduced in [1] and [3], respectively. Id implementations of the 1D FFT are discussed in [4]. Sisal implementations of the 1D FFT are discussed in [5]. A high performance 1D FFT algorithm in Sisal for a vector computer is presented in [6]. The contribution of this paper is the quantitative comparison of the different declarative programming styles using a realistic, well-known benchmark.

In this paper we have studied certain declarative language features and their effect on the time and space efficiency of our programs. More specifically, we have studied three declarative implementations of the NAS FT benchmark: a purely functional, an I-structure, and an M-structure implementation, all written in the programming language Id and executed on the Monsoon Interpreter and Monsoon hardware. The purely functional code provides the most parallelism, but at the cost of a high instruction count. I-structures provides the fastest implementation: the lowest instruction count and a critical path length very close to that of the functional code. However, I-structure code is less space efficient than M-structure code. Only the M-structure code allows the 64^3 problem specified in the FT benchmark to be run. Therefore, we have the ability to trade space for time between the M-structure and I-structure implementations of this benchmark.

References

[1] Arvind, Nikhil R.S., and Pingali, K.K., "I-structures: Data Structures for Parallel Computing" ACM Transactions on Programming Languages and Systems, Vol 11, No 4, October 1989, pp 589-632.

[2] Bailey, D., et. al., "The NAS Parallel Benchmarks", Report RNR-91-002 revision 2, NASA Ames Research Center, 1991.

[3] Barth, Paul S., R. S. Nikhil and Arvind, "M-structures: Extending a parallel, non-strict, functional language with state," **Proc. Functional Prog Languages and Comp Arch**, Cambridge, MA, Aug 1991.

[4] Böhm, A. P. W. and Hiromoto R.E.,"Dataflow Time and Space Complexity of FFTs", **Journal of Parallel and Distributed Computing** no. 18, pp , 1993.

[5] Bollman, D., Sanmiguel, F. and Seguel, J., "Implementing FFT's in Sisal" Proceedings of the Second Sisal Users Conference, December 1992, Lawrence Livermore National Laboratory Report CONF-9210270.

[6] Feo, J. and Cann, D., "Developing a high-performance FFT algorithm in Sisal for a vector supercomputer", Proceedings Sisal'93, October 1993, Lawrence Livermore National Laboratory Report CONF-9310206.

[7] Hicks, James, D. Chiou, B. S. Ang and Arvind, "Performance studies of Id on the Monsoon dataflow system," **Journal of Parallel and Distributed Computing** no. 18, pp 273-300, 1993.

[8] R.S. Nikhil, *Id (version 90.0)* Reference Manual. TR CSG Memo 284-1, MIT LCS 1990.